Parlez-nous de l'expérience de téléchargement de PDF.

Documentation sur le langage C++

Apprenez à utiliser C++ et la bibliothèque standard C++.

Découvrir C++ dans Visual Studio

TÉLÉCHARGER

Télécharger Visual Studio pour Windows

Installer la prise en charge de C/C++ dans Visual Studio

Télécharger uniquement les outils de build à partir de la ligne de commande

BIEN DÉMARRER

Hello world dans Visual Studio avec C++

Créer une calculatrice console en C++

VIDEO

Découvrir C++ - langage et bibliothèque à usage général

ENTRAINEMENT

Re-bienvenue dans C++ - C++ moderne

Exemples et archive des exemples

Nouveautés de C++ dans Visual Studio

NOUVEAUTÉS

Nouveautés de C++ dans Visual Studio

Améliorations de la conformité de C++

VUE D'ENSEMBLE

Présentation du développement C++ dans Visual Studio

Plateformes cibles prises en charge

Aide et ressources de la communauté

Signaler un problème ou faire une suggestion

Utiliser le compilateur et les outils

RÉFÉRENCE

Informations de référence sur les builds C/C++

Projets et systèmes de build

Informations de référence sur le compilateur

Informations de référence sur l'éditeur de liens

Autres outils de build

Erreurs et avertissements

Langage C++

RÉFÉRENCE

Informations de référence sur le langage C++

Mots clés C++

Opérateurs C++

Informations de référence sur le préprocesseur C/C++

Bibliothèque standard C++ (STL)

RÉFÉRENCE

Vue d'ensemble de la bibliothèque standard C++

Informations de référence sur la bibliothèque standard C++ par en-tête

Conteneurs de la bibliothèque standard C++

Iterators

Algorithmes

Allocateurs

Objets de fonction

iostream (programmation)

Expressions régulières

Navigation dans le système de fichiers

Référence du langage C++

Article • 03/04/2023

Cette référence explique le langage de programmation C++ tel qu'implémenté dans le compilateur Microsoft C++. La organization est basée sur *le Manuel de référence C++ annoté* de Margaret Ellis et Bjarne Stroustrup et sur la Norme internationale ANSI/ISO C++ (ISO/IEC FDIS 14882). Les implémentations spécifiques à Microsoft des fonctionnalités du langage C++ sont incluses.

Pour obtenir une vue d'ensemble des pratiques de programmation C++ modernes, consultez Bienvenue dans C++.

Consultez les tableaux suivants pour rechercher rapidement un mot clé ou un opérateur :

- Mots clés C++
- Opérateurs C++

Dans cette section

Conventions lexicales
Éléments lexicaux fondamentaux de programme C++ : jetons, commentaires, opérateurs, mots clés, signes de ponctuation, littéraux. En outre, traduction de fichier, priorité/associativité des opérateurs.

Concepts de base
Portée, liaison, démarrage et arrêt de programme, classes de stockage et types.

Types intégrés Types fondamentaux intégrés au compilateur C++ et leurs plages de valeurs.

Standard Conversions
Conversions de types entre des types intégrés. Et aussi les conversions arithmétiques et les conversions entre le pointeur, la référence et les types pointeur vers membre.

Déclarations et définitions Déclaration et définition de variables, de types et de fonctions.

Opérateurs, priorité et associativité
Les opérateurs dans C++.

Expressions
Types d'expressions, sémantique des expressions, rubriques de référence sur les opérateurs, exécution de cast et opérateurs de cast, informations de type à l'exécution.

Expressions lambda
Technique de programmation qui définit implicitement une classe d'objets de fonction et construit un objet de fonction de ce type de classe.

Instructions
Instructions d'expression, Null, composite, de sélection, d'itération et de déclaration.

Classes et structs
Introduction aux classes, structures et unions. En outre, les fonctions membres, les fonctions membres spéciales, les membres de données, les champs de bits, `this` le pointeur, les classes imbriquées.

Unions
Types définis par l'utilisateur dans lesquels tous les membres partagent le même emplacement de mémoire.

Classes dérivées
Héritage unique et multiple, `virtual` fonctions, classes de base multiples, classes **abstraites** , règles d'étendue. En outre, les `__super` mots clés et `__interface` .

Contrôle d'accès aux membres
Contrôle de l'accès aux membres de la classe : `public`, `private` et `protected` mots clés. Fonctions et classes Friend.

Surcharge
Opérateurs surchargés, règles de surcharge d'opérateur.

Gestion des exceptions
La gestion des exceptions C++, la gestion structurée des exceptions (SEH) et les mots clés utilisés pour écrire des instructions de gestion des exceptions.

Messages d'assertion et de User-Supplied
`#error` directive, le `static_assert` mot clé, la `assert` macro.

Modèles
Spécifications de modèle, modèles de fonction, modèles de classe, `typename` mot clé, modèles et macros, modèles et pointeurs intelligents.

Gestion des événements
Déclaration d'événements et gestionnaires d'événements.

Modificateurs spécifiques à Microsoft

Modificateurs spécifiques à Microsoft C++. Adressage de mémoire, conventions d'appel, `naked` fonctions, attributs de classe de stockage étendus (`__declspec`), `__w64`.

Assembleur inline

Utilisation du langage d'assembly et de C++ dans les `__asm` blocs.

Prise en charge com du compilateur

Référence aux classes spécifiques de Microsoft et aux fonctions globales utilisées pour prendre en charge les types COM.

Microsoft Extensions

Extensions Microsoft en C++.

Comportement non standard

Informations sur le comportement non standard du compilateur Microsoft C++.

Bienvenue dans C++

Vue d'ensemble des pratiques de programmation C++ modernes pour l'écriture de programmes fiables, corrects et efficaces.

Sections connexes

Extensions de composant pour les plateformes Runtime

Documentation de référence sur l'utilisation du compilateur Microsoft C++ pour cibler .NET.

Référence à la génération C/C++

Options du compilateur, options de l'éditeur de liens et autres outils de génération.

Référence du préprocesseur C/C++

Documents de référence sur les pragmas, les directives de préprocesseur, les macros prédéfinies et le préprocesseur.

Bibliothèques Visual C++

Liste de liens vers les pages de démarrage de référence pour les différentes bibliothèques Microsoft C++.

Voir aussi

Informations de référence sur le langage C

Re-bienvenue dans C++ - C++ moderne

Article • 16/06/2023

Depuis sa création, C++ est devenu l'un des langages de programmation les plus utilisés au monde. Les programmes C++ bien écrits sont rapides et efficaces. La langue est plus flexible que les autres langues : elle peut fonctionner au plus haut niveau d'abstraction, et vers le bas au niveau du silicium. C++ fournit des bibliothèques standard hautement optimisées. Il permet d'accéder aux fonctionnalités matérielles de bas niveau, afin d'optimiser la vitesse et de réduire les besoins en mémoire. C++ peut créer presque n'importe quel type de programme : jeux, pilotes d'appareil, HPC, cloud, ordinateur de bureau, applications incorporées et mobiles, et bien plus encore. Même les bibliothèques et les compilateurs pour d'autres langages de programmation sont écrits en C++.

L'une des spécifications d'origine du C++ était la compatibilité descendante avec le langage C. Par conséquent, C++ a toujours autorisé la programmation de style C, avec des pointeurs bruts, des tableaux, des chaînes de caractères null et d'autres fonctionnalités. Ils peuvent permettre d'excellentes performances, mais peuvent également générer des bogues et de la complexité. L'évolution de C++ a mis l'accent sur des fonctionnalités qui réduisent considérablement la nécessité d'utiliser des idiomes de style C. Les anciennes installations de programmation C sont toujours là quand vous en avez besoin. Toutefois, dans le code C++ moderne, vous devez en avoir de moins en moins besoin. Le code C++ moderne est plus simple, plus sûr, plus élégant et toujours aussi rapide que jamais.

Les sections suivantes fournissent une vue d'ensemble des fonctionnalités main de C++ moderne. Sauf indication contraire, les fonctionnalités répertoriées ici sont disponibles en C++11 et versions ultérieures. Dans le compilateur Microsoft C++, vous pouvez définir l'option du /std compilateur pour spécifier la version de la norme à utiliser pour votre projet.

Ressources et pointeurs intelligents

L'une des principales classes de bogues dans la programmation de style C est la *fuite de mémoire*. Les fuites sont souvent causées par un échec d'appel `delete` de la mémoire allouée avec `new`. C++ moderne met l'accent sur le principe de *l'acquisition de ressources est l'initialisation* (RAII). L'idée est simple. Les ressources (mémoire du tas, handles de fichiers, sockets, etc.) doivent *appartenir à* un objet. Cet objet crée ou reçoit la ressource nouvellement allouée dans son constructeur et la supprime dans son destructeur. Le

principe de RAII garantit que toutes les ressources sont correctement retournées au système d'exploitation lorsque l'objet propriétaire sort de l'étendue.

Pour prendre en charge l'adoption facile des principes RAII, la bibliothèque standard C++ fournit trois types de pointeurs intelligents : std::unique_ptr, std::shared_ptret std::weak_ptr. Un pointeur intelligent gère l'allocation et la suppression de la mémoire qu'il possède. L'exemple suivant montre une classe avec un membre de tableau alloué sur le tas dans l'appel à `make_unique()`. Les appels à **`new`** et **`delete`** sont encapsulés par la `unique_ptr` classe . Lorsqu'un `widget` objet sort de l'étendue, le destructeur unique_ptr est appelé et il libère la mémoire allouée pour le tableau.

C++

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

Dans la mesure du possible, utilisez un pointeur intelligent pour gérer la mémoire du tas. Si vous devez utiliser explicitement les **`new`** opérateurs et **`delete`** , suivez le principe de RAII. Pour plus d'informations, consultez Gestion des ressources et de la durée de vie des objets (RAII).

`std::string` et `std::string_view`

Les chaînes de style C sont une autre source majeure de bogues. En utilisant std::string et std::wstring, vous pouvez éliminer pratiquement toutes les erreurs associées aux chaînes de style C. Vous bénéficiez également des fonctions membres pour la recherche, l'ajout, l'imminence, etc. Les deux sont hautement optimisés pour la vitesse. Lorsque vous transmettez une chaîne à une fonction qui ne nécessite qu'un accès en lecture seule, en C++17, vous pouvez l'utiliser std::string_view pour encore plus d'avantages en matière de performances.

`std::vector` et d'autres conteneurs de bibliothèque standard

Les conteneurs de bibliothèque standard suivent tous le principe de RAII. Ils fournissent des itérateurs pour une traversée sûre des éléments. De plus, ils sont hautement optimisés pour les performances et ont été soigneusement testés pour leur exactitude. En utilisant ces conteneurs, vous éliminez le risque de bogues ou d'inefficacités qui peuvent être introduits dans des structures de données personnalisées. Au lieu de tableaux bruts, utilisez vector comme conteneur séquentiel en C++.

```
C++

vector<string> apples;
apples.push_back("Granny Smith");
```

Utilisez map (pas `unordered_map`) comme conteneur associatif par défaut. Utilisez set, multimapet multiset pour les cas dégénérés et multiples.

```
C++

map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

Lorsque l'optimisation des performances est nécessaire, envisagez d'utiliser :

- Le array type lors de l'incorporation est important, par exemple, en tant que membre de classe.
- Conteneurs associatifs non triés tels que unordered_map. Ils ont une charge par élément plus faible et une recherche à temps constant, mais ils peuvent être plus difficiles à utiliser correctement et efficacement.
- Trié `vector`. Pour plus d'informations, consultez Algorithmes.

N'utilisez pas de tableaux de style C. Pour les API plus anciennes qui ont besoin d'un accès direct aux données, utilisez des méthodes d'accesseur telles que `f(vec.data(), vec.size());` à la place. Pour plus d'informations sur les conteneurs, consultez Conteneurs de bibliothèque standard C++.

Algorithmes de bibliothèque standard

Avant de supposer que vous devez écrire un algorithme personnalisé pour votre programme, passez d'abord en revue les algorithmes de la bibliothèque C++ Standard. La bibliothèque standard contient un assortiment toujours croissant d'algorithmes pour de nombreuses opérations courantes telles que la recherche, le tri, le filtrage et la randomisation. La bibliothèque de mathématiques est étendue. En C++17 et versions ultérieures, des versions parallèles de nombreux algorithmes sont fournies.

Voici quelques exemples importants :

- `for_each`, l'algorithme de traversée par défaut (avec les boucles basées sur `for` la plage).
- `transform`, pour la modification non sur place des éléments de conteneur
- `find_if`, l'algorithme de recherche par défaut.
- `sort`, `lower_bound` et les autres algorithmes de tri et de recherche par défaut.

Pour écrire un comparateur, utilisez strict `<` et utilisez *des lambdas nommés* lorsque vous le pouvez.

C++

```
auto comp = [](const widget& w1, const widget& w2)
     { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), widget{0}, comp );
```

`auto` au lieu de noms de type explicites

C++11 a introduit le auto mot clé pour une utilisation dans les déclarations de variables, de fonction et de modèle. `auto` indique au compilateur de déduire le type de l'objet afin que vous n'ayez pas à le taper explicitement. `auto` est particulièrement utile lorsque le type déduit est un modèle imbriqué :

C++

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Boucles basées sur `for` une plage

L'itération de style C sur les tableaux et les conteneurs est sujette aux erreurs d'indexation et est également fastidieuse pour le type. Pour éliminer ces erreurs et rendre votre code plus lisible, utilisez des boucles basées sur `for` des plages avec des conteneurs de bibliothèque standard et des tableaux bruts. Pour plus d'informations, consultez Instruction basée sur une plagefor.

C++

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

`constexpr` expressions au lieu de macros

Les macros en C et C++ sont des jetons qui sont traités par le préprocesseur avant la compilation. Chaque instance d'un jeton de macro est remplacé par sa valeur ou expression définie avant la compilation du fichier. Les macros sont couramment utilisées dans la programmation de style C pour définir des valeurs constantes au moment de la compilation. Toutefois, les macros sont sujettes aux erreurs et difficiles à déboguer. En C++moderne, vous devez préférer les constexpr variables pour les constantes au moment de la compilation :

C++

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Initialisation uniforme

En C++moderne, vous pouvez utiliser l'initialisation d'accolade pour n'importe quel type. Cette forme d'initialisation est particulièrement pratique lors de l'initialisation de tableaux, de vecteurs ou d'autres conteneurs. Dans l'exemple suivant, `v2` est initialisé avec trois instances de `S`. `v3` est initialisé avec trois instances de `S` qui sont elles-mêmes initialisées à l'aide d'accolades. Le compilateur déduit le type de chaque élément en fonction du type déclaré de `v3`.

C++

```
#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}
```

Pour plus d'informations, consultez Initialisation de l'accolade.

Sémantique de mouvement

C++ moderne fournit une *sémantique de déplacement*, qui permet d'éliminer les copies de mémoire inutiles. Dans les versions antérieures de la langue, les copies étaient inévitables dans certaines situations. Une opération de *déplacement* transfère la propriété d'une ressource d'un objet à l'autre sans effectuer de copie. Certaines classes possèdent des ressources telles que la mémoire du tas, les handles de fichier, etc.

Lorsque vous implémentez une classe propriétaire de ressources, vous pouvez définir un *constructeur de déplacement* et *un opérateur d'affectation de déplacement* pour celle-ci. Le compilateur choisit ces membres spéciaux lors de la résolution de surcharge dans les situations où une copie n'est pas nécessaire. Les types de conteneurs de bibliothèque standard appellent le constructeur de déplacement sur des objets s'ils sont définis. Pour plus d'informations, consultez Déplacer les constructeurs et les opérateurs d'affectation de déplacement (C++).

Expressions lambda

Dans la programmation de style C, une fonction peut être passée à une autre fonction à l'aide d'un *pointeur de fonction*. Les pointeurs de fonction ne sont pas pratiques à gérer et à comprendre. La fonction à laquelle ils font référence peut être définie ailleurs dans le code source, loin du point où elle est appelée. En outre, ils ne sont pas de type sécurisé. C++ moderne fournit des *objets de fonction*, des classes qui remplacent l'opérateur operator() , ce qui leur permet d'être appelés comme une fonction. Le moyen le plus pratique de créer des objets de fonction consiste à utiliser des expressions lambda inline. L'exemple suivant montre comment utiliser une expression lambda pour passer un objet de fonction que la `find_if` fonction appelle sur chaque élément du vecteur :

C++

```
    std::vector<int> v {1,2,3,4,5};
    int x = 2;
    int y = 4;
    auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i <
y; });
```

L'expression `[=](int i) { return i > x && i < y; }` lambda peut être lue comme « fonction qui prend un argument unique de type `int` et retourne un booléen qui indique si l'argument est supérieur `x` à et inférieur à `y` ». Notez que les variables `x` et `y` le contexte environnant peuvent être utilisés dans le lambda. spécifie `[=]` que ces variables sont *capturées* par valeur ; en d'autres termes, l'expression lambda a ses propres copies de ces valeurs.

Exceptions

C++ moderne met l'accent sur les exceptions, et non sur les codes d'erreur, comme le meilleur moyen de signaler et de gérer les conditions d'erreur. Pour plus d'informations,

consultez Meilleures pratiques en C++ modernes pour les exceptions et la gestion des erreurs.

`std::atomic`

Utilisez le struct de bibliothèque std::atomic standard C++ et les types associés pour les mécanismes de communication entre threads.

`std::variant` (C++17)

Les unions sont couramment utilisées dans la programmation de style C pour conserver la mémoire en permettant aux membres de différents types d'occuper le même emplacement de mémoire. Toutefois, les unions ne sont pas sécurisées de type et sont sujettes aux erreurs de programmation. C++17 présente la std::variant classe comme une alternative plus robuste et plus sûre aux unions. La std::visit fonction peut être utilisée pour accéder aux membres d'un `variant` type de manière sécurisée.

Voir aussi

Informations de référence sur le langage C++
Expressions lambda
Bibliothèque C++ standard
Conformité du langage Microsoft C/C++

Conventions lexicales

Article • 03/04/2023

Cette section introduit les éléments fondamentaux d'un programme C++. Vous utilisez ces éléments, appelés « éléments lexicaux » ou « jetons », pour élaborer les instructions, les définitions, les déclarations, etc., utilisées pour construire des programmes complets. Les éléments lexicaux suivants sont présentés dans cette section :

- Jetons et jeux de caractères
- Commentaires
- Identificateurs
- Mots clés
- Signes de ponctuation
- Littéraux numériques, booléens et de pointeur
- Littéraux de chaîne et de caractère
- Littéraux définis par l'utilisateur

Pour plus d'informations sur l'analyse des fichiers sources C++, consultez Phases de traduction.

Voir aussi

Informations de référence sur le langage C++
Unités de traduction et liaison

Jetons et jeux de caractères

Article • 16/06/2023

Le texte d'un programme C++ se compose de jetons et *d'espaces blancs*. Un jeton est le plus petit élément d'un programme C++ qui est significatif pour le compilateur. L'analyseur C++ reconnaît les types de jetons suivants :

- Mots clés
- Identificateurs
- Littéraux numériques, booléens et pointeurs
- Littéraux de chaîne et de caractères
- Littéraux définis par l'utilisateur
- Opérateurs
- Signes de ponctuation

Les jetons sont généralement séparés par des *espaces blancs*, qui peuvent être un ou plusieurs :

- Espaces
- Tabulations horizontales ou verticales
- Nouvelles lignes
- Flux de formulaire
- Commentaires

Jeu de caractères sources de base

La norme C++ spécifie un *jeu de caractères source de base* qui peut être utilisé dans les fichiers sources. Pour représenter des caractères qui ne font pas partie de ce jeu, des caractères supplémentaires peuvent être spécifiés à l'aide d'un *nom de caractère universel*. L'implémentation MSVC autorise des caractères supplémentaires. Le *jeu de caractères source de base* se compose de 96 caractères qui peuvent être utilisés dans les fichiers sources. Ce jeu comprend le caractère d'espace, la tabulation horizontale, la tabulation verticale, les caractères de contrôle de saut de page et de nouvelle ligne, ainsi que cet ensemble de caractères graphiques :

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
0 1 2 3 4 5 6 7 8 9
```

`_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '`

Section spécifique à Microsoft

MSVC inclut le `$` caractère en tant que membre du jeu de caractères source de base. MSVC autorise également l'utilisation d'un ensemble supplémentaire de caractères dans les fichiers sources, en fonction de l'encodage du fichier. Par défaut, Visual Studio stocke les fichiers sources en utilisant la page de codes par défaut. Lorsque les fichiers sources sont enregistrés à l'aide d'une page de code spécifique aux paramètres régionaux ou d'une page de code Unicode, MSVC vous permet d'utiliser tous les caractères de cette page de code dans votre code source, à l'exception des codes de contrôle qui ne sont pas explicitement autorisés dans le jeu de caractères source de base. Par exemple, vous pouvez placer des caractères du japonais dans les commentaires, les identificateurs ou les littéraux de chaîne si vous enregistrez le fichier en utilisant une page de codes du japonais. MSVC n'autorise pas les séquences de caractères qui ne peuvent pas être traduites en caractères multioctets ou points de code Unicode valides. Selon les options du compilateur, certains caractères autorisés ne peuvent pas apparaître dans les identificateurs. Pour plus d'informations, consultez Identificateurs.

FIN de la section spécifique à Microsoft

noms de caractères universels

Comme les programmes C++ peuvent utiliser bien plus de caractères que ceux qui sont spécifiés dans le jeu de caractères sources de base, vous pouvez spécifier ces caractères d'une façon portable en utilisant des *noms de caractères universels*. Un nom de caractère universel est constitué d'une séquence de caractères qui représentent un point de code Unicode. Celles-ci peuvent prendre deux formes. Utilisez `\UNNNNNNNN` pour représenter un point de code Unicode de la forme U+NNNNNNNN, où NNNNNNNN est le numéro du point de code hexadécimal sur huit chiffres. Utilisez la séquence de quatre chiffres `\uNNNN` pour représenter un point de code Unicode de la forme U+0000NNNN.

Les noms de caractères universels peuvent être utilisés dans les identificateurs, et dans les littéraux de chaînes et de caractères. Un nom de caractère universel ne peut pas être utilisé pour représenter un point de code de substitution dans la plage 0xD800-0xDFFF. Au lieu de cela, utilisez le point de code souhaité : le compilateur génère automatiquement les caractères de substitution requis. D'autres restrictions s'appliquent aux noms de caractères universels qui peuvent être utilisés dans les identificateurs. Pour plus d'informations, consultez Identifiers et String and Character Literals.

Section spécifique à Microsoft

Le compilateur Microsoft C++ traite indifféremment un caractère sous forme de nom de caractère universel et de forme littérale. Par exemple, vous pouvez déclarer un identificateur en utilisant la forme de nom de caractère universel et l'utiliser sous forme de littéral :

C++

```
auto \u30AD = 42; // \u30AD is 'キ'
if (キ == 42) return true; // \u30AD and キ are the same to the compiler
```

Le format des caractères étendus sur le Presse-papiers Windows est spécifique aux valeurs des paramètres régionaux de l'application. Le fait de couper et de coller ces caractères dans votre code depuis une autre application peut introduire des encodages de caractères inattendus. Ceci peut aboutir à des erreurs d'analyse sans cause visible dans votre code. Nous vous recommandons de définir l'encodage de vos fichiers sources sur une page de codes Unicode avant de coller des caractères étendus. Nous vous recommandons aussi d'utiliser un éditeur de méthode d'entrée (IME) ou l'application Character Map pour générer des caractères étendus.

FIN de la section spécifique à Microsoft

Jeux de caractères d'exécution

Les *jeux de caractères d'exécution* représentent les caractères et les chaînes qui peuvent apparaître dans un programme compilé. Ces jeux de caractères se composent de tous les caractères autorisés dans un fichier source, ainsi que des caractères de contrôle qui représentent l'alerte, le retour arrière, le retour chariot et le caractère null. Le jeu de caractères d'exécution a une représentation spécifique aux paramètres régionaux.

Commentaires (C++)

Article • 16/06/2023

Un commentaire est un texte que le compilateur ignore, mais qui est utile pour les programmeurs. Les commentaires sont normalement utilisés pour annoter du code en vue d'une référence ultérieure. Le compilateur les traite comme des espaces blancs. Vous pouvez utiliser des commentaires dans les tests pour rendre certaines lignes de code inactives ; Toutefois, `#if`/`#endif` les directives de préprocesseur fonctionnent mieux pour cela, car vous pouvez entourer du code qui contient des commentaires, mais vous ne pouvez pas imbriquer des commentaires.

Un commentaire C++ est écrit de l'une des manières suivantes :

- Caractères `/*` (barre oblique, astérisque), suivis de n'importe quelle séquence de caractères (y compris les nouvelles lignes), suivis des `*/` caractères. Cette syntaxe est identique à ANSI C.

- Caractères `//` (deux barres obliques), suivis de n'importe quelle séquence de caractères. Une nouvelle ligne qui n'est pas immédiatement précédée d'une barre oblique inverse met fin à cette forme de commentaire. Par conséquent, elle est communément appelée « commentaire de ligne unique ».

Les caractères de commentaire (`/*`, `*/`, et `//`) n'ont aucune signification particulière dans une constante de caractère, un littéral de chaîne ou un commentaire. Les commentaires utilisant la première syntaxe ne peuvent donc pas être imbriqués.

Voir aussi

Conventions lexicales

Identificateurs (C++)

Article • 03/04/2023

Un identificateur est une séquence de caractères servant à désigner l'un des éléments suivants :

- Nom d'objet ou de variable
- Nom de classe, de structure ou d'union
- Nom de type énuméré
- Membre d'une classe, d'une structure, d'une union ou d'une énumération
- Fonction ou fonction membre de classe
- Nom de typedef
- Nom d'étiquette
- Nom de macro
- Paramètre de macro

Les caractères suivants sont autorisés dans un identificateur :

```
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

Certaines plages de noms de caractères universels sont également autorisées dans un identificateur. Un nom de caractère universel dans un identificateur ne peut pas désigner un caractère de contrôle ou un caractère du jeu de caractères sources de base. Pour plus d'informations, consultez Character Sets. Ces plages de nombres de point de code Unicode sont autorisées en tant que noms de caractères universels pour n'importe quel caractère d'un identificateur :

- 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFD, 10000-1FFFD, 20000-

2FFFD, 30000-3FFFD, 40000-4FFFD, 50000-5FFFD, 60000-6FFFD, 70000-7FFFD, 80000-8FFFD, 90000-9FFFD, A0000-AFFFD, B0000-BFFFD, C0000-CFFFD, D0000-DFFFD, E0000-EFFFD

Les caractères suivants sont autorisés comme caractères quelconques d'un identificateur, à l'exception du premier caractère :

```
0 1 2 3 4 5 6 7 8 9
```

Ces plages de numéros de points de code Unicode sont également autorisées en tant que noms de caractères universels pour n'importe quel caractère d'un identificateur, à l'exception du premier :

- 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F

Section spécifique à Microsoft

Seuls les 2 048 premiers caractères des identificateurs Microsoft C++ sont significatifs. Les noms des types définis par l'utilisateur sont « décorés » par le compilateur de manière à conserver les informations de type. Le nom résultant, qui inclut les informations de type, ne peut pas comporter plus de 2 048 caractères. (Pour plus d'informations, consultez Noms décorés .) Les facteurs qui peuvent influencer la longueur d'un identificateur décoré sont les suivants :

- Si l'identificateur désigne ou non un objet de type défini par l'utilisateur ou un type dérivé d'un type défini par l'utilisateur.
- Si l'identificateur désigne ou non une fonction ou un type dérivé d'une fonction.
- Le nombre d'arguments d'une fonction.

Le signe `$` dollar est un caractère d'identificateur valide dans le compilateur Microsoft C++ (MSVC). MSVC vous permet également d'utiliser les caractères réels représentés par les plages autorisées de noms de caractères universels dans les identificateurs. Pour utiliser ces caractères, vous devez enregistrer le fichier à l'aide d'une page de codes d'encodage de fichier qui les inclut. Cet exemple montre comment utiliser les caractères étendus et les noms de caractères universels de manière interchangeable dans votre code.

C++

```
// extended_identifier.cpp
// In Visual Studio, use File, Advanced Save Options to set
```

```
// the file encoding to Unicode codepage 1200
struct テスト         // Japanese 'test'
{
    void トスト() {}  // Japanese 'toast'
};

int main() {
    テスト \u30D1\u30F3;  // Japanese パン 'bread' in UCN form
    パン.トスト();        // compiler recognizes UCN or literal form
}
```

La plage des caractères autorisés dans un identificateur est moins restrictive durant la compilation du code C++/CLI. Les identificateurs du code compilés à l'aide de /clr doivent suivre la norme ECMA-335 : Common Language Infrastructure (CLI).

FIN de la section spécifique à Microsoft

Le premier caractère d'un identificateur doit être un caractère alphabétique, majuscule ou minuscule, ou un trait de soulignement (_). Comme les identificateurs C++ respectent la casse, `fileName` est différent de `FileName`.

Les identificateurs ne peuvent pas avoir exactement la même orthographe et la même casse que les mots clés. Les identificateurs contenant des mots clés sont autorisés. Par exemple, `Pint` est un identificateur autorisé, bien qu'il contienne **`int`**, qui est un mot clé.

L'utilisation de deux caractères de soulignement séquentiels (__) dans un identificateur, ou d'un trait de soulignement de début unique suivi d'une lettre majuscule, est réservée aux implémentations C++ dans toutes les étendues. Évitez d'utiliser un seul trait de soulignement de début suivi d'une lettre minuscule pour les noms avec une portée de fichier en raison de conflits possibles avec les identificateurs réservés actuels ou futurs.

Voir aussi

Conventions lexicales

Mots clés (C++)

Article • 16/06/2023

Les mots clés sont des identificateurs réservés prédéfinis qui ont des significations particulières. Ils ne peuvent pas être utilisés comme identificateurs dans votre programme. Les mots clés suivants sont réservés pour Microsoft C++. Les noms avec des traits de soulignement de début et des noms spécifiés pour C++/CX et C++/CLI sont des extensions Microsoft.

Mots clés C++ standard

alignas
alignof
and[B]
and_eq[B]
asm[Un]
auto
bitand[B]
bitor[B]
bool
break
case
catch
char
char8_t[C]
char16_t
char32_t
class
compl[B]
`concept`[C]
const
const_cast
`consteval`[C]
constexpr

`constinit`[C]
continue
`co_await`[C]
`co_return`[C]

`co_yield`[C]
decltype
default
delete
do
double
dynamic_cast
else
enum
explicit
`export`[C]
extern
false
float
for
friend
goto
if
inline

int
long
mutable
namespace
new
noexcept
not[B]
not_eq[B]
nullptr
operator
or[B]
or_eq[B]
private
protected
public
register reinterpret_cast
`requires`[C]
return
short
signed
sizeof

static
static_assert

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using Déclaration
Directive using
virtual
void
volatile
wchar_t
while
xor[B]
xor_eq[B]

[a] Le mot clé spécifique `__asm` à Microsoft remplace la syntaxe C++ `asm`. `asm` est réservé pour la compatibilité avec d'autres implémentations C++, mais n'est pas implémenté. Utilisez `__asm` pour l'assembly inline sur des cibles x86. Microsoft C++ ne prend pas en charge l'assembly inline pour d'autres cibles.

[b] Les synonymes de l'opérateur étendu sont des mots clés lorsque /permissive- ou /Za (Désactiver les extensions de langage) est spécifié. Il ne s'agit pas de mots clés lorsque les extensions Microsoft sont activées.

[c] Pris en charge lorsque /std:c++20 ou une version ultérieure (par exemple `/std:c++latest`) est spécifiée.

Mots clés C++ spécifiques à Microsoft

En C++, les identificateurs qui contiennent deux traits de soulignement consécutifs sont réservés aux implémentations du compilateur. La convention Microsoft consiste à faire précéder des mots clés spécifiques à Microsoft avec des traits de soulignement doubles. Ces mots ne peuvent pas être utilisés en tant que noms d'identificateurs.

Les extensions Microsoft sont activées par défaut. Pour vous assurer que vos programmes sont entièrement portables, vous pouvez désactiver les extensions Microsoft en spécifiant l'option ou /Za (Désactiver les extensions de langage) pendant la /permissive- compilation. Ces options désactivent certains mots clés spécifiques à Microsoft.

Lorsque les extensions Microsoft sont activées, vous pouvez utiliser les mots clés Microsoft spécifiques dans vos programmes. Pour la conformité ANSI, ces mots clés sont précédés d'un double trait de soulignement. Pour la compatibilité descendante, les versions à trait de soulignement unique de la plupart des mots clés à double trait de soulignement sont prises en charge. Le `__cdecl` mot clé est disponible sans trait de soulignement de début.

La `__asm` mot clé remplace la syntaxe C++ `asm`. `asm` est réservé pour la compatibilité avec d'autres implémentations C++, mais n'est pas implémenté. Utiliser `__asm`.

Le `__based` mot clé a des utilisations limitées pour les compilations cibles 32 bits et 64 bits.

__alignof[E]
__asm[E]
__assume[E]
__based[E]
__cdecl[E]
__declspec[E]
__event
__except[E]
__fastcall[E]
__finally[E]
__forceinline[E]

__hook[D]
__if_exists
__if_not_exists
__inline[E]
__int16[E]
__int32[E]

__int64[E]
__int8[E]
__interface
__leave[E]
__m128

__m128d
__m128i
__m64
__multiple_inheritance[E]
__ptr32[E]
__ptr64[E]
__raise
__restrict[E]
__single_inheritance[E]
__sptr[E]
__stdcall[E]

__super
__thiscall
__unaligned[E]
__unhook[D]
__uptr[E]
__uuidof[E]
__vectorcall[E]
__virtual_inheritance[E]
__w64[E]
__wchar_t

[d] Fonction intrinsèque utilisée dans la gestion des événements.

[e] Pour une compatibilité descendante avec les versions précédentes, ces mots clés sont disponibles à la fois avec deux traits de soulignement de début et un trait de soulignement de début unique lorsque les extensions Microsoft sont activées (valeur par défaut).

Mots clés Microsoft dans __declspec modificateurs

Ces identificateurs sont des attributs étendus pour le `__declspec` modificateur. Ils sont considérés comme des mots clés dans ce contexte.

align
allocate
allocator
appdomain
code_seg
deprecated

dllexport
dllimport
jitintrinsic
naked
noalias
noinline

noreturn
no_sanitize_address
nothrow
novtable
process
property

restrict
safebuffers
selectany
spectre
thread
uuid

Mots clés C++/CLI et C++/CX

__abstract[F]
__box[F]
__delegate[F]
__gc[F]
__identifier
__nogc[F]
__noop
`__pin`[F]
`__property`[F]
`__sealed`[F]

__try_cast[F]
__value[F]
abstract[G]
array[G]
as_friend
delegate[G]
enum class
enum struct
event[G]

finally
for each in
gcnew[G]
generic[G]
initonly
interface class[G]
interface struct[G]
interior_ptr[G]
literal[G]

new[G]
property[G]
ref class
ref struct
safecast
sealed[G]
typeid
value class[G]
value struct[G]

[f] Applicable à Extensions managées pour C++ uniquement. Cette syntaxe est maintenant déconseillée. Pour plus d'informations, consultez Extensions de composant pour les plateformes Runtime.

[g] Applicable à C++/CLI.

Voir aussi

Conventions lexicales
Opérateurs intégrés C++, priorité et associativité

Signes de ponctuation (C++)

Article • 03/04/2023

Les signes de ponctuation en langage C++ ont une signification syntaxique et sémantique pour le compilateur mais ne spécifient pas d'eux-mêmes une opération qui produit une valeur. Certains signes de ponctuation, seuls ou combinés, peuvent également être des opérateurs C++ ou être significatifs pour le préprocesseur.

Tous les caractères suivants sont considérés comme des signes de ponctuation :

```
! % ^ & * ( ) - + = { } | ~
[ ] \ ; ' : " < > ? , . / #
```

Les ponctuateurs **[ ]**, **( )** et **{ }** doivent apparaître par paires après la phase de traduction 4.

Voir aussi

Conventions lexicales

Littéraux numériques, booléens et de pointeur

Article • 03/04/2023

Un littéral est un élément de programme qui représente directement une valeur. Cet article traite des littéraux de type entier, à virgule flottante, booléenne et pointeur. Pour plus d'informations sur les littéraux de chaîne et de caractères, consultez Littéraux de chaîne et de caractères (C++). Vous pouvez également définir vos propres littéraux en fonction de l'une de ces catégories. Pour plus d'informations, consultez Littéraux définis par l'utilisateur (C++).

Vous pouvez utiliser des littéraux dans de nombreux contextes, mais le plus souvent pour initialiser des variables nommées et passer des arguments à des fonctions :

C++

```
const int answer = 42;      // integer literal
double d = sin(108.87);     // floating point literal passed to sin function
bool b = true;              // boolean literal
MyClass* mc = nullptr;      // pointer literal
```

Parfois, il est important indiquer au compilateur comment interpréter un littéral ou le type spécifique à lui affecter. Pour ce faire, ajoutez des préfixes ou des suffixes au littéral. Par exemple, le préfixe `0x` indique au compilateur d'interpréter le nombre qui le suit comme une valeur hexadécimale, par exemple `0x35`. Le `ULL` suffixe indique au compilateur de traiter la valeur comme un **`unsigned long long`** type, comme dans `5894345ULL`. Consultez les sections suivantes pour obtenir la liste complète des préfixes et suffixes pour chaque type de littéral.

Littéraux d'entier

Les littéraux d'entier commencent par un chiffre et n'ont aucune partie fractionnaire ni exposant. Vous pouvez spécifier des littéraux entiers au format décimal, binaire, octal ou hexadécimal. Vous pouvez éventuellement spécifier un littéral entier comme non signé, et comme type long ou long, à l'aide d'un suffixe.

Quand aucun préfixe ou suffixe n'est présent, le compilateur donne un type **`int`** de valeur littérale intégrale (32 bits), si la valeur est adaptée, sinon il lui donne le type **`long long`** (64 bits).

Pour spécifier un littéral intégral décimal, commencez la spécification par un chiffre différent de zéro. Par exemple :

```
C++
```

```
int i = 157;         // Decimal literal
int j = 0198;        // Not a decimal number; erroneous octal literal
int k = 0365;        // Leading zero specifies octal literal, not decimal
int m = 36'000'000   // digit separators make large values more readable
```

Pour spécifier un littéral intégral octal, commencez la spécification par 0, suivi d'une séquence de chiffres dans la plage 0 à 7. Les chiffres 8 et 9 sont des erreurs lors de la spécification d'un littéral octal. Par exemple :

```
C++
```

```
int i = 0377;   // Octal literal
int j = 0397;   // Error: 9 is not an octal digit
```

Pour spécifier un littéral intégral hexadécimal, commencez la spécification par `0x` ou `0X` (le cas du « x » n'a pas d'importance), suivi d'une séquence de chiffres dans la plage `0` à travers `9` et `a` (ou `A`) à `f` (ou `F`). Les chiffres hexadécimaux `a` (ou `A`) à `f` (ou `F`) représentent des valeurs dans la plage 10 à 15. Par exemple :

```
C++
```

```
int i = 0x3fff;   // Hexadecimal literal
int j = 0X3FFF;   // Equal to i
```

Pour spécifier un type non signé, utilisez le `u` suffixe ou `U` . Pour spécifier un type long, utilisez le `l` suffixe ou `L` . Pour spécifier un type intégral 64 bits, utilisez le suffixe LL ou ll. Le suffixe i64 est toujours pris en charge, mais nous ne le recommandons pas. Il est spécifique à Microsoft et n'est pas portable. Par exemple :

```
C++
```

```
unsigned val_1 = 328u;                  // Unsigned value
long val_2 = 0x7FFFFFL;                 // Long value specified
                                        //  as hex literal
unsigned long val_3 = 0776745ul;        // Unsigned long value
auto val_4 = 108LL;                           // signed long long
auto val_4 = 0x800000000000000ULL << 16;      // unsigned long long
```

Séparateurs de chiffres : vous pouvez utiliser le caractère à guillemet unique (apostrophe) pour séparer les valeurs de place dans des nombres plus grands afin de faciliter leur lecture par les humains. Les séparateurs n'ont pas d'effet sur la compilation.

```
C++
```

```
long long i = 24'847'458'121;
```

Littéraux à virgule flottante

Les littéraux à virgule flottante spécifient des valeurs qui doivent avoir une partie fractionnaire. Ces valeurs contiennent des virgules décimales (`.`) et peuvent contenir des exposants.

Les littéraux à virgule flottante ont une *significande* (parfois appelée *mantisse*), qui spécifie la valeur du nombre. Ils ont un *exposant*, qui spécifie l'ampleur du nombre. De plus, ils ont un suffixe facultatif qui spécifie le type du littéral. La significande est spécifiée sous la forme d'une séquence de chiffres suivie d'un point, suivie d'une séquence facultative de chiffres représentant la partie fractionnaire du nombre. Par exemple :

```
C++
```

```
18.46
38.
```

L'exposant, s'il est présent, spécifie la grandeur du nombre comme puissance de 10, comme indiqué dans l'exemple suivant :

```
C++
```

```
18.46e0      // 18.46
18.46e1      // 184.6
```

L'exposant peut être spécifié à l'aide `e` de ou `E`, qui ont la même signification, suivi d'un signe facultatif (+ ou -) et d'une séquence de chiffres. Si un exposant est présent, la virgule décimale de fin est inutile dans les nombres entiers tels que `18E0`.

Les littéraux à virgule flottante sont de type **`double`** par défaut . En utilisant les suffixes `f` ou `l` ou ou `L` `F` (le suffixe ne respecte pas la casse), le littéral peut être spécifié en tant que **`float`** ou **`long double`**.

Bien que `long double` et `double` aient la même représentation, ils ne sont pas du même type. Par exemple, vous pouvez avoir des fonctions surchargées telles que

```
C++
```

```
void func( double );
```

et

```
C++
```

```
void func( long double );
```

littéraux booléens

Les littéraux booléens sont `true` et `false`.

Littéral de pointeur (C++11)

C++ introduit le nullptr littéral pour spécifier un pointeur initialisé à zéro. Dans le code portable, `nullptr` doit être utilisé à la place de zéro de type intégral ou de macros telles que `NULL`.

Littéraux binaires (C++14)

Un littéral binaire peut être spécifié à l'aide du préfixe `0B` ou `0b`, suivi d'une séquence de 1 et de 0 :

```
C++
```

```
auto x = 0B001101 ; // int
auto y = 0b000001 ; // int
```

Évitez d'utiliser des littéraux en tant que « constantes magiques »

Vous pouvez utiliser des littéraux directement dans les expressions et instructions bien qu'il ne s'agisse pas toujours d'une bonne pratique de programmation :

```
```

C++

```
if (num < 100)
    return "Success";
```

Dans l'exemple précédent, une meilleure pratique consiste à utiliser une constante nommée qui transmet une signification claire, par exemple « MAXIMUM_ERROR_THRESHOLD ». Et si la valeur de retour « Success » est visible par les utilisateurs finaux, il peut être préférable d'utiliser une constante de chaîne nommée. Vous pouvez conserver les constantes de chaîne à un emplacement unique dans un fichier qui peut être localisé dans d'autres langues. L'utilisation de constantes nommées vous aide, ainsi que les autres, à comprendre l'intention du code.

Voir aussi

Conventions lexicales
Littéraux de chaîne C++
Littéraux définis par l'utilisateur C++

Littéraux de chaîne et de caractères (C++)

Article • 03/04/2023

C++ prend en charge divers types de chaîne et de caractère, et fournit les moyens d'exprimer les valeurs littérales de chacun de ces types. Dans votre code source, vous exprimez le contenu de vos littéraux de caractère et de chaîne à l'aide d'un jeu de caractères. Les noms de caractères universels et les caractères d'échappement vous permettent d'exprimer une chaîne en utilisant uniquement le jeu de caractères sources de base. Un littéral de chaîne brut vous permet d'éviter d'utiliser des caractères d'échappement et peut servir à exprimer tous les types de littéral de chaîne. Vous pouvez également créer `std::string` des littéraux sans avoir à effectuer d'étapes de construction ou de conversion supplémentaires.

C++

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string
literals

int main()
{
    // Character literals
    auto c0 =   'A'; // char
    auto c1 = u8'A'; // char
    auto c2 =  L'A'; // wchar_t
    auto c3 =  u'A'; // char16_t
    auto c4 =  U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 =   "hello"; // const char*
    auto s1 = u8"hello"; // const char* before C++20, encoded as UTF-8,
                         // const char8_t* in C++20
    auto s2 =  L"hello"; // const wchar_t*
    auto s3 =  u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 =  U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 =   R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char* before C++20, encoded
as UTF-8,
                                      // const char8_t* in C++20
    auto R2 =  LR"("Hello \ world")"; // const wchar_t*
    auto R3 =  uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
```

```
    auto R4 =  UR("Hello \ world")"; // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 =   "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string before C++20, std::u8string in
C++20
    auto S2 =  L"hello"s; // std::wstring
    auto S3 =  u"hello"s; // std::u16string
    auto S4 =  U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 =   R("Hello \ world")"s; // std::string from a raw const char*
    auto S6 = u8R("Hello \ world")"s; // std::string from a raw const char*
before C++20, encoded as UTF-8,
                                      // std::u8string in C++20
    auto S7 =  LR("Hello \ world")"s; // std::wstring from a raw const
wchar_t*
    auto S8 =  uR("Hello \ world")"s; // std::u16string from a raw const
char16_t*, encoded as UTF-16
    auto S9 =  UR("Hello \ world")"s; // std::u32string from a raw const
char32_t*, encoded as UTF-32
}
```

Les littéraux de chaîne ne peuvent avoir aucun préfixe, ou `u8`, `L`, `u`et `U` des préfixes pour désigner des caractères étroits (un octet ou plusieurs octets), UTF-8, caractères larges (UCS-2 ou UTF-16), UTF-16 et UTF-32, respectivement. Un littéral de chaîne brute peut avoir `R` `u8R`, , `LR` `uR`, et `UR` des préfixes pour les équivalents de version brute de ces encodages. Pour créer des valeurs temporaires ou statiques `std::string` , vous pouvez utiliser des littéraux de chaîne ou des littéraux de chaîne bruts avec un `s` suffixe. Pour plus d'informations, consultez la section Littéraux de chaîne ci-dessous. Pour plus d'informations sur le jeu de caractères source de base, les noms de caractères universels et l'utilisation de caractères de pages de code étendues dans votre code source, consultez Jeux de caractères.

Littéraux de caractère

Un *littéral de caractère* est composé d'une constante caractère. Il est représenté par le caractère entouré de guillemets simples. Il existe cinq types de littéraux de caractères :

- Littéraux de caractères ordinaires de type **`char`**, par exemple `'a'`
- Littéraux de caractères UTF-8 de type **`char`** (**`char8_t`** en C++20), par exemple `u8'a'`
- Littéraux de caractères étendus de type **`wchar_t`**, par exemple `L'a'`

- Littéraux de caractères UTF-16 de type `char16_t`, par exemple `u'a'`
- Littéraux de caractères UTF-32 de type `char32_t`, par exemple `U'a'`

Le caractère utilisé pour un littéral de caractère peut être n'importe quel caractère, à l'exception de la barre oblique inverse des caractères réservés (`\`), d'un guillemet unique (`'`) ou d'une nouvelle ligne. Les caractères réservés peuvent être spécifiés à l'aide d'une séquence d'échappement. Vous pouvez spécifier des caractères à l'aide des noms de caractères universels, tant que le type est suffisamment grand pour contenir le caractère.

Encodage

Les littéraux de caractères sont encodés différemment en fonction de leur préfixe.

- Un littéral de caractère sans préfixe est un littéral de caractère ordinaire. La valeur d'un littéral de caractère ordinaire contenant un caractère unique, une séquence d'échappement ou un nom de caractère universel qui peut être représenté dans le jeu de caractères d'exécution a une valeur égale à la valeur numérique de son encodage dans le jeu de caractères d'exécution. Un littéral de caractères ordinaire qui contient plusieurs caractères, séquence d'échappement ou nom de caractère universel est un *littéral multicharacteur*. Un littéral multicharacter ou un littéral de caractère ordinaire qui ne peut pas être représenté dans le jeu de caractères d'exécution a le type `int`, et sa valeur est définie par l'implémentation. Pour MSVC, consultez la section **spécifique à Microsoft ci-dessous** .
- Un littéral de caractère qui commence par le `L` préfixe est un littéral à caractères larges. La valeur d'un littéral à caractères larges contenant un caractère unique, une séquence d'échappement ou un nom de caractère universel a une valeur égale à la valeur numérique de son encodage dans le jeu de caractères larges d'exécution, sauf si le littéral de caractères de caractère n'a aucune représentation dans le jeu de caractères larges d'exécution, auquel cas la valeur est définie par l'implémentation. La valeur d'un littéral à caractères larges contenant plusieurs caractères, des séquences d'échappement ou des noms de caractères universels est définie par l'implémentation. Pour MSVC, consultez la section **spécifique à Microsoft ci-dessous** .
- Un littéral de caractère commençant par le `u8` préfixe est un littéral de caractères UTF-8. La valeur d'un littéral de caractère UTF-8 contenant un caractère unique, une séquence d'échappement ou un nom de caractère universel a une valeur égale à sa valeur de point de code ISO 10646 s'il peut être représenté par une seule unité de code UTF-8 (correspondant au contrôle C0 et au bloc Unicode latin de base). Si la valeur ne peut pas être représentée par une seule unité de code UTF-8,

le programme est mal formé. Un littéral de caractères UTF-8 contenant plusieurs caractères, une séquence d'échappement ou un nom de caractère universel est mal formé.

- Un littéral de caractère qui commence par le `u` préfixe est un littéral de caractère UTF-16. La valeur d'un littéral de caractères UTF-16 contenant un caractère unique, une séquence d'échappement ou un nom de caractère universel a une valeur égale à sa valeur de point de code ISO 10646 s'il peut être représenté par une seule unité de code UTF-16 (correspondant au plan multilingue de base). Si la valeur ne peut pas être représentée par une seule unité de code UTF-16, le programme est mal formé. Un littéral de caractères UTF-16 contenant plusieurs caractères, une séquence d'échappement ou un nom de caractère universel est mal formé.

- Un littéral de caractère commençant par le `U` préfixe est un littéral de caractères UTF-32. La valeur d'un littéral de caractères UTF-32 contenant un caractère unique, une séquence d'échappement ou un nom de caractère universel a une valeur égale à sa valeur de point de code ISO 10646. Un littéral de caractères UTF-32 contenant plusieurs caractères, une séquence d'échappement ou un nom de caractère universel est mal formé.

Séquences d'échappement

Il existe trois types de séquence d'échappement : simple, octal et hexadécimal. Les séquences d'échappement peuvent être l'une des valeurs suivantes :

Valeur	**Séquence d'échappement**
saut de ligne	\n
barre oblique inverse	\\
tabulation horizontale	\t
point d'interrogation	? ou \?
tabulation verticale	\v
guillemet simple	\'
retour arrière	\b
guillemet double	\"
retour chariot	\r
caractère Null	\0

Valeur	Séquence d'échappement
saut de page	\f
octal	\ooo
alerte (clochette)	\a
hexadécimal	\xhhh

Une séquence d'échappement octale est une barre oblique inverse suivie d'une séquence de un à trois chiffres octaux. Une séquence d'échappement octale se termine au premier caractère qui n'est pas un chiffre octal, s'il est rencontré plus tôt que le troisième chiffre. La valeur octale la plus élevée possible est `\377`.

Une séquence d'échappement hexadécimale est une barre oblique inverse suivie du caractère `x`, suivie d'une séquence d'un ou plusieurs chiffres hexadécimaux. Les zéros non significatifs sont ignorés. Dans un littéral de caractères ordinaire ou préfixé u8, la valeur hexadécimale la plus élevée est 0xFF. Dans un littéral de caractère large avec le préfixe L ou le préfixe u, la valeur hexadécimale la plus élevée est 0xFFFF. Dans un littéral de caractère large avec le préfixe U, la valeur hexadécimale la plus élevée est 0xFFFFFFFF.

Cet exemple de code montre quelques exemples de caractères d'échappement utilisant des littéraux de caractères ordinaires. La même syntaxe de séquence d'échappement est valide pour les autres types littéraux de caractères.

C++

```
#include <iostream>
using namespace std;

int main() {
    char newline = '\n';
    char tab = '\t';
    char backspace = '\b';
    char backslash = '\\';
    char nullChar = '\0';

    cout << "Newline character: " << newline << "ending" << endl;
    cout << "Tab character: " << tab << "ending" << endl;
    cout << "Backspace character: " << backspace << "ending" << endl;
    cout << "Backslash character: " << backslash << "ending" << endl;
    cout << "Null character: " << nullChar << "ending" << endl;
}
/* Output:
Newline character:
ending
Tab character:  ending
```

```
Backspace character:ending
Backslash character: \ending
Null character:  ending
*/
```

La barre oblique inverse (`\`) est un caractère de continuation de ligne lorsqu'il est placé à la fin d'une ligne. Pour qu'une barre oblique inverse apparaisse comme un littéral de caractère, vous devez taper deux barres obliques inverses sur une ligne (`\\`). Pour plus d'informations sur le caractère de continuation de ligne, consultez Phases of Translation.

Spécifique à Microsoft

Pour créer une valeur à partir d'un littéral multicharacteur étroit, le compilateur convertit le caractère ou la séquence de caractères entre guillemets simples en valeurs 8 bits au sein d'un entier 32 bits. Plusieurs caractères dans le littéral remplissent les octets correspondants selon les besoins, des octets de poids fort aux octets poids faible. Le compilateur convertit ensuite l'entier en type de destination en suivant les règles habituelles. Par exemple, pour créer une **`char`** valeur, le compilateur prend l'octet de faible ordre. Pour créer une valeur **`wchar_t`** ou **`char16_t`** , le compilateur prend le mot de poids faible. Le compilateur avertit que le résultat est tronqué si tous les bits sont définis au-dessus de l'octet ou du mot assigné.

C++

```
char c0    = 'abcd';    // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';    // C4305, C4309, truncates to '\x6364'
int i0     = 'abcd';    // 0x61626364
```

Une séquence d'échappement octale qui semble contenir plus de trois chiffres est traitée comme une séquence octale à 3 chiffres, suivie des chiffres suivants sous forme de caractères dans un littéral multicharacter, ce qui peut donner des résultats surprenants. Par exemple :

C++

```
char c1 = '\100';   // '@'
char c2 = '\1000';  // C4305, C4309, truncates to '0'
```

Les séquences d'échappement qui semblent contenir des caractères non octaux sont évaluées comme une séquence octale jusqu'au dernier caractère octal, suivie des caractères restants en tant que caractères suivants dans un littéral multicharacteur.

L'avertissement C4125 est généré si le premier caractère non octal est un chiffre décimal. Par exemple :

```
C++
```

```
char c3 = '\009';   // '9'
char c4 = '\089';   // C4305, C4309, truncates to '9'
char c5 = '\qrs';   // C4129, C4305, C4309, truncates to 's'
```

Une séquence d'échappement octale dont la valeur `\377` est supérieure à ce qui provoque l'erreur C2022 : « *valeur dans la décimale* » : trop grande pour un caractère.

Une séquence d'échappement qui semble avoir des caractères hexadécimaux et non hexadécimaux est évaluée comme un littéral multicaractère qui contient une séquence d'échappement hexadécimale jusqu'au dernier caractère hexadécimal, suivi des caractères non hexadécimaux. Une séquence d'échappement hexadécimale qui ne contient aucun chiffre hexadécimal provoque l'erreur du compilateur C2153 : « Les littéraux hexadécimaux doivent avoir au moins un chiffre hexadécimal ».

```
C++
```

```
char c6 = '\x0050'; // 'P'
char c7 = '\x0pqr'; // C4305, C4309, truncates to 'r'
```

Si un littéral de caractères larges préfixé avec `L` contient une séquence multicaractère, la valeur est extraite du premier caractère et le compilateur déclenche l'avertissement C4066. Les caractères suivants sont ignorés, contrairement au comportement du littéral multicaractère ordinaire équivalent.

```
C++
```

```
wchar_t w1 = L'\100';   // L'@'
wchar_t w2 = L'\1000';  // C4066 L'@', 0 ignored
wchar_t w3 = L'\009';   // C4066 L'\0', 9 ignored
wchar_t w4 = L'\089';   // C4066 L'\0', 89 ignored
wchar_t w5 = L'\qrs';   // C4129, C4066 L'q' escape, rs ignored
wchar_t w6 = L'\x0050'; // L'P'
wchar_t w7 = L'\x0pqr'; // C4066 L'\0', pqr ignored
```

La section **spécifique à Microsoft** se termine ici.

noms de caractères universels

Dans les littéraux de caractère et les littéraux de chaîne natifs (non bruts), un nom de caractère universel peut représenter n'importe quel caractère. Les noms de caractères universels sont formés d'un préfixe `\U` suivi d'un point de code Unicode à huit chiffres ou d'un préfixe `\u` suivi d'un point de code Unicode à quatre chiffres. Tous les points de code Unicode à huit ou quatre chiffres, respectivement, doivent être présents pour constituer un nom de caractère universel bien formé.

C++

```
char u1 = 'A';          // 'A'
char u2 = '\101';       // octal, 'A'
char u3 = '\x41';       // hexadecimal, 'A'
char u4 = '\u0041';     // \u UCN 'A'
char u5 = '\U00000041'; // \U UCN 'A'
```

Paires de substitution

Les noms de caractères universels ne peuvent pas encoder les valeurs dans la plage de points de code de substitution D800-DFFF. Pour les paires de substitution Unicode, spécifiez le nom de caractère universel à l'aide de `\UNNNNNNNN`, où NNNNNNNN représente le point de code à huit chiffres du caractère. Le compilateur génère une paire de substitution si nécessaire.

En C++03, la langue permettait uniquement à un sous-ensemble de caractères d'être représentés par leurs noms de caractères universels et a autorisé certains noms de caractères universels qui ne représentaient pas réellement de caractères Unicode valides. Cette erreur a été corrigée dans la norme C++11. En C++11, les littéraux de caractère et de chaîne, ainsi que les identificateurs, peuvent utiliser des noms de caractères universels. Pour plus d'informations sur les noms de caractères universels, consultez Character Sets. Pour plus d'informations sur Unicode, consultez Unicode. Pour plus d'informations sur les paires de substitution, consultez Paires de substitution et caractères supplémentaires.

Littéraux de chaîne

Un littéral de chaîne représente une séquence de caractères qui, ensemble, forment une chaîne terminée par le caractère Null. Les caractères doivent être placés entre guillemets doubles. Il existe les genres suivants de littéraux de chaîne :

Littéraux de chaînes étroites

Un littéral de chaîne étroite est un tableau non préfixé, délimité par des guillemets doubles, avec fin null de type `const char[n]`, où n est la longueur du tableau en octets. Un littéral de chaîne étroit peut contenir n'importe quel caractère graphique à l'exception du guillemet double (`"`), de la barre oblique inverse (`\`) ou du caractère de nouvelle ligne. Un littéral de chaîne étroit peut également contenir les séquences d'échappement répertoriées ci-dessus, ainsi que les noms de caractères universels qui tiennent dans un octet.

C++

```
const char *narrow = "abcd";

// represents the string: yes\no
const char *escaped = "yes\\no";
```

Chaînes encodées UTF-8

Une chaîne encodée UTF-8 est un tableau de type null délimité par un préfixe u8, délimité par deux guillemets, de type `const char[n]`null, où *n* est la longueur du tableau encodé en octets. Un littéral de chaîne ayant le préfixe u8 peut contenir n'importe quel caractère graphique à l'exception du guillemet double (`"`), de la barre oblique inverse (`\`) ou du caractère de nouvelle ligne. Un littéral de chaîne ayant le préfixe u8 peut également contenir les séquences d'échappement répertoriées ci-dessus, ainsi que des noms de caractères universels.

C++20 introduit le type de caractère portable **`char8_t`** (Unicode 8 bits encodé en UTF-8). En C++20, `u8` les préfixes littéraux spécifient des caractères ou des chaînes de **`char8_t`** au lieu de **`char`**.

C++

```
// Before C++20
const char* str1 = u8"Hello World";
const char* str2 = u8"\U0001F607 is O:-)";
// C++20 and later
const char8_t* u8str1 = u8"Hello World";
const char8_t* u8str2 = u8"\U0001F607 is O:-)";
```

Littéraux de chaîne large

Un littéral de chaîne large est un tableau de constante **`wchar_t`** avec fin null qui est précédé de « `L` » et contient n'importe quel caractère graphique, à l'exception du

guillemet double (`"`), de la barre oblique inverse (`\`) ou du caractère de nouvelle ligne. Un littéral de chaîne large peut contenir les séquences d'échappement répertoriées ci-dessus, ainsi que des noms de caractères universels.

```
C++
```

```
const wchar_t* wide = L"zyxw";
const wchar_t* newline = L"hello\ngoodbye";
```

char16_t et char32_t (C++11)

C++11 présente les types de caractère portables **`char16_t`** (Unicode 16 bits) et **`char32_t`** (Unicode 32 bits) :

```
C++
```

```
auto s3 = u"hello"; // const char16_t*
auto s4 = U"hello"; // const char32_t*
```

Littéraux de chaîne bruts (C++11)

Un littéral de chaîne brute est un tableau avec fin null, de n'importe quel type de caractère, qui contient n'importe quel caractère graphique, y compris le guillemet double (`"`), la barre oblique inverse (`\`) ou le caractère de ligne nouvelle. Les littéraux de chaîne bruts sont souvent utilisés dans les expressions régulières qui utilisent des classes de caractères et dans des chaînes XML et des chaînes HTML. Pour obtenir des exemples, consultez l'article du FAQ de Bjarne Stroustrup sur C++11 ↗.

```
C++
```

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t*  raw_wide  = LR"(An unescaped \ character)";
const char*     raw_utf8a = u8R"(An unescaped \ character)"; // Before C++20
const char8_t*  raw_utf8b = u8R"(An unescaped \ character)"; // C++20
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character)";
```

Un délimiteur est une séquence définie par l'utilisateur de jusqu'à 16 caractères qui précède immédiatement la parenthèse d'ouverture d'un littéral de chaîne brute et suit immédiatement sa parenthèse fermante. Par exemple, dans `R"abc(Hello"\()abc"` , la séquence du délimiteur est `abc` , et le contenu de la chaîne est `Hello"\(`. Vous pouvez

utiliser un délimiteur pour distinguer les chaînes brutes qui contiennent à la fois des guillemets doubles et des parenthèses. Ce littéral de chaîne provoque une erreur du compilateur :

```
C++
```

```
// meant to represent the string: )"
const char* bad_parens = R"()")";  // error C2059
```

Mais un délimiteur résout cette erreur :

```
C++
```

```
const char* good_parens = R"xyz()")xyz";
```

Vous pouvez construire un littéral de chaîne brute qui contient une nouvelle ligne (et non le caractère d'échappement) dans la source :

```
C++
```

```
// represents the string: hello
//goodbye
const wchar_t* newline = LR"(hello
goodbye)";
```

std::string literals (C++14)

`std::string` les littéraux sont des implémentations de bibliothèque standard de littéraux définis par l'utilisateur (voir ci-dessous) qui sont représentés comme `"xyz"s` (avec un `s` suffixe). Ce type de littéral de chaîne produit un objet temporaire de type `std::string`, `std::wstring`, `std::u32string` ou `std::u16string`, en fonction du préfixe spécifié. Lorsqu'aucun préfixe n'est utilisé, comme ci-dessus, un `std::string` est produit. `L"xyz"s` produit un `std::wstring`. `u"xyz"s` produit une std::u16string et `U"xyz"s` une std::u32string.

```
C++
```

```
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };      // Before C++20
u8string u8str2{ u8"Hello World" }; // C++20
wstring str3{ L"hello"s };
```

```
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

Le `s` suffixe peut également être utilisé sur les littéraux de chaîne bruts :

```
C++

u32string str6{ UR"(She said "hello.")"s };
```

`std::string` les littéraux sont définis dans l'espace de noms `std::literals::string_literals` dans le fichier d'en-tête <de chaîne> . Étant donné que `std::literals::string_literals`, et `std::literals` sont tous deux déclarés en tant qu'espaces de noms inline, `std::literals::string_literals` est automatiquement traité comme s'il appartenait directement à l'espace de noms `std`.

Taille des littéraux de chaîne

Pour les chaînes ANSI `char*` et autres encodages sur un octet (mais pas UTF-8), la taille (en octets) d'un littéral de chaîne correspond au nombre de caractères plus 1 pour le caractère null de fin. Pour tous les autres types de chaînes, la taille n'est pas strictement liée au nombre de caractères. UTF-8 utilise jusqu'à quatre **`char`** éléments pour encoder certaines *unités de code*, et **`char16_t`** ou **`wchar_t`** encodé en UTF-16 peut utiliser deux éléments (pour un total de quatre octets) pour encoder une seule unité de *code*. Cet exemple illustre la taille d'un littéral de chaîne large en octets :

```
C++

const wchar_t* str = L"Hello!";
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

Notez que `strlen()` et `wcslen()` n'incluent pas la taille du caractère null de fin, dont la taille est égale à la taille d'élément du type de chaîne : un octet sur une `char*` chaîne ou `char8_t*` , deux octets sur `wchar_t*` des chaînes ou `char16_t*` et quatre octets sur `char32_t*` des chaînes.

Dans les versions de Visual Studio antérieures à Visual Studio 2022 version 17.0, la longueur maximale d'un littéral de chaîne est de 65 535 octets. Cette limite s'applique à la fois aux littéraux de chaîne étroits et étendus. Dans Visual Studio 2022 version 17.0 et ultérieures, cette restriction est levée et la longueur des chaînes est limitée par les ressources disponibles.

Modification des littéraux de chaîne

Étant donné que les littéraux de chaîne (sans les `std::string` littéraux) sont des constantes, la tentative de les modifier (par exemple, `str[2] = 'A'`) provoque une erreur du compilateur.

Spécifique à Microsoft

Dans Microsoft C++, vous pouvez utiliser un littéral de chaîne pour initialiser un pointeur vers non const `char` ou `wchar_t`. Cette initialisation non const est autorisée dans le code C99, mais est déconseillée en C++98 et supprimée en C++11. Toute tentative de modification de la chaîne provoque une violation d'accès, comme dans cet exemple :

```
C++

wchar_t* str = L"hello";
str[2] = L'a'; // run-time error: access violation
```

Vous pouvez provoquer l'émission d'une erreur quand un littéral de chaîne est converti en pointeur de caractères non const lorsque vous définissez l'option du compilateur (Désactiver la/Zc:strictStrings conversion de type littéral de chaîne). Nous le recommandons pour le code portable conforme aux normes. Il est également recommandé d'utiliser le `auto` mot clé pour déclarer des pointeurs initialisés littérals de chaîne, car il résout le type correct (const). Par exemple, cet exemple de code intercepte une tentative d'écriture dans un littéral de chaîne au moment de la compilation :

```
C++

auto str = L"hello";
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

Dans certains cas, des littéraux de chaîne identiques peuvent être regroupés pour économiser de l'espace dans le fichier exécutable. Dans un regroupement de littéraux de chaîne, le compilateur fait en sorte que toutes les références à un littéral de chaîne particulier pointent vers le même emplacement en mémoire, au lieu que chaque référence pointe vers une instance distincte du littéral de chaîne. Pour activer le regroupement de chaînes, utilisez l'option du /GF compilateur.

La section **spécifique à Microsoft** se termine ici.

Concaténation de littéraux de chaîne adjacents

Les littéraux de chaîne étendus ou étroits adjacents sont concaténés. Cette déclaration :

```
C++

char str[] = "12" "34";
```

est identique à la déclaration suivante :

```
C++

char atr[] = "1234";
```

et à cette déclaration :

```
C++

char atr[] =  "12\
34";
```

L'utilisation de codes d'échappement hexadécimaux incorporés pour spécifier des littéraux de chaîne peut provoquer des résultats inattendus. L'exemple suivant tente de créer un littéral de chaîne qui contient le caractère ASCII 5, suivi des caractères f, i, v et e :

```
C++

"\x05five"
```

Le résultat réel est un hexadécimal 5F, qui correspond au code ASCII pour un trait de soulignement, suivi des caractères i, v et e. Pour obtenir le résultat correct, vous pouvez utiliser l'une des séquences d'échappement suivantes :

```
C++

"\005five"     // Use octal literal.
"\x05" "five"  // Use string splicing.
```

`std::string` les littéraux (et les , et) associés `std::u8string` peuvent être concaténés avec l'opérateur `+` défini pour les basic_string types. `std::u32string` `std::u16string` Ils peuvent également être concaténés de la même façon que des littéraux de chaîne adjacents. Dans les deux cas, l'encodage de chaîne et le suffixe doivent correspondre :

C++

```
auto x1 = "hello" " " " world"; // OK
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix
auto x3 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

Littéraux de chaîne avec des noms de caractères universels

Les littéraux de chaîne natifs (non bruts) peuvent utiliser des noms de caractères universels pour représenter un caractère, du moment que le nom de caractère universel peut être encodé sous forme d'un ou de plusieurs caractères dans le type de chaîne. Par exemple, un nom de caractère universel représentant un caractère étendu ne peut pas être encodé dans une chaîne étroite à l'aide de la page de code ANSI, mais il peut être encodé dans des chaînes étroites dans certaines pages de code multi-octets, ou dans des chaînes UTF-8 ou dans une chaîne large. En C++11, la prise en charge d'Unicode est étendue par les `char16_t*` types de chaîne et `char32_t*` C++20 l'étend `char8_t` au type :

C++

```
// ASCII smiling face
const char*     s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*  s2 = L"😉 = \U0001F609 is ;-)";

// UTF-8  encoded SMILING FACE WITH HALO (U+1F607)
const char*     s3a = u8"😇 = \U0001F607 is O:-)"; // Before C++20
const char8_t*  s3b = u8"😇 = \U0001F607 is O:-)"; // C++20

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t* s4 = u"😃 = \U0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t* s5 = U"😎 = \U0001F60E is B-)";
```

Voir aussi

Jeux de caractères
Littéraux numériques, booléens et pointeurs
Littéraux définis par l'utilisateur

Littéraux définis par l'utilisateur

Article • 16/06/2023

Il existe six catégories principales de littéraux en C++: entier, caractère, virgule flottante, chaîne, booléen et pointeur. À partir de C++ 11, vous pouvez définir vos propres littéraux en fonction de ces catégories, pour fournir des raccourcis syntaxiques pour les idiomes courants et améliorer la sécurité de type. Par exemple, supposons que vous avez une `Distance` classe. Vous pouvez définir un littéral pour les kilomètres et un autre pour les miles, et encourager l'utilisateur à être explicite sur les unités de mesure en écrivant : `auto d = 42.0_km` ou `auto d = 42.0_mi`. Les littéraux définis par l'utilisateur ne présentent aucun avantage ou inconvénient en termes de performances ; ils sont principalement pour des raisons pratiques ou pour la déduction de type au moment de la compilation. La bibliothèque standard a des littéraux définis par l'utilisateur pour `std::string`, pour `std::complex` et pour les unités dans les opérations de temps et de durée dans l'en-tête <chrono> :

C++

```
Distance d = 36.0_mi + 42.0_km;          // Custom UDL (see below)
std::string str = "hello"s + "World"s;   // Standard Library <string> UDL
complex<double> num =
   (2.0 + 3.01i) * (5.0 + 4.3i);         // Standard Library <complex> UDL
auto duration = 15ms + 42h;              // Standard Library <chrono> UDLs
```

Signatures d'opérateur littéral défini par l'utilisateur

Vous implémentez un littéral défini par l'utilisateur en définissant un **opérateur** « » au niveau de l'étendue de l'espace de noms avec l'une des formes suivantes :

C++

```
ReturnType operator "" _a(unsigned long long int);   // Literal operator for
user-defined INTEGRAL literal
ReturnType operator "" _b(long double);              // Literal operator for
user-defined FLOATING literal
ReturnType operator "" _c(char);                     // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _d(wchar_t);                  // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _e(char16_t);                 // Literal operator for
user-defined CHARACTER literal
ReturnType operator "" _f(char32_t);                 // Literal operator for
```

```
user-defined CHARACTER literal
ReturnType operator "" _g(const char*, size_t);      // Literal operator for
user-defined STRING literal
ReturnType operator "" _h(const wchar_t*, size_t);   // Literal operator for
user-defined STRING literal
ReturnType operator "" _i(const char16_t*, size_t);  // Literal operator for
user-defined STRING literal
ReturnType operator "" _g(const char32_t*, size_t);  // Literal operator for
user-defined STRING literal
ReturnType operator "" _r(const char*);              // Raw literal operator
template<char...> ReturnType operator "" _t();       // Literal operator
template
```

Les noms d'opérateur dans l'exemple précédent sont des espaces réservés pour le nom que vous fournissez ; toutefois, le caractère de soulignement de début est requis. (Seule la bibliothèque standard est autorisée à définir des littéraux sans trait de soulignement.) Le type de retour est l'endroit où vous personnalisez la conversion ou d'autres opérations effectuées par le littéral. En outre, chacun de ces opérateurs peut être défini en tant que `constexpr`.

Littéraux traités

Dans le code source, tout littéral, défini ou non par l'utilisateur, est essentiellement une séquence de caractères alphanumériques, tels que `101`, ou `54.7`, ou `"hello"` ou `true`. Le compilateur interprète la séquence comme une valeur entière, float, const char*, de chaîne, et ainsi de suite. Un littéral défini par l'utilisateur qui accepte comme entrée tout type que le compilateur a affecté à la valeur littérale est officieusement appelé *littéral cuit*. Tous les opérateurs ci-dessus, à l'exception de `_r` et `_t`, sont des littéraux traités. Par exemple, un littéral `42.0_km` établit une liaison avec un opérateur nommé _km ayant une signature similaire à _b et le littéral `42_km` établit une liaison avec un opérateur ayant une signature similaire à _a.

L'exemple suivant montre comment des littéraux définis par l'utilisateur peuvent encourager les appelants à être explicites sur leur entrée. Pour créer une `Distance`, l'utilisateur doit spécifier explicitement des kilomètres ou des miles à l'aide du littéral défini par l'utilisateur approprié. Vous pouvez obtenir le même résultat d'autres façons, mais les littéraux définis par l'utilisateur sont moins détaillés que les alternatives.

C++

```
// UDL_Distance.cpp

#include <iostream>
#include <string>
```

```
struct Distance
{
private:
    explicit Distance(long double val) : kilometers(val)
    {}

    friend Distance operator"" _km(long double val);
    friend Distance operator"" _mi(long double val);

    long double kilometers{ 0 };
public:
    const static long double km_per_mile;
    long double get_kilometers() { return kilometers; }

    Distance operator+(Distance other)
    {
        return Distance(get_kilometers() + other.get_kilometers());
    }
};

const long double Distance::km_per_mile = 1.609344L;

Distance operator"" _km(long double val)
{
    return Distance(val);
}

Distance operator"" _mi(long double val)
{
    return Distance(val * Distance::km_per_mile);
}

int main()
{
    // Must have a decimal point to bind to the operator we defined!
    Distance d{ 402.0_km }; // construct using kilometers
    std::cout << "Kilometers in d: " << d.get_kilometers() << std::endl; //
402

    Distance d2{ 402.0_mi }; // construct using miles
    std::cout << "Kilometers in d2: " << d2.get_kilometers() << std::endl;
//646.956

    // add distances constructed with different units
    Distance d3 = 36.0_mi + 42.0_km;
    std::cout << "d3 value = " << d3.get_kilometers() << std::endl; //
99.9364

    // Distance d4(90.0); // error constructor not accessible

    std::string s;
    std::getline(std::cin, s);
    return 0;
}
```

Le nombre littéral doit utiliser une décimale. Sinon, le nombre serait interprété comme un entier et le type ne serait pas compatible avec l'opérateur . Pour l'entrée à virgule flottante, le type doit être `long double`, et pour les types intégraux, il doit être `long long`.

Littéraux bruts

Dans un littéral brut défini par l'utilisateur, l'opérateur que vous définissez accepte le littéral comme une séquence de valeurs char. C'est à vous d'interpréter cette séquence comme un nombre, une chaîne ou un autre type. Dans la liste des opérateurs présentée plus haut dans cette page, `_r` et `_t` peuvent être utilisés pour définir des littéraux bruts :

C++

```
ReturnType operator "" _r(const char*);              // Raw literal operator
template<char...> ReturnType operator "" _t();       // Literal operator
template
```

Vous pouvez utiliser des littéraux bruts pour fournir une interprétation personnalisée d'une séquence d'entrée différente du comportement normal du compilateur. Par exemple, vous pouvez définir un littéral qui convertit la séquence `4.75987` en un type Decimal personnalisé au lieu d'un type à virgule flottante IEEE 754. Les littéraux bruts, comme les littéraux cuits, peuvent également être utilisés pour la validation au moment de la compilation des séquences d'entrée.

Exemple : Limitations des littéraux bruts

L'opérateur littéral brut et le modèle d'opérateur littéral ne fonctionnent que pour les littéraux définis par l'utilisateur intégraux et à virgule flottante, comme le montre l'exemple suivant :

C++

```
#include <cstddef>
#include <cstdio>

// Literal operator for user-defined INTEGRAL literal
void operator "" _dump(unsigned long long int lit)
{
    printf("operator \"\" _dump(unsigned long long int) : ===>%llu<===\n",
lit);
};
```

```
// Literal operator for user-defined FLOATING literal
void operator "" _dump(long double lit)
{
    printf("operator \"\" _dump(long double)           : ===>%Lf<===\n",
lit);
};

// Literal operator for user-defined CHARACTER literal
void operator "" _dump(char lit)
{
    printf("operator \"\" _dump(char)                  : ===>%c<===\n",
lit);
};

void operator "" _dump(wchar_t lit)
{
    printf("operator \"\" _dump(wchar_t)               : ===>%d<===\n",
lit);
};

void operator "" _dump(char16_t lit)
{
    printf("operator \"\" _dump(char16_t)              : ===>%d<===\n",
lit);
};

void operator "" _dump(char32_t lit)
{
    printf("operator \"\" _dump(char32_t)              : ===>%d<===\n",
lit);
};

// Literal operator for user-defined STRING literal
void operator "" _dump(const     char* lit, size_t)
{
    printf("operator \"\" _dump(const     char*, size_t): ===>%s<===\n",
lit);
};

void operator "" _dump(const  wchar_t* lit, size_t)
{
    printf("operator \"\" _dump(const  wchar_t*, size_t): ===>%ls<===\n",
lit);
};

void operator "" _dump(const char16_t* lit, size_t)
{
    printf("operator \"\" _dump(const char16_t*, size_t):\n"
);
};

void operator "" _dump(const char32_t* lit, size_t)
{
    printf("operator \"\" _dump(const char32_t*, size_t):\n"
);
```

```
};

// Raw literal operator
void operator "" _dump_raw(const char* lit)
{
    printf("operator \"\" _dump_raw(const char*)        : ===>%s<===\n",
lit);
};

template<char...> void operator "" _dump_template();       // Literal
operator template

int main(int argc, const char* argv[])
{
    42_dump;
    3.1415926_dump;
    3.14e+25_dump;
     'A'_dump;
    L'B'_dump;
    u'C'_dump;
    U'D'_dump;
      "Hello World"_dump;
     L"Wide String"_dump;
    u8"UTF-8 String"_dump;
     u"UTF-16 String"_dump;
     U"UTF-32 String"_dump;
    42_dump_raw;
    3.1415926_dump_raw;
    3.14e+25_dump_raw;

    // There is no raw literal operator or literal operator template support
on these types:
    //  'A'_dump_raw;
    // L'B'_dump_raw;
    // u'C'_dump_raw;
    // U'D'_dump_raw;
    //   "Hello World"_dump_raw;
    //  L"Wide String"_dump_raw;
    // u8"UTF-8 String"_dump_raw;
    //  u"UTF-16 String"_dump_raw;
    //  U"UTF-32 String"_dump_raw;
}
```

Output

```
operator "" _dump(unsigned long long int) : ===>42<===
operator "" _dump(long double)            : ===>3.141593<===
operator "" _dump(long double)            :
===>3139999999999998506827776.000000<===
operator "" _dump(char)                   : ===>A<===
operator "" _dump(wchar_t)                : ===>66<===
operator "" _dump(char16_t)               : ===>67<===
operator "" _dump(char32_t)               : ===>68<===
```

```
operator "" _dump(const     char*, size_t): ===>Hello World<===
operator "" _dump(const  wchar_t*, size_t): ===>Wide String<===
operator "" _dump(const     char*, size_t): ===>UTF-8 String<===
operator "" _dump(const char16_t*, size_t):
operator "" _dump(const char32_t*, size_t):
operator "" _dump_raw(const char*)        : ===>42<===
operator "" _dump_raw(const char*)        : ===>3.1415926<===
operator "" _dump_raw(const char*)        : ===>3.14e+25<===
```

Concepts de base (C++)

Article • 16/06/2023

Cette section explique les concepts essentiels à la compréhension de C++. Les programmeurs C connaissent un grand nombre de ces concepts, mais il existe des différences subtiles qui peuvent entraîner des résultats inattendus du programme. Cet article contient les rubriques suivantes :

- Système de type C++
- Portée
- Unités de traduction et liaison
- Fonction main et arguments de ligne de commande
- Arrêt du programme
- Lvalues et Rvalues
- Objets temporaires
- Alignment
- Trivial, standard-layout et pod types

Voir aussi

Informations de référence sur le langage C++

Système de type C++

Article • 03/04/2023

Le concept de *type* est important en C++. Chaque variable, argument de fonction et valeur de retour de fonction doit avoir un type pour être compilé. En outre, toutes les expressions (y compris les valeurs littérales) reçoivent implicitement un type par le compilateur avant d'être évaluées. Certains exemples de types incluent des types intégrés tels que `int` pour stocker des valeurs entières, `double` pour stocker des valeurs à virgule flottante ou des types bibliothèque standard tels que la classe std::basic_string pour stocker du texte. Vous pouvez créer votre propre type en définissant un `class` ou `struct`. Le type spécifie la quantité de mémoire allouée pour la variable (ou le résultat de l'expression). Le type spécifie également les types de valeurs qui peuvent être stockés, la façon dont le compilateur interprète les modèles de bits dans ces valeurs et les opérations que vous pouvez effectuer sur celles-ci. Cet article contient une présentation informelle des principales fonctionnalités du système de type C++.

Terminologie

Type scalaire : type qui contient une valeur unique d'une plage définie. Les scalaires incluent les types arithmétiques (valeurs intégrales ou à virgule flottante), les membres de type énumération, les types de pointeur, les types pointeur vers membre et `std::nullptr_t`. Les types fondamentaux sont généralement des types scalaires.

Type composé : type qui n'est pas un type scalaire. Les types composés incluent les types de tableau, les types de fonction, les types de classe (ou struct), les types d'union, les énumérations, les références et les pointeurs vers des membres de classe non statiques.

Variable : nom symbolique d'une quantité de données. Le nom peut être utilisé pour accéder aux données qu'il fait référence dans l'étendue du code où il est défini. En C++, *la variable* est souvent utilisée pour faire référence à des instances de types de données scalaires, tandis que les instances d'autres types sont généralement *appelées objets*.

Objet : par souci de simplicité et de cohérence, cet article utilise le terme *objet* pour faire référence à n'importe quelle instance d'une classe ou d'une structure. Lorsqu'il est utilisé au sens général, il inclut tous les types, même les variables scalaires.

Type POD (anciennes données simples) : cette catégorie informelle de types de données en C++ fait référence à des types scalaires (voir la section Types fondamentaux) ou *des classes POD*. Une classe POD n'a pas de membres de données statiques qui ne sont pas

également des POD et n'a pas de constructeurs définis par l'utilisateur, de destructeurs définis par l'utilisateur ou d'opérateurs d'affectation définis par l'utilisateur. En outre, une classe POD n'a aucune fonction virtuelle, aucune classe de base et aucune donnée membre non static privée ou protégée. Les types POD sont souvent utilisés pour l'échange de données externes, par exemple avec un module écrit en langage C (disposant de types POD uniquement).

Spécification des types de variable et de fonction

C++ est à la fois un langage *fortement typé* et un langage *statiquement typé* ; chaque objet a un type et ce type ne change jamais. Lorsque vous déclarez une variable dans votre code, vous devez spécifier son type explicitement ou utiliser le `auto` mot clé pour indiquer au compilateur de déduire le type de l'initialiseur. Lorsque vous déclarez une fonction dans votre code, vous devez spécifier le type de sa valeur de retour et de chaque argument. Utilisez le type `void` de valeur de retour si aucune valeur n'est retournée par la fonction . L'exception est lorsque vous utilisez des modèles de fonction, qui autorisent les arguments de types arbitraires.

Une fois que vous avez déclaré une variable pour la première fois, vous ne pouvez pas modifier son type ultérieurement. Toutefois, vous pouvez copier la valeur de la variable ou la valeur de retour d'une fonction dans une autre variable d'un type différent. Ces opérations sont appelées *conversions de type*, qui sont parfois nécessaires, mais qui sont également des sources potentielles de perte de données ou d'erreur.

Lorsque vous déclarez une variable de type POD, nous vous recommandons vivement de *l'initialiser* , ce qui signifie de lui donner une valeur initiale. Tant que vous n'avez pas initialisé une variable, elle a la valeur « garbage » qui se compose des bits, quels qu'ils soient, présents précédemment dans cet emplacement mémoire. C'est un aspect important de C++ à retenir, surtout si vous venez d'un autre langage qui gère l'initialisation pour vous. Lorsque vous déclarez une variable de type de classe non POD, le constructeur gère l'initialisation.

L'exemple suivant présente des déclarations de variable simples avec quelques descriptions. L'exemple montre également comment le compilateur utilise les informations de type pour autoriser ou interdire certaines opérations ultérieures sur la variable.

C++

```
int result = 0;              // Declare and initialize an integer.
double coefficient = 10.8;   // Declare and initialize a floating
```

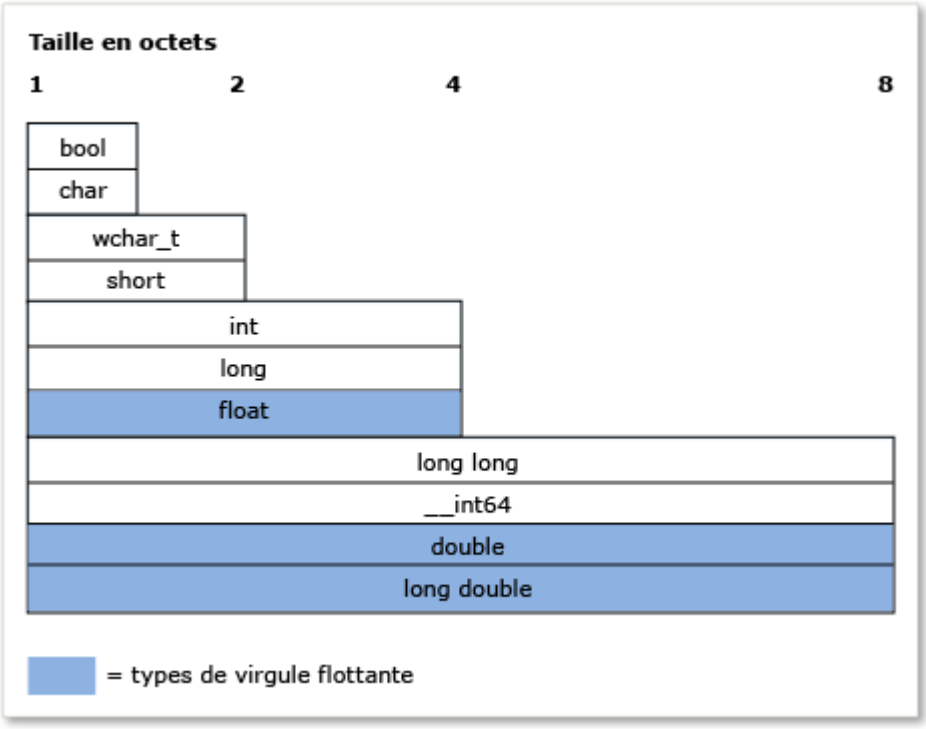
```
                            // point value.
auto name = "Lady G.";      // Declare a variable and let compiler
                            // deduce the type.
auto address;               // error. Compiler cannot deduce a type
                            // without an intializing value.
age = 12;                   // error. Variable declaration must
                            // specify a type or use auto!
result = "Kenny G.";        // error. Can't assign text to an int.
string result = "zero";     // error. Can't redefine a variable with
                            // new type.
int maxValue;               // Not recommended! maxValue contains
                            // garbage bits until it is initialized.
```

Types fondamentaux (intégrés)

Contrairement à certains langages, C++ n'a aucun type de base universel dont tous les autres types sont dérivés. Le langage inclut de nombreux *types fondamentaux*, également appelés *types intégrés*. Ces types incluent des types numériques tels que , , , ainsi que `int` les types et `wchar_t` pour les `char` caractères ASCII et UNICODE, respectivement. `bool` `long` `double` La plupart des types fondamentaux intégraux (à l'exception `bool` de , `double`, `wchar_t` et des types associés) ont tous des `unsigned` versions qui modifient la plage de valeurs que la variable peut stocker. Par exemple, un `int`, qui stocke un entier signé 32 bits, peut représenter une valeur comprise entre -2 147 483 648 et 2 147 483 647. Un `unsigned int`, qui est également stocké sous forme de 32 bits, peut stocker une valeur comprise entre 0 et 4 294 967 295. Le nombre total de valeurs possibles dans chaque cas est identique ; seule la plage est différente.

Le compilateur reconnaît ces types intégrés, et il a des règles intégrées qui régissent les opérations que vous pouvez effectuer sur eux, et comment elles peuvent être converties en d'autres types fondamentaux. Pour obtenir la liste complète des types intégrés, ainsi que leur taille et leurs limites numériques, consultez Types intégrés.

L'illustration suivante montre les tailles relatives des types intégrés dans l'implémentation Microsoft C++ :



Le tableau suivant répertorie les types fondamentaux les plus fréquemment utilisés, ainsi que leurs tailles dans l'implémentation Microsoft C++ :

Type	Taille	Commentaire
`int`	4 octets	Choix par défaut pour les valeurs intégrales.
`double`	8 octets	Choix par défaut pour les valeurs à virgule flottante.
`bool`	1 octet	Représente des valeurs qui peuvent être true ou false.
`char`	1 octet	À utiliser pour les caractères ASCII dans les chaînes de style C plus anciennes ou les objets std::string qui ne devront jamais être convertis en UNICODE.
`wchar_t`	2 octets	Représente les valeurs à caractères « larges » qui peuvent être encodées au format UNICODE (UTF-16 sur Windows, mais peut varier sur les autres systèmes d'exploitation). `wchar_t` est le type de caractère utilisé dans les chaînes de type `std::wstring`.
`unsigned char`	1 octet	C++ n'a pas de type d'octet intégré. Utilisez `unsigned char` pour représenter une valeur d'octet.
`unsigned int`	4 octets	Option par défaut pour les bits indicateurs.
`long long`	8 octets	Représente une plage beaucoup plus grande de valeurs entières.

D'autres implémentations C++ peuvent utiliser des tailles différentes pour certains types numériques. Pour plus d'informations sur les tailles et les relations de taille requises par

la norme C++, consultez Types intégrés.

Le type `void`

Le `void` type est un type spécial ; vous ne pouvez pas déclarer une variable de type `void`, mais vous pouvez déclarer une variable de type `void *` (pointeur vers `void`), ce qui est parfois nécessaire lors de l'allocation de mémoire brute (non typée). Toutefois, les pointeurs vers `void` ne sont pas de type sécurisé et leur utilisation est déconseillée en C++moderne. Dans une déclaration de fonction, une `void` valeur de retour signifie que la fonction ne retourne pas de valeur ; l'utiliser comme type de retour est une utilisation courante et acceptable de `void`. Alors que le langage C exigeait des fonctions qui n'ont aucun paramètre à déclarer `void` dans la liste de paramètres, par exemple, `fn(void)`, cette pratique est déconseillée en C++ moderne ; une fonction sans paramètre doit être déclarée `fn()`. Pour plus d'informations, consultez Conversions de type et sécurité des types.

`const` qualificateur de type

Tout type intégré ou défini par l'utilisateur peut être qualifié par le `const` mot clé. En outre, les fonctions membres peuvent être `const`qualifiées et même `const`surchargées. La valeur d'un `const` type ne peut pas être modifiée après son initialisation.

C++

```
const double PI = 3.1415;
PI = .75; //Error. Cannot modify const variable.
```

Le `const` qualificateur est largement utilisé dans les déclarations de fonction et de variable, et « const correctness » est un concept important en C++; cela signifie essentiellement utiliser `const` pour garantir, au moment de la compilation, que les valeurs ne sont pas modifiées involontairement. Pour plus d'informations, consultez const.

Un `const` type est distinct de sa version non-`const` ; par exemple, `const int` est un type distinct de `int`. Vous pouvez utiliser l'opérateur C++ `const_cast` dans les rares cas où vous devez supprimer *la const-ness* d'une variable. Pour plus d'informations, consultez Conversions de type et sécurité des types.

Types chaîne

À proprement parler, le langage C++ n'a pas de type de chaîne intégré ; `char` et `wchar_t` stockent des caractères uniques : vous devez déclarer un tableau de ces types pour se rapprocher d'une chaîne, en ajoutant une valeur null de fin (par exemple, ASCII `'\0'`) à l'élément de tableau après le dernier caractère valide (également appelé *chaîne de style C*). Les chaînes de style C exigent beaucoup plus de code pour être écrites ou l'utilisation de fonctions de bibliothèque d'utilitaires de chaînes externes. Mais dans C++moderne, nous avons les types `std::string` bibliothèque standard (pour les chaînes de caractères de type 8 bits `char`) ou `std::wstring` (pour les chaînes de caractères de type -16 bits `wchar_t`). Ces conteneurs de bibliothèque standard C++ peuvent être considérés comme des types de chaînes natifs, car ils font partie des bibliothèques standard incluses dans n'importe quel environnement de build C++ conforme. Utilisez la `#include <string>` directive pour rendre ces types disponibles dans votre programme. (Si vous utilisez MFC ou ATL, la `CString` classe est également disponible, mais ne fait pas partie de la norme C++.) L'utilisation de tableaux de caractères terminés par un caractère Null (chaînes de style C mentionnées précédemment) est déconseillée en C++ moderne.

Types définis par l'utilisateur

Lorsque vous définissez un `class`, `struct`, `union` ou `enum`, cette construction est utilisée dans le reste de votre code comme s'il s'agissait d'un type fondamental. Il a une taille connue en mémoire, et certaines règles relatives à son utilisation s'appliquent à celui-ci pour la vérification au moment de la compilation et, au moment de l'exécution, pour la durée de vie de votre programme. Les principales différences entre les types intégrés fondamentaux et les types définis par l'utilisateur sont les suivantes :

- Le compilateur n'a pas connaissance de manière intégrée d'un type défini par l'utilisateur. Il apprend le type lorsqu'il rencontre la définition pour la première fois pendant le processus de compilation.

- Vous spécifiez les opérations qui peuvent être exécutées sur votre type et comment il peut être converti en d'autres types, en définissant (via la surcharge) les opérateurs appropriés, comme membres de classe ou comme fonctions non membres. Pour plus d'informations, consultez Surcharge des fonctions

Types de pointeur

Comme dans les premières versions du langage C, C++ continue de vous permettre de déclarer une variable d'un type de pointeur à l'aide du déclarateur `*` spécial (astérisque). Un type pointeur stocke l'adresse de l'emplacement dans la mémoire où la

valeur réelle des données est stockée. En C++ moderne, ces types de pointeurs sont appelés *pointeurs bruts*, et ils sont accessibles dans votre code via des opérateurs spéciaux : `*` (astérisque) ou `->` (tiret avec supérieur à, souvent appelé *flèche*). Cette opération d'accès à la mémoire est appelée *déréférencement*. L'opérateur que vous utilisez dépend de la déréférencement d'un pointeur vers un scalaire ou d'un pointeur vers un membre dans un objet.

L'utilisation des types pointeur a longtemps été l'un des aspects les plus importants et les plus perturbants du développement de programmes en C et C++. Cette section décrit quelques faits et pratiques pour vous aider à utiliser des pointeurs bruts si vous le souhaitez. Toutefois, en C++moderne, il n'est plus nécessaire (ou recommandé) d'utiliser des pointeurs bruts pour la propriété de l'objet, en raison de l'évolution du pointeur intelligent (plus en détail à la fin de cette section). Il est toujours utile et sûr d'utiliser des pointeurs bruts pour observer des objets. Toutefois, si vous devez les utiliser pour la propriété de l'objet, vous devez le faire avec prudence et en tenant compte soigneusement de la façon dont les objets qui leur appartiennent sont créés et détruits.

La première chose que vous devez savoir est qu'une déclaration de variable de pointeur brute alloue uniquement suffisamment de mémoire pour stocker une adresse : l'emplacement de mémoire auquel le pointeur fait référence lorsqu'il est déréférencé. La déclaration de pointeur n'alloue pas la mémoire nécessaire pour stocker la valeur des données. (Cette mémoire est également appelée *magasin de stockage*.) En d'autres termes, en déclarant une variable de pointeur brute, vous créez une variable d'adresse mémoire, et non une variable de données réelle. Si vous déréférencez une variable de pointeur avant de vous assurer qu'elle contient une adresse valide pour un magasin de stockage, cela entraîne un comportement non défini (généralement une erreur irrécupérable) dans votre programme. L'exemple suivant illustre ce type d'erreur :

C++

```
int* pNumber;       // Declare a pointer-to-int variable.
*pNumber = 10;      // error. Although this may compile, it is
                    // a serious error. We are dereferencing an
                    // uninitialized pointer variable with no
                    // allocated memory to point to.
```

L'exemple déréférence un type pointeur sans avoir de mémoire allouée pour stocker les données entières réelles, ni d'adresse mémoire valide assignée. Le code suivant corrige les erreurs suivantes :

C++

```
    int number = 10;          // Declare and initialize a local integer
                              // variable for data backing store.
```

```
    int* pNumber = &number;   // Declare and initialize a local integer
                              // pointer variable to a valid memory
                              // address to that backing store.
...
    *pNumber = 41;            // Dereference and store a new value in
                              // the memory pointed to by
                              // pNumber, the integer variable called
                              // "number". Note "number" was changed, not
                              // "pNumber".
```

L'exemple de code corrigé utilise la mémoire de la pile locale pour créer le magasin de stockage vers lequel pointe `pNumber`. Nous utilisons un type fondamental pour des raisons de simplicité. Dans la pratique, les magasins de stockage pour les pointeurs sont le plus souvent des types définis par l'utilisateur qui sont alloués dynamiquement dans une zone de mémoire appelée *tas* (ou *magasin libre*) à l'aide d'une `new` expression mot clé (dans la programmation de style C, l'ancienne `malloc()` fonction de bibliothèque runtime C a été utilisée). Une fois allouées, ces variables sont normalement *appelées objets*, en particulier si elles sont basées sur une définition de classe. La mémoire allouée avec `new` doit être supprimée par une instruction correspondante `delete` (ou, si vous avez utilisé la fonction pour l'allouer `malloc()` , la fonction `free()`runtime C).

Toutefois, il est facile d'oublier de supprimer un objet alloué dynamiquement, en particulier dans le code complexe, ce qui provoque un bogue de ressource appelé *fuite de mémoire*. Pour cette raison, l'utilisation de pointeurs bruts est déconseillée en C++moderne. Il est presque toujours préférable d'encapsuler un pointeur brut dans un pointeur intelligent, qui libère automatiquement la mémoire lorsque son destructeur est appelé. (Autrement dit, lorsque le code sort de l'étendue du pointeur intelligent.) En utilisant des pointeurs intelligents, vous éliminez pratiquement toute une classe de bogues dans vos programmes C++. Dans l'exemple suivant, supposez que `MyClass` est un type défini par l'utilisateur qui a une méthode `DoSomeWork();`publique

C++

```
void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
  // No memory leak. Out-of-scope automatically calls the destructor
  // for the unique_ptr, freeing the resource.
```

Pour plus d'informations sur les pointeurs intelligents, consultez Pointeurs intelligents.

Pour plus d'informations sur les conversions de pointeur, consultez Conversions de type et sécurité de type.

Pour plus d'informations sur les pointeurs en général, consultez Pointeurs.

Types de données Windows

Dans la programmation Win32 classique pour C et C++, la plupart des fonctions utilisent des typedefs et `#define` des macros spécifiques à Windows (définis dans `windef.h`) pour spécifier les types de paramètres et de valeurs de retour. Ces types de données Windows sont principalement des noms spéciaux (alias) donnés aux types intégrés C/C++. Pour obtenir la liste complète de ces définitions typedefs et préprocesseur, consultez Types de données Windows. Certains de ces typedefs, tels que `HRESULT` et `LCID`, sont utiles et descriptifs. D'autres, comme `INT`, n'ont pas de signification particulière et ne sont que des alias pour les types C++ fondamentaux. D'autres types de données Windows ont des noms qui proviennent de l'époque de la programmation en C et des processeurs 16 bits ; ils n'ont aucune finalité ou signification particulière par rapport au matériel ou aux systèmes d'exploitation modernes. Il existe également des types de données spéciaux associés à la bibliothèque Windows Runtime, répertoriés comme Windows Runtime types de données de base. En C++moderne, la règle générale consiste à préférer les types fondamentaux C++, sauf si le type Windows communique une signification supplémentaire sur la façon dont la valeur doit être interprétée.

Informations complémentaires

Pour plus d'informations sur le système de type C++, consultez les articles suivants.

Types de valeur
Décrit les *types de valeurs* ainsi que les problèmes liés à leur utilisation.

Conversions de types et sécurité de type
Décrit les problèmes de conversion des types courants et indique comment les éviter.

Voir aussi

Bienvenue dans C++
Informations de référence sur le langage C++
Bibliothèque C++ standard

Portée (C++)

Article • 03/04/2023

Lorsque vous déclarez un élément de programme tel qu'une classe, une fonction ou une variable, son nom ne peut être « vu » et utilisé que dans certaines parties de votre programme. Le contexte dans lequel un nom est visible est appelé son *étendue*. Par exemple, si vous déclarez une variable `x` au sein d'une fonction, `x` est visible uniquement dans ce corps de fonction. Il a *une étendue locale*. Vous pouvez avoir d'autres variables portant le même nom dans votre programme ; tant qu'ils se trouvent dans des étendues différentes, ils ne violent pas la règle d'une définition unique et aucune erreur n'est générée.

Pour les variables automatiques non statiques, l'étendue détermine également quand elles sont créées et détruites dans la mémoire du programme.

Il existe six types d'étendue :

- **Étendue globale** Un nom global est un nom qui est déclaré en dehors d'une classe, d'une fonction ou d'un espace de noms. Toutefois, en C++, même ces noms existent avec un espace de noms global implicite. L'étendue des noms globaux s'étend du point de déclaration à la fin du fichier dans lequel ils sont déclarés. Pour les noms globaux, la visibilité est également régie par les règles de liaison qui déterminent si le nom est visible dans d'autres fichiers du programme.

- **Étendue de l'espace de noms** Un nom déclaré dans un espace de noms, en dehors d'un bloc de fonction ou d'une définition de classe ou d'énumération, est visible à partir de son point de déclaration jusqu'à la fin de l'espace de noms. Un espace de noms peut être défini dans plusieurs blocs dans différents fichiers.

- **Étendue locale** Un nom déclaré dans une fonction ou lambda, y compris les noms de paramètres, a une étendue locale. Ils sont souvent appelés « locaux ». Ils ne sont visibles qu'à partir de leur point de déclaration jusqu'à la fin de la fonction ou du corps lambda. L'étendue locale est une sorte d'étendue de bloc, qui est abordée plus loin dans cet article.

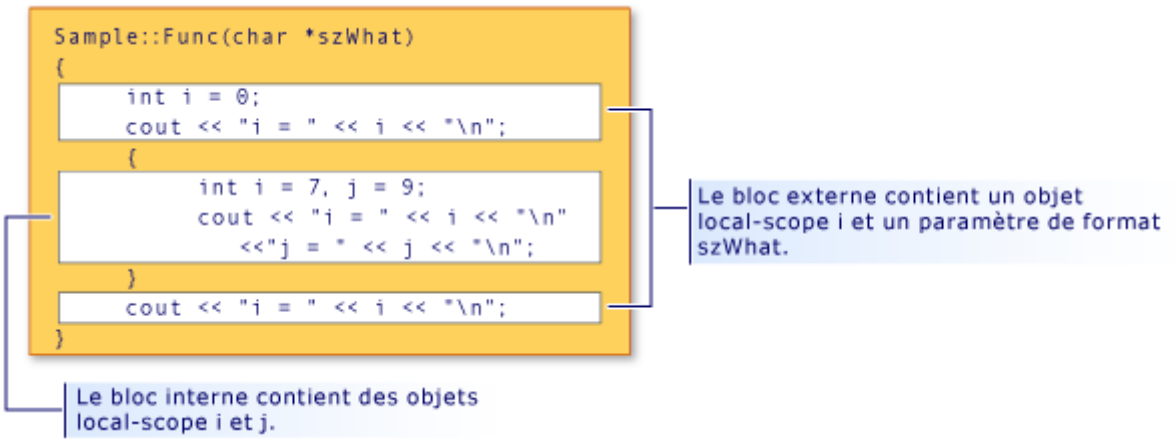
- **Étendue de classe** Les noms des membres de classe ont une étendue de classe, qui s'étend tout au long de la définition de classe, quel que soit le point de déclaration. L'accessibilité des membres de classe est davantage contrôlée par les `public` mots clés , `private` et `protected` . Les membres publics ou protégés sont accessibles uniquement à l'aide des opérateurs de sélection de membres (. ou ->) ou des opérateurs pointeur vers membre (.* ou ->*).

- **Étendue de l'instruction** Les noms déclarés dans une `for` instruction , `if`, `while` ou `switch` sont visibles jusqu'à la fin du bloc d'instruction.

- **Étendue de la fonction** Une étiquette a une étendue de fonction, ce qui signifie qu'elle est visible dans tout le corps d'une fonction avant même son point de déclaration. L'étendue de fonction permet d'écrire des instructions comme `goto cleanup` avant la déclaration de l'étiquette `cleanup` .

Masquage des noms

Vous pouvez masquer un nom en le déclarant dans un bloc englobé. Dans l'illustration suivante, `i` est redéclaré dans le bloc interne, masquant ainsi la variable associée à `i` dans la portée de bloc externe.



Bloquer l'étendue et le masquage de nom

La sortie du programme représentée dans l'illustration est la suivante :

```
C++
```

```
i = 0
i = 7
j = 9
i = 0
```

> ⓘ Notes
>
> L'argument `szWhat` est considéré comme étant dans la portée de la fonction. Par conséquent, il est traité comme s'il avait été déclaré dans le bloc le plus à l'extérieur de la fonction.

Masquage des noms de classes

Vous pouvez masquer les noms de classe en déclarant une fonction, un objet, une variable ou un énumérateur dans la même portée. Toutefois, le nom de la classe est toujours accessible lorsqu'il est préfixé par le mot clé **`class`**.

C++

```
// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at global scope.
class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
private:
    double balance;
};

double Account = 15.37;            // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                       //  class name

    cout << "Opening account with a balance of: "
         << Checking.GetBalance() << "\n";
}
//Output: Opening account with a balance of: 15.37
```

> ⓘ **Notes**
>
> Quel que soit l'endroit pour lequel le nom de classe (`Account`) est appelé, la classe mot clé doit être utilisée pour la différencier du compte de variable d'étendue globale. Cette règle ne s'applique pas lorsque le nom de classe apparaît à gauche de l'opérateur de résolution de portée (::). Les noms à gauche de l'opérateur de résolution de portée sont toujours considérés comme des noms de classe.

L'exemple suivant montre comment déclarer un pointeur vers un objet de type `Account` à l'aide du **`class`** mot clé :

C++

```
class Account *Checking = new class Account( Account );
```

dans `Account` l'initialiseur (entre parenthèses) de l'instruction précédente a une étendue globale ; il est de type **`double`**.

> ⓘ Notes
>
> La réutilisation des noms d'identificateur, comme l'indique cet exemple, est considérée comme un style de programmation médiocre.

Pour plus d'informations sur la déclaration et l'initialisation d'objets de classe, consultez Classes, structures et unions. Pour plus d'informations sur l'utilisation des **`new`** opérateurs et **`delete`** de stockage gratuit, consultez opérateurs nouveaux et supprimer.

Masquage de noms avec une étendue globale

Vous pouvez masquer les noms avec une étendue globale en déclarant explicitement le même nom dans l'étendue du bloc. Toutefois, les noms d'étendue globale sont accessibles à l'aide de l'opérateur de résolution d'étendue (`::`).

C++

```
#include <iostream>

int i = 7;   // i has global scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
   int i = 5;   // i has block scope, hides i at global scope
   cout << "Block-scoped i has the value: " << i << "\n";
   cout << "Global-scoped i has the value: " << ::i << "\n";
}
```

Output

```
Block-scoped i has the value: 5
Global-scoped i has the value: 7
```

Voir aussi

Concepts de base

Fichiers d'en-tête (C++)

Article • 16/06/2023

Les noms des éléments de programme tels que les variables, les fonctions, les classes, etc., doivent être déclarés avant de pouvoir être utilisés. Par exemple, vous ne pouvez pas simplement écrire `x = 42` sans déclarer au préalable « x ».

C++

```
int x; // declaration
x = 42; // use x
```

La déclaration indique au compilateur si l'élément est un `int`, un `double`, une **fonction**, un `class` ou une autre chose. En outre, chaque nom doit être déclaré (directement ou indirectement) dans chaque fichier .cpp dans lequel il est utilisé. Lorsque vous compilez un programme, chaque fichier .cpp est compilé indépendamment dans une unité de compilation. Le compilateur ne connaît pas les noms déclarés dans d'autres unités de compilation. Cela signifie que si vous définissez une classe, une fonction ou une variable globale, vous devez fournir une déclaration de cette chose dans chaque fichier .cpp supplémentaire qui l'utilise. Chaque déclaration de cette chose doit être exactement identique dans tous les fichiers. Une légère incohérence provoque des erreurs ou un comportement involontaire lorsque l'éditeur de liens tente de fusionner toutes les unités de compilation dans un seul programme.

Pour réduire le risque d'erreurs, C++ a adopté la convention d'utilisation de *fichiers d'en-tête* pour contenir des déclarations. Vous effectuez les déclarations dans un fichier d'en-tête, puis utilisez la directive #include dans chaque fichier .cpp ou tout autre fichier d'en-tête qui nécessite cette déclaration. La directive #include insère une copie du fichier d'en-tête directement dans le fichier .cpp avant la compilation.

> ⓘ **Notes**
>
> Dans Visual Studio 2019, la fonctionnalité *modules* C++20 est introduite en tant qu'amélioration et remplacement éventuel des fichiers d'en-tête. Pour plus d'informations, consultez **Vue d'ensemble des modules en C++**.

Exemple

L'exemple suivant montre un moyen courant de déclarer une classe, puis de l'utiliser dans un autre fichier source. Nous allons commencer par le fichier d'en-tête, `my_class.h`. Il contient une définition de classe, mais notez que la définition est incomplète ; la fonction `do_something` membre n'est pas définie :

C++

```
// my_class.h
namespace N
{
    class my_class
    {
    public:
        void do_something();
    };

}
```

Ensuite, créez un fichier d'implémentation (généralement avec une extension .cpp ou similaire). Nous allons appeler le fichier my_class.cpp et fournir une définition pour la déclaration de membre. Nous ajoutons une `#include` directive pour le fichier « my_class.h » afin que la déclaration my_class soit insérée à ce stade dans le fichier .cpp, et nous incluons pour extraire `<iostream>` la déclaration pour `std::cout`. Notez que les guillemets sont utilisés pour les fichiers d'en-tête dans le même répertoire que le fichier source, et que les crochets angulaires sont utilisés pour les en-têtes de bibliothèque standard. En outre, de nombreux en-têtes de bibliothèque standard n'ont pas .h ou toute autre extension de fichier.

Dans le fichier d'implémentation, nous pouvons éventuellement utiliser une **`using`** instruction pour éviter d'avoir à qualifier chaque mention de « my_class » ou « cout » avec « N:: » ou « std:: ». Ne placez **`using`** pas d'instructions dans vos fichiers d'en-tête !

C++

```
// my_class.cpp
#include "my_class.h" // header in local directory
#include <iostream> // header in standard library

using namespace N;
using namespace std;

void my_class::do_something()
{
    cout << "Doing something!" << endl;
}
```

Nous pouvons maintenant utiliser `my_class` dans un autre fichier .cpp. Nous #include le fichier d'en-tête afin que le compilateur extrait la déclaration. Tout ce que le compilateur doit savoir, c'est que my_class est une classe qui a une fonction membre publique appelée `do_something()`.

C++

```
// my_program.cpp
#include "my_class.h"

using namespace N;

int main()
{
    my_class mc;
    mc.do_something();
    return 0;
}
```

Une fois que le compilateur a terminé de compiler chaque fichier .cpp dans des fichiers .obj, il transmet les fichiers .obj à l'éditeur de liens. Lorsque l'éditeur de liens fusionne les fichiers d'objets, il trouve exactement une définition pour my_class ; il se trouve dans le fichier .obj produit pour my_class.cpp, et la build réussit.

Inclure des gardes

En règle générale, les fichiers d'en-tête ont un *include guard* ou une `#pragma once` directive pour s'assurer qu'ils ne sont pas insérés plusieurs fois dans un seul fichier .cpp.

C++

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

namespace N
{
    class my_class
    {
    public:
        void do_something();
    };
}

#endif /* MY_CLASS_H */
```

Éléments à placer dans un fichier d'en-tête

Étant donné qu'un fichier d'en-tête peut potentiellement être inclus par plusieurs fichiers, il ne peut pas contenir de définitions susceptibles de produire plusieurs définitions du même nom. Les éléments suivants ne sont pas autorisés ou sont considérés comme de très mauvaises pratiques :

- définitions de type intégrées au niveau de l'espace de noms ou de l'étendue globale
- définitions de fonction non inline
- définitions de variables non const
- définitions d'agrégats
- espaces de noms sans nom
- Directives using

L'utilisation de la `using` directive ne provoque pas nécessairement une erreur, mais peut potentiellement provoquer un problème, car elle place l'espace de noms dans l'étendue de chaque fichier .cpp qui inclut directement ou indirectement cet en-tête.

Exemple de fichier d'en-tête

L'exemple suivant montre les différents types de déclarations et de définitions autorisés dans un fichier d'en-tête :

C++

```
// sample.h
#pragma once
#include <vector> // #include directive
#include <string>

namespace N  // namespace declaration
{
    inline namespace P
    {
        //...
    }

    enum class colors : short { red, blue, purple, azure };

    const double PI = 3.14;  // const and constexpr definitions
    constexpr int MeaningOfLife{ 42 };
    constexpr int get_meaning()
    {
        static_assert(MeaningOfLife == 42, "unexpected!"); // static_assert
        return MeaningOfLife;
    }
```

```
    using vstr = std::vector<int>;  // type alias
    extern double d; // extern variable

#define LOG   // macro definition

#ifdef LOG   // conditional compilation directive
    void print_to_log();
#endif

    class my_class   // regular class definition,
    {                // but no non-inline function definitions

        friend class other_class;
    public:
        void do_something();   // definition in my_class.cpp
        inline void put_value(int i) { vals.push_back(i); } // inline OK

    private:
        vstr vals;
        int i;
    };

    struct RGB
    {
        short r{ 0 };  // member initialization
        short g{ 0 };
        short b{ 0 };
    };

    template <typename T>  // template definition
    class value_store
    {
    public:
        value_store<T>() = default;
        void write_value(T val)
        {
            //... function definition OK in template
        }
    private:
        std::vector<T> vals;
    };

    template <typename T>  // template declaration
    class value_widget;
}
```

Unités de traduction et liaison

Article • 03/04/2023

Dans un programme C++, un *symbole*, par exemple un nom de variable ou de fonction, peut être déclaré un nombre quelconque de fois dans son étendue. Toutefois, il ne peut être défini qu'une seule fois. Cette règle est la « règle de définition unique » (ODR). Une *déclaration* introduit (ou réintroduise) un nom dans le programme, ainsi que suffisamment d'informations pour associer ultérieurement le nom à une définition. Une *définition* introduit un nom et fournit toutes les informations nécessaires pour le créer. Si le nom représente une variable, une définition crée explicitement le stockage et l'initialise. Une *définition de fonction* se compose de la signature et du corps de la fonction. Une définition de classe se compose du nom de classe suivi d'un bloc qui répertorie tous les membres de la classe. (Les corps des fonctions membres peuvent éventuellement être définis séparément dans un autre fichier.)

L'exemple suivant montre certaines déclarations :

C++

```
int i;
int f(int x);
class C;
```

L'exemple suivant montre certaines définitions :

C++

```
int i{42};
int f(int x){ return x * i; }
class C {
public:
   void DoSomething();
};
```

Un programme se compose d'une ou plusieurs *unités de traduction*. Une unité de traduction se compose d'un fichier d'implémentation et de tous les en-têtes qu'elle inclut directement ou indirectement. Les fichiers d'implémentation ont généralement une extension de fichier de `.cpp` ou `.cxx`. Les fichiers d'en-tête ont généralement une extension de `.h` ou `.hpp`. Chaque unité de traduction est compilée indépendamment par le compilateur. Une fois la compilation terminée, l'éditeur de liens fusionne les unités de traduction compilées dans un seul *programme*. Les violations de la règle ODR s'affichent généralement sous forme d'erreurs d'éditeur de liens. Des erreurs d'éditeur

de liens se produisent lorsque le même nom est défini dans plusieurs unités de traduction.

En général, la meilleure façon de rendre une variable visible dans plusieurs fichiers consiste à la déclarer dans un fichier d'en-tête. Ajoutez ensuite une `#include` directive dans chaque `.cpp` fichier qui nécessite la déclaration. En ajoutant *des protections include* autour du contenu de l'en-tête, vous vous assurez que les noms déclarés par un en-tête ne sont déclarés qu'une seule fois pour chaque unité de traduction. Définissez le nom dans un seul fichier d'implémentation.

Dans C++20, les modules sont introduits comme une alternative améliorée aux fichiers d'en-tête.

Dans certains cas, il peut être nécessaire de déclarer une variable ou une classe globale dans un `.cpp` fichier. Dans ce cas, vous avez besoin d'un moyen d'indiquer au compilateur et à l'éditeur de liens le type de *liaison* du nom. Le type de liaison spécifie si le nom de l'objet n'est visible que dans un seul fichier ou dans tous les fichiers. Le concept de liaison s'applique uniquement aux noms globaux. Le concept de liaison ne s'applique pas aux noms déclarés dans une étendue. Une étendue est spécifiée par un ensemble d'accolades englobantes, comme dans les définitions de fonction ou de classe.

Liaison externe et interne

Une *fonction libre* est une fonction qui est définie au niveau de l'étendue globale ou de l'espace de noms. Les variables globales non const et les fonctions libres ont par défaut une *liaison externe* ; elles sont visibles à partir de n'importe quelle unité de traduction du programme. Aucun autre objet global ne peut avoir ce nom. Un symbole avec *liaison interne* ou *aucune liaison n'est* visible que dans l'unité de traduction dans laquelle il est déclaré. Lorsqu'un nom a une liaison interne, le même nom peut exister dans une autre unité de traduction. Les variables déclarées dans les définitions de classe ou les corps de fonction n'ont aucune liaison.

Vous pouvez forcer un nom global à avoir une liaison interne en le déclarant explicitement comme `static`. Cette mot clé limite sa visibilité à l'unité de traduction dans laquelle elle est déclarée. Dans ce contexte, `static` signifie quelque chose de différent de celui appliqué à des variables locales.

Les objets suivants ont une liaison interne par défaut :

- Objets `const`
- Objets `constexpr`

- Objets `typedef`
- `static` objets dans l'étendue de l'espace de noms

Pour attribuer une liaison externe d'objet `const` , déclarez-le en tant que `extern` et affectez-lui une valeur :

```
C++
```

```
extern const int value = 42;
```

Pour plus d'informations, consultez extern.

Voir aussi

Concepts de base

`main` fonction et arguments de ligne de commande

Article • 03/04/2023

Tous les programmes C++ doivent avoir une `main` fonction. Si vous essayez de compiler un programme C++ sans `main` fonction, le compilateur génère une erreur. (Les bibliothèques et static bibliothèques de liens dynamiques n'ont pas de `main` fonction.) La `main` fonction est l'endroit où votre code source commence l'exécution, mais avant qu'un programme n'entre dans la `main` fonction, tous les static membres de classe sans initialiseurs explicites sont définis sur zéro. Dans Microsoft C++, les objets globaux static sont également initialisés avant l'entrée dans `main`. Plusieurs restrictions s'appliquent à la `main` fonction qui ne s'appliquent à aucune autre fonction C++. La fonction `main` :

- Ne peut pas être surchargé (voir Surcharge des fonctions).
- Ne peut pas être déclaré en tant que **`inline`**.
- Ne peut pas être déclaré en tant que **`static`**.
- Son adresse ne peut pas être prise.
- Ne peut pas être appelée à partir de votre programme.

Signature de la fonction `main`

La fonction `main` n'a pas de déclaration, car elle est intégrée au langage. Si elle en avait, la syntaxe de déclaration pour `main` ressemblerait à ceci :

C++

```
int main();
int main(int argc, char *argv[]);
```

Si aucune valeur de retour n'est spécifiée dans `main`, le compilateur fournit une valeur de retour de zéro.

Arguments de ligne de commande standard

Les arguments pour `main` permettent l'analyse pratique de la ligne de commande des arguments. Les types pour `argc` et `argv` sont définis par le langage. Les noms `argc` et `argv` sont traditionnels, mais vous pouvez les nommer comme vous le souhaitez.

Les définitions d'argument sont les suivantes :

`argc`
Entier qui contient le nombre d'arguments qui suivent dans *argv*. Le paramètre *argc* est toujours supérieur ou égal à 1.

`argv`
Tableau de chaînes terminées par le caractère NULL qui représentent les arguments de ligne de commande entrés par l'utilisateur du programme. Par convention, `argv[0]` est la commande avec laquelle le programme est appelé. `argv[1]` est le premier argument de ligne de commande. Le dernier argument de la ligne de commande est `argv[argc - 1]`, et `argv[argc]` est toujours NULL.

Pour plus d'informations sur la suppression du traitement en ligne de commande, consultez Personnaliser le traitement en ligne de commande C++.

> ⓘ **Notes**
>
> Par convention, `argv[0]` est le nom de fichier du programme. Toutefois, sur Windows, il est possible de générer un processus à l'aide **CreateProcess**de . Si vous utilisez à la fois le premier et le deuxième arguments (`lpApplicationName` et `lpCommandLine`), `argv[0]` il est possible que le nom de l'exécutable ne soit pas le nom de l'exécutable. Vous pouvez utiliser **GetModuleFileName** pour récupérer le nom de l'exécutable et son chemin d'accès complet.

Extensions spécifiques à Microsoft

Les sections suivantes décrivent le comportement propre à Microsoft.

Fonction `wmain` et `_tmain` macro

Si vous concevez votre code source de façon à utiliser des acters unicode larges char, vous pouvez utiliser le point d'entrée spécifique à `wmain` Microsoft, qui est la version wide-acterchar de `main`. Voici la syntaxe de déclaration effective pour `wmain`:

C++

```
int wmain();
int wmain(int argc, wchar_t *argv[]);
```

Vous pouvez également utiliser la macro spécifique à `_tmain`Microsoft , qui est une macro de préprocesseur définie dans *`tchar.h`*. `_tmain` est résolu par `main` sauf si `_UNICODE` est défini. Dans ce cas, `_tmain` est résolu à `wmain`. La `_tmain` macro et les autres macros qui commencent par `_t` sont utiles pour le code qui doit générer des versions distinctes pour les jeux d'acteurs étroits et larges char. Pour plus d'informations, consultez Utilisation de mappages de texte génériques.

Retour `void` de main

En tant qu'extension Microsoft, les `main` fonctions et `wmain` peuvent être déclarées comme retournant **`void`** (aucune valeur de retour). Cette extension est également disponible dans d'autres compilateurs, mais son utilisation n'est pas recommandée. Il est disponible pour la symétrie lorsque `main` ne retourne pas de valeur.

Si vous déclarez `main` ou `wmain` que vous retournez **`void`**, vous ne pouvez pas retourner de exit code au processus parent ou au système d'exploitation à l'aide d'une return instruction . Pour retourner un exit code lorsque `main` ou `wmain` est déclaré comme **`void`**, vous devez utiliser la exit fonction .

Argument de ligne de commande `envp`

Les `main` signatures ou `wmain` autorisent une extension facultative spécifique à Microsoft pour l'accès aux variables d'environnement. Cette extension est également courante dans d'autres compilateurs pour les systèmes Windows et UNIX. Le nom *`envp`* est traditionnel, mais vous pouvez nommer le paramètre d'environnement comme vous le souhaitez. Voici les déclarations effectives pour les listes d'arguments qui incluent le paramètre d'environnement :

C++

```
int main(int argc, char* argv[], char* envp[]);
int wmain(int argc, wchar_t* argv[], wchar_t* envp[]);
```

`envp`

Le paramètre facultatif *`envp`* est un tableau de chaînes représentant les variables définies dans l'environnement de l'utilisateur. Ce tableau se termine par une entrée NULL. Il peut être déclaré en tant que tableau de pointeurs vers **`char`** (`char *envp[]`) ou en tant que pointeur vers **`char`** (`char **envp`). Si votre programme utilise `wmain` au lieu de `main`, utilisez le **`wchar_t`** type de données au lieu de **`char`**.

Le bloc d'environnement passé à `main` et `wmain` est une copie « figée » de l'environnement actuel. Si vous modifiez ultérieurement l'environnement en effectuant un appel à `putenv` ou `_wputenv`, l'environnement actuel (tel que retourné par `getenv` ou `_wgetenv` et la `_environ` variable ou `_wenviron`) change, mais le bloc pointé par *`envp`* ne change pas. Pour plus d'informations sur la suppression du traitement de l'environnement, consultez Personnaliser le traitement en ligne de commande C++. L'argument *`envp`* est compatible avec la norme C89, mais pas avec les normes C++.

Exemples d'arguments à `main`

L'exemple suivant montre comment utiliser les *`argc`* arguments , *`argv`* et *`envp`* pour `main`:

C++

```
// argument_definitions.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;
int main( int argc, char *argv[], char *envp[] )
{
    bool numberLines = false;    // Default is no line numbers.

    // If /n is passed to the .exe, display numbered listing
    // of environment variables.
    if ( (argc == 2) && _stricmp( argv[1], "/n" ) == 0 )
         numberLines = true;

    // Walk through list of strings until a NULL is encountered.
    for ( int i = 0; envp[i] != NULL; ++i )
    {
        if ( numberLines )
            cout << i << ": "; // Prefix with numbers if /n specified
        cout << envp[i] << "\n";
    }
}
```

Analyse des arguments de ligne de commande C++

Les règles d'analyse de ligne de commande utilisées par le code Microsoft C/C++ sont spécifiques à Microsoft. Le code de démarrage du runtime utilise ces règles lors de

l'interprétation des arguments donnés sur la ligne de commande du système d'exploitation :

- Les arguments sont délimités par un espace blanc, qui peut être un espace ou une tabulation.
- Le premier argument (`argv[0]`) est traité de manière spéciale. Il représente le nom du programme. Comme il doit s'agir d'un nom de chemin d'accès valide, les parties entourées de guillemets doubles (`"`) sont autorisées. Les guillemets doubles ne sont pas inclus dans la sortie `argv[0]`. Les parties entourées de guillemets doubles empêchent l'interprétation d'un espace ou d'une tabulation charcomme fin de l'argument. Les règles suivantes répertoriées dans cette liste ne s'appliquent pas.
- Une chaîne entourée de guillemets doubles est interprétée comme un argument unique, qui peut contenir des espaces blancs char. Une chaîne entre guillemets peut être incorporée dans un argument. L'insertion (`^`) n'est pas reconnue en tant qu'élément d'échappement charou délimiteur. Dans une chaîne entre guillemets, une paire de guillemets doubles est interprétée comme un seul guillemet double échappé. Si la ligne de commande se termine avant qu'un guillemet double fermant ne soit trouvé, tous les characters lus jusqu'à présent sont générés comme dernier argument.
- Un guillemet double précédé d'une barre oblique inverse (`\"`) est interprété comme un guillemet double littéral (`"`).
- Les barres obliques inverses sont interprétées littéralement, sauf si elles précèdent immédiatement un guillemet double.
- Si un nombre pair de barres obliques inverses est suivi d'un guillemet double, une barre oblique inverse (`\`) est placée dans le tableau `argv` pour chaque paire de barres obliques inverses (`\\`) et le guillemet double (`"`) est interprété comme un délimiteur de chaîne.
- Si un nombre impair de barres obliques inverses est suivi d'un guillemet double, une barre oblique inverse (`\`) est placée dans le tableau `argv` pour chaque paire de barres obliques inverses (`\\`). Le guillemet double est interprété comme une séquence d'échappement par la barre oblique inverse de réinitulationmain, ce qui entraîne le passage d'un guillemet double littéral (`"`) dans `argv`.

Exemple d'analyse d'argument de ligne de commande

Le programme suivant montre comment les arguments de ligne de commande sont passés :

C++

```
// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main( int argc,      // Number of strings in array argv
          char *argv[],   // Array of command-line argument strings
          char *envp[] )  // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
         cout << "  argv[" << count << "]   "
                << argv[count] << "\n";
}
```

Résultats de l'analyse des lignes de commande

Le tableau suivant montre un exemple d'entrée et de sortie attendue, montrant les règles de la liste précédente.

Entrée de ligne de commande	argv[1]	argv[2]	argv[3]
`"abc" d e`	`abc`	`d`	`e`
`a\\b d"e f"g h`	`a\\b`	`de fg`	`h`
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`
`a"b"" c d`	`ab" c d`		

Développement des caractères génériques

Le compilateur Microsoft vous permet éventuellement d'utiliser des acteurs *génériques*char, le point d'interrogation (`?`) et l'astérisque (`*`), pour spécifier les arguments nom de fichier et chemin d'accès sur la ligne de commande.

Les arguments de ligne de commande sont gérés par une routine interne dans le code de démarrage du runtime, qui, par défaut, ne développe pas les caractères génériques dans des chaînes distinctes dans le `argv` tableau de chaînes. Vous pouvez activer l'extension de caractères génériques en incluant le *`setargv.obj`* fichier (*`wsetargv.obj`* fichier pour `wmain`) dans vos **`/link`** options du compilateur ou dans votre **`LINK`** ligne de commande.

Pour plus d'informations sur les options de l'éditeur de liens de démarrage du runtime, consultez Options de lien.

Personnaliser le traitement en ligne de commande C++

Si votre programme ne prend pas d'arguments de ligne de commande, vous pouvez supprimer la routine de traitement de ligne de commande pour économiser une petite quantité d'espace. Pour supprimer son utilisation, incluez le fichier *`noarg.obj`* (pour `main` et `wmain`) dans vos options de compilateur **`/link`** ou votre ligne de commande **`LINK`**.

De même, si vous n'accédez jamais à la table d'environnement par le biais de l'argument *`envp`*, vous pouvez supprimer la routine de traitement de l'environnement interne. Pour supprimer son utilisation, incluez le fichier *`noenv.obj`* (pour `main` et `wmain`) dans vos options de compilateur **`/link`** ou votre ligne de commande **`LINK`**.

Votre programme peut effectuer des appels à la famille de routines `spawn` ou `exec` dans la bibliothèque Runtime C. Si c'est le cas, vous ne devez pas supprimer la routine de traitement de l'environnement, car elle est utilisée pour transmettre un environnement du processus parent au processus enfant.

Voir aussi

Concepts de base

Arrêt du programme C++

Article • 16/06/2023

En C++, vous pouvez quitter un programme de ces manières :

- Appelez la fonction exit.
- Appelez la fonction abort.
- Exécutez une return instruction à partir de `main`.

Fonction `exit`

La exit fonction, déclarée dans <stdlib.h>, met fin à un programme C++. La valeur fournie en tant qu'argument à `exit` est retournée au système d'exploitation en tant que code de retour ou code de sortie du programme. Par convention, un code de retour égal à zéro signifie que le programme s'est terminé correctement. Vous pouvez utiliser les constantes `EXIT_FAILURE` et `EXIT_SUCCESS`, également définies dans <stdlib.h>, pour indiquer la réussite ou l'échec de votre programme.

Fonction `abort`

La abort fonction, également déclarée dans le fichier <include standard stdlib.h>, met fin à un programme C++. La différence entre `exit` et `abort` est que `exit` permet au traitement de l'arrêt du runtime C++ d'avoir lieu (les destructeurs d'objets globaux sont appelés). `abort` arrête immédiatement le programme. La `abort` fonction contourne le processus de destruction normal pour les objets statiques globaux initialisés. Il ignore également tout traitement spécial qui a été spécifié avec la fonction atexit.

Spécifique à Microsoft : pour des raisons de compatibilité Windows, l'implémentation Microsoft de peut autoriser l'exécution du code d'arrêt de `abort` DLL dans certaines circonstances. Pour plus d'informations, consultez abort.

Fonction `atexit`

Utilisez la atexit fonction pour spécifier les actions qui s'exécutent avant l'arrêt du programme. Aucun objet statique global initialisé avant l'appel à n'est `atexit` détruit avant l'exécution de la fonction de traitement de sortie.

`return` instruction dans `main`

L'instruction `return` vous permet de spécifier une valeur de retour à partir de `main`. Une return instruction dans agit d'abord `main` comme n'importe quelle autre `return` instruction. Toutes les variables automatiques sont détruites. Ensuite, `main` appelle `exit` avec la valeur de retour en tant que paramètre. Prenons l'exemple suivant :

C++

```
// return_statement.cpp
#include <stdlib.h>
struct S
{
    int value;
};
int main()
{
    S s{ 3 };

    exit( 3 );
    // or
    return 3;
}
```

Les `exit` instructions et `return` de l'exemple précédent ont un comportement similaire. Les deux arrêtent le programme et retournent une valeur de 3 au système d'exploitation. La différence est que `exit` ne détruit pas la variable `s`automatique , tandis que l'instruction le `return` fait.

Normalement, C++ nécessite que les fonctions qui ont des types de retour autres que `void` retournent une valeur. La `main` fonction est une exception ; elle peut se terminer sans `return` instruction. Dans ce cas, elle retourne une valeur spécifique à l'implémentation au processus d'appel. (Par défaut, MSVC retourne 0.)

Destruction des threads et des objets statiques

Lorsque vous appelez `exit` directement (ou lorsqu'il est appelé après une `return` instruction de `main`), les objets thread associés au thread actuel sont détruits. Ensuite, les objets statiques sont détruits dans l'ordre inverse de leur initialisation (après les appels aux fonctions spécifiées à `atexit`, le cas échéant). L'exemple suivant montre comment ce type d'initialisation et de nettoyage fonctionne.

Exemple

Dans l'exemple suivant, les objets `sd1` statiques et `sd2` sont créés et initialisés avant l'entrée dans `main`. Une fois ce programme terminé à l'aide de l'instruction **`return`** , est d'abord `sd2` détruit, puis `sd1`. Le destructeur de la classe `ShowData` ferme les fichiers associés à ces objets statiques.

C++

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
public:
   // Constructor opens a file.
   ShowData( const char *szDev ) {
   errno_t err;
      err = fopen_s(&OutputDev, szDev, "w" );
   }

   // Destructor closes the file.
   ~ShowData() { fclose( OutputDev ); }

   // Disp function shows a string on the output device.
   void Disp( char *szData ) {
      fputs( szData, OutputDev );
   }
private:
   FILE *OutputDev;
};

//  Define a static object of type ShowData. The output device
//   selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

//  Define another static object of type ShowData. The output
//   is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
   sd1.Disp( "hello to default device\n" );
   sd2.Disp( "hello to file hello.dat\n" );
}
```

Une autre façon d'écrire ce code consiste à déclarer les objets avec une `ShowData` étendue de bloc, ce qui les détruit implicitement lorsqu'ils sortent de l'étendue :

C++

```
int main() {
   ShowData sd1( "CON" ), sd2( "hello.dat" );

   sd1.Disp( "hello to default device\n" );
```

```
    sd2.Disp( "hello to file hello.dat\n" );
}
```

Voir aussi

main fonction et arguments de ligne de commande

Lvalues et Rvalues (C++)

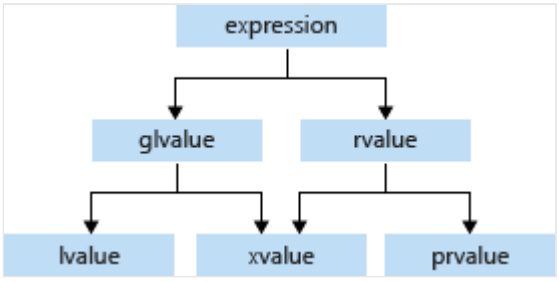
Article • 17/06/2023

Chaque expression C++ a un type et appartient à une *catégorie de valeur*. Les catégories de valeurs constituent la base des règles que les compilateurs doivent suivre lors de la création, de la copie et du déplacement d'objets temporaires pendant l'évaluation d'expression.

La norme C++17 définit les catégories de valeurs d'expression comme suit :

- Une *glvalue* est une expression dont l'évaluation détermine l'identité d'un objet, d'un champ binaire ou d'une fonction.
- Une *prvalue* est une expression dont l'évaluation initialise un objet ou un champ de bits, ou calcule la valeur de l'opérande d'un opérateur, comme spécifié par le contexte dans lequel il apparaît.
- Une *valeur x* est une glvalue qui désigne un objet ou un champ de bits dont les ressources peuvent être réutilisées (généralement parce qu'il est proche de la fin de sa durée de vie). Exemple : certains types d'expressions impliquant des références rvalue (8.3.2) produisent des valeurs xvalues, telles qu'un appel à une fonction dont le type de retour est une référence rvalue ou un cast en type référence rvalue.
- Une *lvalue* est une glvalue qui n'est pas une valeur xvalue.
- Une *rvalue* est une prvalue ou une xvalue.

Le diagramme suivant illustre les relations entre les catégories :



Une valeur l a une adresse à laquelle votre programme peut accéder. Les exemples d'expressions lvalue incluent des noms de variables, notamment `const` des variables, des éléments de tableau, des appels de fonction qui retournent une référence lvalue, des champs de bits, des unions et des membres de classe.

Une expression prvalue n'a aucune adresse accessible par votre programme. Les exemples d'expressions prvalue incluent les littéraux, les appels de fonction qui retournent un type sans référence et les objets temporaires qui sont créés pendant l'évaluation d'expression, mais accessibles uniquement par le compilateur.

Une expression xvalue a une adresse qui n'est plus accessible par votre programme, mais qui peut être utilisée pour initialiser une référence rvalue, qui fournit l'accès à l'expression. Par exemple, les appels de fonction qui retournent une référence rvalue, ainsi que l'indice de tableau, le membre et le pointeur vers des expressions membres où le tableau ou l'objet est une référence rvalue.

Exemple

L'exemple suivant illustre plusieurs utilisations correctes et incorrectes des valeurs lvalues et des rvalues :

C++

```
// lvalues_and_rvalues2.cpp
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a
prvalue.
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106).`j * 4`
is a prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}
```

> ⓘ **Notes**
>
> Les exemples de cette rubrique illustrent une utilisation correcte et incorrecte lorsque les opérateurs ne sont pas surchargés. En surchargeant les opérateurs, vous pouvez faire d'une expression telle que `j * 4` une lvalue.

Les termes *lvalue* et *rvalue* sont souvent utilisés lorsque vous faites référence à des références d'objets. Pour plus d'informations sur les références, consultez Déclarateur de

référence Lvalue : & et Rvalue Reference Declarator : &&.

Voir aussi

Concepts de base
Déclarateur de référence Lvalue : &
Déclarateur de référence Rvalue : &&

Objets temporaires

Article • 03/04/2023

Un objet temporaire est un objet sans nom créé par le compilateur pour stocker une valeur temporaire.

Notes

Dans certains cas, il est nécessaire que le compilateur crée des objets temporaires. Ces objets temporaires peuvent être créés pour les raisons suivantes :

- Pour initialiser une `const` référence avec un initialiseur d'un type différent du type sous-jacent de la référence initialisée.
- Pour stocker la valeur de retour d'une fonction qui retourne un type défini par l'utilisateur (UDT). Ces temporaires sont créés uniquement si votre programme ne copie pas la valeur de retour dans un objet. Par exemple :

  ```
  C++

  UDT Func1();    //  Declare a function that returns a user-defined
                  //   type.

  ...

  Func1();        //  Call Func1, but discard return value.
                  //  A temporary object is created to store the return
                  //   value.
  ```

  Étant donné que la valeur de retour n'est pas copiée dans un autre objet, un objet temporaire est créé. Un des cas les plus courants où des objets temporaires sont créés est l'évaluation d'une expression où des fonctions surchargées d'opérateur doivent être appelées. Ces fonctions d'opérateur surchargées retournent un type défini par l'utilisateur qui n'est souvent pas copié dans un autre objet.

  Prenons l'exemple de l'expression `ComplexResult = Complex1 + Complex2 + Complex3`. L'expression `Complex1 + Complex2` est évaluée, et le résultat est stocké dans un objet temporaire. Ensuite, l'expression *temporaire*`+ Complex3` est évaluée et le résultat est copié dans `ComplexResult` (en supposant que l'opérateur d'affectation n'est pas surchargé).

- Pour stocker le résultat d'un cast en un type défini par l'utilisateur. Lorsqu'un objet d'un type donné est converti explicitement en un type défini par l'utilisateur, ce nouvel objet est construit sous la forme d'un objet temporaire.

Les objets temporaires ont une durée de vie, définie par leur point de création et le point auquel ils sont détruits. Toute expression qui crée plusieurs objets temporaires les détruit dans l'ordre inverse de création.

La destruction d'un temporaire dépend de la façon dont il est utilisé :

- Temporaires utilisés pour l'initialisation des `const` références :
  Si un initialiseur n'est pas une valeur l du même type que la référence en cours d'initialisation, un temporaire du type d'objet sous-jacent est créé. Il est initialisé par l'expression d'initialisation. Cet objet temporaire est détruit immédiatement après la destruction de l'objet de référence auquel il est lié. Comme cette destruction peut se produire bien après l'expression qui a créé le temporaire, elle est parfois appelée *extension de durée de vie*.

- Temporaires créés en tant qu'effet de l'évaluation d'expression :
  Tous les éléments temporaires qui ne correspondent pas à la première catégorie et qui sont créés en tant qu'effet d'évaluation d'expression, sont détruits à la fin de l'instruction d'expression (c'est-à-dire, au point-virgule) ou à la fin des expressions de contrôle pour `for`les instructions , , `if` `while`, `do`, et`switch`.

Voir aussi

Blog d'Herb Sutter sur les références, simplement

Alignment

Article • 03/04/2023

Une des fonctionnalités de bas niveau de C++ est la possibilité de spécifier avec précision l'alignement des objets en mémoire afin d'exploiter au mieux les capacités d'une architecture matérielle spécifique. Par défaut, le compilateur aligne les membres de classe et de struct sur leur valeur de taille : `bool` et `char` sur les limites de 1 octet, `short` sur les limites de 2 octets, `int`, `long` et `float` sur les limites de 4 octets et `long long`, `double` et `long double` sur les limites de 8 octets.

Dans la plupart des scénarios, vous n'avez jamais à vous soucier de l'alignement, car l'alignement par défaut est déjà optimal. Dans certains cas, toutefois, vous pouvez obtenir des améliorations significatives des performances ou des économies de mémoire en spécifiant un alignement personnalisé pour vos structures de données. Avant Visual Studio 2015, vous pouviez utiliser les mots clés `__alignof` spécifiques à Microsoft et `__declspec(align)` spécifier un alignement supérieur à la valeur par défaut. À compter de Visual Studio 2015, vous devez utiliser les mots clés `alignof` standard C++11 et `alignas` pour une portabilité maximale du code. Les nouveaux mots clés se comportent de la même façon que les extensions spécifiques à Microsoft. La documentation de ces extensions s'applique également aux nouveaux mots clés. Pour plus d'informations, consultez alignof Opérateur et alignement. La norme C++ ne spécifie pas le comportement d'empaquetage pour l'alignement sur des limites inférieures à la valeur par défaut du compilateur pour la plateforme cible. Vous devez donc toujours utiliser Microsoft #pragma pack dans ce cas.

Utilisez la classe aligned_storage pour l'allocation de mémoire des structures de données avec des alignements personnalisés. La classe aligned_union est destinée à spécifier l'alignement pour les unions avec des constructeurs ou des destructeurs non triviaux.

Alignement et adresses mémoire

L'alignement est une propriété d'une adresse mémoire, exprimée sous la forme de l'adresse numérique modulo une puissance de 2. Par exemple, l'adresse 0x0001103F modulo 4 est 3. Cette adresse est dite alignée sur 4n+3, où 4 indique la puissance choisie de 2. L'alignement d'une adresse dépend de la puissance choisie de 2. La même adresse modulo 8 est 7. Une adresse est considérée comme étant alignée sur X si son alignement est Xn+0.

Les processeurs exécutent des instructions qui fonctionnent sur les données stockées en mémoire. Les données sont identifiées par leurs adresses en mémoire. Une donnée unique a également une taille. Nous appelons une donnée *naturellement alignée* si son adresse est alignée sur sa taille. C'est ce qu'on appelle *une mauvaise alignement* dans le cas contraire. Par exemple, une donnée à virgule flottante de 8 octets est naturellement alignée si l'adresse utilisée pour l'identifier a un alignement de 8 octets.

Gestion par le compilateur de l'alignement des données

Les compilateurs tentent d'effectuer des allocations de données d'une manière qui empêche l'alignement incorrect des données.

Pour les types de données simples, le compilateur assigne des adresses qui sont des multiples de la taille en octets du type de données. Par exemple, le compilateur affecte des adresses à des variables de type `long` qui sont des multiples de 4, en définissant les 2 bits inférieurs de l'adresse sur zéro.

Le compilateur place également les structures d'une manière qui aligne naturellement chaque élément de la structure. Considérez la structure `struct x_` dans l'exemple de code suivant :

C++

```
struct x_
{
   char a;     // 1 byte
   int b;      // 4 bytes
   short c;    // 2 bytes
   char d;     // 1 byte
} bar[3];
```

Le compilateur remplit cette structure pour appliquer naturellement l'alignement.

L'exemple de code suivant montre comment le compilateur place la structure rembourrée en mémoire :

C++

```
// Shows the actual memory layout
struct x_
{
   char a;            // 1 byte
   char _pad0[3];     // padding to put 'b' on 4-byte boundary
   int b;            // 4 bytes
```

```
    short c;          // 2 bytes
    char d;           // 1 byte
    char _pad1[1];    // padding to make sizeof(x_) multiple of 4
} bar[3];
```

Les deux déclarations retournent `sizeof(struct x_)` 12 octets.

La deuxième déclaration inclut deux éléments de remplissage :

1. `char _pad0[3]` pour aligner le `int b` membre sur une limite de 4 octets.
2. `char _pad1[1]` pour aligner les éléments de tableau de la structure `struct _x bar[3];` sur une limite de quatre octets.

Le remplissage aligne les éléments de `bar[3]` de manière à permettre un accès naturel.

L'exemple de code suivant montre la disposition du `bar[3]` tableau :

Output

```
adr offset   element
------   -------
0x0000   char a;         // bar[0]
0x0001   char pad0[3];
0x0004   int b;
0x0008   short c;
0x000a   char d;
0x000b   char _pad1[1];

0x000c   char a;         // bar[1]
0x000d   char _pad0[3];
0x0010   int b;
0x0014   short c;
0x0016   char d;
0x0017   char _pad1[1];

0x0018   char a;         // bar[2]
0x0019   char _pad0[3];
0x001c   int b;
0x0020   short c;
0x0022   char d;
0x0023   char _pad1[1];
```

`alignof` et `alignas`

Le **`alignas`** spécificateur de type est un moyen portable standard C++ de spécifier l'alignement personnalisé des variables et des types définis par l'utilisateur. L'opérateur

`alignof` est également un moyen standard et portable d'obtenir l'alignement d'un type ou d'une variable spécifié.

Exemple

Vous pouvez utiliser `alignas` sur une classe, un struct ou une union, ou sur des membres individuels. Lorsque plusieurs `alignas` spécificateurs sont rencontrés, le compilateur choisit celui le plus strict (celui qui a la valeur la plus élevée).

C++

```
// alignas_alignof.cpp
// compile with: cl /EHsc alignas_alignof.cpp
#include <iostream>

struct alignas(16) Bar
{
    int i;       // 4 bytes
    int n;      // 4 bytes
    alignas(4) char arr[3];
    short s;          // 2 bytes
};

int main()
{
    std::cout << alignof(Bar) << std::endl; // output: 16
}
```

Voir aussi

Alignement de la structure de données

Types triviaux, à disposition standard, POD et littéraux

Article • 03/04/2023

Le terme *disposition* fait référence à la façon dont les membres d'un objet de classe, d'un struct ou d'un type union sont organisés dans la mémoire. Dans certains cas, la disposition est bien définie par la spécification du langage. Toutefois, quand une classe ou un struct contiennent certaines fonctionnalités du langage C++, comme des classes de base virtuelles, des fonctions virtuelles, des membres avec un contrôle d'accès différent, le compilateur est libre de choisir une disposition. Cette disposition peut varier selon les optimisations qui sont apportées. Dans de nombreux cas, l'objet peut même ne pas occuper une zone contiguë de mémoire. Par exemple, si une classe a des fonctions virtuelles, toutes les instances de cette classe partagent probablement une seule table de fonctions virtuelles. Ces types sont très utiles, mais ils présentent également des limites. Étant donné que la disposition n'est pas définie, ils ne peuvent pas être passés aux programmes écrits dans d'autres langages, comme C, et parce qu'ils risquent de ne pas être contigus, ils ne peuvent pas non plus être copiés de façon fiable avec des fonctions rapides de bas niveau comme `memcopy`, ni sérialisés sur un réseau.

Pour activer les compilateurs, ainsi que les programmes et métaprogrammes C++, afin qu'un type donné reste adapté aux opérations qui dépendent d'une disposition de mémoire particulière, C++14 a introduit trois catégories de classes et de structs simples : *trivial*, *disposition standard* et *POD* ou Plain Old Data. La bibliothèque Standard contient les modèles de fonction `is_trivial<T>`, `is_standard_layout<T>` et `is_pod<T>` qui déterminent si un type donné appartient à une catégorie donnée.

Types triviaux

Quand une classe ou un struct dans C++ a des fonctions membres spéciales définies explicitement par défaut ou fournies par le compilateur, il s'agit d'un type trivial. Il occupe une zone de mémoire contiguë. Il peut avoir des membres avec des spécificateurs d'accès différents. En C++, le compilateur est libre de choisir comment classer les membres dans cette situation. Par conséquent, vous pouvez utiliser memcopy pour copier ces objets, mais vous ne pouvez pas les consommer de façon fiable dans un programme C. Un type trivial T peut être copié dans un tableau de caractères ou de caractères non signés, puis copié sans problème dans une variable T. Notez qu'en raison des conditions d'alignement, il peut y avoir des octets de remplissage entre les membres de type.

Les types triviaux ont un constructeur par défaut trivial, un constructeur de copie trivial, un opérateur d'attribution de copie trivial et un destructeur trivial. Dans chaque cas, *trivial* signifie que le constructeur/opérateur/destructeur n'est pas fourni par l'utilisateur et appartient à une classe qui n'a

- aucune fonction virtuelle, aucune classe de base virtuelle,
- aucune classe de base avec un constructeur/opérateur/destructeur non trivial correspondant
- aucun membre de données de type de classe avec un constructeur/opérateur/destructeur non trivial correspondant

Les exemples suivants montrent des types triviaux. Dans Trivial2, la présence du constructeur `Trivial2(int a, int b)` vous demande de fournir un constructeur par défaut. Pour que le type soit considéré comme trivial, vous devez définir explicitement ce constructeur comme constructeur par défaut.

C++

```
struct Trivial
{
   int i;
private:
   int j;
};

struct Trivial2
{
   int i;
   Trivial2(int a, int b) : i(a), j(b) {}
   Trivial2() = default;
private:
   int j;   // Different access control
};
```

Types de disposition standard

Quand une classe ou un struct ne contiennent pas certaines fonctionnalités du langage C++ (comme des fonctions virtuelles) qui ne figurent pas dans le langage C, et que tous les membres ont le même contrôle d'accès, il s'agit d'un type à disposition standard. Il peut utiliser memcopy et la disposition est suffisamment définie pour être consommée par des programmes C. Les types de disposition standard peuvent avoir des fonctions membres spéciales définies par l'utilisateur. Par ailleurs, les types de disposition standard ont ces caractéristiques :

- aucune fonction virtuelle ni aucune classe de base virtuelle
- tous les membres de données non statiques ont le même contrôle d'accès
- tous les membres non statiques de type de classe sont à disposition standard
- toutes les classes de base sont à disposition standard
- aucune classe de base du même type que le premier membre de données non statique.
- remplit l'une des conditions suivantes :
  - aucun membre de données non statique dans la classe la plus dérivée et pas plus d'une classe de base avec des membres de données non statiques, ou
  - aucune classe de base avec des membres de données non statiques

Le code suivant montre un exemple d'un type à disposition standard :

C++

```
struct SL
{
   // All members have same access:
   int i;
   int j;
   SL(int a, int b) : i(a), j(b) {} // User-defined constructor OK
};
```

Les deux dernières conditions peuvent éventuellement être mieux illustrées avec du code. Dans l'exemple suivant, même si Base est à disposition standard, `Derived` n'est pas à disposition standard, car les deux (la classe la plus dérivée) et `Base` ont des membres de données non statiques :

C++

```
struct Base
{
   int i;
   int j;
};

// std::is_standard_layout<Derived> == false!
struct Derived : public Base
{
   int x;
   int y;
};
```

Dans cet exemple, `Derived` est à disposition standard, car `Base` n'a aucun membre de données non statique :

C++

```
struct Base
{
   void Foo() {}
};

// std::is_standard_layout<Derived> == true
struct Derived : public Base
{
   int x;
   int y;
};
```

Derived serait également à disposition standard si `Base` avait des membres de données et si `Derived` avait uniquement des fonctions membres.

Types POD

Quand une classe ou un struct sont à la fois triviaux et à disposition standard, il s'agit d'un type POD (Plain Old Data). La disposition en mémoire des types POD est par conséquent contiguë et chaque membre a une adresse supérieure à celle du membre qui a été déclaré avant lui, afin que les copies octet pour octet et les E/S binaires puissent être effectuées sur ces types. Les types scalaires comme int sont également des types POD. Les types POD qui sont des classes peuvent avoir uniquement des types POD comme membres de données non statiques.

Exemple

L'exemple suivant montre les différences entre les types triviaux, à disposition standard et POD :

C++

```
#include <type_traits>
#include <iostream>

using namespace std;

struct B
{
```

```
protected:
   virtual void Foo() {}
};

// Neither trivial nor standard-layout
struct A : B
{
   int a;
   int b;
   void Foo() override {} // Virtual function
};

// Trivial but not standard-layout
struct C
{
   int a;
private:
   int b;   // Different access control
};

// Standard-layout but not trivial
struct D
{
   int a;
   int b;
   D() {} //User-defined constructor
};

struct POD
{
   int a;
   int b;
};

int main()
{
   cout << boolalpha;
   cout << "A is trivial is " << is_trivial<A>() << endl; // false
   cout << "A is standard-layout is " << is_standard_layout<A>() << endl;
// false

   cout << "C is trivial is " << is_trivial<C>() << endl; // true
   cout << "C is standard-layout is " << is_standard_layout<C>() << endl;
// false

   cout << "D is trivial is " << is_trivial<D>() << endl;  // false
   cout << "D is standard-layout is " << is_standard_layout<D>() << endl; //
true

   cout << "POD is trivial is " << is_trivial<POD>() << endl; // true
   cout << "POD is standard-layout is " << is_standard_layout<POD>() <<
endl; // true

   return 0;
}
```

Types de littéral

Un type de littéral est un type dont la disposition peut être déterminée au moment de la compilation. Voici les types de littéral disponibles :

- void
- Types scalaires
- references
- Tableaux de types void, de types scalaires ou de références
- Classe contenant un destructeur trivial, et un ou plusieurs constructeurs constexpr autres que des constructeurs de copie ou de déplacement. Tous les membres de données non statiques et classes de base doivent également être des types de littéral et être non volatiles.

Voir aussi

Concepts de base

Classes C++ en tant que types de valeurs

Article • 16/06/2023

Les classes C++ sont, par défaut, des types valeur. Ils peuvent être spécifiés en tant que types de référence, ce qui permet un comportement polymorphe pour prendre en charge la programmation orientée objet. Les types valeur sont parfois consultés du point de vue de la mémoire et du contrôle de disposition, tandis que les types de référence concernent les classes de base et les fonctions virtuelles à des fins polymorphes. Par défaut, les types valeur peuvent être copiés, ce qui signifie qu'il existe toujours un constructeur de copie et un opérateur d'affectation de copie. Pour les types de référence, vous devez rendre la classe non copyable (désactiver le constructeur de copie et l'opérateur d'affectation de copie) et utiliser un destructeur virtuel, qui prend en charge leur polymorphisme prévu. Les types de valeurs concernent également le contenu, qui, lorsqu'ils sont copiés, vous donnent toujours deux valeurs indépendantes qui peuvent être modifiées séparément. Les types de référence concernent l'identité : quel type d'objet s'agit-il ? Pour cette raison, les « types de référence » sont également appelés « types polymorphes ».

Si vous souhaitez vraiment un type de type référence (classe de base, fonctions virtuelles), vous devez désactiver explicitement la copie, comme indiqué dans la `MyRefType` classe dans le code suivant.

C++

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);
public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

La compilation du code ci-dessus entraîne l'erreur suivante :

Output

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private
member declared in class 'MyRefType'
        meow.cpp(5) : see declaration of 'MyRefType::operator ='
        meow.cpp(3) : see declaration of 'MyRefType'
```

Types de valeurs et efficacité du déplacement

La surcharge d'allocation de copie est évitée en raison de nouvelles optimisations de copie. Par exemple, lorsque vous insérez une chaîne au milieu d'un vecteur de chaînes, il n'y a pas de surcharge de réallocation de copie, seulement un déplacement, même si cela entraîne une croissance du vecteur lui-même. Ces optimisations s'appliquent également à d'autres opérations : pour instance, l'exécution d'une opération d'ajout sur deux objets volumineux. Comment activez-vous ces optimisations d'opération de valeur ? Le compilateur les active implicitement, tout comme les constructeurs de copie peuvent être générés automatiquement par le compilateur. Toutefois, votre classe doit « accepter » pour déplacer des devoirs et déplacer des constructeurs en les déclarant dans votre définition de classe. Move utilise la référence rvalue double ampersand (&&) dans les déclarations de fonction membre appropriées et définit les méthodes d'affectation de déplacement et de constructeur de déplacement. Vous devez également insérer le code correct pour « voler les tripes » de l'objet source.

Comment décidez-vous si vous devez activer les opérations de déplacement ? Si vous savez déjà que vous devez activer la construction de copie, vous souhaitez probablement également activer la construction de déplacement, en particulier si elle est moins chère qu'une copie profonde. Toutefois, si vous savez que vous avez besoin d'une prise en charge du déplacement, cela ne signifie pas nécessairement que vous souhaitez activer les opérations de copie. Ce dernier cas serait appelé « type de déplacement uniquement ». Un exemple déjà dans la bibliothèque standard est `unique_ptr`. Notez que l'ancien `auto_ptr` est déprécié et a été remplacé par `unique_ptr` précisément en raison de l'absence de prise en charge de la sémantique de déplacement dans la version précédente de C++.

À l'aide de la sémantique de déplacement, vous pouvez retourner par valeur ou insérer au milieu. Le déplacement est une optimisation de la copie. Il n'est pas nécessaire d'utiliser l'allocation de tas comme solution de contournement. Considérez le pseudocode suivant :

C++

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData();   // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" );   // efficient, no deep copy-
shuffle
v.insert( begin(v)+v.size()/2, "Andrei" );  // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix& , const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix& ,       HugeMatrix&&);
HugeMatrix operator+(      HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+(      HugeMatrix&&,       HugeMatrix&&);
//...
hm5 = hm1+hm2+hm3+hm4+hm5;   // efficient, no extra copies
```

Activation du déplacement pour les types de valeurs appropriés

Pour une classe de type valeur où le déplacement peut être moins cher qu'une copie approfondie, activez la construction de déplacement et l'affectation de déplacement pour plus d'efficacité. Considérez le pseudocode suivant :

C++

```
#include <memory>
#include <stdexcept>
using namespace std;
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other )   // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other )   // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() {   // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient
```

```
resources!");
    }
};
```

Si vous activez la construction/l'affectation de copie, activez également la construction/l'affectation de déplacement si elle peut être moins coûteuse qu'une copie approfondie.

Certains types *sans valeur* sont déplacés uniquement, par exemple quand vous ne pouvez pas cloner une ressource, mais uniquement transférer la propriété. Exemple : `unique_ptr`.

Voir aussi

Système de type C++
Bienvenue dans C++
Informations de référence sur le langage C++
Bibliothèque C++ standard

Conversions de types et sécurité de type

Article • 03/04/2023

Ce document identifie les problèmes courants de conversion de type et explique comment les éviter dans votre code C++.

Lorsque vous écrivez un programme C++, il est important de s'assurer qu'il est de type sécurisé. Cela signifie que chaque variable, argument de fonction et valeur de retour de fonction stocke un type de données acceptable, et que les opérations qui impliquent des valeurs de différents types « sont logiques » et ne provoquent pas de perte de données, d'interprétation incorrecte des modèles de bits ou d'altération de la mémoire. Un programme qui ne convertit jamais explicitement ou implicitement des valeurs d'un type à un autre est un type sécurisé par définition. Toutefois, les conversions de type, même les conversions non sécurisées, sont parfois requises. Par exemple, vous devrez peut-être stocker le résultat d'une opération à virgule flottante dans une variable de type `int`, ou vous devrez peut-être passer la valeur dans un `unsigned int` à une fonction qui prend un `signed int`. Ces deux exemples illustrent des conversions non sécurisées, car elles peuvent entraîner une perte de données ou une ré-interprétation d'une valeur.

Lorsque le compilateur détecte une conversion non sécurisée, il émet une erreur ou un avertissement. Une erreur arrête la compilation ; un avertissement permet la poursuite de la compilation, mais indique une erreur possible dans le code. Toutefois, même si votre programme se compile sans avertissements, il peut toujours contenir du code qui conduit à des conversions de type implicites qui produisent des résultats incorrects. Les erreurs de type peuvent également être introduites par des conversions explicites, ou des casts, dans le code.

Conversions de type implicites

Lorsqu'une expression contient des opérandes de différents types intégrés et qu'aucune cast explicite n'est présente, le compilateur utilise *des conversions standard* intégrées pour convertir l'un des opérandes afin que les types correspondent. Le compilateur tente les conversions dans une séquence bien définie jusqu'à ce que l'une d'elles réussisse. Si la conversion sélectionnée est une promotion, le compilateur n'émet pas d'avertissement. Si la conversion est un rétrécissement, le compilateur émet un avertissement concernant une perte de données possible. La perte réelle de données dépend des valeurs réelles impliquées, mais nous vous recommandons de traiter cet avertissement comme une erreur. Si un type défini par l'utilisateur est impliqué, le compilateur tente d'utiliser les conversions que vous avez spécifiées dans la définition

de classe. S'il ne trouve pas de conversion acceptable, le compilateur génère une erreur et ne compile pas le programme. Pour plus d'informations sur les règles qui régissent les conversions standard, consultez Conversions standard. Pour plus d'informations sur les conversions définies par l'utilisateur, consultez Conversions définies par l'utilisateur (C++/CLI) .

Élargissement des conversions (promotion)

Dans une conversion étendue, une valeur d'une variable plus petite est affectée à une variable plus grande sans perte de données. Étant donné que les conversions étendues sont toujours sécurisées, le compilateur les effectue en mode silencieux et n'émet pas d'avertissements. Les conversions suivantes sont des conversions d'élargissement.

Du	À
Tout `signed` type ou `unsigned` intégral à l'exception `long long` de ou `__int64`	`double`
`bool` ou `char`	Tout autre type intégré
`short` ou `wchar_t`	`int`, `long`, `long long`
`int`, `long`	`long long`
`float`	`double`

Conversions restrictives (coercition)

Le compilateur effectue implicitement des conversions restrictives, mais il vous avertit d'une perte de données potentielle. Prenez ces avertissements très au sérieux. Si vous êtes certain qu'aucune perte de données ne se produira parce que les valeurs de la variable plus grande seront toujours ajustées dans la variable plus petite, ajoutez un cast explicite afin que le compilateur n'émette plus d'avertissement. Si vous n'êtes pas sûr que la conversion est sécurisée, ajoutez à votre code un type de case activée d'exécution pour gérer la perte de données possible afin qu'elle ne provoque pas que votre programme produise des résultats incorrects.

Toute conversion d'un type à virgule flottante en type intégral est une conversion restrictive, car la partie fractionnaire de la valeur à virgule flottante est ignorée et perdue.

L'exemple de code suivant montre certaines conversions restrictives implicites et les avertissements que le compilateur émet pour celles-ci.

C++

```
int i = INT_MAX + 1; //warning C4307:'+':integral constant overflow
wchar_t wch = 'A'; //OK
char c = wch; // warning C4244:'initializing':conversion from 'wchar_t'
              // to 'char', possible loss of data
unsigned char c2 = 0xfffe; //warning C4305:'initializing':truncation from
                           // 'int' to 'unsigned char'
int j = 1.9f; // warning C4244:'initializing':conversion from 'float' to
              // 'int', possible loss of data
int k = 7.7; // warning C4244:'initializing':conversion from 'double' to
             // 'int', possible loss of data
```

Signé - conversions non signées

Un type intégral signé et son équivalent non signé ont toujours la même taille, mais ils diffèrent par la façon dont le modèle de bits est interprété pour la transformation de valeur. L'exemple de code suivant montre ce qui se passe lorsque le même modèle de bits est interprété comme une valeur signée et comme une valeur non signée. Le modèle de bits stocké dans les deux `num` et `num2` ne change jamais de ce qui est illustré dans l'illustration précédente.

C++

```
using namespace std;
unsigned short num = numeric_limits<unsigned short>::max(); // #include
<limits>
short num2 = num;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"

// Go the other way.
num2 = -1;
num = num2;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"
```

Notez que les valeurs sont réinterprétées dans les deux sens. Si votre programme produit des résultats impairs dans lesquels le signe de la valeur semble inversé par rapport à ce que vous attendez, recherchez les conversions implicites entre les types intégraux signés et non signés. Dans l'exemple suivant, le résultat de l'expression (0 - 1) est implicitement converti de **`int`** à **`unsigned int`** lorsqu'il est stocké dans `num`. Cela entraîne une réinterprétation du modèle de bits.

C++

```
unsigned int u3 = 0 - 1;
cout << u3 << endl; // prints 4294967295
```

Le compilateur n'avertit pas des conversions implicites entre les types intégraux signés et non signés. Par conséquent, nous vous recommandons d'éviter les conversions signées à non signées. Si vous ne pouvez pas les éviter, ajoutez une case activée runtime pour détecter si la valeur en cours de conversion est supérieure ou égale à zéro et inférieure ou égale à la valeur maximale du type signé. Les valeurs de cette plage sont transférées de signées à non signées ou de non signées à signées sans être réinterprétées.

Conversions de pointeurs

Dans de nombreuses expressions, un tableau de style C est implicitement converti en pointeur vers le premier élément du tableau, et les conversions constantes peuvent se produire en mode silencieux. Bien que cela soit pratique, il est également potentiellement sujet aux erreurs. Par exemple, l'exemple de code mal conçu suivant semble absurde, et pourtant, il sera compilé et produit un résultat de 'p'. Tout d'abord, le littéral de la constante de chaîne « Help » est converti en un `char*` qui pointe vers le premier élément du tableau ; ce pointeur est ensuite incrémenté de trois éléments afin qu'il pointe maintenant vers le dernier élément 'p'.

C++

```
char* s = "Help" + 3;
```

Conversions explicites (casts)

À l'aide d'une opération de cast, vous pouvez demander au compilateur de convertir une valeur d'un type en un autre. Le compilateur génère une erreur dans certains cas si les deux types ne sont pas liés, mais dans d'autres cas, il ne génère pas d'erreur même si l'opération n'est pas de type sécurisé. Utilisez les casts avec parcimonie, car toute conversion d'un type à un autre est une source potentielle d'erreur de programme. Cependant, les casts sont parfois nécessaires, et tous les casts ne sont pas aussi dangereux. Une utilisation efficace d'une cast est lorsque votre code effectue une conversion de rétrécissement et que vous savez que la conversion n'entraîne pas de résultats incorrects de votre programme. En effet, cela indique au compilateur que vous savez ce que vous faites et qu'il doit cesser de vous déranger avec des avertissements à ce sujet. Une autre utilisation consiste à convertir d'une classe pointeur vers une classe

dérivée en classe pointeur vers base. Une autre utilisation consiste à écarter la constité d'une variable pour la passer à une fonction qui nécessite un argument non const. La plupart de ces opérations de cast impliquent un certain risque.

Dans la programmation de style C, le même opérateur de cast de style C est utilisé pour tous les types de casts.

```
C++
```

```
(int) x; // old-style cast, old-style syntax
int(x); // old-style cast, functional syntax
```

L'opérateur de cast de style C est identique à l'opérateur d'appel () et est donc peu visible dans le code et facile à ignorer. Les deux sont mauvais, car ils sont difficiles à reconnaître en un coup d'œil ou à rechercher, et ils sont suffisamment disparates pour appeler n'importe quelle combinaison de `static`, `const` et `reinterpret_cast`. Déterminer ce que fait réellement un cast de style ancien peut être difficile et sujet aux erreurs. Pour toutes ces raisons, lorsqu'un cast est nécessaire, nous vous recommandons d'utiliser l'un des opérateurs de cast C++ suivants, qui, dans certains cas, sont beaucoup plus sûrs pour le type et qui expriment beaucoup plus explicitement l'intention de programmation :

- `static_cast`, pour les casts qui sont vérifiés au moment de la compilation uniquement. `static_cast` retourne une erreur si le compilateur détecte que vous essayez de convertir entre des types totalement incompatibles. Vous pouvez également l'utiliser pour effectuer un cast entre pointeur à base et pointeur vers dérivé, mais le compilateur ne peut pas toujours déterminer si ces conversions seront sécurisées au moment de l'exécution.

  ```
  C++
  ```

  ```
  double d = 1.58947;
  int i = d;  // warning C4244 possible loss of data
  int j = static_cast<int>(d);       // No warning.
  string s = static_cast<string>(d); // Error C2440:cannot convert from
                                     // double to std:string

  // No error but not necessarily safe.
  Base* b = new Base();
  Derived* d2 = static_cast<Derived*>(b);
  ```

  Pour plus d'informations, consultez static_cast.

- `dynamic_cast`, pour les casts sécurisés et vérifiés au moment de l'exécution de pointeur à base en pointeur vers dérivé. Un `dynamic_cast` est plus sûr que pour `static_cast` les rétrodiffusions, mais le runtime case activée entraîne une certaine surcharge.

  ```
  C++

  Base* b = new Base();

  // Run-time check to determine whether b is actually a Derived*
  Derived* d3 = dynamic_cast<Derived*>(b);

  // If b was originally a Derived*, then d3 is a valid pointer.
  if(d3)
  {
     // Safe to call Derived method.
     cout << d3->DoSomethingMore() << endl;
  }
  else
  {
     // Run-time check failed.
     cout << "d3 is null" << endl;
  }

  //Output: d3 is null;
  ```

  Pour plus d'informations, consultez dynamic_cast.

- `const_cast`, pour la conversion de la `const` valeur -ness d'une variable ou la conversion d'une variable non-en `const` `const`. La conversion de `const`-ness à l'aide de cet opérateur est tout aussi sujette aux erreurs que l'utilisation d'un cast de style C, sauf qu'avec `const_cast` vous êtes moins susceptible d'effectuer la cast accidentellement. Parfois, vous devez caster la `const` valeur -ness d'une variable, par exemple, pour passer une `const` variable à une fonction qui prend un paramètre non-`const` . L'exemple suivant vous montre comment procéder.

  ```
  C++

  void Func(double& d) { ... }
  void ConstCast()
  {
     const double pi = 3.14;
     Func(const_cast<double&>(pi)); //No error.
  }
  ```

  Pour plus d'informations, consultez const_cast.

- `reinterpret_cast`, pour les casts entre des types non liés tels qu'un type de pointeur et un `int`.

  > ⓘ Notes
  >
  > Cet opérateur de cast n'est pas utilisé aussi souvent que les autres, et il n'est pas garanti qu'il soit portable pour d'autres compilateurs.

  L'exemple suivant montre comment `reinterpret_cast` diffère de `static_cast`.

  C++

```
const char* str = "hello";
int i = static_cast<int>(str);//error C2440: 'static_cast' : cannot
                              // convert from 'const char *' to 'int'
int j = (int)str; // C-style cast. Did the programmer really intend
                  // to do this?
int k = reinterpret_cast<int>(str);// Programming intent is clear.
                                   // However, it is not 64-bit safe.
```

  Pour plus d'informations, consultez reinterpret_cast Opérateur.

Voir aussi

Système de type C++
Bienvenue dans C++
Informations de référence sur le langage C++
Bibliothèque C++ standard

Conversions standard

Article • 17/06/2023

Le langage C++ définit les conversions entre ses types fondamentaux. Il définit également les conversions pour les types dérivés de pointeur, de référence et de pointeur vers membre. Ces conversions sont appelées *conversions standard*.

Cette section décrit les conversions standard suivantes :

- Promotions intégrales
- Conversions intégrales
- Conversions flottantes
- Conversions flottantes et intégrales
- Conversions arithmétiques
- Conversions de pointeurs
- Conversions de références
- Conversions de pointeur vers membre

> ⓘ **Notes**
>
> Les types définis par l'utilisateur peuvent spécifier leurs propres conversions. La conversion de types définis par l'utilisateur est traitée dans **Constructeurs et conversions**.

Le code suivant provoque des conversions (dans cet exemple, promotions intégrales) :

C++

```
long  long_num1, long_num2;
int   int_num;

// int_num promoted to type long prior to assignment.
long_num1 = int_num;

// int_num promoted to type long prior to multiplication.
long_num2 = int_num * long_num2;
```

Le résultat d'une conversion est une l-value uniquement si elle produit un type référence. Par exemple, une conversion définie par l'utilisateur déclarée comme `operator int&()` retourne une référence et est une valeur l. Toutefois, une conversion déclarée comme `operator int()` renvoie un objet et n'est pas une valeur l.

Promotions intégrales

Les objets d'un type intégral peuvent être convertis en un autre type intégral plus large, c'est-à-dire un type qui peut représenter un ensemble plus grand de valeurs. Ce type de conversion étendue est appelé *promotion intégrale*. Avec la promotion intégrale, vous pouvez utiliser les types suivants dans une expression partout où un autre type intégral peut être utilisé :

- Objets, littéraux et constantes de type **`char`** et **`short int`**
- Types d'énumération
- **`int`** champs bit
- Énumérateurs

Les promotions C++ sont « préservant la valeur », car la valeur après la promotion est garantie identique à la valeur avant la promotion. Dans les promotions de préservation de la valeur, les objets de types intégraux plus courts (tels que les champs de bits ou les objets de type **`char`**) sont promus en type **`int`** si **`int`** peut représenter la plage complète du type d'origine. Si **`int`** ne peut pas représenter la plage complète de valeurs, l'objet est promu en type **`unsigned int`**. Bien que cette stratégie soit identique à celle utilisée par Standard C, les conversions de conservation de valeur ne préservent pas la « signature » de l'objet.

Les promotions de type conservation-valeur et les promotions qui conservent l'entier non signé produisent normalement les mêmes résultats. Toutefois, ils peuvent produire des résultats différents si l'objet promu apparaît comme suit :

- Opérande de `/`, `%`, `/=`, `%=`, `<`, `<=`, `>`, ou `>=`

  Ces opérateurs se basent sur un signe pour déterminer le résultat. Les promotions de conservation de valeur et de conservation des signes produisent des résultats différents lorsqu'elles sont appliquées à ces opérandes.

- Opérande gauche de `>>` ou `>>=`

Ces opérateurs traitent différemment les quantités signées et non signées dans une opération de décalage. Pour les quantités signées, une opération de décalage vers la droite propage le bit de signe dans les positions de bits libérées, tandis que les positions de bits libérées sont remplies à zéro dans les quantités non signées.

- Argument d'une fonction surchargée, ou de l'opérande d'un opérateur surchargé, qui dépend de la signature du type d'opérande pour la correspondance d'argument. Pour plus d'informations sur la définition d'opérateurs surchargés, consultez Opérateurs surchargés.

Conversions intégrales

Les conversions intégrales sont des conversions entre des types intégraux. Les types intégraux sont , (ou `short int`), `int`, `long`, et `long long`. `short` `char` Ces types peuvent être qualifiés avec `signed` ou , et `unsigned` peuvent être utilisés comme abréviation pour `unsigned int` `unsigned`.

Type signé en type non signé

Les objets de types intégraux signés peuvent être convertis en types non signés correspondants. Lorsque ces conversions se produisent, le modèle de bits réel ne change pas. Toutefois, l'interprétation des données change. Prenons le code suivant :

C++

```
#include <iostream>

using namespace std;
int main()
{
    short  i = -3;
    unsigned short u;

    cout << (u = i) << "\n";
}
// Output: 65533
```

Dans l'exemple précédent, un `signed short`, `i` est défini et initialisé à un nombre négatif. L'expression `(u = i)` provoque `i` la conversion en avant `unsigned short` l'affectation en `u`.

Type non signé en type signé

Les objets de types intégraux non signés peuvent être convertis en types signés correspondants. Toutefois, si la valeur non signée se trouve en dehors de la plage représentable du type signé, le résultat n'aura pas la valeur correcte, comme illustré dans l'exemple suivant :

C++

```
#include <iostream>

using namespace std;
int main()
{
short  i;
unsigned short u = 65533;

cout << (i = u) << "\n";
}
//Output: -3
```

Dans l'exemple précédent, `u` est un **`unsigned short`** objet intégral qui doit être converti en quantité signée pour évaluer l'expression `(i = u)`. Étant donné que sa valeur ne peut pas être correctement représentée dans un **`signed short`**, les données sont mal interprétées comme indiqué.

Conversions de nombres à virgule flottante

Un objet d'un type flottant peut être converti en toute sécurité en un type flottant plus précis, c'est-à-dire que la conversion n'entraîne aucune perte de signification. Par exemple, les conversions de **`float`** vers **`double`** ou de **`double`** vers **`long double`** sont sécurisées, et la valeur est inchangée.

Un objet d'un type flottant peut également être converti en un type moins précis, s'il se trouve dans une plage pouvant être représentée par ce type. (Consultez Limites flottantes pour connaître les plages de types flottants.) Si la valeur d'origine n'est pas représentable précisément, elle peut être convertie en valeur représentable supérieure ou inférieure suivante. Le résultat n'est pas défini si cette valeur n'existe pas. Prenons l'exemple suivant :

C++

```
cout << (float)1E300 << endl;
```

La valeur maximale pouvant être représentée par type `float` est 3,402823466E38, ce qui est un nombre beaucoup plus petit que 1E300. Par conséquent, le nombre est converti en infini et le résultat est « inf ».

Conversions entre types intégraux et à virgule flottante

Certaines expressions peuvent provoquer la conversion des objets de type flottant en types intégraux, ou vice versa. Lorsqu'un objet de type intégral est converti en type flottant et que la valeur d'origine n'est pas représentable exactement, le résultat est la valeur suivante plus élevée ou la valeur représentable inférieure suivante.

Lorsqu'un objet de type flottant est converti en type intégral, la partie fractionnaire est *tronquée ou arrondie* à zéro. Un nombre comme 1.3 est converti en 1 et -1.3 est converti en -1. Si la valeur tronquée est supérieure à la valeur représentable la plus élevée ou inférieure à la valeur représentable la plus faible, le résultat n'est pas défini.

Conversions arithmétiques

De nombreux opérateurs binaires (décrits dans Expressions avec des opérateurs binaires) provoquent des conversions d'opérandes et produisent des résultats de la même façon. Les conversions que ces opérateurs provoquent sont appelées *conversions arithmétiques habituelles*. Les conversions arithmétiques d'opérandes ayant différents types natifs sont effectuées comme indiqué dans le tableau suivant. Les types typedef se comportent selon leurs types natifs sous-jacents.

Conditions pour la conversion de type

Conditions remplies	Conversion
L'un ou l'autre opérande est de type `long double`.	Un autre opérande est converti en type `long double`.
Condition précédente non remplie et l'un des opérandes est de type `double`.	Un autre opérande est converti en type `double`.
Les conditions précédentes ne sont pas remplies et l'un des opérandes est de type `float`.	Un autre opérande est converti en type `float`.

Conditions remplies	Conversion
Conditions précédentes non remplies (aucun des opérandes n'est de type flottant).	Les opérandes bénéficient de promotions intégrales comme suit : - Si l'un des opérandes est de type `unsigned long`, l'autre opérande est converti en type `unsigned long`. - Si la condition précédente n'est pas remplie, et si l'un des opérandes est de type `long` et l'autre de type `unsigned int`, les deux opérandes sont convertis en type `unsigned long`. - Si les deux conditions précédentes ne sont pas remplies et si l'un des opérandes est de type `long`, l'autre opérande est converti en type `long`. - Si les trois conditions précédentes ne sont pas remplies et si l'un des opérandes est de type `unsigned int`, l'autre opérande est converti en type `unsigned int`. - Si aucune des conditions précédentes n'est remplie, les deux opérandes sont convertis en type `int`.

Le code suivant illustre les règles de conversion décrites dans le tableau :

C++

```
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main() {
   // iVal converted to unsigned long
   // result of multiplication converted to double
   dVal = iVal * ulVal;

   // ulVal converted to float
   // result of addition converted to double
   dVal = ulVal + fVal;
}
```

Dans l'exemple précédent, la première instruction affiche la multiplication de deux types intégraux, `iVal` et `ulVal`. La condition remplie est qu'aucun opérande n'est de type flottant et qu'un opérande est de type `unsigned int`. Ainsi, l'autre opérande, `iVal`, est converti en type `unsigned int`. Le résultat est ensuite affecté à `dVal`. La condition remplie ici est qu'un opérande est de type `double`, de sorte que le `unsigned int` résultat de la multiplication est converti en type `double`.

La deuxième instruction de l'exemple précédent montre l'ajout d'un `float` et d'un type intégral : `fVal` et `ulVal`. La `ulVal` variable est convertie en type `float` (troisième

condition dans la table). Le résultat de l'ajout est converti en type `double` (deuxième condition dans la table) et affecté à `dVal`.

Conversions de pointeurs

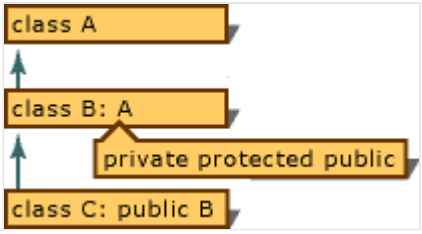
Les pointeurs peuvent être convertis durant l'assignation, l'initialisation, la comparaison et d'autres expressions.

Pointeur vers classes

Il existe deux cas dans lesquels un pointeur vers une classe peut être converti en un pointeur vers une classe de base.

Dans le premier cas, la classe de base spécifiée est accessible et la conversion est n'est pas ambiguë. Pour plus d'informations sur les références ambiguës de classe de base, consultez Plusieurs classes de base.

L'accessibilité d'une classe de base dépend du type d'héritage utilisé dans la dérivation. Considérez l'héritage illustré dans la figure suivante :



Graphique d'héritage illustrant l'accessibilité de la classe de base

Le tableau suivant indique l'accessibilité de la classe de base d'après la situation illustrée.

Type de fonction	**Dérivation**	**Conversion de `B*` à légal `A*` ?**
Fonction externe (hors portée de classe)	Privées	Non
	Protected	Non
	Public	Yes
Fonction membre B (dans la portée B)	Privées	Oui
	Protected	Yes

Type de fonction	Dérivation	Conversion de `B*` à légal `A*` ?
	Public	Yes
Fonction membre C (dans la portée C)	Privées	Non
	Protected	Yes
	Public	Yes

Dans le second cas, un pointeur vers une classe peut être converti en pointeur vers une classe de base lorsque vous utilisez une conversion de type explicite. Pour plus d'informations sur les conversions de type explicites, consultez Opérateur de conversion de type explicite.

Le résultat d'une telle conversion est un pointeur vers le *sous-objet*, la partie de l'objet qui est entièrement décrite par la classe de base.

Le code suivant définit deux classes, `A` et `B`, où `B` est dérivée de `A`. (Pour plus d'informations sur l'héritage, consultez Classes dérivées.) Il définit `bObject`ensuite , un objet de type `B`, et deux pointeurs (`pA` et `pB`) qui pointent vers l'objet .

C++

```
// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};
int main()
{
   B bObject;
   A *pA = &bObject;
   B *pB = &bObject;

   pA->AMemberFunc();   // OK in class A
   pB->AMemberFunc();   // OK: inherited from class A
   pA->BMemberFunc();   // Error: not in class A
}
```

Le pointeur `pA` est de type `A *`, ce qui peut être interprété comme signifiant « pointeur vers un objet de type `A` ». Les membres de `bObject` (tels que `BComponent` et `BMemberFunc`) sont propres au type `B` et sont donc inaccessibles via `pA`. Le pointeur `pA` autorise l'accès uniquement aux caractéristiques (fonctions membres et données) de l'objet définies dans la classe `A`.

Pointeur vers fonction

Un pointeur vers une fonction peut être converti en type `void *`, si le type `void *` est suffisamment grand pour contenir ce pointeur.

Pointeur vers void

Les pointeurs vers le type **`void`** peuvent être convertis en pointeurs vers n'importe quel autre type, mais uniquement avec un cast de type explicite (contrairement à C). Un pointeur vers n'importe quel type peut être converti implicitement en pointeur vers le type **`void`**. Un pointeur vers un objet incomplet d'un type peut être converti en pointeur vers **`void`** (implicitement) et retour (explicitement). Le résultat de ce type de conversion est égal à la valeur du pointeur d'origine. Un objet est considéré comme incomplet s'il est déclaré, mais les informations disponibles sont insuffisantes pour déterminer sa taille ou sa classe de base.

Pointeur vers n'importe quel objet qui n'est pas **`const`** ou **`volatile`** peut être converti implicitement en pointeur de type `void *`.

Pointeurs const et volatile

C++ ne fournit pas de conversion standard d'un **`const`** type ou **`volatile`** en un type qui n'est pas **`const`** ou **`volatile`**. Cependant, toute sorte de conversion peut être spécifiée à l'aide de casts de type explicite (y compris les conversions qui ne sont pas sécurisées).

> ⓘ **Notes**
>
> Les pointeurs C++ vers les membres, à l'exception des pointeurs vers les membres statiques, sont différents des pointeurs normaux et n'ont pas les mêmes conversions standard. Les pointeurs vers des membres statiques sont des pointeurs normaux et ils ont les mêmes conversions que les pointeurs normaux.

Conversions de pointeurs null

Une expression constante intégrale qui est évaluée à zéro, ou une telle expression cast en type pointeur, est convertie en pointeur appelé *pointeur null*. Ce pointeur est toujours comparé à un pointeur vers n'importe quel objet ou fonction valide. Une exception est les pointeurs vers des objets basés, qui peuvent avoir le même décalage et toujours pointer vers des objets différents.

En C++11, le type nullptr doit être préféré au pointeur null de style C.

Conversions d'expression de pointeur

Toute expression avec un type de tableau peut être convertie en un pointeur du même type. Le résultat de la conversion est un pointeur vers le premier élément du tableau. L'exemple suivant illustre cette conversion :

C++

```
char szPath[_MAX_PATH]; // Array of type char.
char *pszPath = szPath; // Equals &szPath[0].
```

Une expression qui entraîne une fonction qui retourne un type particulier est convertie en un pointeur vers une fonction qui retourne ce type, sauf lorsque :

- L'expression est utilisée comme opérande de l'opérateur d'adresse (**&**).
- L'expression est utilisée comme opérande de l'opérateur d'appel de fonction.

Conversions de références

Une référence à une classe peut être convertie en référence à une classe de base dans les cas suivants :

- La classe de base spécifiée est accessible.
- La conversion n'est pas ambiguë. (Pour plus d'informations sur les références ambiguës de classe de base, consultez Plusieurs classes de base.)

Le résultat de la conversion est un pointeur vers le sous-objet qui représente la classe de base.

Pointeur vers membre

Les pointeurs vers des membres de classe peuvent être convertis durant l'assignation, l'initialisation, la comparaison et d'autres expressions. Cette section décrit les conversions de pointeur vers membre suivantes :

Pointeur vers membre de classe de base

Un pointeur vers un membre d'une classe de base peut être converti en un pointeur vers un membre d'une classe dérivée de cette classe lorsque les conditions suivantes sont remplies :

- La conversion inverse, du pointeur vers la classe dérivée vers le pointeur de classe de base, est accessible.
- La classe dérivée n'hérite pas virtuellement de la classe de base.

Lorsque l'opérande gauche est un pointeur vers membre, l'opérande droite doit être de type pointeur vers membre ou une expression constante qui correspond à 0. Cette assignation est valide uniquement dans les cas suivants :

- L'opérande droite est un pointeur vers un membre de la même classe que l'opérande gauche.
- L'opérande gauche est un pointeur vers un membre d'une classe dérivée de façon publique et non ambiguë de la classe de l'opérande droite.

Conversion du pointeur null vers les membres

Une expression constante intégrale qui est évaluée à zéro est convertie en pointeur null. Ce pointeur est toujours comparé à un pointeur vers n'importe quel objet ou fonction valide. Une exception est les pointeurs vers des objets basés, qui peuvent avoir le même décalage et toujours pointer vers des objets différents.

Le code suivant illustre la définition d'un pointeur désignant le membre `i` dans la classe `A`. Le pointeur, `pai`, est initialisé à 0, qui est le pointeur null.

C++

```
class A
{
public:
int i;
};

int A::*pai = 0;
```

```
int main()
{
}
```

Voir aussi

Informations de référence sur le langage C++

Types intégrés (C++)

Article • 03/04/2023

Les types intégrés (également appelés *types fondamentaux*) sont spécifiés par la norme de langage C++ et sont intégrés au compilateur. Les types intégrés ne sont définis dans aucun fichier d'en-tête. Les types intégrés sont divisés en trois catégories main : *intégrale*, *à virgule flottante* et *vide*. Les types intégraux représentent des nombres entiers. Les types à virgule flottante peuvent spécifier des valeurs qui peuvent avoir des parties fractionnaires. La plupart des types intégrés sont traités comme des types distincts par le compilateur. Toutefois, certains types sont *des synonymes* ou sont traités comme des types équivalents par le compilateur.

Type void

Le void type décrit un ensemble vide de valeurs. Aucune variable de type `void` ne peut être spécifiée. Le `void` type est principalement utilisé pour déclarer des fonctions qui ne retournent aucune valeur ou pour déclarer des pointeurs génériques vers des données non typées ou arbitrairement typées. Toute expression peut être convertie explicitement ou castée en type `void`. Toutefois, ces expressions sont limitées aux utilisations suivantes :

- Instruction d'expression. (Pour plus d'informations, consultez Expressions.)
- Opérande gauche de l'opérateur virgule. (Pour plus d'informations, consultez Opérateur virgule.)
- Deuxième ou troisième opérande de l'opérateur conditionnel (`? :`). (Pour plus d'informations, consultez Expressions avec l'opérateur conditionnel.)

std::nullptr_t

Le mot clé `nullptr` est une constante de pointeur Null de type `std::nullptr_t`, qui est convertible en n'importe quel type de pointeur brut. Pour plus d'informations, consultez nullptr.

Type booléen

Le bool type peut avoir des valeurs true et false. La taille du type est spécifique à l'implémentation `bool` . Pour plus d'informations sur l'implémentation spécifique à

Microsoft, consultez Tailles des types intégrés .

Types de caractères

Le `char` type est un type de représentation de caractères qui encode efficacement les membres du jeu de caractères d'exécution de base. Le compilateur C++ traite les variables de type `char`, `signed char` et `unsigned char` comme ayant différents types.

Spécifique à Microsoft : les variables de type `char` sont promues `int` comme si de type `signed char` par défaut, sauf si l'option /J de compilation est utilisée. Dans ce cas, ils sont traités en tant que type `unsigned char` et sont promus `int` en sans extension de signe.

Une variable de type `wchar_t` est un type de caractère large ou multioctet. Utilisez le `L` préfixe avant un littéral de caractère ou de chaîne pour spécifier le type de caractères larges.

Spécifique à Microsoft : par défaut, `wchar_t` est un type natif, mais vous pouvez utiliser /Zc:wchar_t- pour créer `wchar_t` un typedef pour `unsigned short`. Le type `__wchar_t` est un synonyme spécifique à Microsoft pour le type natif `wchar_t` .

Le `char8_t` type est utilisé pour la représentation de caractères UTF-8. Il a la même représentation que `unsigned char`, mais est traité comme un type distinct par le compilateur. Le `char8_t` type est nouveau dans C++20. **Spécifique à Microsoft** : l'utilisation de `char8_t` nécessite l'option du /std:c++20 compilateur ou une version ultérieure (par `/std:c++latest` exemple).

Le `char16_t` type est utilisé pour la représentation de caractères UTF-16. Elle doit être suffisamment grande pour représenter toute unité de code UTF-16. Il est traité comme un type distinct par le compilateur.

Le `char32_t` type est utilisé pour la représentation de caractères UTF-32. Elle doit être suffisamment grande pour représenter toute unité de code UTF-32. Il est traité comme un type distinct par le compilateur.

Types virgule flottante

Les types à virgule flottante utilisent une représentation IEEE-754 pour fournir une approximation des valeurs fractionnaires sur une large plage de grandeurs. Le tableau suivant répertorie les types à virgule flottante en C++ et les restrictions comparatives sur les tailles de type à virgule flottante. Ces restrictions sont imposées par la norme

C++ et sont indépendantes de l'implémentation Microsoft. La taille absolue des types à virgule flottante intégrée n'est pas spécifiée dans la norme.

Type	Contenu
`float`	Type `float` est le plus petit type à virgule flottante en C++.
`double`	Le type `double` est un type virgule flottante supérieur ou égal au type `float`, mais inférieur ou égal à la taille du type `long double`.
`long double`	Le type `long double` est un type virgule flottante supérieur ou égal au type `double`.

Spécifique à Microsoft : la représentation de `long double` et `double` est identique. Toutefois, `long double` et `double` sont traités comme des types distincts par le compilateur. Le compilateur Microsoft C++ utilise les représentations à virgule flottante IEEE-754 de 4 et 8 octets. Pour plus d'informations, consultez Représentation à virgule flottante IEEE.

Types d'entier

Le `int` type est le type entier de base par défaut. Il peut représenter tous les nombres entiers sur une plage spécifique à l'implémentation.

Une représentation entière *signée* est une représentation qui peut contenir des valeurs positives et négatives. Il est utilisé par défaut ou lorsque le `signed` modificateur mot clé est présent. Le `unsigned` modificateur mot clé spécifie une représentation *non signée* qui ne peut contenir que des valeurs non négatives.

Un modificateur de taille spécifie la largeur en bits de la représentation entière utilisée. La langue prend en charge `short` les modificateurs , `long` et `long long` . Un `short` type doit avoir au moins 16 bits de large. Un `long` type doit avoir au moins 32 bits de large. Un `long long` type doit avoir au moins 64 bits de large. La norme spécifie une relation de taille entre les types intégraux :

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

Une implémentation doit conserver les exigences de taille minimale et la relation de taille pour chaque type. Toutefois, les tailles réelles peuvent varier d'une implémentation à l'autre. Pour plus d'informations sur l'implémentation spécifique à Microsoft, consultez Tailles des types intégrés .

Le `int` mot clé peut être omis lorsque `signed` des modificateurs de taille , `unsigned` ou sont spécifiés. Les modificateurs et `int` le type, s'ils sont présents, peuvent apparaître dans n'importe quel ordre. Par exemple, `short unsigned` et `unsigned int short` font référence au même type.

Synonymes de type entier

Les groupes de types suivants sont considérés comme des synonymes par le compilateur :

- `short`, `short int`, `signed short`, `signed short int`
- `unsigned short`, `unsigned short int`
- `int`, `signed`, `signed int`
- `unsigned`, `unsigned int`
- `long`, `long int`, `signed long`, `signed long int`
- `unsigned long`, `unsigned long int`
- `long long`, `long long int`, `signed long long`, `signed long long int`
- `unsigned long long`, `unsigned long long int`

Les types **entiers spécifiques à Microsoft** incluent les types de largeur `__int8` spécifique , `__int16`, `__int32` et `__int64` . Ces types peuvent utiliser les `signed` modificateurs et .`unsigned` Le type de données `__int8` est synonyme du type `char`, le type `__int16` est synonyme du type `short`, le type `__int32` est synonyme du type `int`, et le type `__int64` est synonyme du type `long long`.

Tailles des types intégrés

La plupart des types intégrés ont des tailles définies par l'implémentation. Le tableau suivant répertorie la quantité de stockage requise pour les types intégrés dans Microsoft C++. En particulier, `long` est de 4 octets même sur les systèmes d'exploitation 64 bits.

Type	Taille
`bool`, `char`, `char8_t`, `unsigned char`, `signed char`, `__int8`	1 octet
`char16_t`, `__int16`, `short`, `unsigned short`, `wchar_t`, `__wchar_t`	2 octets

Type	Taille
`char32_t`, `float`, `__int32`, `int`, `unsigned int`, `long`, `unsigned long`	4 octets
`double`, `__int64`, `long double`, `long long`, `unsigned long long`	8 octets

Pour obtenir un résumé de la plage de valeurs de chaque type, consultez Plages de types de données.

Pour plus d'informations sur la conversion de type, consultez Conversions standard.

Voir aussi

Plages de types de données

plages de types de données

Article • 03/04/2023

Les compilateurs Microsoft C++ 32 bits et 64 bits reconnaissent les types dans la table plus loin dans cet article.

- `int` (`unsigned int`)
- `__int8` (`unsigned __int8`)
- `__int16` (`unsigned __int16`)
- `__int32` (`unsigned __int32`)
- `__int64` (`unsigned __int64`)
- `short` (`unsigned short`)
- `long` (`unsigned long`)
- `long long` (`unsigned long long`)

Si son nom commence par deux traits de soulignement (`__`), un type de données est non standard.

Les plages spécifiées dans le tableau ci-dessous sont inclusives-inclusives.

Nom de type	Octets	Autres noms	Plage de valeurs
`int`	4	`signed`	-2 147 483 648 à 2 147 483 647
`unsigned int`	4	`unsigned`	de 0 à 4 294 967 295
`__int8`	1	`char`	-128 à 127
`unsigned __int8`	1	`unsigned char`	0 à 255
`__int16`	2	`short`, `short int`, `signed short int`	-32 768 à 32 767
`unsigned __int16`	2	`unsigned short`, `unsigned short int`	0 à 65 535
`__int32`	4	`signed`, `signed int`, `int`	-2 147 483 648 à 2 147 483 647

Nom de type	Octets	Autres noms	Plage de valeurs
`unsigned __int32`	4	`unsigned`, `unsigned int`	de 0 à 4 294 967 295
`__int64`	8	`long long`, `signed long long`	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
`unsigned __int64`	8	`unsigned long long`	de 0 à 18 446 744 073 709 551 615
`bool`	1	aucun	`false` ou `true`
`char`	1	aucun	-128 à 127 par défaut 0 à 255 en cas de compilation à l'aide de /J
`signed char`	1	aucun	-128 à 127
`unsigned char`	1	aucun	0 à 255
`short`	2	`short int`, `signed short int`	-32 768 à 32 767
`unsigned short`	2	`unsigned short int`	0 à 65 535
`long`	4	`long int`, `signed long int`	-2 147 483 648 à 2 147 483 647
`unsigned long`	4	`unsigned long int`	de 0 à 4 294 967 295
`long long`	8	none (mais équivalent à `__int64`)	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
`unsigned long long`	8	none (mais équivalent à `unsigned __int64`)	de 0 à 18 446 744 073 709 551 615
`enum`	varie	aucun	
`float`	4	aucun	3.4E +/- 38 (7 chiffres)
`double`	8	aucun	1.7E +/- 308 (15 chiffres)
`long double`	identique à `double`	aucun	Identique à `double`
`wchar_t`	2	`__wchar_t`	0 à 65 535

Selon la façon dont elle est utilisée, une variable de `__wchar_t` désigne un type caractères larges ou un type caractères multioctets. Utilisez le préfixe `L` avant une constante caractère ou chaîne pour désigner la constante de type caractères larges.

`signed` et `unsigned` sont des modificateurs que vous pouvez utiliser avec tout type d'entier sauf `bool`. Notez que `char`, `signed char` et `unsigned char` sont trois types distincts qui servent aux mécanismes tels que la surcharge et les modèles.

Les types `int` et `unsigned int` ont une taille de quatre octets. Toutefois, le code portable ne doit pas dépendre de la taille d' `int` , car la norme du langage permet que cela soit propre à l'implémentation.

C/C++ dans Visual Studio prend également en charge les types entier dimensionnés. Pour plus d'informations, consultez __int8, __int16, __int32, __int64Limites d'entiers.

Pour plus d'informations sur les restrictions des tailles de chaque type, consultez Types intégrés.

La plage de types énumérés varie selon le contexte du langage et les indicateurs spécifiés du compilateur. Pour plus d'informations, consultez Déclarations d'énumération C et Énumérations.

Voir aussi

Mots clés
Types intégrés

nullptr

Article • 16/06/2023

Le `nullptr` mot clé spécifie une constante de pointeur null de type `std::nullptr_t`, qui est convertible en n'importe quel type de pointeur brut. Bien que vous puissiez utiliser le mot clé `nullptr` sans inclure d'en-têtes, si votre code utilise le type `std::nullptr_t`, vous devez le définir en incluant l'en-tête `<cstddef>`.

> ⓘ Notes
>
> Le `nullptr` mot clé est également défini en C++/CLI pour les applications de code managé et n'est pas interchangeable avec le mot clé ISO Standard C++. Si votre code peut être compilé à l'aide de l'option **/clr** du compilateur, qui cible le code managé, utilisez `__nullptr` dans n'importe quelle ligne de code où vous devez garantir que le compilateur utilise l'interprétation C++ native. Pour plus d'informations, consultez **nullptr (C++/CLI et C++/CX).**

Notes

Évitez d'utiliser `NULL` ou zéro (`0`) comme constante de pointeur null ; `nullptr` est moins vulnérable aux mauvaises utilisations et fonctionne mieux dans la plupart des situations. Par exemple, étant donné `func(std::pair<const char *, double>)`, l'appel à `func(std::make_pair(NULL, 3.14))` provoque une erreur du compilateur. La macro `NULL` se développe sur `0`, de sorte que l'appel `std::make_pair(0, 3.14)` retourne `std::pair<int, double>`, qui n'est pas convertible en type de `std::pair<const char *, double>` paramètre dans `func`. L'appel à `func(std::make_pair(nullptr, 3.14))` entraîne une compilation correcte, car `std::make_pair(nullptr, 3.14)` retourne `std::pair<std::nullptr_t, double>`, qui peut être convertie en `std::pair<const char *, double>`.

Voir aussi

Mots clés
nullptr (C++/CLI et C++/CX)

void (C++)

Article • 16/06/2023

Lorsqu'elle est utilisée comme type de retour de fonction, la `void` mot clé spécifie que la fonction ne retourne pas de valeur. Lorsqu'elle est utilisée pour la liste de paramètres d'une fonction, `void` spécifie que la fonction ne prend aucun paramètre. Lorsqu'il est utilisé dans la déclaration d'un pointeur, `void` spécifie que le pointeur est « universel ».

Si le type d'un pointeur est `void*`, le pointeur peut pointer vers n'importe quelle variable qui n'est pas déclarée avec ou `const` `volatile` mot clé. Un `void*` pointeur ne peut pas être déréférencé, sauf s'il est converti en un autre type. Un `void*` pointeur peut être converti en n'importe quel autre type de pointeur de données.

En C++, un `void` pointeur peut pointer vers une fonction libre (une fonction qui n'est pas membre d'une classe), ou vers une fonction membre statique, mais pas vers une fonction membre non statique.

Vous ne pouvez pas déclarer une variable de type `void`.

Pour une question de style, les instructions C++ Core vous recommandent de ne pas utiliser `void` pour spécifier une liste de paramètres formelle vide. Pour plus d'informations, consultez C++ Core Guidelines NL.25 : Ne pas utiliser void comme type d'argument.

Exemple

C++

```
// void.cpp

void return_nothing()
{
   // A void function can have a return with no argument,
   // or no return statement.
}

void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main()
{
   pv = &i;
   // Cast is optional in C, required in C++
```

```
    pint = (int *)pv;
}
```

Voir aussi

Mots clés

Types intégrés

bool (C++)

Article • 16/06/2023

Ce mot clé est un type intégré. Une variable de ce type peut avoir des valeurs true et false. Les expressions conditionnelles ont le type `bool` et ont donc des valeurs de type `bool`. Par exemple, `i != 0` a `true` maintenant ou `false` en fonction de la valeur de `i`.

Visual Studio 2017 version 15.3 et ultérieure (disponible avec /std:c++17 et versions ultérieures) : L'opérande d'un opérateur d'incrémentation ou de décrémentation de préfixe ou de postfix peut ne pas être de type `bool`. En d'autres termes, étant donné une variable `b` de type `bool`, ces expressions ne sont plus autorisées :

```
C++

    b++;
    ++b;
    b--;
    --b;
```

Les valeurs `true` et `false` ont la relation suivante :

```
C++

!false == true
!true == false
```

Dans l'instruction suivante :

```
C++

if (condexpr1) statement1;
```

Si `condexpr1` est `true`, `statement1` est toujours exécuté ; si `condexpr1` est `false`, `statement1` n'est jamais exécuté.

Lorsqu'un opérateur postfix ou préfixe `++` est appliqué à une variable de type `bool`, la variable est définie sur `true`.

Visual Studio 2017 version 15.3 et ultérieure : `operator++` pour `bool` a été supprimé du langage et n'est plus pris en charge.

L'opérateur postfix ou préfixe `--` ne peut pas être appliqué à une variable de ce type.

Le `bool` type participe aux promotions intégrales par défaut. Une valeur r de type `bool` peut être convertie en une valeur r de type `int`, devenant `false` zéro et `true` devenant un. En tant que type distinct, `bool` participe à la résolution de surcharge.

Voir aussi

Mots clés
Types intégrés

false (C++)

Article • 16/06/2023

Le mot clé est l'une des deux valeurs d'une variable de type bool ou d'une expression conditionnelle (une expression conditionnelle est désormais une `true` expression booléenne). Par exemple, si `i` est une variable de type `bool`, l'instruction `i = false;` assigne `false` à `i`.

Exemple

C++

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

Voir aussi

Mots clés

true (C++)

Article • 03/04/2023

Syntaxe

```
bool-identifier = true ;
bool-expression logical-operator true ;
```

Notes

Cette mot clé est l'une des deux valeurs d'une variable de type bool ou d'une expression conditionnelle (une expression conditionnelle est désormais une expression booléenne vraie). Si `i` est de type **`bool`**, l'instruction `i = true;` affecte **`true`** à `i`.

Exemple

C++

```
// bool_true.cpp
#include <stdio.h>
int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

Output

```
1
0
```

Voir aussi

Mots clés

char, wchar_t, char8_t, char16_t, char32_t

Article • 16/06/2023

Les types `char`, `wchar_t`, `char8_t`, `char16_t` et `char32_t` sont des types intégrés qui représentent des caractères alphanumériques, des glyphes non alphanumériques et des caractères non imprimables.

Syntaxe

C++

```
char     ch1{ 'a' };  // or { u8'a' }
wchar_t  ch2{ L'a' };
char16_t ch3{ u'a' };
char32_t ch4{ U'a' };
```

Notes

Le `char` type était le type de caractère d'origine en C et C++. Le `char` type peut être utilisé pour stocker des caractères du jeu de caractères ASCII ou de l'un des jeux de caractères ISO-8859, ainsi que des octets individuels de caractères multi-octets tels que Shift-JIS ou l'encodage UTF-8 du jeu de caractères Unicode. Dans le compilateur Microsoft, `char` est un type 8 bits. Il s'agit d'un type distinct de et `signed char` `unsigned char`. Par défaut, les variables de type `char` sont promues comme `int` si à partir du type `signed char` , sauf si l'option du /J compilateur est utilisée. Sous `/J`, ils sont traités comme type `unsigned char` et sont promus vers `int` sans extension de signe.

Le type `unsigned char` est souvent utilisé pour représenter un *octet*, qui n'est pas un type intégré en C++.

Le `wchar_t` type est un type de caractère large défini par l'implémentation. Dans le compilateur Microsoft, il représente un caractère large 16 bits utilisé pour stocker Unicode encodé en UTF-16LE, le type de caractère natif sur les systèmes d'exploitation Windows. Les versions à caractères larges des fonctions de bibliothèque UCRT (Universal C Runtime) utilisent `wchar_t` et ses types de pointeur et de tableau en tant que paramètres et valeurs de retour, comme le font les versions à caractères larges de l'API Windows native.

Les `char8_t` types , `char16_t` et `char32_t` représentent respectivement des caractères larges 8 bits, 16 bits et 32 bits. (`char8_t` est nouveau en C++20 et nécessite l'option du

/std:c++20 compilateur ou `/std:c++latest` .) Unicode encodé en UTF-8 peut être stocké dans le `char8_t` type . Les chaînes de `char8_t` et `char` de type sont appelées chaînes *étroites* , même lorsqu'elles sont utilisées pour encoder des caractères Unicode ou multi-octets. Unicode encodé en UTF-16 peut être stocké dans le `char16_t` type, et Unicode encodé en UTF-32 peut être stocké dans le `char32_t` type . Chaînes de ces types et `wchar_t` sont toutes appelées chaînes *larges* , bien que le terme se réfère souvent spécifiquement aux chaînes de `wchar_t` type.

Dans la bibliothèque standard C++, le `basic_string` type est spécialisé pour les chaînes étroites et larges. Utilisez `std::string` lorsque les caractères sont de type `char`, `std::u8string` lorsque les caractères sont de type `char8_t`, `std::u16string` lorsque les caractères sont de type `char16_t`, `std::u32string` lorsque les caractères sont de type `char32_t`et `std::wstring` lorsque les caractères sont de type `wchar_t`. Autres types qui représentent du texte, y compris `std::stringstream` et `std::cout` ont des spécialisations pour les chaînes étroites et larges.

__int8, __int16, __int32, __int64

Article • 03/04/2023

Spécifique à Microsoft

Prise en charge des fonctionnalités Microsoft C/C++ pour les types d'entiers dimensionnés. Vous pouvez déclarer des variables entières 8, 16, 32 ou 64 bits à l'aide du `__intN` spécificateur de type, où `N` est 8, 16, 32 ou 64.

L'exemple suivant déclare une variable pour chacun de ces types d'entiers dimensionnés :

C++

```
__int8 nSmall;      // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

Les types `__int8`, `__int16` et `__int32` sont synonymes des types ANSI qui ont la même taille et sont utiles pour écrire du code portable qui se comporte de manière identique sur plusieurs plateformes. Le `__int8` type de données est synonyme de type `char`, `__int16` est synonyme de type `short` et `__int32` est synonyme de type `int`. Le `__int64` type est synonyme de type `long long`.

Pour la compatibilité avec les versions précédentes, `_int8`, , `_int16` `_int32` et `_int64` sont synonymes de `__int8`, `__int16`, `__int32`, et `__int64` sauf si l'option /Za du compilateur (Désactiver les extensions de langage) est spécifiée.

Exemple

L'exemple suivant montre qu'un `__intN` paramètre sera promu en `int`:

C++

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
```

```
{
    __int8 i8 = 100;
    func(i8);   // no void func(__int8 i8) function
                // __int8 will be promoted to int
}
```

Output

```
func
```

Voir aussi

Mots clés

Types intégrés

Plages de types de données

__m64

Article • 16/06/2023

Section spécifique à Microsoft

Le `__m64` type de données est destiné à être utilisé avec MMX et 3DNow ! intrinsèques, et est défini dans <xmmintrin.h>.

C++

```
// data_types__m64.cpp
#include <xmmintrin.h>
int main()
{
   __m64 x;
}
```

Notes

Vous ne devez pas accéder directement aux `__m64` champs. Vous pouvez toutefois visualiser ces types dans le débogueur. Une variable de type `__m64` est mappée aux registres MM[0-7].

Les variables de type **_m64** sont automatiquement alignées sur les limites de 8 octets.

Le `__m64` type de données n'est pas pris en charge sur les processeurs x64. Les applications qui utilisent __m64 dans le cadre des intrinsèques MMX doivent être réécrites pour utiliser les intrinsèques SSE et SSE2 équivalentes.

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Types intégrés
Plages de types de données

__m128

Article • 16/06/2023

Section spécifique à Microsoft

Le **type de données __m128** , à utiliser avec les intrinsèques d'instructions Streaming SIMD Extensions et Streaming SIMD Extensions 2, est défini dans <xmmintrin.h>.

C++

```
// data_types__m128.cpp
#include <xmmintrin.h>
int main() {
   __m128 x;
}
```

Notes

Vous ne devez pas accéder directement aux `__m128` champs. Vous pouvez toutefois visualiser ces types dans le débogueur. Une variable de type `__m128` est mappée aux registres XMM[0-7].

Les variables de type `__m128` sont automatiquement alignées sur les limites de 16 octets.

Le `__m128` type de données n'est pas pris en charge sur les processeurs ARM.

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Types intégrés
Plages de types de données

__m128d

Article • 16/06/2023

Section spécifique à Microsoft

Le `__m128d` type de données, à utiliser avec les instructions intrinsèques des extensions SIMD 2 de streaming, est défini dans <emmintrin.h>.

C++

```
// data_types__m128d.cpp
#include <emmintrin.h>
int main() {
   __m128d x;
}
```

Notes

Vous ne devez pas accéder directement aux `__m128d` champs. Vous pouvez toutefois visualiser ces types dans le débogueur. Une variable de type `__m128` est mappée aux registres XMM[0-7].

Les variables de type **_m128d** sont automatiquement alignées sur les limites de 16 octets.

Le `__m128d` type de données n'est pas pris en charge sur les processeurs ARM.

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Types intégrés
Plages de types de données

__m128i

Article • 16/06/2023

Section spécifique à Microsoft

Le `__m128i` type de données, à utiliser avec les instructions SSE2 (Streaming SIMD Extensions 2), est défini dans <emmintrin.h>.

C++

```
// data_types__m128i.cpp
#include <emmintrin.h>
int main() {
   __m128i x;
}
```

Notes

Vous ne devez pas accéder directement aux `__m128i` champs. Vous pouvez toutefois visualiser ces types dans le débogueur. Une variable de type `__m128i` est mappée aux registres XMM[0-7].

Les variables de type `__m128i` sont automatiquement alignées sur les limites de 16 octets.

> ⓘ **Notes**
>
> L'utilisation de variables de type `__m128i` entraîne la génération de l'instruction SSE2 `movdqa` par le compilateur. Cette instruction ne provoque pas d'erreur sur les processeurs Pentium III, mais entraîne une défaillance silencieuse, avec des effets secondaires possibles causés par les instructions `movdqa` qui se traduisent par sur les processeurs Pentium III.

Le `__m128i` type de données n'est pas pris en charge sur les processeurs ARM.

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Types intégrés
Plages de types de données

__ptr32, __ptr64

Article • 16/06/2023

Section spécifique à Microsoft

`__ptr32` représente un pointeur natif sur un système 32 bits, tandis que `__ptr64` représente un pointeur natif sur un système 64 bits.

L'exemple suivant montre comment déclarer chacun de ces types pointeur :

C++

```
int * __ptr32 p32;
int * __ptr64 p64;
```

Sur un système 32 bits, un pointeur déclaré avec `__ptr64` est tronqué en pointeur 32 bits. Sur un système 64 bits, un pointeur déclaré avec `__ptr32` est converti en pointeur 64 bits.

> ⓘ Notes
>
> Vous ne pouvez pas utiliser `__ptr32` ou `__ptr64` lors de la compilation avec **/clr:pure**. Sinon, l'erreur du compilateur C2472 est générée. Les options du compilateur **/clr:pure** et **/clr:safe** sont déconseillées dans Visual Studio 2015 et non prises en charge dans Visual Studio 2017.

Pour la compatibilité avec les versions précédentes, **_ptr32** et **_ptr64** sont des synonymes de `__ptr32` et `__ptr64` , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Exemple

L'exemple suivant montre comment déclarer et allouer des pointeurs avec les `__ptr32` mots clés et `__ptr64` .

C++

```
#include <cstdlib>
#include <iostream>

int main()
{
```

```
    using namespace std;

    int * __ptr32 p32;
    int * __ptr64 p64;

    p32 = (int * __ptr32)malloc(4);
    *p32 = 32;
    cout << *p32 << endl;

    p64 = (int * __ptr64)malloc(4);
    *p64 = 64;
    cout << *p64 << endl;
}
```

Output

```
32
64
```

FIN de la section spécifique à Microsoft

Voir aussi

Types intégrés

Limites numériques (C++)

Article • 03/04/2023

Les deux standard incluent des fichiers, <limits.h> et <float.h>, définissent les limites numériques, ou les valeurs minimales et maximales qu'une variable d'un type donné peut contenir. La portabilité de ces valeurs minimales et maximales est garantie pour tout compilateur C++ qui utilise la même représentation de données que ANSI C. Le <fichier include limits.h> définit les limites numériques pour les types intégraux et <float.h> définit les limites numériques pour les types flottants.

Voir aussi

Concepts de base

Limites d'entier

Article • 03/04/2023

Spécifique à Microsoft

Les limites pour les types d'entiers sont répertoriées dans le tableau ci-dessous. Les macros de préprocesseur pour ces limites sont également définies lorsque vous incluez les limites du fichier <d'en-tête> standard.

Limites appliquées aux constantes entières

Constant	Signification	Valeur
`CHAR_BIT`	Nombre de bits dans la plus petite variable qui n'est pas un champ de bits	8
`SCHAR_MIN`	Valeur minimale d'une variable de type `signed char`	-128
`SCHAR_MAX`	Valeur maximale d'une variable de type `signed char`	127
`UCHAR_MAX`	Valeur maximale d'une variable de type `unsigned char`	255 (0xff)
`CHAR_MIN`	Valeur minimale d'une variable de type `char`	-128; 0 si `/J` l'option utilisée
`CHAR_MAX`	Valeur maximale d'une variable de type `char`	127; 255 si `/J` l'option utilisée
`MB_LEN_MAX`	Nombre maximal d'octets dans une constante à multiples caractères	5
`SHRT_MIN`	Valeur minimale d'une variable de type `short`	-32768
`SHRT_MAX`	Valeur maximale d'une variable de type `short`	32767
`USHRT_MAX`	Valeur maximale d'une variable de type `unsigned short`	65535 (0xffff)
`INT_MIN`	Valeur minimale d'une variable de type `int`	-2147483648
`INT_MAX`	Valeur maximale d'une variable de type `int`	2147483647
`UINT_MAX`	Valeur maximale d'une variable de type `unsigned int`	4294967295 (0xffffffff)
`LONG_MIN`	Valeur minimale d'une variable de type `long`	-2147483648
`LONG_MAX`	Valeur maximale d'une variable de type `long`	2147483647

Constant	Signification	Valeur
`ULONG_MAX`	Valeur maximale d'une variable de type **`unsigned long`**	4294967295 (0xffffffff)
`LLONG_MIN`	Valeur minimale d'une variable de type **`long long`**	-9223372036854775808
`LLONG_MAX`	Valeur maximale d'une variable de type **`long long`**	9223372036854775807
`ULLONG_MAX`	Valeur maximale d'une variable de type **`unsigned long long`**	18446744073709551615 (0xffffffffffffffff)

Si une valeur dépasse la plus grande représentation d'entier, le compilateur Microsoft génère une erreur.

Voir aussi

Limites flottantes

Limites des valeurs à virgule flottante

Article • 16/06/2023

Section spécifique à Microsoft

Le tableau suivant répertorie les limites applicables aux valeurs des constantes à virgule flottante. Ces limites sont également définies dans le fichier <d'en-tête standard float.h>.

Limites des constantes à virgule flottante

Constant	Signification	Valeur

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
	Nombre de chiffres, q, tel qu'un nombre à virgule flottante avec q chiffres décimaux peut être arrondi en une représentation à virgule flottante puis restauré à sa valeur initiale sans perte de précision.	6
		15
		15

`FLT_EPSILON`
`DBL_EPSILON`
`LDBL_EPSILON`
	Plus petit nombre positif x, de sorte que x + 1.0 ne soit pas égal à 1.0.	1,192092896e-07F
		2,2204460492503131e-016
		2,2204460492503131e-016

`FLT_GUARD`		0

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
	Nombre de chiffres dans le radix spécifié par `FLT_RADIX` dans la significande à virgule flottante. La base est 2 ; par conséquent, ces valeurs spécifient des bits.	24
		53
		53

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX`
	Nombre à virgule flottante maximal pouvant être représenté	3,402823466e+38F
		1.7976931348623158e+308
		1.7976931348623158e+308

`FLT_MAX_10_EXP`
`DBL_MAX_10_EXP`
`LDBL_MAX_10_EXP`
	Entier maximal tel que 10 élevé à la puissance de ce nombre est un nombre à virgule flottante pouvant être représenté.	38
		308
		308

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
	Entier maximal tel que élevé `FLT_RADIX` à ce nombre est un nombre à virgule flottante représentable.	128
		1 024
		1 024

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
	Valeur positive minimale.	1,175494351e-38F
		2.2250738585072014e-308
		2.2250738585072014e-308

Constant	Signification	Valeur

`FLT_MIN_10_EXP`
`DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`
	Entier négatif minimal tel que 10 élevé à ce nombre est un nombre à virgule flottante pouvant être représenté.	-37
		-307
		-307

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
	Entier négatif minimal tel que `FLT_RADIX` élevé à ce nombre est un nombre à virgule flottante pouvant être représenté.	-125
		-1021
		-1021

`FLT_NORMALIZE`		0

`FLT_RADIX`
`_DBL_RADIX`
`_LDBL_RADIX`
	Base de représentation d'exposant	2
		2
		2

`FLT_ROUNDS`
`_DBL_ROUNDS`
`_LDBL_ROUNDS`
	Mode d'arrondi pour l'addition à virgule flottante	1 (arrondi)
		1 (arrondi)
		1 (arrondi)

> **Notes**
>
> Les informations contenues dans le tableau peuvent différer dans les versions ultérieures du produit.

FIN de la section spécifique à Microsoft

Voir aussi

Limites des entiers

Déclarations et définitions (C++)

Article • 03/04/2023

Un programme C++ se compose de différentes entités telles que des variables, des fonctions, des types et des espaces de noms. Chacune de ces entités doit être *déclarée* avant de pouvoir être utilisée. Une déclaration spécifie un nom unique pour l'entité, ainsi que des informations sur son type et d'autres caractéristiques. En C++, le point auquel un nom est déclaré est le point auquel il devient visible par le compilateur. Vous ne pouvez pas faire référence à une fonction ou à une classe déclarée ultérieurement dans l'unité de compilation. Les variables doivent être déclarées aussi près que possible avant le moment où elles sont utilisées.

L'exemple suivant montre certaines déclarations :

C++

```
#include <string>

int f(int i); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    C obj; // error! C not yet declared.
    std::string str; // OK std::string is declared in <string> header
    j = 0; // error! No type specified.
    auto k = 0; // OK. type inferred as int by compiler.
}

int f(int i)
{
    return i + 42;
}

namespace N {
   class C{/*...*/};
}
```

À la ligne 5, la `main` fonction est déclarée. À la ligne 7, une **`const`** variable nommée `pi` est déclarée et *initialisée*. À la ligne 8, un entier `i` est déclaré et initialisé avec la valeur produite par la fonction `f`. Le nom `f` est visible par le compilateur en raison de la *déclaration de transfert* à la ligne 3.

À la ligne 9, une variable nommée `obj` de type `C` est déclarée. Toutefois, cette déclaration génère une erreur, car `C` elle n'est déclarée que plus tard dans le

programme et n'est pas déclarée par transfert. Pour corriger l'erreur, vous pouvez déplacer l'intégralité de la *définition* de `C` before `main` ou bien ajouter une déclaration de transfert pour celle-ci. Ce comportement est différent des autres langages tels que C#. Dans ces langages, les fonctions et les classes peuvent être utilisées avant leur point de déclaration dans un fichier source.

À la ligne 10, une variable nommée `str` de type `std::string` est déclarée. Le nom `std::string` est visible, car il est introduit dans le `string`fichier d'en-tête, qui est fusionné dans le fichier source à la ligne 1. `std` est l'espace de noms dans lequel la `string` classe est déclarée.

À la ligne 11, une erreur est générée car le nom `j` n'a pas été déclaré. Une déclaration doit fournir un type, contrairement à d'autres langages tels que JavaScript. À la ligne 12, le **`auto`** mot clé est utilisé, ce qui indique au compilateur de déduire le type de en fonction de `k` la valeur avec laquelle il est initialisé. Dans ce cas, le compilateur choisit **`int`** pour le type.

Portée de la déclaration

Le nom introduit par une déclaration est valide dans *l'étendue* où la déclaration se produit. Dans l'exemple précédent, les variables déclarées à l'intérieur de la `main` fonction sont *des variables locales*. Vous pouvez déclarer une autre variable nommée `i` en dehors de main, à *l'étendue globale*, et il s'agirait d'une entité distincte. Toutefois, une telle duplication de noms peut entraîner une confusion et des erreurs du programmeur et doit être évitée. À la ligne 21, la classe `C` est déclarée dans l'étendue de l'espace de noms `N`. L'utilisation d'espaces de noms permet d'éviter *les collisions de noms*. La plupart des noms de bibliothèque standard C++ sont déclarés dans l'espace de `std` noms. Pour plus d'informations sur la façon dont les règles d'étendue interagissent avec les déclarations, consultez Étendue.

Définitions

Certaines entités, notamment les fonctions, les classes, les énumérations et les variables constantes, doivent être définies et déclarées. Une *définition* fournit au compilateur toutes les informations dont il a besoin pour générer du code machine lorsque l'entité est utilisée ultérieurement dans le programme. Dans l'exemple précédent, la ligne 3 contient une déclaration pour la fonction `f` , mais la *définition* de la fonction est fournie aux lignes 15 à 18. À la ligne 21, la classe `C` est déclarée et définie (bien que la classe ne fasse rien). Une variable constante doit être définie, c'est-à-dire une valeur affectée,

dans l'instruction dans laquelle elle est déclarée. Une déclaration d'un type intégré tel que `int` est automatiquement une définition, car le compilateur sait combien d'espace lui allouer.

L'exemple suivant montre des déclarations qui sont également des définitions :

C++

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
public:
    Boolean IsChecked();
    virtual int     ChangeState() = 0;
};
```

Voici quelques déclarations qui ne sont pas des définitions :

C++

```
extern int i;
char *strchr( const char *Str, const char Target );
```

Instructions Typedefs et using

Dans les versions antérieures de C++, le typedef mot clé est utilisé pour déclarer un nouveau nom qui est un *alias* pour un autre nom. Par exemple, le type `std::string` est un autre nom pour `std::basic_string<char>`. Il devrait être évident pourquoi les programmeurs utilisent le nom typedef et non le nom réel. En C++moderne, le using mot clé est préféré à `typedef`, mais l'idée est la même : un nouveau nom est déclaré pour une entité, qui est déjà déclarée et définie.

Membres de la classe statique

Les membres de données de classe statique sont des variables discrètes partagées par tous les objets de la classe . Étant donné qu'ils sont partagés, ils doivent être définis et

initialisés en dehors de la définition de classe. Pour plus d'informations, consultez Classes.

déclarations extern

Un programme C++ peut contenir plusieurs unités de compilation. Pour déclarer une entité définie dans une unité de compilation distincte, utilisez la extern mot clé. Les informations contenues dans la déclaration sont suffisantes pour le compilateur. Toutefois, si la définition de l'entité est introuvable à l'étape de liaison, l'éditeur de liens génère une erreur.

Contenu de cette section

Classes de stockage
const
constexpr
extern
Initialiseurs
Alias et typedefs
using Déclaration
volatile
decltype
Attributs en C++

Voir aussi

Concepts de base

Classes de stockage

Article • 16/06/2023

Une *classe de stockage* dans le contexte des déclarations de variables C++ est un spécificateur de type qui régit la durée de vie, la liaison et l'emplacement de mémoire des objets. Un objet donné ne peut avoir qu'une seule classe de stockage. Les variables définies dans un bloc ont un stockage automatique, sauf indication contraire à l'aide des `extern` spécificateurs , `static` ou `thread_local` . Les objets et variables automatiques n'ont aucune liaison ; ils ne sont pas visibles pour le code en dehors du bloc. La mémoire est allouée automatiquement pour eux lorsque l'exécution entre dans le bloc, et elle est désaffectée lorsque le bloc est supprimé.

Notes

- Le mutable mot clé peut être considéré comme un spécificateur de classe de stockage. Toutefois, il n'est disponible que dans la liste des membres d'une définition de classe.
- **Visual Studio 2010 et versions ultérieures :** Le `auto` mot clé n'est plus un spécificateur de classe de stockage C++, et le `register` mot clé est déprécié. **Visual Studio 2017 version 15.7 et ultérieure :** (disponible en mode et versions /std:c++17 ultérieures) : le `register` mot clé est supprimé du langage C++. Son utilisation génère un message de diagnostic :

  C++

  ```
  // c5033.cpp
  // compile by using: cl /c /std:c++17 c5033.cpp
  register int value; // warning C5033: 'register' is no longer a
  supported storage class
  ```

`static`

Le `static` mot clé peut être utilisé pour déclarer des variables et des fonctions à l'étendue globale, à l'étendue de l'espace de noms et à l'étendue de la classe. Les variables statiques peuvent également être déclarées dans la portée locale.

La durée statique signifie que l'objet ou la variable est alloué au démarrage du programme et est libéré à la fin de l'exécution du programme. La liaison externe signifie que le nom de la variable est visible à partir de l'extérieur du fichier où la variable est

déclarée. À l'inverse, la liaison interne signifie que le nom n'est pas visible en dehors du fichier où la variable est déclarée. Par défaut, un objet ou une variable défini dans l'espace de noms global a une durée statique et une liaison externe. Le `static` mot clé peut être utilisé dans les situations suivantes.

1. Lorsque vous déclarez une variable ou une fonction à l'étendue du fichier (étendue globale et/ou d'espace de noms), le `static` mot clé spécifie que la variable ou la fonction a une liaison interne. Lorsque vous déclarez une variable, la variable a une durée statique et le compilateur l'initialise à 0, sauf si vous spécifiez une autre valeur.

2. Lorsque vous déclarez une variable dans une fonction, le `static` mot clé spécifie que la variable conserve son état entre les appels à cette fonction.

3. Lorsque vous déclarez un membre de données dans une déclaration de classe, le `static` mot clé spécifie qu'une copie du membre est partagée par toutes les instances de la classe. Un `static` membre de données doit être défini au niveau de l'étendue du fichier. Un membre de données intégral que vous déclarez comme `const static` peut avoir un initialiseur.

4. Lorsque vous déclarez une fonction membre dans une déclaration de classe, le `static` mot clé spécifie que la fonction est partagée par toutes les instances de la classe. Une `static` fonction membre ne peut pas accéder à un membre instance, car la fonction n'a pas de pointeur implicite `this`. Pour accéder à un membre instance, déclarez la fonction avec un paramètre qui est un pointeur ou une référence instance.

5. Vous ne pouvez pas déclarer les membres d'un `union` en tant que `static`. Toutefois, un anonyme `union` déclaré globalement doit être déclaré `static` explicitement.

Cet exemple montre comment une variable déclarée `static` dans une fonction conserve son état entre les appels à cette fonction.

C++

```
// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
   static int nStatic;    // Value of nStatic is retained
                          // between each function call
   nStatic += curr;
```

```
   cout << "nStatic is " << nStatic << endl;
}

int main() {
   for ( int i = 0; i < 5; i++ )
      showstat( i );
}
```

Output

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

Cet exemple montre l'utilisation de `static` dans une classe.

C++

```
// static2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CMyClass {
public:
   static int m_i;
};

int CMyClass::m_i = 0;
CMyClass myObject1;
CMyClass myObject2;

int main() {
   cout << myObject1.m_i << endl;
   cout << myObject2.m_i << endl;

   myObject1.m_i = 1;
   cout << myObject1.m_i << endl;
   cout << myObject2.m_i << endl;

   myObject2.m_i = 2;
   cout << myObject1.m_i << endl;
   cout << myObject2.m_i << endl;

   CMyClass::m_i = 3;
   cout << myObject1.m_i << endl;
   cout << myObject2.m_i << endl;
}
```

Output

```
0
0
1
1
2
2
3
3
```

L'exemple suivant montre une variable locale déclarée `static` dans une fonction membre. La `static` variable est disponible pour l'ensemble du programme ; toutes les instances du type partagent la même copie de la `static` variable.

C++

```
// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
   void Test(int value) {
      static int var = 0;
      if (var == value)
         cout << "var == value" << endl;
      else
         cout << "var != value" << endl;

      var = value;
   }
};

int main() {
   C c1;
   C c2;
   c1.Test(100);
   c2.Test(100);
}
```

Output

```
var != value
var == value
```

À compter de C++11, une `static` initialisation de variable locale est garantie comme thread-safe. Cette fonctionnalité est parfois appelée *statique magique*. Toutefois, dans une application multithread, toutes les assignations suivantes doivent être

synchronisées. La fonctionnalité d'initialisation statique thread-safe peut être désactivée à l'aide de l'indicateur /Zc:threadSafeInit- pour éviter de prendre une dépendance sur le CRT.

`extern`

Les objets et les variables déclarés en tant `extern` que déclarent un objet qui est défini dans une autre unité de traduction ou dans une étendue englobante comme ayant une liaison externe. Pour plus d'informations, consultez externunités de traduction et liaison.

`thread_local` (C++11)

Une variable déclarée avec le `thread_local` spécificateur n'est accessible que sur le thread sur lequel elle est créée. La variable est créée lorsque le thread est créé, et elle est détruite lorsque le thread est détruit. Chaque thread possède sa propre copie de la variable. Sur Windows, `thread_local` est fonctionnellement équivalent à l'attribut propre __declspec(thread) à Microsoft.

C++

```
thread_local float f = 42.0; // Global namespace. Not implicitly static.

struct S // cannot be applied to type definition
{
    thread_local int i; // Illegal. The member must be static.
    thread_local static char buf[10]; // OK
};

void DoSomething()
{
    // Apply thread_local to a local variable.
    // Implicitly "thread_local static S my_struct".
    thread_local S my_struct;
}
```

Éléments à noter sur le `thread_local` spécificateur :

- Les variables locales de threads initialisées dynamiquement dans les DLL peuvent ne pas être correctement initialisées sur tous les threads appelants. Pour plus d'informations, consultez thread.
- Le `thread_local` spécificateur peut être combiné avec `static` ou `extern`.

- Vous pouvez vous appliquer `thread_local` uniquement aux déclarations et définitions de données ; `thread_local` ne peuvent pas être utilisées sur les déclarations ou définitions de fonction.

- Vous pouvez spécifier `thread_local` uniquement sur les éléments de données avec une durée de stockage statique, ce qui inclut les objets de données globaux (à la fois et `extern`), les `static` objets statiques locaux et les membres de données statiques des classes. Toute variable locale déclarée `thread_local` est implicitement statique si aucune autre classe de stockage n'est fournie ; en d'autres termes, l'étendue `thread_local` du bloc équivaut à `thread_local static`.

- Vous devez spécifier `thread_local` à la fois pour la déclaration et la définition d'un objet local de thread, que la déclaration et la définition se produisent dans le même fichier ou dans des fichiers séparés.

- Nous vous déconseillons d'utiliser des `thread_local` variables avec `std::launch::async`. Pour plus d'informations, consultez <future> fonctions.

Sur Windows, `thread_local` est fonctionnellement équivalent à, __declspec(thread) sauf que `*__declspec(thread)*` peut être appliqué à une définition de type et est valide dans le code C. Dans la mesure du possible, utilisez `thread_local` car il fait partie de la norme C++ et est donc plus portable.

inscription

Visual Studio 2017 version 15.3 et ultérieure (disponible en mode et versions ultérieures) : le `register` mot clé n'est plus une classe de stockage prise en /std:c++17 charge. Son utilisation provoque un diagnostic. Le mot clé est toujours réservé dans la norme pour une utilisation ultérieure.

```
C++
```

```
   register int val; // warning C5033: 'register' is no longer a supported
storage class
```

Exemple : initialisation automatique et statique

Une variable automatique ou un objet local est initialisé chaque fois que l'ordre d'exécution atteint sa définition. Une variable automatique ou un objet statique est initialisé la première fois que l'ordre d'exécution atteint sa définition.

Prenons l'exemple suivant, qui définit une classe qui stocke l'initialisation et la destruction des objets, puis définit trois objets, `I1`, `I2`, et `I3`:

C++

```
// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
    szObjName(NULL), sizeofObjName(0) {
    if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
        // Allocate storage for szObjName, then copy
        // initializer szWhat into szObjName, using
        // secured CRT functions.
        sizeofObjName = strlen( szWhat ) + 1;

        szObjName = new char[ sizeofObjName ];
        strcpy_s( szObjName, sizeofObjName, szWhat );

        cout << "Initializing: " << szObjName << "\n";
    }
    else {
        szObjName = 0;
    }
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
    if( szObjName != 0 ) {
        cout << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" ); {
        cout << "In block.\n";
        InitDemo I2( "Auto I2" );
```

```
        static InitDemo I3( "Static I3" );
    }
    cout << "Exited block.\n";
}
```

Output

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3
```

Cet exemple montre comment et quand les objets `I1`, `I2` et `I3` sont initialisés et quand ils sont détruits.

Il y a plusieurs points à noter au sujet du programme :

- Tout d'abord, `I1` et `I2` sont automatiquement détruits lorsque le flux de contrôle quitte le bloc dans lequel ils sont définis.
- Deuxièmement, en C++, il n'est pas nécessaire de déclarer des objets ou des variables au début d'un bloc. En outre, ces objets sont initialisés uniquement lorsque l'ordre d'exécution atteint leurs définitions. (`I2` et `I3` sont des exemples de telles définitions.) La sortie indique exactement quand ils sont initialisés.
- Enfin, les variables locales statiques telles que `I3` conserver leurs valeurs pendant l'exécution du programme, mais sont détruites à l'arrêt du programme.

Voir aussi

Déclarations et définitions

`auto` (C++)

Article • 03/04/2023

Déduit le type d'une variable déclarée de son expression d'initialisation.

> ⓘ **Notes**
>
> La norme C++ définit une signification originale et une signification révisée pour cette mot clé. Avant Visual Studio 2010, le mot clé déclare une variable dans la classe de stockage *automatique*, c'est-à-dire une variable dont la durée de vie est locale. À compter de Visual Studio 2010, le `auto` mot clé déclare une variable dont le type est déduit de l'expression d'initialisation dans sa déclaration. L'option **/Zc:auto[-]** du compilateur contrôle la signification du `auto` mot clé.

Syntaxe

> `auto`*initialiseur dedéclarateur*`;`

> `[](auto`*param1*`, auto`*param2*`) {};`

Notes

Le `auto` mot clé indique au compilateur d'utiliser l'expression d'initialisation d'une variable déclarée, ou d'un paramètre d'expression lambda, pour déduire son type.

Nous vous recommandons d'utiliser la mot clé pour la `auto` plupart des situations, sauf si vous souhaitez vraiment une conversion, car elle offre les avantages suivants :

- **Robustesse:** Si le type de l'expression est modifié, y compris lorsqu'un type de retour de fonction est modifié, cela fonctionne simplement.
- **Performance:** Vous êtes assuré qu'il n'y a pas de conversion.
- **Utilisabilité:** Vous n'avez pas à vous soucier des problèmes d'orthographe de nom de type et des fautes de frappe.
- **Efficacité:** Votre codage peut être plus efficace.

Cas de conversion dans lesquels vous ne souhaitez peut-être pas utiliser `auto`:

- Vous voulez un type spécifique et rien d'autre ne fera.
- Dans les types d'assistance de modèle d'expression, par exemple . `(valarray+valarray)`

Pour utiliser le **`auto`** mot clé, utilisez-le au lieu d'un type pour déclarer une variable et spécifier une expression d'initialisation. En outre, vous pouvez modifier les mot clé à l'aide **`auto`** de spécificateurs et de déclarateurs tels que **`const`**, , **`volatile`**pointeur (**`*`**), référence (**`&`**) et référence rvalue (**`&&`**). Le compilateur évalue l'expression d'initialisation et utilise ensuite ces informations pour déduire le type de la variable.

L'expression **`auto`** d'initialisation peut prendre plusieurs formes :

- Syntaxe d'initialisation universelle, telle que `auto a { 42 };`.
- Syntaxe d'affectation, telle que `auto b = 0;`.
- Syntaxe d'affectation universelle, qui combine les deux formes précédentes, telles que `auto c = { 3.14159 };`.
- Initialisation directe, ou syntaxe de style constructeur, telle que `auto d( 1.41421f );`.

Pour plus d'informations, consultez Initialiseurs et exemples de code plus loin dans ce document.

Quand **`auto`** est utilisé pour déclarer le paramètre de boucle dans une instruction basée sur **`for`** une plage, il utilise une syntaxe d'initialisation différente, par exemple `for (auto& i : iterable) do_action(i);`. Pour plus d'informations, consultez Instruction basée sur une plage for (C++).

Le **`auto`** mot clé est un espace réservé pour un type, mais ce n'est pas lui-même un type. Par conséquent, le **`auto`** mot clé ne peut pas être utilisé dans des casts ou des opérateurs tels que sizeof et (pour C++/CLI). typeid

Utilité

Le **`auto`** mot clé est un moyen simple de déclarer une variable dont le type est complexe. Par exemple, vous pouvez utiliser **`auto`** pour déclarer une variable où l'expression d'initialisation implique des modèles, des pointeurs vers des fonctions ou des pointeurs vers des membres.

Vous pouvez également utiliser **`auto`** pour déclarer et initialiser une variable dans une expression lambda. Vous ne pouvez pas déclarer le type de la variable vous-même, car

le type d'une expression lambda est connu uniquement du compilateur. Pour plus d'informations, consultez Exemples d'expressions lambda.

Types de retour de fin

Vous pouvez utiliser `auto`, avec le spécificateur de `decltype` type, pour aider à écrire des bibliothèques de modèles. Utilisez `auto` et `decltype` pour déclarer un modèle de fonction dont le type de retour dépend des types de ses arguments de modèle. Vous pouvez également utiliser `auto` et `decltype` pour déclarer un modèle de fonction qui encapsule un appel à une autre fonction, puis retourne le type de retour de cette autre fonction. Pour plus d'informations, consultez decltype.

Références et qualificateurs cv

L'utilisation de `auto` supprime des références, `const` des qualificateurs et `volatile` des qualificateurs. Prenons l'exemple suivant :

C++

```
// cl.exe /analyze /EHsc /W4
#include <iostream>

using namespace std;

int main( )
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

Dans l'exemple précédent, myAuto est un `int`, et non une `int` référence, de sorte que la sortie est `11 11`, pas `11 12` comme ce serait le cas si le qualificateur de référence n'avait pas été supprimé par `auto`.

Déduction de type avec des initialiseurs avec accolades (C++14)

L'exemple de code suivant montre comment initialiser une `auto` variable à l'aide d'accolades. Notez la différence entre B et C et entre A et E.

C++

```
#include <initializer_list>

int main()
{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
    auto C{ 4 };

    // C3535: cannot deduce type for 'auto' from initializer list'
    auto D = { 5, 6.7 };

    // C3518 in a direct-list-initialization context the type for 'auto'
    // can only be deduced from a single initializer expression
    auto E{ 8, 9 };

    return 0;
}
```

Restrictions et messages d'erreur

Le tableau suivant répertorie les restrictions relatives à l'utilisation du `auto` mot clé et le message d'erreur de diagnostic correspondant émis par le compilateur.

Numéro d'erreur	Description
C3530	Le `auto` mot clé ne peut pas être combiné avec un autre spécificateur de type.
C3531	Un symbole déclaré avec le `auto` mot clé doit avoir un initialiseur.
C3532	Vous avez utilisé incorrectement le `auto` mot clé pour déclarer un type. Par exemple, vous avez déclaré un type de retour de méthode ou un tableau.
C3533, C3539	Un paramètre ou un argument de modèle ne peut pas être déclaré avec le `auto` mot clé.
C3535	Un paramètre de méthode ou de modèle ne peut pas être déclaré avec le `auto` mot clé.

Numéro d'erreur	Description
C3536	Un symbole ne peut pas être utilisé avant son initialisation. Dans la pratique, cela signifie qu'une variable ne peut pas être utilisée pour s'initialiser.
C3537	Vous ne pouvez pas effectuer un cast vers un type déclaré avec le `auto` mot clé.
C3538	Tous les symboles d'une liste de déclarateurs déclarés avec le `auto` mot clé doivent être résolus au même type. Pour plus d'informations, consultez Déclarations et définitions.
C3540, C3541	Les opérateurs sizeof et typeid ne peuvent pas être appliqués à un symbole déclaré avec le `auto` mot clé.

Exemples

Ces fragments de code illustrent certaines des façons dont le `auto` mot clé peut être utilisé.

Les déclarations suivantes sont équivalentes. Dans la première instruction, la variable `j` est déclarée de type `int`. Dans la deuxième instruction, la variable `k` est déduite comme étant de type `int` , car l'expression d'initialisation (0) est un entier.

C++

```
int j = 0;  // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

Les déclarations suivantes sont équivalentes, mais la seconde déclaration est plus simple que la première. L'une des raisons les plus convaincantes d'utiliser le `auto` mot clé est la simplicité.

C++

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

Le fragment de code suivant déclare le type de variables `iter` et `elem` quand les `for` boucles de plage `for` et démarrent.

C++

```
// cl /EHsc /nologo /W4
#include <deque>
```

```
using namespace std;

int main()
{
    deque<double> dqDoubleData(10, 0.1);

    for (auto iter = dqDoubleData.begin(); iter != dqDoubleData.end();
++iter)
    { /* ... */ }

    // prefer range-for loops with the following information in mind
    // (this applies to any range-for with auto, not just deque)

    for (auto elem : dqDoubleData) // COPIES elements, not much better than
the previous examples
    { /* ... */ }

    for (auto& elem : dqDoubleData) // observes and/or modifies elements IN-
PLACE
    { /* ... */ }

    for (const auto& elem : dqDoubleData) // observes elements IN-PLACE
    { /* ... */ }
}
```

Le fragment de code suivant utilise l'opérateur et la `new` déclaration de pointeur pour déclarer des pointeurs.

C++

```
double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);
```

Le fragment de code suivant déclare plusieurs symboles dans chaque instruction de déclaration. Notez que tous les symboles dans chaque instruction sont résolus en un même type.

C++

```
auto x = 1, *y = &x, **z = &y; // Resolves to int.
auto a(2.01), *b (&a);         // Resolves to double.
auto c = 'a', *d(&c);          // Resolves to char.
auto m = 1, &n = m;            // Resolves to int.
```

Ce fragment de code utilise l'opérateur conditionnel (`?:`) pour déclarer la variable `x` en tant qu'entier avec la valeur 200 :

C++

```
int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;
```

Le fragment de code suivant initialise la variable `x` à type `int`, la variable `y` à une référence au type `const int` et la variable `fp` à un pointeur vers une fonction qui retourne le type `int`.

C++

```
int f(int x) { return x; }
int main()
{
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x);
    p = f;
    auto fp = p;
    //...
}
```

Voir aussi

Mots clés
/Zc:auto (Déduire le type de variable)
sizeof, opérateur
typeid
operator new
Déclarations et définitions
Exemples d'expressions lambda
Initialiseurs
decltype

`const` (C++)

Article • 14/03/2023

Lorsqu'il modifie une déclaration de données, le **`const`** mot clé spécifie que l'objet ou la variable n'est pas modifiable.

Syntaxe

> *`declarator`*:
>
> *`ptr-declarator`*
>
> *`noptr-declarator`* *`parameters-and-qualifiers`* *`trailing-return-type`*
>
> *`ptr-declarator`*:
>
> *`noptr-declarator`*
>
> *`ptr-operator`* *`ptr-declarator`*
>
> *`noptr-declarator`*:
>
> *`declarator-id`* *`attribute-specifier-seq`*opt
>
> *`noptr-declarator`* *`parameters-and-qualifiers`*
>
> *`noptr-declarator`* **`[`** *`constant-expression`*Opter **`]`** *`attribute-specifier-seq`*Opter
>
> **`(`** *`ptr-declarator`* **`)`**
>
> *`parameters-and-qualifiers`*:
>
> **`(`** *`parameter-declaration-clause`* **`)`** *`cv-qualifier-seq`*opt
>
> *`ref-qualifier`*Opter *`noexcept-specifier`*Opter *`attribute-specifier-seq`*Opter
>
> *`trailing-return-type`*:
>
> **`->`** *`type-id`*
>
> *`ptr-operator`*:
>
> **`*`** *`attribute-specifier-seq`*opt *`cv-qualifier-seq`*opt
>
> **`&`** *`attribute-specifier-seq`*opt
>
> **`&&`** *`attribute-specifier-seq`*opt
>
> *`nested-name-specifier`* **`*`** *`attribute-specifier-seq`*opt *`cv-qualifier-seq`*opt
>
> *`cv-qualifier-seq`*:
>
> *`cv-qualifier`* *`cv-qualifier-seq`*opt
>
> *`cv-qualifier`*:
>
> **`const`**
>
> **`volatile`**
>
> *`ref-qualifier`*:
>
> **`&`**
>
> **`&&`**

> *declarator-id*:
>
> ...opt *id-expression*

`const` Valeurs

Le **`const`** mot clé spécifie que la valeur d'une variable est constante et indique au compilateur d'empêcher le programmeur de la modifier.

C++

```
// constant_values1.cpp
int main() {
   const int i = 5;
   i = 10;   // C3892
   i++;   // C2105
}
```

En C++, vous pouvez utiliser le **`const`** mot clé au lieu de la #define directive de préprocesseur pour définir des valeurs constantes. Les valeurs définies avec **`const`** sont soumises à la vérification de type et peuvent être utilisées à la place d'expressions constantes. En C++, vous pouvez spécifier la taille d'un tableau avec une **`const`** variable comme suit :

C++

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray];  // allowed in C++; not allowed in C
```

En C, les valeurs de constante utilisent par défaut la liaison externe. Elles ne peuvent donc apparaître que dans des fichiers sources. En C++, les valeurs de constante utilisent par défaut la liaison interne, ce qui leur permet d'apparaître dans les fichiers d'en-tête.

Le **`const`** mot clé peut également être utilisé dans les déclarations de pointeur.

C++

```
// constant_values3.cpp
int main() {
   char this_char{'a'}, that_char{'b'};
   char *mybuf = &this_char, *yourbuf = &that_char;
   char *const aptr = mybuf;
   *aptr = 'c';   // OK
```

```
    aptr = yourbuf;   // C3892
}
```

Un pointeur vers une variable déclarée comme `const` peut être affecté uniquement à un pointeur qui est également déclaré en tant que `const`.

C++

```
// constant_values4.cpp
#include <stdio.h>
int main() {
   const char *mybuf = "test";
   char *yourbuf = "test2";
   printf_s("%s\n", mybuf);

   const char *bptr = mybuf;   // Pointer to constant data
   printf_s("%s\n", bptr);

   // *bptr = 'a';   // Error
}
```

Vous pouvez utiliser des pointeurs vers des données constantes comme paramètres de fonction pour empêcher la fonction de modifier un paramètre passé par l'intermédiaire d'un pointeur.

Pour les objets déclarés en tant que `const`, vous pouvez uniquement appeler des fonctions membres constantes. Le compilateur garantit que l'objet constant n'est jamais modifié.

C++

```
birthday.getMonth();    // Okay
birthday.setMonth( 4 ); // Error
```

Vous pouvez appeler des fonctions membres constantes ou non constantes pour un objet non constant. Vous pouvez également surcharger une fonction membre à l'aide du `const` mot clé ; cette fonctionnalité permet d'appeler une autre version de la fonction pour les objets constants et non constants.

Vous ne pouvez pas déclarer des constructeurs ou des destructeurs avec le `const` mot clé.

`const` fonctions membres

La déclaration d'une fonction membre avec le `const` mot clé spécifie que la fonction est une fonction « en lecture seule » qui ne modifie pas l'objet pour lequel elle est appelée. Une fonction membre constante ne peut pas modifier des membres de données non statiques ou appeler des fonctions membres qui ne sont pas constantes. Pour déclarer une fonction membre constante, placez le `const` mot clé après la parenthèse fermante de la liste d'arguments. Le `const` mot clé est requis dans la déclaration et dans la définition.

C++

```
// constant_member_function.cpp
class Date
{
public:
   Date( int mn, int dy, int yr );
   int getMonth() const;     // A read-only function
   void setMonth( int mn );   // A write function; can't be const
private:
   int month;
};

int Date::getMonth() const
{
   return month;        // Doesn't modify anything
}
void Date::setMonth( int mn )
{
   month = mn;          // Modifies data member
}
int main()
{
   Date MyDate( 7, 4, 1998 );
   const Date BirthDate( 1, 18, 1953 );
   MyDate.setMonth( 4 );    // Okay
   BirthDate.getMonth();    // Okay
   BirthDate.setMonth( 4 ); // C2662 Error
}
```

Différences C et C++ `const`

Lorsque vous définissez une `const` variable dans un fichier de code source C, vous procédez comme suit :

C

```
const int i = 2;
```

Vous pouvez ensuite utiliser cette variable dans un autre module comme suit :

```
C
```

```
extern const int i;
```

Toutefois, pour obtenir le même comportement en C++, vous devez définir votre `const` variable comme suit :

```
C++
```

```
extern const int i = 2;
```

Comme pour C, vous pouvez ensuite utiliser cette variable dans un autre module comme suit :

```
C++
```

```
extern const int i;
```

Si vous souhaitez définir une `extern` variable dans un fichier de code source C++ à utiliser dans un fichier de code source C, utilisez :

```
C++
```

```
extern "C" const int x=10;
```

pour empêcher que le compilateur C++ altère les noms.

Remarques

Lorsque vous suivez la liste de paramètres d'une fonction membre, le `const` mot clé spécifie que la fonction ne modifie pas l'objet pour lequel elle est appelée.

Pour plus d'informations sur `const`, consultez les articles suivants :

- const et volatile pointeurs
- Qualificateurs de type (référence du langage C)
- volatile
- #define

Voir aussi

Mots clés

constexpr (C++)

Article • 03/04/2023

Le mot clé `constexpr` a été introduit en C++11 et amélioré en C++14. Cela signifie *constexpression de fourmi*. Comme `const`, il peut être appliqué à des variables : une erreur du compilateur est générée lorsqu'un code tente de modifierifla valeur. Contrairement à `const`, `constexpr` peut également être appliqué aux fonctions et aux ructors de classe const. `constexpr` indique que la valeur, ou valeur de retour, est constune fourmi et, si possible, est calculée au moment de la compilation.

Une `constexpr` valeur intégrale peut être utilisée partout où un const entier est requis, par exemple dans les arguments de modèle et les déclarations de tableau. Et quand une valeur est calculée au moment de la compilation plutôt qu'au moment de l'exécution, cela permet à votre programme de s'exécuter plus rapidement et d'utiliser moins de mémoire.

Pour limiter la complexité des calculs de fourmis au moment constde la compilation et leurs impacts potentiels sur le temps de compilation, la norme C++14 exige que les types dans constles expressions de fourmi soient des types littéraux.

Syntaxe

```
constexpr literal-typeidentifier=constant-expression;
constexpr literal-typeidentifier{constant-expression};
constexpr literal-typeidentifier(params);
constexpr ctor(params);
```

Paramètres

params
Un ou plusieurs paramètres, chacun d'eux doit être un type littéral et doit lui-même être une constexpression de fourmi.

Valeur retournée

Une `constexpr` variable ou une fonction doit retourner un type littéral.

Variables constexpr

La principale inférence difentre `const` les variables et est `constexpr` que l'initialisation d'une `const` variable peut être différée jusqu'à l'exécution. Une `constexpr` variable doit être initialisée au moment de la compilation. Toutes les `constexpr` variables sont `const`.

- Une variable peut être déclarée avec `constexpr`, lorsqu'elle a un type littéral et est initialisée. Si l'initialisation est effectuée parformé par un constructor, le constructor doit être déclaré en tant que `constexpr`.

- Une référence peut être déclarée comme `constexpr` lorsque ces deux conditions sont remplies : l'objet référencé est initialisé par une constexpression de fourmi, et toutes les conversions implicites appelées pendant l'initialisation sont également constdes expressions fourmis.

- Toutes les déclarations d'une variable ou d'une `constexpr` fonction doivent avoir la `constexpr` spécificationifla plus spécifique.

C++

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

fonctionsconstexpr

Une `constexpr` fonction est une fonction dont la valeur de retour est calculable au moment de la compilation lorsque la consommation de code l'exige. La consommation de code nécessite la valeur de retour au moment de la compilation pour initialiser une `constexpr` variable ou pour fournir un argument de modèle sans type. Lorsque ses arguments sont `constexpr` des valeurs, une `constexpr` fonction produit une fourmi au moment constde la compilation. Lorsqu'elle est appelée avec des arguments non-`constexpr` ou lorsque sa valeur n'est pas requise au moment de la compilation, elle produit une valeur au moment de l'exécution, comme une fonction normale. (Ce double comportement vous évite d'avoir à écrire `constexpr` et des versions non-`constexpr` de la même fonction.)

Une `constexpr` fonction ou constun ructoreur est implicitement `inline`.

Les règles suivantes s'appliquent aux constexpr fonctions :

- Une `constexpr` fonction doit accepter et retourner uniquement les types littéraux.

- Une `constexpr` fonction peut être récursive.
- Êtrefore C++20, une `constexpr` fonction ne peut pas être virtuelle et un constructor ne peut pas être défini comme `constexpr` lorsque la classe englobante a des classes de base virtuelles. Dans C++20 et versions ultérieures, une `constexpr` fonction peut être virtuelle. Visual Studio 2019 version 16.10 et versions ultérieures prennent en charge `constexpr` les fonctions virtuelles lorsque vous utilisezifl'option de /std:c++20 compilateur ou ultérieure.
- Le corps de la fonction peut être défini avec la valeur `= default` ou `= delete`.
- Le corps ne peut contenir aucune `goto` instruction ni `try` aucun bloc.
- Une spécialisation explicite d'un modèle non-peut`constexpr` être déclarée comme `constexpr`suit :
- Une spécialisation explicite d'un `constexpr` modèle n'a pas non plus besoin d'être `constexpr`:

Les règles suivantes s'appliquent aux `constexpr` fonctions dans Visual Studio 2017 et versions ultérieures :

- Il peut contenir des `if` instructions et `switch` , ainsi que toutes les instructions de bouclage, y compris `for`, basées sur `for`une plage , `while`et **do-while**.
- Il peut contenir des déclarations de variable locale, mais la variable doit être initialisée. Il doit s'agir d'un type littéral et ne peut pas être `static` ou thread-local. La variable déclarée localement n'a pas besoin d'être `const`, et peut muter.
- Une `constexpr` fonction non-membre`static` n'a pas besoin d'être implicitement `const`.

C++

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

Conseil

Dans le débogueur Visual Studio, lors du débogage d'une build Debug non optimisée, vous pouvez déterminer si une `constexpr` fonction est évaluée au moment de la compilation en y plaçant un point d'arrêt. Si le point d'arrêt est atteint, la fonction a été appelée à l'exécution. Sinon, la fonction a été appelée au moment de la compilation.

Extern constexpr

L'option du compilateur /Zc:externConstexpr oblige le compilateur à appliquer une liaison externe aux variables déclarées à l'aide de **l'extern constexpr**. Dans les versions antérieures de Visual Studio, par défaut ou lorsque **/Zc:externConstexpr-** est spécifiéif, Visual Studio applique la liaison interne aux `constexpr` variables même lorsque le `extern` mot clé est utilisé. L'option **/Zc:externConstexpr** est disponible à partir de Visual Studio 2017 Update 15.6 et est désactivée par défaut. L'option /permissive- n'active pas **/Zc:externConstexpr**.

Exemple

L'exemple suivant montre des `constexpr` variables, des fonctions et un type défini par l'utilisateur. Dans la dernière instruction de `main()`, la `constexpr` fonction `GetValue()` membre est un appel d'exécution, car la valeur n'est pas obligatoirement connue au moment de la compilation.

C++

```
// constexpr.cpp
// Compile with: cl /EHsc /W4 constexpr.cpp
#include <iostream>

using namespace std;

// Pass by value
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}

// Pass by reference
constexpr float exp2(const float& x, const int& n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
```

```
        exp2(x * x, (n - 1) / 2) * x;
}

// Compile-time computation of array length
template<typename T, int N>
constexpr int length(const T(&)[N])
{
    return N;
}

// Recursive constexpr function
constexpr int fac(int n)
{
    return n == 1 ? 1 : n * fac(n - 1);
}

// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const
    {
        return _i;
    }
private:
    int _i;
};

int main()
{
    // foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!

    // Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;
}
```

Configuration requise

Visual Studio 2015 ou version ultérieure.

Voir aussi

Déclarations et définitions

const

`extern` (C++)

Article • 03/04/2023

Le `extern` mot clé peut être appliqué à une variable globale, une fonction ou une déclaration de modèle. Il spécifie que le symbole a *externune liaison al*. Pour obtenir des informations générales sur la liaison et la raison pour laquelle l'utilisation de variables globales est déconseillée, consultez Unités de traduction et liaison.

Le `extern` mot clé a quatre significations selon le contexte :

- Dans une déclaration de variable non `const` globale, `extern` spécifie que la variable ou la fonction est définie dans une autre unité de traduction. Doit `extern` être appliqué dans tous les fichiers, à l'exception de celui où la variable est définie.
- Dans une `const` déclaration de variable, elle spécifie que la variable a externune liaison al. Doit `extern` être appliqué à toutes les déclarations de tous les fichiers. (Les variables globales `const` ont une liaison interne par défaut.)
- `extern "C"` spécifie que la fonction est définie ailleurs et utilise la convention d'appel en langage C. Le `extern "C"` modificateur peut également être appliqué à plusieurs déclarations de fonction dans un bloc.
- Dans une déclaration de modèle, `extern` spécifie que le modèle a déjà été instancié ailleurs. `extern` indique au compilateur qu'il peut réutiliser l'autre instanciation, plutôt que d'en créer une à l'emplacement actuel. Pour plus d'informations sur cette utilisation de `extern`, consultez Instanciation explicite.

`extern` liaison pour les non-globals `const`

Lorsque l'éditeur de liens voit `extern` avant une déclaration de variable globale, il recherche la définition dans une autre unité de traduction. Les déclarations de non-variables `const` de portée globale sont external par défaut. S'appliquent `extern` uniquement aux déclarations qui ne fournissent pas la définition.

C++

```
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
extern int i;  // declaration only. same as i in FileA
```

```
//fileC.cpp
extern int i;  // declaration only. same as i in FileA

//fileD.cpp
int i = 43; // LNK2005! 'i' already has a definition.
extern int i = 43; // same error (extern is ignored on definitions)
```

`extern` liaison pour `const` les globals

Une `const` variable globale a une liaison interne par défaut. Si vous souhaitez que la variable ait externune liaison al, appliquez la `extern` mot clé à la définition et à toutes les autres déclarations dans d'autres fichiers :

C++

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i;  // declaration only. same as i in FileA
```

Liaison `extern constexpr`

Dans Visual Studio 2017 version 15.3 et antérieures, le compilateur a toujours donné une `constexpr` liaison interne de variable, même lorsque la variable a été marquée `extern`. Dans Visual Studio 2017 version 15.5 et ultérieures, le commutateur de /Zc:externConstexpr compilateur permet un comportement conforme aux normes correct. Finalement, l'option devient la valeur par défaut. L'option /permissive- n'active `/Zc:externConstexpr` pas .

C++

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

Si un fichier d'en-tête contient une variable déclarée `extern constexpr`, il doit être marqué `__declspec(selectany)` pour que ses déclarations dupliquées soient correctement combinées :

C++

```
extern constexpr __declspec(selectany) int x = 10;
```

`extern "C"` déclarations de fonction et `extern "C++"`

En C++, lorsqu'il est utilisé avec une chaîne, **`extern`** spécifie que les conventions de liaison d'un autre langage sont utilisées pour le ou les déclarateurs. Les fonctions et les données C sont accessibles uniquement si elles ont été déclarées comme ayant une liaison C. Toutefois, elles doivent être définies dans une unité de traduction compilée séparément.

Microsoft C++ prend en charge les chaînes **`"C"`** et **`"C++"`** dans le champ *littéral de chaîne* . Tous les fichiers include standard utilisent la **`extern "C"`** syntaxe pour permettre aux fonctions de bibliothèque d'exécution d'être utilisées dans les programmes C++.

Exemple

L'exemple suivant montre comment déclarer des noms qui ont une liaison C :

C++

```
// Declare printf with C linkage.
extern "C" int printf(const char *fmt, ...);

//  Cause everything in the specified
//  header files to have C linkage.
extern "C" {
    // add your #include statements here
#include <stdio.h>
}

//  Declare the two functions ShowChar
//  and GetChar with C linkage.
extern "C" {
    char ShowChar(char ch);
    char GetChar(void);
}

//  Define the two functions
//  ShowChar and GetChar with C linkage.
extern "C" char ShowChar(char ch) {
    putchar(ch);
    return ch;
}

extern "C" char GetChar(void) {
    char ch;
    ch = getchar();
    return ch;
}
```

```
// Declare a global variable, errno, with C linkage.
extern "C" int errno;
```

Si une fonction a plusieurs spécifications de liaison, elles doivent être d'accord. Il s'agit d'une erreur de déclarer les fonctions comme ayant à la fois une liaison C et C++. En outre, si deux déclarations pour une fonction se produisent dans un programme, l'une avec une spécification de liaison et l'autre sans, la déclaration avec la spécification de liaison doit être la première. Toutes les déclarations redondantes de fonctions qui ont déjà une spécification de liaison sont affectées de la liaison spécifiée dans la première déclaration. Par exemple :

C++

```
extern "C" int CFunc1();
...
int CFunc1();            // Redeclaration is benign; C linkage is
                         //  retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                         //  CFunc2;  cannot contain linkage
                         //  specifier.
```

À compter de Visual Studio 2019, lorsque `/permissive-` est spécifié, le compilateur vérifie que les déclarations des paramètres de `extern "C"` fonction correspondent également. Vous ne pouvez pas surcharger une fonction déclarée en tant que `extern "C"`. À partir de Visual Studio 2019 version 16.3, vous pouvez remplacer cette case activée à l'aide de l'option du /Zc:externC- compilateur après l'option `/permissive-` .

Voir aussi

Mots clés
Unités de traduction et liaison
extern Spécificateur Storage-Class en C
Comportement des identificateurs en C
Liaison en C

Initialiseurs

Article • 06/04/2023

Un initialiseur spécifie la valeur initiale d'une variable. Vous pouvez initialiser des variables dans les contextes suivants :

- Dans la définition d'une variable :

```
C++

int i = 3;
Point p1{ 1, 2 };
```

- Comme l'un des paramètres d'une fonction :

```
C++

set_point(Point{ 5, 6 });
```

- En tant que valeur de retour d'une fonction :

```
C++

Point get_new_point(int x, int y) { return { x, y }; }
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

Les initialiseurs peuvent prendre les formes suivantes :

- Une expression (ou une liste d'expressions séparées par des virgules) entre parenthèses :

```
C++

Point p1(1, 2);
```

- Un signe égal suivi d'une expression :

```
C++

string s = "hello";
```

- Une liste d'initialiseurs entre accolades. La liste peut être vide ou comprendre un ensemble de listes, comme dans l'exemple suivant :

C++

```
struct Point{
    int x;
    int y;
};
class PointConsumer{
public:
    void set_point(Point p){};
    void set_points(initializer_list<Point> my_list){};
};
int main() {
    PointConsumer pc{};
    pc.set_point({});
    pc.set_point({ 3, 4 });
    pc.set_points({ { 3, 4 }, { 5, 6 } });
}
```

Types d'initialisations

Il existe plusieurs types d'initialisations, qui peuvent se produire à différents moments durant l'exécution du programme. Différents types d'initialisation ne s'excluent pas mutuellement: par exemple, l'initialisation de liste peut déclencher l'initialisation de valeur et, dans d'autres circonstances, elle peut déclencher l'initialisation d'agrégation.

Initialisation à zéro

L'initialisation à zéro attribue à une variable une valeur zéro convertie implicitement en type :

- Les variables numériques sont initialisées à 0 (ou 0,0, ou 0,0000000000, etc.).
- Les variables char sont initialisées en `'\0'`.
- Les pointeurs sont initialisés vers **`nullptr`**.
- Les tableaux, les classes POD , les structs et les unions ont leurs membres initialisés à une valeur zéro.

L'initialisation à zéro peut être effectuée à différents moments :

- Au démarrage du programme, pour toutes les variables nommées dont la durée est statique. Ces variables pourront être initialisées à nouveau.
- Pendant l'initialisation de valeurs, pour les types scalaires et ceux de classe POD qui sont initialisés à l'aide d'accolades vides.

- Pour les tableaux dont seul un sous-ensemble de leurs membres est initialisé.

Voici quelques exemples d'initialisation à zéro :

C++

```
struct my_struct{
    int i;
    char c;
};

int i0;              // zero-initialized to 0
int main() {
    static float f1;  // zero-initialized to 0.000000000
    double d{};     // zero-initialized to 0.00000000000000000
    int* ptr{};     // initialized to nullptr
    char s_array[3]{'a', 'b'};  // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 };  // the fourth and fifth ints are
initialized to 0
    my_struct a_struct{};   // i = 0, c = '\0'
}
```

Initialisation par défaut

L'initialisation par défaut des classes, des structs et des unions utilise un constructeur par défaut. Le constructeur par défaut peut être appelé sans expression d'initialisation ou avec le `new` mot clé :

C++

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

Si la classe, le struct ou l'union n'a pas de constructeur par défaut, le compilateur émet une erreur.

Les variables scalaires sont initialisées par défaut lorsqu'elles sont définies sans expression d'initialisation. Leurs valeurs sont indéterminées.

C++

```
int i1;
float f;
char c;
```

Les tableaux sont initialisés par défaut lorsqu'ils sont définis sans expression d'initialisation. Lorsqu'un tableau est initialisé par défaut, ses membres sont initialisés par défaut et ont des valeurs indéterminées, comme dans l'exemple suivant :

C++

```
int int_arr[3];
```

Si les membres du tableau n'ont pas de constructeur par défaut, le compilateur émet une erreur.

Initialisation par défaut des variables constantes

Les variables constantes doivent être déclarées en même temps que l'initialiseur. S'il s'agit de types scalaires, ils provoquent une erreur du compilateur, et s'il s'agit de types de classes qui ont un constructeur par défaut, ils provoquent un avertissement :

C++

```
class MyClass{};
int main() {
    //const int i2;   // compiler error C2734: const object must be
initialized if not extern
    //const char c2;  // same error
    const MyClass mc1; // compiler error C4269: 'const automatic data
initialized with compiler generated default constructor produces unreliable
results
}
```

Initialisation par défaut des variables statiques

Les variables statiques déclarées sans initialiseur sont initialisées à 0 (implicitement converties en type).

C++

```
class MyClass {
private:
    int m_int;
    char m_char;
};

int main() {
    static int int1;       // 0
    static char char1;     // '\0'
    static bool bool1;   // false
```

```
    static MyClass mc1;     // {0, '\0'}
}
```

Pour plus d'informations sur l'initialisation d'objets statiques globaux, consultez main fonction et arguments de ligne de commande.

Initialisation de valeurs

L'initialisation de valeurs se produit dans les cas suivants :

- une valeur nommée est initialisée à l'aide d'accolades vides ;
- un objet temporaire anonyme est initialisé à l'aide de parenthèses ou d'accolades vides ;
- un objet est initialisé avec le `new` mot clé plus des parenthèses ou des accolades vides

L'initialisation de valeurs entraîne ce qui suit :

- pour les classes qui possèdent au moins un constructeur public, le constructeur par défaut est appelé ;
- pour les classes non-union sans constructeur déclaré, l'objet est initialisé à zéro et le constructeur par défaut est appelé
- pour les tableaux, chaque élément est initialisé par une valeur ;
- dans tous les autres cas, la variable est initialisée à zéro.

C++

```
class BaseClass {
private:
    int m_int;
};

int main() {
    BaseClass bc{};     // class is initialized
    BaseClass*  bc2 = new BaseClass();  // class is initialized, m_int value
is 0
    int int_arr[3]{};  // value of all members is 0
    int a{};     // value of a is 0
    double b{};  // value of b is 0.00000000000000000
}
```

Initialisation de copie

L'initialisation de copie correspond à l'initialisation d'un objet à l'aide d'un autre objet. Elle se produit dans les cas suivants :

- Une variable est initialisée à l'aide d'un signe égal.
- Un argument est passé à une fonction.
- Un objet est retourné par une fonction.
- Une exception est levée ou interceptée.
- Des données membres non statiques sont initialisées à l'aide d'un signe égal.
- Les membres des classes, des structs et des unions sont initialisés par initialisation de copie pendant l'initialisation des agrégats. Pour obtenir des exemples, consultez Initialisation d'agrégation .

Le code suivant montre plusieurs exemples d'initialisation de copie :

C++

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass(int myInt) {}
    void set_int(int myInt) { m_int = myInt; }
    int get_int() const { return m_int; }
private:
    int m_int = 7; // copy initialization of m_int

};
class MyException : public exception{};
int main() {
    int i = 5;              // copy initialization of i
    MyClass mc1{ i };
    MyClass mc2 = mc1;      // copy initialization of mc2 from mc1
    MyClass mc1.set_int(i);    // copy initialization of parameter from i
    int i2 = mc2.get_int(); // copy initialization of i2 from return value
of get_int()

    try{
        throw MyException();
    }
    catch (MyException ex){ // copy initialization of ex
        cout << ex.what();
    }
}
```

L'initialisation de copie ne peut pas appeler des constructeurs explicites.

```
C++
```

```
vector<int> v = 10; // the constructor is explicit; compiler error C2440:
can't convert from 'int' to 'std::vector<int,std::allocator<_Ty>>'
regex r = "a.*b"; // the constructor is explicit; same error
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same
error
```

Dans certains cas, si le constructeur de copie de la classe est supprimé ou inaccessible, l'initialisation de copie entraîne une erreur du compilateur.

Initialisation directe

L'initialisation directe utilise des accolades ou des parenthèses non vides. Contrairement à l'initialisation de copie, elle peut appeler des constructeurs explicites. Elle se produit dans les cas suivants :

- Une variable est initialisée à l'aide d'accolades ou de parenthèses non vides.
- une variable est initialisée avec le `new` mot clé plus des accolades ou parenthèses non vides
- Une variable est initialisée à l'aide de .`static_cast`
- Dans un constructeur, les classes de base et les membres non statiques sont initialisés à l'aide d'une liste d'initialiseurs.
- Dans la copie d'une variable capturée d'une expression lambda.

Le code suivant montre des exemples d'initialisation directe :

```
C++
```

```
class BaseClass{
public:
    BaseClass(int n) :m_int(n){} // m_int is direct initialized
private:
    int m_int;
};

class DerivedClass : public BaseClass{
public:
    // BaseClass and m_char are direct initialized
    DerivedClass(int n, char c) : BaseClass(n), m_char(c) {}
private:
```

```
    char m_char;
};
int main(){
    BaseClass bc1(5);
    DerivedClass dc1{ 1, 'c' };
    BaseClass* bc2 = new BaseClass(7);
    BaseClass bc3 = static_cast<BaseClass>(dc1);

    int a = 1;
    function<int()> func = [a](){  return a + 1; }; // a is direct
initialized
    int n = func();
}
```

Initialisation de liste

L'initialisation de liste se produit lorsqu'une variable est initialisée à l'aide d'une liste d'initialiseurs entre accolades. Les listes d'initialiseurs entre accolades peuvent être utilisées dans les cas suivants :

- Une variable est initialisée.
- une classe est initialisée avec le `new` mot clé
- Un objet est retourné par une fonction.
- Un argument est passé à une fonction.
- L'un des arguments est compris dans une initialisation directe.
- Dans un initialiseur de données membres non statiques.
- Dans une liste d'initialiseurs de constructeur.

Le code suivant montre des exemples d'initialisation de liste :

C++

```
class MyClass {
public:
    MyClass(int myInt, char myChar) {}
private:
    int m_int[]{ 3 };
    char m_char;
};
class MyClassConsumer{
public:
    void set_class(MyClass c) {}
    MyClass get_class() { return MyClass{ 0, '\0' }; }
};
```

```
struct MyStruct{
    int my_int;
    char my_char;
    MyClass my_class;
};
int main() {
    MyClass mc1{ 1, 'a' };
    MyClass* mc2 = new MyClass{ 2, 'b' };
    MyClass mc3 = { 3, 'c' };

    MyClassConsumer mcc;
    mcc.set_class(MyClass{ 3, 'c' });
    mcc.set_class({ 4, 'd' });

    MyStruct ms1{ 1, 'a', { 2, 'b' } };
}
```

Initialisation d'agrégats

L'initialisation d'agrégats est un type d'initialisation de liste utilisé pour les tableaux et les types de classes (souvent des structs ou des unions) qui n'ont :

- aucun membre privé ou protégé ;
- aucun constructeur fourni par l'utilisateur, à l'exception des constructeurs supprimés ou utilisés par défaut de manière explicite ;
- aucune classe de base ;
- aucune fonction membre virtuelle ;

> ⓘ **Notes**
> Dans Visual Studio 2015 et versions antérieures, un agrégat n'est pas autorisé à avoir des initialiseurs accolades ou égaux pour les membres non statiques. Cette restriction a été supprimée dans la norme C++14 et implémentée dans Visual Studio 2017.

Les initialiseurs d'agrégats se composent d'une liste d'initialisation entre accolades, avec ou sans signe égal, comme dans l'exemple suivant :

C++

```
#include <iostream>
using namespace std;

struct MyAggregate{
    int myInt;
```

```
    char myChar;
};

struct MyAggregate2{
    int myInt;
    char myChar = 'Z'; // member-initializer OK in C++14
};

int main() {
    MyAggregate agg1{ 1, 'c' };
    MyAggregate2 agg2{2};
    cout << "agg1: " << agg1.myChar << ": " << agg1.myInt << endl;
    cout << "agg2: " << agg2.myChar << ": " << agg2.myInt << endl;

    int myArr1[]{ 1, 2, 3, 4 };
    int myArr2[3] = { 5, 6, 7 };
    int myArr3[5] = { 8, 9, 10 };

    cout << "myArr1: ";
    for (int i : myArr1){
        cout << i << " ";
    }
    cout << endl;

    cout << "myArr3: ";
    for (auto const &i : myArr3) {
        cout << i << " ";
    }
    cout << endl;
}
```

Vous devez normalement voir la sortie suivante :

Output

```
agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0
```

ⓘ Important

Les membres du tableau qui sont déclarés mais qui ne sont pas explicitement initialisés pendant l'initialisation d'agrégation sont initialisés à zéro, comme indiqué ci-dessus `myArr3` .

Initialisation des unions et des structs

Si une union n'a pas de constructeur, vous pouvez l'initialiser avec une valeur unique (ou avec une autre instance d'une union). La valeur est utilisée pour initialiser le premier champ non static. C'est donc différent de l'initialisation de struct, durant laquelle la première valeur de l'initialiseur est utilisée pour initialiser le premier champ, la deuxième pour initialiser le deuxième champ, et ainsi de suite. L'exemple suivant vous permet de comparer l'initialisation des unions et des structs :

C++

```
struct MyStruct {
    int myInt;
    char myChar;
};
union MyUnion {
    int my_int;
    char my_char;
    bool my_bool;
    MyStruct my_struct;
};

int main() {
    MyUnion mu1{ 'a' };  // my_int = 97, my_char = 'a', my_bool = true,
{myInt = 97, myChar = '\0'}
    MyUnion mu2{ 1 };   // my_int = 1, my_char = 'x1', my_bool = true,
{myInt = 1, myChar = '\0'}
    MyUnion mu3{};      // my_int = 0, my_char = '\0', my_bool = false,
{myInt = 0, myChar = '\0'}
    MyUnion mu4 = mu3;  // my_int = 0, my_char = '\0', my_bool = false,
{myInt = 0, myChar = '\0'}
    //MyUnion mu5{ 1, 'a', true };  // compiler error: C2078: too many
initializers
    //MyUnion mu6 = 'a';            // compiler error: C2440: cannot convert
from 'char' to 'MyUnion'
    //MyUnion mu7 = 1;              // compiler error: C2440: cannot convert
from 'int' to 'MyUnion'

    MyStruct ms1{ 'a' };            // myInt = 97, myChar = '\0'
    MyStruct ms2{ 1 };              // myInt = 1, myChar = '\0'
    MyStruct ms3{};                 // myInt = 0, myChar = '\0'
    MyStruct ms4{1, 'a'};           // myInt = 1, myChar = 'a'
    MyStruct ms5 = { 2, 'b' };      // myInt = 2, myChar = 'b'
}
```

Initialisation d'agrégats contenant des agrégats

Les types d'agrégats peuvent contenir d'autres types d'agrégats. Par exemple, les tableaux peuvent contenir des tableaux, des structs, etc. Ces types sont initialisés à l'aide de jeux imbriqués d'accolades, par exemple :

C++

```
struct MyStruct {
    int myInt;
    char myChar;
};
int main() {
    int intArr1[2][2]{{ 1, 2 }, { 3, 4 }};
    int intArr3[2][2] = {1, 2, 3, 4};
    MyStruct structArr[]{ { 1, 'a' }, { 2, 'b' }, {3, 'c'} };
}
```

Initialisation de références

Les variables de type référence doivent être initialisées avec un objet du type à partir duquel le type référence est dérivé, ou avec un objet d'un type pouvant être converti en type à partir duquel le type référence est dérivé. Par exemple :

C++

```
// initializing_references.cpp
int iVar;
long lVar;
int main()
{
    long& LongRef1 = lVar;        // No conversion required.
    long& LongRef2 = iVar;        // Error C2440
    const long& LongRef3 = iVar;  // OK
    LongRef1 = 23L;               // Change lVar through a reference.
    LongRef2 = 11L;               // Change iVar through a reference.
    LongRef3 = 11L;               // Error C3892
}
```

La seule façon d'initialiser une référence avec un objet temporaire consiste à initialiser un objet constant temporaire. Une fois initialisée, une variable de type référence pointe toujours vers le même objet ; il ne peut pas être modifié pour pointer vers un autre objet.

Bien que la syntaxe puisse être identique, l'initialisation des variables de type référence et l'assignation à des variables de type référence sont sémantiquement différentes. Dans l'exemple précédent, les assignations qui modifient `iVar` et `lVar` ressemblent aux initialisations, mais leurs effets sont différents. L'initialisation spécifie l'objet vers lequel pointe la variable de type référence ; l'assignation assigne à l'objet référencé via la référence.

Étant donné que le passage d'un argument de type référence à une fonction et le retour d'une valeur de type référence à partir d'une fonction sont des initialisations, les arguments formels d'une fonction sont initialisés correctement, de même que les références retournées.

Les variables de type référence peuvent être déclarées sans initialiseurs uniquement dans les éléments suivants :

- Déclarations de fonction (prototypes). Par exemple :

```
C++

int func( int& );
```

- Déclarations de type retour de fonction. Par exemple :

```
C++

int& func( int& );
```

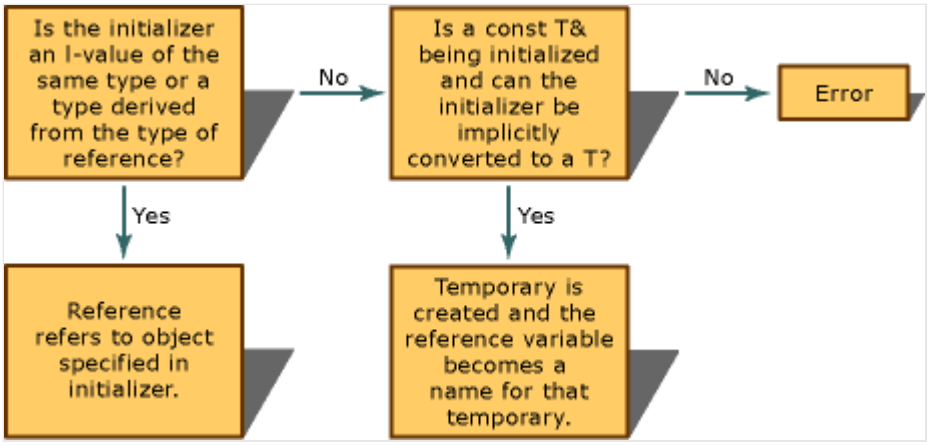
- Déclaration d'un membre de classe de type référence. Par exemple :

```
C++

class c {public:   int& i;};
```

- Déclaration d'une variable explicitement spécifiée en tant que `extern`. Par exemple :

```
C++

extern int& iVal;
```

Lors de l'initialisation d'une variable de type référence, le compilateur utilise le graphique de décision illustré dans la figure suivante pour choisir entre la création d'une référence à un objet ou la création d'un objet temporaire vers lequel la référence pointe :



Graphique de décision pour l'initialisation des types de référence

Les références aux `volatile` types (déclarés en tant*qu'identificateur* `volatile` *de typename*&) peuvent être initialisées avec `volatile` des objets du même type ou avec des objets qui n'ont pas été déclarés en tant que `volatile`. Toutefois, ils ne peuvent pas être initialisés avec `const` des objets de ce type. De même, les références aux `const` types (déclarées en tant*qu'identificateur* `const` *de typename*&) peuvent être initialisées avec `const` des objets du même type (ou tout ce qui a une conversion vers ce type ou avec des objets qui n'ont pas été déclarés en tant que `const`). Toutefois, ils ne peuvent pas être initialisés avec `volatile` des objets de ce type.

Les références qui ne sont pas qualifiées avec ou `const` `volatile` mot clé ne peuvent être initialisées qu'avec des objets déclarés comme ni `const` ni `volatile`.

Initialisation des variables externes

Les déclarations de variables automatiques, statiques et externes peuvent contenir des initialiseurs. Toutefois, les déclarations de variables externes peuvent contenir des initialiseurs uniquement si les variables ne sont pas déclarées en tant que `extern`.

Alias et typedefs (C++)

Article • 03/04/2023

Vous pouvez utiliser une *déclaration d'alias* pour déclarer un nom à utiliser comme synonyme d'un type précédemment déclaré. (Ce mécanisme est également appelé *alias de type* de manière informelle). Vous pouvez également utiliser ce mécanisme pour créer un *modèle d'alias*, ce qui peut être utile pour les répartiteurs personnalisés.

Syntaxe

```
C++
```

```
using identifier = type;
```

Notes

identifier
Nom de l'alias.

type
Identificateur de type pour lequel vous créez un alias.

Un alias n'introduit pas de nouveau type et ne peut pas modifier la signification d'un nom de type existant.

La forme la plus simple d'un alias équivaut au `typedef` mécanisme de C++03 :

```
C++
```

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Ces deux formulaires permettent la création de variables de type `counter`. Un élément plus utile est un alias de type comme celui-ci pour `std::ios_base::fmtflags` :

```
C++
```

```
// C++11
using fmtfl = std::ios_base::fmtflags;
```

```
// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase |
std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

Les alias fonctionnent également avec les pointeurs de fonction, mais sont beaucoup plus lisibles que le typedef équivalent :

C++

```
// C++11
using func = void(*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

Une limitation du `typedef` mécanisme est qu'il ne fonctionne pas avec les modèles. Toutefois, la syntaxe de l'alias de type en C++11 permet la création de modèles d'alias :

C++

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

Exemple

L'exemple suivant montre comment utiliser un modèle d'alias avec un allocateur personnalisé, dans ce cas un type de vecteur entier. Vous pouvez remplacer n'importe quel type pour créer un alias pratique pour `int` masquer les listes de paramètres complexes dans votre code fonctionnel main. En utilisant l'allocateur personnalisé dans tout votre code, vous pouvez améliorer la lisibilité et réduire le risque d'introduction de bogues causés par les fautes de frappe.

C++

```
#include <stdlib.h>
#include <new>

template <typename T> struct MyAlloc {
    typedef T value_type;

    MyAlloc() { }
    template <typename U> MyAlloc(const MyAlloc<U>&) { }

    bool operator==(const MyAlloc&) const { return true; }
    bool operator!=(const MyAlloc&) const { return false; }

    T * allocate(const size_t n) const {
        if (n == 0) {
            return nullptr;
        }

        if (n > static_cast<size_t>(-1) / sizeof(T)) {
            throw std::bad_array_new_length();
        }

        void * const pv = malloc(n * sizeof(T));

        if (!pv) {
            throw std::bad_alloc();
        }

        return static_cast<T *>(pv);
    }

    void deallocate(T * const p, size_t) const {
        free(p);
    }
};

#include <vector>
using MyIntVector = std::vector<int, MyAlloc<int>>;

#include <iostream>

int main ()
{
    MyIntVector foov = { 1701, 1764, 1664 };

    for (auto a: foov) std::cout << a << " ";
    std::cout << "\n";

    return 0;
}
```

Output

```
1701 1764 1664
```

Typedefs

Une `typedef` déclaration introduit un nom qui, dans son étendue, devient synonyme du type donné par la partie *type-déclaration* de la déclaration.

Vous pouvez utiliser des déclarations typedef pour construire des noms plus courts ou plus explicites pour les types déjà définis par la langue ou pour les types que vous avez déclarés. Les noms typedef vous permettent d'encapsuler des détails d'implémentation susceptibles de changer.

Contrairement aux `class` déclarations , `struct`, `union` et , `typedef` les `enum` déclarations n'introduisent pas de nouveaux types ; elles introduisent de nouveaux noms pour les types existants.

Les noms déclarés à l'aide `typedef` de occupent le même espace de noms que les autres identificateurs (à l'exception des étiquettes d'instruction). Par conséquent, ils ne peuvent pas utiliser le même identificateur qu'un nom précédemment déclaré, sauf dans une déclaration de type classe. Prenons l'exemple suivant :

C++

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;   // Declare a typedef name, UL.
int UL;                     // C2377: redefined.
```

Les règles de masquage de noms qui se rapportent à d'autres identificateurs régissent également la visibilité des noms déclarés à l'aide `typedef` de . Par conséquent, l'exemple suivant est autorisé en C++ :

C++

```
// typedef_names2.cpp
typedef unsigned long UL;   // Declare a typedef name, UL
int main()
{
   unsigned int UL;   // Redeclaration hides typedef name
}

// typedef UL back in scope
```

Autre instance de masquage de nom :

C++

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

Lorsque vous déclarez un identificateur d'étendue locale sous le même nom qu'un `typedef`, ou lorsque vous déclarez un membre d'une structure ou d'une union dans la même étendue ou dans une étendue interne, le spécificateur de type doit être spécifié. Par exemple :

C++

```
typedef char FlagType;
const FlagType x;
```

Pour réutiliser le nom `FlagType` pour un identificateur, un membre de structure ou un membre d'union, le type doit être fourni :

C++

```
const int FlagType;  // Type specifier required
```

Il ne suffit pas de déclarer

C++

```
const FlagType;      // Incomplete specification
```

car est `FlagType` considéré comme faisant partie du type, et non comme un identificateur qui est en cours de redéclaré. Cette déclaration est considérée comme une déclaration illégale, similaire à :

C++

```
int;  // Illegal declaration
```

Vous pouvez déclarer tout type avec `typedef`, y compris des types de pointeur, de fonction et de tableau. Vous pouvez déclarer un nom typedef pour un pointeur vers une structure ou un type union avant de définir la structure ou le type union, tant que la définition a la même visibilité que la déclaration.

Exemples

L'une des utilisations des `typedef` déclarations consiste à rendre les déclarations plus uniformes et compactes. Par exemple :

C++

```
typedef char CHAR;          // Character type.
typedef CHAR * PSTR;        // Pointer to a string (char *).
PSTR strchr( PSTR source, CHAR target );
typedef unsigned long ulong;
ulong ul;     // Equivalent to "unsigned long ul;"
```

`typedef` Pour spécifier des types fondamentaux et dérivés dans la même déclaration, vous pouvez séparer les déclarateurs par des virgules. Par exemple :

C++

```
typedef char CHAR, *PSTR;
```

L'exemple suivant fournit le type `DRAWF` pour une fonction qui ne retourne aucune valeur et qui accepte deux arguments int :

C++

```
typedef void DRAWF( int, int );
```

Après l'instruction ci-dessus `typedef` , la déclaration

C++

```
DRAWF box;
```

équivaut à la déclaration

```
```

C++

```
void box( int, int );
```

`typedef` est souvent combiné avec `struct` pour déclarer et nommer les types définis par l'utilisateur :

C++

```
// typedef_specifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int   i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d   %f\n", ms.i, ms.f);
}
```

Output

```
10   0.990000
```

Redeclaration de typedefs

La `typedef` déclaration peut être utilisée pour redéclarer le même nom pour faire référence au même type. Par exemple :

Fichier source `file1.h` :

C++

```
// file1.h
typedef char CHAR;
```

Fichier source `file2.h` :

C++

```
// file2.h
typedef char CHAR;
```

Fichier source `prog.cpp` :

C++

```
// prog.cpp
#include "file1.h"
#include "file2.h"   // OK
```

Le fichier *`prog.cpp`* comprend deux fichiers d'en-tête, qui contiennent **`typedef`** tous deux des déclarations pour le nom `CHAR`. Tant que les deux déclarations désignent le même type, cette redéclaration est acceptable.

Un **`typedef`** ne peut pas redéfinir un nom qui a été déclaré précédemment en tant que type différent. Envisagez cette alternative `file2.h`:

C++

```
// file2.h
typedef int CHAR;     // Error
```

Le compilateur émet une erreur dans en *`prog.cpp`* raison de la tentative de nouvelle déclaration du nom `CHAR` pour faire référence à un autre type. Cette stratégie s'étend aux constructions telles que :

C++

```
typedef char CHAR;
typedef CHAR CHAR;      // OK: redeclared as same type

typedef union REGS      // OK: name REGS redeclared
{                       //  by typedef name with the
    struct wordregs x;  //  same meaning.
    struct byteregs h;
} REGS;
```

typedefs en C++ par rapport à C

L'utilisation du **`typedef`** spécificateur avec des types de classes est en grande partie prise en charge en raison de la pratique ANSI C consistant à déclarer des structures sans

nom dans les `typedef` déclarations. Par exemple, de nombreux programmeurs C utilisent l'idiome suivant :

C++

```
// typedef_with_class_types1.cpp
// compile with: /c
typedef struct {   // Declare an unnamed structure and give it the
                   // typedef name POINT.
   unsigned x;
   unsigned y;
} POINT;
```

L'avantage de ce type de déclaration est qu'il permet des déclarations telles que :

C++

```
POINT ptOrigin;
```

plutôt que :

C++

```
struct point_t ptOrigin;
```

En C++, la différence entre `typedef` les noms et les types réels (déclarés avec les `class` mots clés , `struct`, `union` et `enum`) est plus distincte. Bien que la pratique C consistant à déclarer une structure sans nom dans une `typedef` instruction fonctionne toujours, elle n'offre aucun avantage de notation comme elle le fait dans C.

C++

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
   int POINT();
   unsigned x;
   unsigned y;
} POINT;
```

L'exemple précédent déclare une classe nommée `POINT` à l'aide de la syntaxe de classe `typedef` sans nom. `POINT` est considérée comme un nom de classe. Toutefois, les restrictions suivantes s'appliquent aux noms introduits de cette façon :

- Le nom (synonyme) ne peut pas apparaître après un `class` préfixe , `struct` ou `union` .
- Le nom ne peut pas être utilisé comme nom de constructeur ou de destructeur dans une déclaration de classe.

En résumé, cette syntaxe ne fournit aucun mécanisme pour l'héritage, la construction ou la destruction.

déclaration using

Article • 03/04/2023

La `using` déclaration introduit un nom dans la région déclarative dans laquelle la déclaration using apparaît.

Syntaxe

```
using [typename] nested-name-specifier unqualified-id ;
using declarator-list ;
```

Paramètres

nested-name-specifier Séquence de noms d'espace de noms, de classes ou d'énumérations et d'opérateurs de résolution d'étendue (::), terminée par un opérateur de résolution d'étendue. Un opérateur de résolution d'étendue unique peut être utilisé pour introduire un nom à partir de l'espace de noms global. Le mot clé `typename` est facultatif et peut être utilisé pour résoudre les noms dépendants lorsqu'il est introduit dans un modèle de classe à partir d'une classe de base.

unqualified-id Id-expression non qualifié, qui peut être un identificateur, un nom d'opérateur surchargé, un opérateur littéral défini par l'utilisateur ou un nom de fonction de conversion, un nom de destructeur de classe ou un nom de modèle et une liste d'arguments.

declarator-list Liste séparée par des virgules de déclarateurs *unqualified-id* [`typename`] *nested-name-specifier*, suivie éventuellement d'un point de suspension.

Notes

Une déclaration using introduit un nom non qualifié comme synonyme d'une entité déclarée ailleurs. Il permet d'utiliser un nom unique d'un espace de noms spécifique sans qualification explicite dans la région de déclaration dans laquelle il apparaît. Cela contraste avec la directive using, qui permet d'utiliser *tous les* noms d'un espace de noms sans qualification. Le `using` mot clé est également utilisé pour les alias de type.

Exemple : `using` déclaration dans le champ de classe

Une déclaration using peut être utilisée dans une définition de classe.

C++

```
// using_declaration1.cpp
#include <stdio.h>
class B {
public:
   void f(char) {
      printf_s("In B::f()\n");
   }

   void g(char) {
      printf_s("In B::g()\n");
   }
};

class D : B {
public:
   using B::f;    // B::f(char) is now visible as D::f(char)
   using B::g;    // B::g(char) is now visible as D::g(char)
   void f(int) {
      printf_s("In D::f()\n");
      f('c');     // Invokes B::f(char) instead of recursing
   }

   void g(int) {
      printf_s("In D::g()\n");
      g('c');     // Invokes B::g(char) instead of recursing
   }
};

int main() {
   D myD;
   myD.f(1);
   myD.g('a');
}
```

Output

```
In D::f()
In B::f()
In B::g()
```

Exemple : `using` déclaration pour déclarer un membre

Lorsqu'elle est utilisée pour déclarer un membre, une déclaration using doit faire référence à un membre d'une classe de base.

C++

```
// using_declaration2.cpp
#include <stdio.h>

class B {
public:
   void f(char) {
      printf_s("In B::f()\n");
   }

   void g(char) {
      printf_s("In B::g()\n");
   }
};

class C {
public:
   int g();
};

class D2 : public B {
public:
   using B::f;   // ok: B is a base of D2
   // using C::g;   // error: C isn't a base of D2
};

int main() {
   D2 MyD2;
   MyD2.f('a');
}
```

Output

```
In B::f()
```

Exemple : `using` déclaration avec qualification explicite

Les membres déclarés à l'aide d'une déclaration using peuvent être référencés à l'aide d'une qualification explicite. Le `::` préfixe fait référence à l'espace de noms global.

C++

```
// using_declaration3.cpp
#include <stdio.h>

void f() {
   printf_s("In f\n");
}

namespace A {
   void g() {
      printf_s("In A::g\n");
   }
}

namespace X {
   using ::f;   // global f is also visible as X::f
   using A::g;   // A's g is now visible as X::g
}

void h() {
   printf_s("In h\n");
   X::f();   // calls ::f
   X::g();   // calls A::g
}

int main() {
   h();
}
```

Output

```
In h
In f
In A::g
```

Exemple : `using` synonymes et alias de déclaration

Lorsqu'une déclaration using est effectuée, le synonyme créé par la déclaration fait référence uniquement aux définitions valides au point de la déclaration using. Les définitions ajoutées à un espace de noms après la déclaration using ne sont pas des synonymes valides.

Un nom défini par une `using` déclaration est un alias pour son nom d'origine. Elle n'affecte pas le type, la liaison ou d'autres attributs de la déclaration d'origine.

C++

```
// post_declaration_namespace_additions.cpp
// compile with: /c
namespace A {
   void f(int) {}
}

using A::f;   // f is a synonym for A::f(int) only

namespace A {
   void f(char) {}
}

void f() {
   f('a');   // refers to A::f(int), even though A::f(char) exists
}

void b() {
   using A::f;   // refers to A::f(int) AND A::f(char)
   f('a');   // calls A::f(char);
}
```

Exemple : déclarations et `using` déclarations locales

En ce qui concerne les fonctions dans les espaces de noms, si un ensemble de déclarations locales et des déclarations utilisant pour un nom unique sont donnés dans une région déclarative, elles doivent toutes faire référence à la même entité, ou elles doivent toutes faire référence à des fonctions.

C++

```
// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}

void g() {
    int i;
    using B::i;   // error: i declared twice
    void f(char);
```

```
    using B::f;   // ok: each f is a function
}
```

Dans l'exemple ci-dessus, l'instruction `using B::i` entraîne la déclaration d'une seconde `int i` dans la `g()` fonction . L'instruction `using B::f` n'est pas en conflit avec la `f(char)` fonction, car les noms de fonction introduits par `B::f` ont des types de paramètres différents.

Exemple : déclarations et `using` déclarations de fonction locale

Une déclaration de fonction locale ne peut pas avoir le même nom et le même type qu'une fonction introduite à l'aide de la déclaration. Par exemple :

C++

```
// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;          // introduces B::f(int) and B::f(double)
    using C::f;          // C::f(int), C::f(double), and C::f(char)
    f('h');              // calls C::f(char)
    f(1);                // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);         // C2883 conflicts with B::f(int) and C::f(int)
}
```

Exemple : `using` déclaration et héritage

En ce qui concerne l'héritage, lorsqu'une déclaration using introduit un nom d'une classe de base dans une étendue de classe dérivée, les fonctions membres de la classe dérivée remplacent les fonctions membres virtuelles avec le même nom et les mêmes types d'arguments dans la classe de base.

```
```

C++

```cpp
// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
   virtual void f(int) {
      printf_s("In B::f(int)\n");
   }

   virtual void f(char) {
      printf_s("In B::f(char)\n");
   }

   void g(int) {
      printf_s("In B::g\n");
   }

   void h(int);
};

struct D : B {
   using B::f;
   void f(int) {   // ok: D::f(int) overrides B::f(int)
      printf_s("In D::f(int)\n");
   }

   using B::g;
   void g(char) {   // ok: there is no B::g(char)
      printf_s("In D::g(char)\n");
   }

   using B::h;
   void h(int) {}   // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
   pd->f(1);     // calls D::f(int)
   pd->f('a');   // calls B::f(char)
   pd->g(1);     // calls B::g(int)
   pd->g('a');   // calls D::g(char)
}

int main() {
   D * myd = new D();
   f(myd);
}
```

Output

```
In D::f(int)
In B::f(char)
In B::g
In D::g(char)
```

Exemple : `using` accessibilité des déclarations

Toutes les instances d'un nom mentionné dans une déclaration using doivent être accessibles. En particulier, si une classe dérivée utilise une déclaration using pour accéder à un membre d'une classe de base, le nom du membre doit être accessible. Si le nom est celui d'une fonction membre surchargée, toutes les fonctions nommées doivent être accessibles.

Pour plus d'informations sur l'accessibilité des membres, consultez Access Control membres.

C++

```
// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
   void f(char);
public:
   void f(int);
protected:
   void g();
};

class B : public A {
   using A::f;   // C2876: A::f(char) is inaccessible
public:
   using A::g;   // B::g is a public synonym for A::g
};
```

Voir aussi

Espaces de noms

Mots clés

volatile (C++)

Article • 16/06/2023

Qualificateur de type que vous pouvez utiliser pour déclarer qu'un objet peut être modifié dans le programme par le matériel.

Syntaxe

```
volatile declarator ;
```

Notes

Vous pouvez utiliser le commutateur de compilateur /volatile pour modifier la façon dont le compilateur interprète cette mot clé.

Visual Studio interprète les `volatile` mot clé différemment en fonction de l'architecture cible. Pour ARM, si aucune option de compilateur **/volatile** n'est spécifiée, le compilateur fonctionne comme si **/volatile:iso** était spécifié. Pour les architectures autres qu'ARM, si aucune option de compilateur **/volatile** n'est spécifiée, le compilateur fonctionne comme si **/volatile:ms** était spécifié ; Par conséquent, pour les architectures autres qu'ARM, nous vous recommandons vivement de spécifier **/volatile:iso** et d'utiliser des primitives de synchronisation explicites et des intrinsèques du compilateur lorsque vous traitez de la mémoire partagée entre des threads.

Vous pouvez utiliser le `volatile` qualificateur pour fournir l'accès aux emplacements de mémoire utilisés par les processus asynchrones tels que les gestionnaires d'interruptions.

Quand `volatile` est utilisé sur une variable qui a également le __restrict mot clé, `volatile` est prioritaire.

Si un `struct` membre est marqué comme `volatile`, `volatile` est propagé à l'ensemble de la structure. Si une structure n'a pas de longueur qui peut être copiée sur l'architecture actuelle à l'aide d'une seule instruction, `volatile` peut être complètement perdue sur cette structure.

L'mot clé `volatile` peut ne pas avoir d'effet sur un champ si l'une des conditions suivantes est remplie :

- La longueur du champ volatile dépasse la taille maximale pouvant être copiée sur l'architecture actuelle en utilisant une instruction.
- La longueur du contenant `struct` le plus à l'extérieur (ou s'il s'agit d'un membre d'un éventuellement imbriqué `struct`) dépasse la taille maximale qui peut être copiée sur l'architecture actuelle à l'aide d'une instruction.

Bien que le processeur ne réorganise pas les accès à la mémoire non mis en cache, les variables non mises en cache doivent être marquées comme `volatile` pour garantir que le compilateur ne réorganise pas les accès à la mémoire.

Les objets déclarés comme `volatile` ne sont pas utilisés dans certaines optimisations, car leurs valeurs peuvent changer à tout moment. Le système lit toujours la valeur actuelle d'un objet volatile lorsque la demande en est faite, même si une instruction précédente a demandé une valeur du même objet. De même, la valeur de l'objet est écrite immédiatement sur l'assignation.

Conforme à la norme ISO

Si vous êtes familiarisé avec les mot clé volatiles C#, ou si vous connaissez le comportement de `volatile` dans les versions antérieures du compilateur Microsoft C++ (MSVC), n'oubliez pas que le mot clé ISO Standard `volatile` C++11 est différent et est pris en charge dans MSVC lorsque l'option de compilateur /volatile:iso est spécifiée. (Pour ARM, elle est spécifiée par défaut). Le `volatile` mot clé dans le code ISO Standard C++11 doit être utilisé uniquement pour l'accès au matériel; ne l'utilisez pas pour la communication entre threads. Pour la communication entre threads, utilisez des mécanismes tels que std::atomic<T> de la bibliothèque standard C++.

Fin de la conformité ISO

Section spécifique à Microsoft

Lorsque l'option de compilateur **/volatile:ms** est utilisée (par défaut lorsque des architectures autres qu'ARM sont ciblées), le compilateur génère du code supplémentaire pour maintenir l'ordre entre les références aux objets volatiles en plus de maintenir l'ordre des références à d'autres objets globaux. En particulier :

- L'écriture dans un objet volatile (également appelée "écriture volatile") a une sémantique Release, c'est-à-dire, une référence à un objet global ou statique qui se produit avant qu'une écriture dans un objet volatile de la séquence

d’instructions ne se produise avant cette écriture volatile dans le fichier binaire compilé.

- La lecture d'un objet volatile (également appelée "lecture volatile") a une sémantique Acquire, c'est-à-dire, une référence à un objet global ou statique qui se produit après qu'une lecture de la mémoire volatile dans la séquence d'instructions se produit après cette lecture volatile dans le fichier binaire compilé.

Cela permet aux objets volatiles d'être utilisés pour les verrous et les libérations de mémoire dans les applications multithread.

> ⓘ **Notes**
>
> Lorsqu’il s’appuie sur la garantie améliorée fournie lors de l’utilisation de l’option de compilateur **/volatile:ms** , le code n’est pas portable.

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
const
Pointeurs const et volatiles

`decltype` (C++)

Article • 16/06/2023

Le **`decltype`** spécificateur de type génère le type d'une expression spécifiée. Le **`decltype`** spécificateur de type, ainsi que le auto mot clé, sont principalement utiles aux développeurs qui écrivent des bibliothèques de modèles. Utilisez **`auto`** et **`decltype`** pour déclarer un modèle de fonction dont le type de retour dépend des types de ses arguments de modèle. Vous pouvez également utiliser **`auto`** et **`decltype`** pour déclarer un modèle de fonction qui encapsule un appel à une autre fonction, puis retourne le type de retour de la fonction encapsulée.

Syntaxe

> **`decltype(`** *`expression`* **`)`**

Paramètres

`expression`
Expression. Pour plus d'informations, consultez Expressions.

Valeur retournée

Type du paramètre *`expression`*.

Notes

Le **`decltype`** spécificateur de type est pris en charge dans Visual Studio 2010 ou versions ultérieures et peut être utilisé avec du code natif ou managé. `decltype(auto)` (C++14) est pris en charge dans Visual Studio 2015 et versions ultérieures.

Le compilateur utilise les règles suivantes pour déterminer le type du *`expression`* paramètre .

- Si le *`expression`* paramètre est un identificateur ou un accès membre de classe, `decltype(expression)` est le type de l'entité nommée par *`expression`*. S'il n'existe aucune entité de ce type ou si le *`expression`* paramètre nomme un ensemble de fonctions surchargées, le compilateur génère un message d'erreur.

- Si le *`expression`* paramètre est un appel à une fonction ou une fonction d'opérateur surchargée, `decltype(expression)` est le type de retour de la fonction. Les parenthèses autour d'un opérateur surchargé sont ignorées.
- Si le *`expression`* paramètre est une rvalue, `decltype(expression)` est le type de *`expression`*. Si le *`expression`* paramètre est une lvalue, `decltype(expression)` est une référence lvalue au type de *`expression`*.

L'exemple de code suivant illustre certaines utilisations du spécificateur de **`decltype`** type. Tout d'abord, supposons que vous avez codé les instructions suivantes.

```
C++
```

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

Ensuite, examinez les types retournés par les quatre **`decltype`** instructions du tableau suivant.

.	Type	Notes
`decltype(fx());`	`const int&&`	Référence rvalue à un **`const int`**.
`decltype(var);`	**`int`**	Type de la variable `var`.
`decltype(a->x);`	**`double`**	Type de l'accès au membre.
`decltype((a->x));`	`const double&`	Les parenthèses internes provoquent l'évaluation de l'instruction en tant qu'expression plutôt qu'en tant qu'accès au membre. Et comme `a` est déclaré en tant que **`const`** pointeur, le type est une référence à **`const double`**.

`decltype` et `auto`

Dans C++14, vous pouvez utiliser `decltype(auto)` sans type de retour de fin pour déclarer un modèle de fonction dont le type de retour dépend des types de ses arguments de modèle.

En C++11, vous pouvez utiliser le **`decltype`** spécificateur de type sur un type de retour de fin, avec le **`auto`** mot clé, pour déclarer un modèle de fonction dont le type de retour dépend des types de ses arguments de modèle. Par exemple, considérez l'exemple de

code suivant dans lequel le type de retour du modèle de fonction dépend des types des arguments du modèle. Dans l'exemple de code, l'espace `UNKNOWN` réservé indique que le type de retour ne peut pas être spécifié.

```
C++
```

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

L'introduction du **`decltype`** spécificateur de type permet à un développeur d'obtenir le type de l'expression retournée par le modèle de fonction. Utilisez la *syntaxe de déclaration de fonction alternative* qui est présentée ultérieurement, la **`auto`** mot clé et le **`decltype`** spécificateur de type pour déclarer un type de retour *spécifié tardivement*. Le type de retour spécifié tardivement est déterminé lors de la compilation de la déclaration, au lieu du moment où elle est codée.

Le prototype suivant illustre la syntaxe d'une autre déclaration de fonction. Les **`const`** qualificateurs et **`volatile`** et la spécification d'exception **`throw`** sont facultatifs. L'espace *`function_body`* réservé représente une instruction composée qui spécifie ce que fait la fonction. En guise de meilleure pratique de codage, l'espace *`expression`* réservé dans l'instruction **`decltype`** doit correspondre à l'expression spécifiée par l'instruction, le **`return`** cas échéant, dans le *`function_body`*.

`auto` *`function_name`* **`(`** *`parameters`*_{Opter} **`)`** **`const`**_{Opter} **`volatile`**_{Opter} **`->`** **`decltype(`** *`expression`* **`)`** **`noexcept`**_{Opter} **`{`** *`function_body`* **`};`**

Dans l'exemple de code suivant, le type de retour spécifié tardivement du `myFunc` modèle de fonction est déterminé par les types des arguments de `t` modèle et `u` . Comme meilleure pratique de codage, l'exemple de code utilise également des références rvalue et le modèle de `forward` fonction, qui prennent en charge le *transfert parfait*. Pour plus d'informations, consultez Déclarateur de référence Rvalue : &&.

```
C++
```

```
//C++11
template<typename T, typename U>
auto myFunc(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
        { return forward<T>(t) + forward<U>(u); };

//C++14
template<typename T, typename U>
decltype(auto) myFunc(T&& t, U&& u)
        { return forward<T>(t) + forward<U>(u); };
```

`decltype` et fonctions de transfert (C++11)

Les fonctions de transfert encapsulent les appels à d'autres fonctions. Considérez un modèle de fonction qui transfère ses arguments, ou les résultats d'une expression qui implique ces arguments, à une autre fonction. En outre, la fonction de transfert retourne le résultat de l'appel à l'autre fonction. Dans ce scénario, le type de retour de la fonction de transfert doit être identique au type de retour de la fonction encapsulée.

Dans ce scénario, vous ne pouvez pas écrire une expression de type appropriée sans le spécificateur de `decltype` type. Le `decltype` spécificateur de type active les fonctions de transfert génériques, car il ne perd pas les informations requises pour savoir si une fonction retourne un type référence. Pour obtenir un exemple de code d'une fonction de transfert, consultez l'exemple de modèle de fonction précédent `myFunc` .

Exemples

L'exemple de code suivant déclare le type de retour spécifié tardivement du modèle `Plus()`de fonction . La `Plus` fonction traite ses deux opérandes avec la `operator+` surcharge. Ainsi, l'interprétation de l'opérateur plus (`+`) et du type de retour de la `Plus` fonction dépend des types des arguments de fonction.

C++

```
// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
   decltype(forward<T1>(t1) + forward<T2>(t2))
{
   return forward<T1>(t1) + forward<T2>(t2);
}

class X
{
   friend X operator+(const X& x1, const X& x2)
   {
      return X(x1.m_data + x2.m_data);
   }
```

```
public:
   X(int data) : m_data(data) {}
   int Dump() const { return m_data;}
private:
   int m_data;
};

int main()
{
   // Integer
   int i = 4;
   cout <<
      "Plus(i, 9) = " <<
      Plus(i, 9) << endl;

   // Floating point
   float dx = 4.0;
   float dy = 9.5;
   cout <<
      setprecision(3) <<
      "Plus(dx, dy) = " <<
      Plus(dx, dy) << endl;

   // String
   string hello = "Hello, ";
   string world = "world!";
   cout << Plus(hello, world) << endl;

   // Custom type
   X x1(20);
   X x2(22);
   X x3 = Plus(x1, x2);
   cout <<
      "x3.Dump() = " <<
      x3.Dump() << endl;
}
```

Output

```
Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42
```

Visual Studio 2017 et versions ultérieures : Le compilateur analyse `decltype` les arguments lorsque les modèles sont déclarés plutôt qu'instanciés. Par conséquent, si une spécialisation non dépendante est trouvée dans l'argument `decltype` , elle ne sera pas différée au moment de l'instanciation ; elle est traitée immédiatement et toutes les erreurs résultantes sont diagnostiquées à ce moment-là.

L'exemple suivant montre une erreur de compilateur de ce type générée au moment de la déclaration :

```
C++
```

```
#include <utility>
template <class T, class ReturnT, class... ArgsT> class IsCallable
{
public:
   struct BadType {};
   template <class U>
   static decltype(std::declval<T>()(std::declval<ArgsT>()...)) Test(int);
//C2064. Should be declval<U>
   template <class U>
   static BadType Test(...);
   static constexpr bool value = std::is_convertible<decltype(Test<T>(0)),
ReturnT>::value;
};

constexpr bool test1 = IsCallable<int(), int>::value;
static_assert(test1, "PASS1");
constexpr bool test2 = !IsCallable<int*, int>::value;
static_assert(test2, "PASS2");
```

Configuration requise

Visual Studio 2010 ou versions ultérieures.

`decltype(auto)` nécessite Visual Studio 2015 ou version ultérieure.

Attributs en C++

Article • 11/05/2023

La norme C++ définit un ensemble commun d'attributs. Il permet également aux fournisseurs de compilateur de définir leurs propres attributs au sein d'un espace de noms propre au fournisseur. Toutefois, les compilateurs sont uniquement requis pour reconnaître les attributs définis dans la norme.

Dans certains cas, les attributs standard se chevauchent avec les paramètres spécifiques du `__declspec` compilateur. Dans Microsoft C++, vous pouvez utiliser l'attribut `[[deprecated]]` au lieu d'utiliser `__declspec(deprecated)`. L'attribut `[[deprecated]]` est reconnu par n'importe quel compilateur conforme. Pour tous les autres `__declspec` paramètres tels que `dllimport` et `dllexport`, jusqu'à présent, il n'existe aucun équivalent d'attribut. Vous devez donc continuer à utiliser la `__declspec` syntaxe. Les attributs n'affectent pas le système de type et ne modifient pas la signification d'un programme. Les compilateurs ignorent les valeurs d'attribut qu'ils ne reconnaissent pas.

Visual Studio 2017 version 15.3 et ultérieure (disponible avec /std:c++17 et versions ultérieures) : dans l'étendue d'une liste d'attributs, vous pouvez spécifier l'espace de noms pour tous les noms avec un seul **`using`** introduction :

C++

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel,
rpr::target(cpu,gpu) ]]
    do task();
}
```

Attributs standard C++

En C++11, les attributs fournissent un moyen standardisé d'annoter des constructions C++ (y compris, mais sans s'y limiter, les classes, les fonctions, les variables et les blocs) avec des informations supplémentaires. Les attributs peuvent ou non être spécifiques au fournisseur. Un compilateur peut utiliser ces informations pour générer des messages d'information ou pour appliquer une logique spéciale lors de la compilation du code attribué. Le compilateur ignore tous les attributs qu'il ne reconnaît pas, ce qui signifie que vous ne pouvez pas définir vos propres attributs personnalisés à l'aide de cette syntaxe. Les attributs sont placés entre crochets doubles :

C++

```
[[deprecated]]
void Foo(int);
```

Les attributs représentent une alternative standardisée aux extensions spécifiques au fournisseur, `__declspec()` telles que `#pragma` les directives (Visual C++) ou `__attribute__` (GNU). Toutefois, vous devrez toujours utiliser les constructions propres au fournisseur pour la plupart des fins. La norme spécifie actuellement les attributs suivants qu'un compilateur conforme doit reconnaître.

`[[carries_dependency]]`

L'attribut `[[carries_dependency]]` spécifie que la fonction propage l'ordre des dépendances de données pour la synchronisation de threads. L'attribut peut être appliqué à un ou plusieurs paramètres pour spécifier que l'argument transmis porte une dépendance dans le corps de la fonction. L'attribut peut être appliqué à la fonction elle-même, pour spécifier que la valeur de retour porte une dépendance hors de la fonction. Le compilateur peut utiliser ces informations pour générer du code plus efficace.

`[[deprecated]]`

Visual Studio 2015 et versions ultérieures : L'attribut `[[deprecated]]` spécifie qu'une fonction n'est pas destinée à être utilisée. Ou, qu'il n'existe peut-être pas dans les versions futures d'une interface de bibliothèque. L'attribut `[[deprecated]]` peut être appliqué à la déclaration d'une classe, d'un typedef-name, d'une variable, d'un membre de données non statique, d'une fonction, d'un espace de noms, d'une énumération, d'un énumérateur ou d'une spécialisation de modèle. Le compilateur peut utiliser cet attribut pour générer un message d'information lorsque le code client tente d'appeler la fonction. Lorsque le compilateur Microsoft C++ détecte l'utilisation d'un `[[deprecated]]` élément, il déclenche l'avertissement du compilateur C4996.

`[[fallthrough]]`

Visual Studio 2017 et versions ultérieures : (Disponible avec /std:c++17 et versions ultérieures.) L'attribut `[[fallthrough]]` peut être utilisé dans le contexte d'instructions switch pour indiquer au compilateur (ou à toute personne lisant le code) que le comportement de basculement est destiné. Le compilateur Microsoft C++ n'avertit pas actuellement du comportement de basculement. Cet attribut n'a donc aucun effet sur le comportement du compilateur.

`[[likely]]`

Visual Studio 2019 version 16.6 et ultérieure : (Disponible avec /std:c++20 et versions ultérieures.) L'attribut `[[likely]]` spécifie un indicateur au compilateur indiquant que le chemin de code de l'étiquette ou de l'instruction attribuée est plus susceptible de s'exécuter que les autres. Dans le compilateur Microsoft, l'attribut `[[likely]]` marque les blocs en tant que « code chaud », ce qui incrémente un score d'optimisation interne. Le score est incrémenté davantage lors de l'optimisation pour la vitesse, et moins lors de l'optimisation pour la taille. Le score net affecte la probabilité d'inlining, de déroulement de boucles et d'optimisations de vectorisation. L'effet de `[[likely]]` et `[[unlikely]]` est similaire à l'optimisation guidée par profil, mais limité dans l'étendue à l'unité de traduction actuelle. L'optimisation de la réorganisation des blocs n'est pas encore implémentée pour cet attribut.

`[[maybe_unused]]`

Visual Studio 2017 version 15.3 et ultérieure : (Disponible avec /std:c++17 et versions ultérieures.) L'attribut `[[maybe_unused]]` spécifie qu'une variable, une fonction, une classe, un typedef, un membre de données non statique, une énumération ou un modèle de spécialisation peut être intentionnellement inutilisée. Le compilateur n'avertit pas lorsqu'une entité marquée `[[maybe_unused]]` n'est pas utilisée. Une entité déclarée sans l'attribut peut être redeclarée ultérieurement avec l'attribut et inversement. Une entité est considérée comme *marquée* après que sa première déclaration marquée `[[maybe_unused]]` est analysée, et pour le reste de l'unité de traduction actuelle.

`[[nodiscard]]`

Visual Studio 2017 version 15.3 et ultérieure : (Disponible avec /std:c++17 et versions ultérieures.) Spécifie que la valeur de retour d'une fonction n'est pas destinée à être ignorée. Déclenche l'avertissement C4834, comme illustré dans cet exemple :

C++

```
[[nodiscard]]
int foo(int i) { return i * i; }

int main()
{
    foo(42); //warning C4834: discarding return value of function with
'nodiscard' attribute
    return 0;
}
```

`[[noreturn]]`

L'attribut `[[noreturn]]` spécifie qu'une fonction ne retourne jamais ; en d'autres termes, elle lève toujours une exception ou se ferme. Le compilateur peut ajuster ses règles de compilation pour les `[[noreturn]]` entités.

`[[unlikely]]`

Visual Studio 2019 version 16.6 et ultérieure : (Disponible avec /std:c++20 et versions ultérieures.) L'attribut `[[unlikely]]` indique au compilateur que le chemin d'accès du code de l'étiquette ou de l'instruction attribuée est moins susceptible de s'exécuter que les autres. Dans le compilateur Microsoft, l'attribut `[[unlikely]]` marque les blocs en tant que « code froid », ce qui décrémente un score d'optimisation interne. Le score est décrémenté plus lors de l'optimisation pour la taille, et pas autant lors de l'optimisation pour la vitesse. Le score net affecte la probabilité d'inlining, de déroulement de boucles et d'optimisations de vectorisation. L'optimisation de la réorganisation des blocs n'est pas encore implémentée pour cet attribut.

Attributs spécifiques à Microsoft

`[[gsl::suppress(rules)]]`

L'attribut spécifique `[[gsl::suppress(rules)]]` à Microsoft est utilisé pour supprimer les avertissements des vérificateurs qui appliquent des règles GSL (Guidelines Support Library) dans le code. Par exemple, considérez cet extrait de code :

C++

```
int main()
{
    int arr[10]; // GSL warning C26494 will be fired
    int* p = arr; // GSL warning C26485 will be fired
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
    {
        int* q = p + 1; // GSL warning C26481 suppressed
        p = q--; // GSL warning C26481 suppressed
    }
}
```

L'exemple déclenche les avertissements suivants :

- C26494 (Règle de type 5 : initialiser toujours un objet.)

- C26485 (Règle de limites 3 : Aucune décomposition de tableau en pointeur.)
- C26481 (Règle de limites 1 : N'utilisez pas d'arithmétique de pointeur. Utilisez plutôt l'étendue.)

Les deux premiers avertissements se déclenchent lorsque vous compilez ce code avec l'outil d'analyse du code CppCoreCheck installé et activé. Mais le troisième avertissement ne se déclenche pas en raison de l'attribut . Vous pouvez supprimer l'intégralité du profil de limites en écrivant `[[gsl::suppress(bounds)]]` sans inclure de numéro de règle spécifique. Les instructions C++ Core sont conçues pour vous aider à écrire du code meilleur et plus sûr. L'attribut suppress permet de désactiver facilement les avertissements lorsqu'ils ne sont pas souhaités.

`[[msvc::flatten]]`

L'attribut `[[msvc::flatten]]` spécifique à Microsoft est très similaire à `[[msvc::forceinline_calls]]`, et peut être utilisé aux mêmes emplacements et de la même manière. La différence est que tous les appels dans l'étendue à laquelle ils sont appliqués de manière récursive, jusqu'à ce qu'aucun `[[msvc::flatten]]` `[[msvc::forceinline_calls]]` appel ne soit laissé. Cela peut avoir des conséquences sur la croissance de la taille du code de la fonction ou du débit du compilateur, que vous devez gérer manuellement.

`[[msvc::forceinline]]`

Lorsqu'il est placé avant une déclaration de fonction, l'attribut `[[msvc::forceinline]]` spécifique à Microsoft a la même signification que `__forceinline`.

`[[msvc::forceinline_calls]]`

L'attribut `[[msvc::forceinline_calls]]` spécifique à Microsoft peut être placé sur ou avant une instruction ou un bloc. Cela entraîne la tentative de l'heuristique inline à `[[msvc::forceinline]]` tous les appels de cette instruction ou de ce bloc :

C++

```
void f() {
    [[msvc::forceinline_calls]]
    {
        foo();
        bar();
    }
    ...
```

```
    [[msvc::forceinline_calls]]
    bar();

    foo();
}
```

Le premier appel à `foo`, et les deux appels à `bar`, sont traités comme s'ils avaient été déclarés `__forceinline`. Le deuxième appel à `foo` n'est pas traité comme `__forceinline`.

`[[msvc::intrinsic]]`

L'attribut spécifique `[[msvc::intrinsic]]` à Microsoft indique au compilateur d'inliner une métafonction qui agit comme un cast nommé du type de paramètre au type de retour. Lorsque l'attribut est présent sur une définition de fonction, le compilateur remplace tous les appels à cette fonction par un cast simple. L'attribut `[[msvc::intrinsic]]` est disponible dans Visual Studio 2022 version 17.5 preview 2 et versions ultérieures. Cet attribut s'applique uniquement à la fonction spécifique qui le suit.

L'attribut `[[msvc::intrinsic]]` a trois contraintes sur la fonction à laquelle il est appliqué :

- La fonction ne peut pas être récursive ; son corps ne doit avoir qu'une instruction de retour avec un cast.
- La fonction ne peut accepter qu'un seul paramètre.
- L'option **`/permissive-`** du compilateur est requise. (Les **`/std:c++20`** options et ultérieures impliquent **`/permissive-`** par défaut.)

Exemple

Dans cet exemple de code, l'attribut `[[msvc::intrinsic]]` appliqué à la `my_move` fonction fait que le compilateur remplace les appels à la fonction par le cast statique inline dans son corps :

C++

```
template <typename T>
[[msvc::intrinsic]] T&& my_move(T&& t) { return static_cast<T&&>(t); }

void f() {
    int i = 0;
    i = my_move(i);
}
```

`[[msvc::noinline]]`

Lorsqu'il est placé avant une déclaration de fonction, l'attribut `[[msvc::noinline]]` spécifique à Microsoft a la même signification que `declspec(noinline)`.

`[[msvc::noinline_calls]]`

L'attribut `[[msvc::noinline_calls]]` spécifique à Microsoft a la même utilisation que `[[msvc::forceinline_calls]]`. Il peut être placé avant n'importe quelle instruction ou bloc. Au lieu de forcer tous les appels de ce bloc, cela a pour effet de désactiver l'inlining pour l'étendue à laquelle il est appliqué.

`[[msvc::no_tls_guard]]`

L'attribut spécifique `[[msvc::no_tls_guard]]` à Microsoft désactive les vérifications d'initialisation lors du premier accès aux variables locales de thread dans les DLL. Les vérifications sont activées par défaut dans le code généré à l'aide de Visual Studio 2019 version 16.5 et versions ultérieures. Cet attribut s'applique uniquement à la variable spécifique qui le suit. Pour désactiver les vérifications globalement, utilisez l'option du /Zc:tlsGuards- compilateur.

Opérateurs intégrés C++ , précédence et associativité

Article • 03/04/2023

Le langage C++ inclut tous les opérateurs C et en ajoute plusieurs nouveaux. Les opérateurs spécifient une évaluation à effectuer sur un ou plusieurs opérandes.

Priorité et associativité

La *priorité* de l'opérateur spécifie l'ordre des opérations dans les expressions qui contiennent plusieurs opérateurs. *L'associativité* de l'opérateur spécifie si, dans une expression qui contient plusieurs opérateurs avec la même priorité, un opérande est regroupé avec celui situé à gauche ou à droite.

Orthographes alternatives

C++ spécifie d'autres orthographes pour certains opérateurs. En C, les autres orthographes sont fournies sous forme de macros dans l'en-tête <iso646.h> . En C++, ces alternatives sont des mots clés, et l'utilisation de <iso646.h> ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer les autres orthographes.

Table de priorité et d'associativité de l'opérateur C++

Le tableau suivant indique la priorité et l'associativité des opérateurs C++ (de priorité décroissante). Les opérateurs ayant le même numéro de priorité ont une priorité identique, à moins qu'une autre relation soit explicitement forcée par des parenthèses.

Description des opérateurs	Opérateur	Alternative
Priorité du groupe 1, aucune associativité		
Résolution d'étendue	::	
Priorité du groupe 2, associativité de gauche à droite		
Sélection de membre (objet ou pointeur)	. ou ->	
Indice de tableau	[]	

Description des opérateurs	Opérateur	Alternative
Appel de fonction	()	
Incrément suffixé	++	
Décrément suffixé	--	
Nom du type	typeid	
Conversion de type constant	const_cast	
Conversion de type dynamique	dynamic_cast	
Conversion de type réinterprété	reinterpret_cast	
Conversion de type statique	static_cast	
Priorité du groupe 3, associativité de droite à gauche		
Taille d'objet ou de type	sizeof	
Incrément préfixé	++	
Décrément préfixé	--	
Le complément de l'un	~	`compl`
Non logique	!	`not`
Négation unaire	-	
Plus unaire	+	
Adresse-de	&	
Adressage indirect	*	
Créer un objet	new	
Supprimer un objet	delete	
Cast	()	
Priorité du groupe 4, associativité de gauche à droite		
Pointeur vers membre (objets ou pointeurs)	.* ou ->*	
Priorité du groupe 5, associativité de gauche à droite		
Multiplication	*	
Division	/	

Description des opérateurs	Opérateur	Alternative
Module	%	
Priorité du groupe 6, associativité de gauche à droite		
Addition	+	
Soustraction	-	
Priorité du groupe 7, associativité de gauche à droite		
Décalage vers la gauche	<<	
Décalage vers la droite	>>	
Priorité du groupe 8, associativité de gauche à droite		
Inférieur à	<	
Supérieur à	>	
Inférieur ou égal à	<=	
Supérieur ou égal à	>=	
Priorité du groupe 9, associativité de gauche à droite		
Égalité	==	
Inégalité	!=	`not_eq`
Priorité du groupe 10 de gauche à droite associativité		
ET au niveau du bit	&	`bitand`
Priorité du groupe 11, associativité de gauche à droite		
OU exclusif au niveau du bit	^	`xor`
Priorité du groupe 12, associativité de gauche à droite		
Opération OR inclusive au niveau du bit	|	`bitor`
Priorité du groupe 13, associativité de gauche à droite		
ET logique	&&	`and`
Priorité du groupe 14, associativité de gauche à droite		
OU logique	||	`or`
Priorité du groupe 15, associativité de droite à gauche		

Description des opérateurs	Opérateur	Alternative
Logique conditionnelle	? :	
Cession	=	
Assignation de multiplication	*=	
Assignation de division	/=	
Assignation de modulo	%=	
Assignation d'addition	+=	
Assignation de soustraction	-=	
Assignation de décalage vers la gauche	<<=	
Assignation de décalage vers la droite	>>=	
Assignation d'opération AND au niveau du bit	&=	`and_eq`
Assignation d'opération OR inclusive au niveau du bit	|=	`or_eq`
Assignation d'opération OR exclusive au niveau du bit	^=	`xor_eq`
expression throw	throw	
Priorité du groupe 16, associativité de gauche à droite		
Comma	,	

Voir aussi

Surcharge d’opérateur

L'opérateur `alignof`

Article • 16/06/2023

L'opérateur `alignof` retourne l'alignement en octets du type spécifié en tant que valeur de type `size_t`.

Syntaxe

```
C++

alignof( type )
```

Notes

Par exemple :

Expression	Valeur
`alignof( char )`	1
`alignof( short )`	2
`alignof( int )`	4
`alignof( long long )`	8
`alignof( float )`	4
`alignof( double )`	8

La `alignof` valeur est identique à la valeur pour pour `sizeof` les types de base. Observons toutefois l'exemple suivant :

```
C++

typedef struct { int a; double b; } S;
// alignof(S) == 8
```

Dans ce cas, la `alignof` valeur est l'exigence d'alignement du plus grand élément de la structure.

De même, pour

```
C++
```

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` est égal à `32`.

Une utilisation de **`alignof`** serait comme paramètre pour l'une de vos propres routines d'allocation de mémoire. Par exemple, dans la structure définie `S`suivante, vous pouvez appeler une routine d'allocation de mémoire nommée `aligned_malloc` pour allouer la mémoire sur une limite d'alignement particulière.

```
C++
```

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), alignof(S));
```

Pour plus d'informations sur la modification de l'alignement, consultez :

- pack
- align
- __unaligned
- /Zp (Alignement du membre de struct)
- Exemples d'alignement de structure x64

Pour plus d'informations sur les différences d'alignement dans le code pour x86 et x64, consultez :

- Conflits avec le compilateur x86

Spécifique à Microsoft

`alignof` et **`__alignof`** sont des synonymes dans le compilateur Microsoft. Avant de faire partie de la norme en C++11, l'opérateur spécifique à **`__alignof`** Microsoft fournissait cette fonctionnalité. Pour une portabilité maximale, vous devez utiliser l'opérateur **`alignof`** au lieu de l'opérateur spécifique à **`__alignof`** Microsoft.

Pour la compatibilité avec les versions précédentes, **`_alignof`** est un synonyme de **`__alignof`** l'option /Za de compilateur (Désactiver les extensions de langage) spécifiée.

Voir aussi

Expressions avec opérateurs unaires
Mots clés

`__uuidof`, opérateur

Article • 16/06/2023

Section spécifique à Microsoft

Récupère le GUID attaché à l'expression.

Syntaxe

> `__uuidof (`*expression*`)`

Notes

L'expression peut être un nom de type, un pointeur, une référence ou un tableau de ce type, un modèle spécialisé sur ces types ou une variable de ces types. L'argument est valide tant que le compilateur peut l'utiliser pour rechercher le GUID attaché.

Un cas particulier de cette intrinsèque est lorsque **0** ou NULL est fourni en tant qu'argument. Dans ce cas, `__uuidof` retourne un GUID composé de zéros.

Utilisez cette mot clé pour extraire le GUID attaché à :

- Objet par l'attribut uuid étendu.
- Bloc de bibliothèque créé avec l'attribut module .

> ⓘ **Notes**
>
> Dans une build de débogage, `__uuidof` initialise toujours un objet de manière dynamique (au moment de l'exécution). Dans une build de mise en production, `__uuidof` peut initialiser statiquement (au moment de la compilation) un objet.

Pour la compatibilité avec les versions précédentes, `_uuidof` est un synonyme de `__uuidof` l'option /Za de compilateur (Désactiver les extensions de langage) spécifiée.

Exemple

Le code suivant (compilé avec ole32.lib) affiche l'uuid d'un bloc de bibliothèque créé avec l'attribut module :

C++

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include "stdio.h"
#include "windows.h"

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
   int i;
};

int main() {
   LPOLESTR lpolestr;
   StringFromCLSID(__uuidof(MyLib), &lpolestr);
   wprintf_s(L"%s", lpolestr);
   CoTaskMemFree(lpolestr);
}
```

Commentaires

Dans les cas où le nom de la bibliothèque n'est plus dans l'étendue, vous pouvez utiliser `__LIBID_` au lieu de `__uuidof`. Par exemple :

C++

```
StringFromCLSID(__LIBID_, &lpolestr);
```

FIN de la section spécifique à Microsoft

Voir aussi

Expressions avec opérateurs unaires

Mots clés

Opérateurs additifs : + et -

Article • 03/04/2023

Syntaxe

```
expression + expression
expression - expression
```

Notes

Les opérateurs additifs sont les suivants :

- Addition (+)
- Soustraction (-)

Ces opérateurs binaires ont une associativité de droite à gauche.

Les opérateurs additifs prennent des opérandes de type arithmétique ou pointeur. Le résultat de l'opérateur addition (+) est la somme des opérandes. Le résultat de l'opérateur de soustraction (-) est la différence entre les opérandes. Si l'un des opérandes ou les deux sont des pointeurs, ils doit d'agir de pointeurs vers des objets, et non des fonctions. Si les deux opérandes sont des pointeurs, les résultats ne sont pas significatifs à moins que tous deux ne soient des pointeurs vers des objets dans le même tableau.

Les opérateurs additifs prennent des opérandes de types *arithmétiques*, *intégraux* et *scalaires* . Ils sont définis dans le tableau suivant.

Types utilisés avec les opérateurs additifs

Type	Signification
Opérateurs arithmétiques	Les types intégraux et flottants sont appelés collectivement des types « arithmétiques ».
Intégrale	Les types char et int de toutes tailles (longs, courts) et les énumérations sont des types « intégraux ».

Type	Signification
scalaire	Les opérandes scalaires sont des opérandes de type arithmétique ou pointeur.

Les combinaisons valides pour ces opérateurs sont les suivantes :

Arithmétique + Arithmétique

Scalaire + Intégrale

Intégrale + Scalaire

Arithmétique - Arithmétique

Scalaire - Scalaire

Notez que l'addition et la soustraction ne sont pas des opérations équivalentes.

Si les deux opérandes sont de type arithmétique, les conversions couvertes dans conversions standard sont appliquées aux opérandes et le résultat est du type converti.

Exemple

C++

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
   int i = 5, j = 10;
   int n[SIZE] = { 0, 1, 2, 3, 4 };
   cout  << "5 + 10 = " << i + j << endl
         << "5 - 10 = " << i - j << endl;

   // use pointer arithmetic on array

   cout << "n[3] = " << *( n + 3 ) << endl;
}
```

Addition de pointeur

Si l'un des opérandes d'une addition est un pointeur vers un tableau d'objets, l'autre doit être de type intégral. Le résultat est un pointeur qui est du même type que le

pointeur d'origine et qui pointe vers un autre élément de tableau. Le fragment de code suivant illustre ce concept :

```
C++
```

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

Bien que la valeur intégrale 1 soit ajoutée à `pIntArray`, cela ne signifie pas « ajouter 1 à l'adresse ». Cela signifie plutôt « ajuster le pointeur pour qu'il pointe vers l'objet suivant du tableau », qui se trouve à une distance de 2 octets (ou `sizeof( int )`).

> ⓘ **Notes**
>
> Le code du formulaire `pIntArray = pIntArray + 1` figure rarement dans les programmes C++. Pour effectuer un incrément, il est préférable d'opter pour les formulaires `pIntArray++` ou `pIntArray += 1`.

Soustraction de pointeur

Si les deux opérandes sont des pointeurs, le résultat de la soustraction est la différence (en éléments de tableau) entre les opérandes. L'expression de soustraction génère un résultat intégral signé de type `ptrdiff_t` (défini dans le fichier <include standard stddef.h>).

L'un des opérandes peut être de type intégral, à condition que ce soit le second opérande. Le résultat de la soustraction est du même type que le pointeur d'origine. La valeur de la soustraction est un pointeur vers l'élément de tableau (*n* - *i*)th array, où *n* est l'élément pointé par le pointeur d'origine et *i* la valeur intégrale du deuxième opérande.

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés, priorité et associativité C++

Opérateurs additifs C

Opérateur d'adresse : `&`

Article • 16/06/2023

Syntaxe

`address-of-expression`:
    `&` *`cast-expression`*

Notes

L'opérateur d'adresse unaire (**`&`**) retourne l'adresse de (c'est-à-dire un pointeur vers) son opérande. L'opérande de l'opérateur d'adresse peut être un indicateur de fonction ou un lvalue qui fait référence à un objet qui n'est pas un champ de bits.

L'opérateur address-of ne peut être appliqué qu'à certaines expressions lvalue : soit à des variables de types fondamentaux, de structure, de classe ou d'union, soit à des références de tableau en indice. Dans ces expressions, une expression constante (qui n'inclut pas l'opérateur address-of) peut être ajoutée ou soustraite de l'expression address-of.

Lorsqu'elle est appliquée à des fonctions ou lvalues, le résultat de l'expression est un type pointeur (une rvalue) dérivé du type de l'opérande. Par exemple, si l'opérande est de type **`char`**, le résultat de l'expression est de type pointeur vers **`char`**. L'opérateur address-of, appliqué aux **`const`** objets ou **`volatile`** , prend la valeur `const type *` ou `volatile type *`, où `type` est le type de l'objet d'origine.

Vous ne pouvez prendre l'adresse d'une fonction surchargée que lorsqu'il est clair quelle version de la fonction est référencée. Pour plus d'informations sur la façon d'obtenir l'adresse d'une fonction surchargée particulière, consultez Surcharge de fonction.

Lorsque l'opérateur address-of est appliqué à un nom qualifié, le résultat varie selon que le *nom qualifié* spécifie ou non un membre statique. Si oui, le résultat est un pointeur vers le type spécifié dans la déclaration du membre. Pour un membre qui n'est pas statique, le résultat est un pointeur vers le *nom* de membre de la classe indiqué par *qualified-class-name*. Pour plus d'informations sur *qualified-class-name*, consultez Expressions principales.

Exemple : adresse du membre statique

Le fragment de code suivant montre comment le résultat de l'opérateur address-of diffère, selon qu'un membre de classe est statique ou non :

C++

```
// expre_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
    int iValue;
    static float fValue;
};

int main() {
   int   PTM::*piValue = &PTM::iValue;  // OK: non-static
   float PTM::*pfValue = &PTM::fValue;  // C2440 error: static
   float *spfValue     = &PTM::fValue;  // OK
}
```

Dans cet exemple, l'expression `&PTM::fValue` permet d'obtenir le type `float *` à la place du type `float PTM::*`, car `fValue` est un membre statique.

Exemple : Adresse d'un type référence

L'application de l'opérateur d'adresse à un type référence fournit le même résultat que l'application de l'opérateur à l'objet auquel la référence est liée. Par exemple :

C++

```
// expre_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
   double d;        // Define an object of type double.
   double& rd = d;  // Define a reference to the object.

   // Obtain and compare their addresses
   if( &d == &rd )
      cout << "&d equals &rd" << endl;
}
```

Output

```
&d equals &rd
```

Exemple : Adresse de fonction en tant que paramètre

L'exemple ci-dessous utilise l'opérateur d'adresse pour passer un argument de pointeur à une fonction :

C++

```
// expre_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
   return (*n) * (*n);
}

int main() {
   int mynum = 5;
   cout << square( &mynum ) << endl;   // pass address of int
}
```

Output

```
25
```

Voir aussi

Expressions avec opérateurs unaires
Opérateurs intégrés C++ , précédence et associativité
Déclarateur de référence Lvalue : &
Opérateurs d'indirection et d'adresse

Opérateurs d’affectation

Article • 16/06/2023

Syntaxe

`expression` `assignment-operator` `expression`

`assignment-operator` : l'un des éléments suivants :

`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

Notes

Les opérateurs d’affectation stockent une valeur dans l’objet spécifié par l’opérande gauche. Il existe deux types d’opérations d’affectation :

- *affectation simple*, dans laquelle la valeur du deuxième opérande est stockée dans l’objet spécifié par le premier opérande.
- *affectation composée*, dans laquelle une opération arithmétique, shift ou au niveau du bit est effectuée avant de stocker le résultat.

Tous les opérateurs d’affectation dans le tableau suivant, à l’exception de l’opérateur `=` , sont des opérateurs d’affectation composés.

Table des opérateurs d’affectation

Opérateur	Signification
`=`	Enregistre la valeur du deuxième opérande dans l’objet spécifié par le premier opérande (assignation simple).
`*=`	Multiplie la valeur du premier opérande par la valeur du deuxième opérande ; enregistre le résultat dans l’objet spécifié par le premier opérande.
`/=`	Divise la valeur du premier opérande par la valeur du deuxième opérande ; enregistre le résultat dans l’objet spécifié par le premier opérande.
`%=`	Accepte le module du premier opérande spécifié par la valeur du deuxième opérande ; enregistre le résultat dans l’objet spécifié par le premier opérande.
`+=`	Ajoute la valeur du deuxième opérande à la valeur du premier opérande ; enregistre le résultat dans l’objet spécifié par le premier opérande.

Opérateur	Signification
`-=`	Soustrait la valeur du deuxième opérande de la valeur du premier opérande ; enregistre le résultat dans l'objet spécifié par le premier opérande.
`<<=`	Déplace la valeur du premier opérande à gauche du nombre de bits spécifiés par la valeur du deuxième opérande ; enregistre le résultat dans l'objet spécifié par le premier opérande.
`>>=`	Déplace la valeur du premier opérande à droite du nombre de bits spécifiés par la valeur du deuxième opérande ; enregistre le résultat dans l'objet spécifié par le premier opérande.
`&=`	Obtient l'opérateur de bits AND du premier et du deuxième opérandes ; enregistre le résultat dans l'objet spécifié par le premier opérande.
`^=`	Obtient l'opérateur de bits OR exclusif du premier et du deuxième opérandes ; enregistre le résultat dans l'objet spécifié par le premier opérande.
`|=`	Obtient l'opérateur de bits OR inclusif du premier et du deuxième opérandes ; enregistre le résultat dans l'objet spécifié par le premier opérande.

Mots clés d'opérateur

Trois des opérateurs d'affectation composée ont mot clé équivalents. Il s'agit des éléments suivants :

Opérateur	Équivalent
`&=`	`and_eq`
`|=`	`or_eq`
`^=`	`xor_eq`

C++ spécifie ces mots clés d'opérateur en tant qu'orthographes alternatives pour les opérateurs d'affectation composée. En C, les autres orthographes sont fournies sous forme de macros dans l'en-tête <iso646.h> . En C++, les autres orthographes sont des mots clés ; l'utilisation d'iso646.h <> ou l'équivalent <C++ ciso646> est déconseillé. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

```
C++
```

```
// expre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
#include <iostream>
using namespace std;
int main() {
   int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

   a += b;      // a is 9
   b %= a;      // b is 6
   c >>= 1;      // c is 5
   d |= e;      // Bitwise--d is 0xFFFF

   cout  << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
         << "a += b yields " << a << endl
         << "b %= a yields " << b << endl
         << "c >>= 1 yields " << c << endl
         << "d |= e yields " << hex << d << endl;
}
```

Assignation simple

L'opérateur d'affectation simple (`=`) permet de stocker la valeur du deuxième opérande dans l'objet spécifié par le premier opérande. Si les deux objets sont de types arithmétiques, l'opérande de droite est converti en type de gauche, avant de stocker la valeur.

Les objets de **`const`** types et **`volatile`** peuvent être affectés à des valeurs l de types qui sont uniquement **`volatile`**, ou qui ne sont pas **`const`** ou **`volatile`**.

L'affectation à des objets de type classe (**`struct`**, , et**`class`**) est effectuée par une fonction nommée `operator=`**`union`**. Le comportement par défaut de cette fonction d'opérateur consiste à effectuer une copie de bits ; toutefois, ce comportement peut être modifié à l'aide d'opérateurs surchargés. Pour plus d'informations, consultez Surcharge d'opérateur. Les types de classes peuvent également avoir des opérateurs *d'affectation de copie* et *de déplacement* . Pour plus d'informations, consultez Copier des constructeurs et des opérateurs d'affectation de copie et Déplacer des constructeurs et déplacer des opérateurs d'affectation.

Un objet de toute classe clairement dérivée d'une classe de base donnée peut être assigné à un objet de la classe de base. L'inverse n'est pas vrai, car il existe une conversion implicite de la classe dérivée en classe de base, mais pas de la classe de base en classe dérivée. Par exemple :

```
C++
```

```
// expre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}
```

Les assignations aux types référence se comportent comme si l'assignation était effectuée à l'objet auquel la référence pointe.

Pour les objets de type classe, l'assignation est différente de l'initialisation. Pour illustrer les différences entre l'assignation et l'initialisation, prenez le code

C++

```
UserType1 A;
UserType2 B = A;
```

Le code précédent illustre un initialiseur ; il appelle le constructeur de `UserType2` qui accepte un argument de type `UserType1`. Étant donné le code

C++

```
UserType1 A;
UserType2 B;

B = A;
```

l'instruction d'assignation

```
C++
```

```
B = A;
```

peut avoir l'un des effets suivants :

- Appelez la fonction `operator=` pour `UserType2`, à condition qu'un `operator=` `UserType1` argument soit fourni.
- Appelez la fonction de conversion explicite `UserType1::operator UserType2`, si une telle fonction existe.
- Appelez un constructeur `UserType2::UserType2`, si un tel constructeur existe, qui accepte un argument `UserType1` et copie le résultat.

Assignation composée

Les opérateurs d'affectation composés sont indiqués dans la table Opérateurs d'affectation. Ces opérateurs ont la forme *e1op= e2*, où *e1* est une valeur l non `const` modifiable et *e2* est :

- type arithmétique
- un pointeur, si *op* est `+` ou `-`

Le formulaire *e1op= e2* se comporte comme *e1* `=` *e1ope2*, mais *e1* n'est évalué qu'une seule fois.

L'assignation composée en type énuméré génère un message d'erreur. Si l'opérande gauche est d'un type pointeur, l'opérande droit doit être d'un type pointeur, ou il doit s'agir d'une expression constante qui prend la valeur 0. Lorsque l'opérande gauche est d'un type intégral, l'opérande droit ne doit pas être de type pointeur.

Résultat des opérateurs d'assignation

Les opérateurs d'assignation retournent la valeur de l'objet spécifié par l'opérande gauche après l'assignation. Le type résultant est le type de l'opérande gauche. Le résultat d'une expression d'assignation est toujours une l-value. Ces opérateurs ont une associativité de droite à gauche. L'opérande gauche doit être une l-value modifiable.

Dans ANSI C, le résultat d'une expression d'affectation n'est pas une valeur l. Cela signifie que l'expression `(a += b) += c` C++ légale n'est pas autorisée en C.

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés C++, priorité et associativité
Opérateurs d'affectation C

Opérateur AND au niveau du bit : `&`

Article • 16/06/2023

Syntaxe

`and-expression`:

`equality-expression`

`and-expression` **`&`** *`equality-expression`*

Notes

L'opérateur AND au niveau du bit (**`&`**) compare chaque bit du premier opérande au bit correspondant du deuxième opérande. Si les deux bits ont pour valeur 1, le bit obtenu correspondant a pour valeur 1. Sinon, il a pour valeur 0.

Les deux opérandes de l'opérateur AND au niveau du bit doivent avoir des types intégraux. Les conversions arithmétiques habituelles couvertes dans les conversions Standard sont appliquées aux opérandes.

Opérateur mot clé pour `&`

C++ spécifie **`bitand`** comme une autre orthographe pour **`&`**. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'autre orthographe est un mot clé ; l'utilisation d'iso646.h><ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
   unsigned short a = 0xCCCC;      // pattern 1100 ...
   unsigned short b = 0xAAAA;      // pattern 1010 ...
```

```
    cout  << hex << ( a & b ) << endl;   // prints "8888", pattern 1000 ...
}
```

Voir aussi

Opérateurs intégrés C++, priorité et associativité

Opérateurs au niveau du bit C

Opérateur de bits OU exclusif : ^

Article • 16/06/2023

Syntaxe

> *Expression* `^` *Expression*

Notes

L'opérateur OR exclusif au niveau du bit (`^`) compare chaque bit de son premier opérande au bit correspondant de son deuxième opérande. Si le bit dans l'un des opérandes est 0 et le bit dans l'autre opérande est 1, le bit de résultat correspondant est défini sur 1. Sinon, il a pour valeur 0.

Les deux opérandes de l'opérateur doivent avoir des types intégraux. Les conversions arithmétiques habituelles couvertes dans les conversions standard sont appliquées aux opérandes.

Pour plus d'informations sur l'utilisation alternative du `^` caractère dans C++/CLI et C++/CX, consultez Handle to Object Operator (^) (C++/CLI et C++/CX).

Opérateur mot clé pour ^

C++ spécifie **`xor`** comme autre orthographe pour `^`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'orthographe alternative est une mot clé ; l'utilisation de <iso646.h> ou de l'équivalent <C++ ciso646> est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
   unsigned short a = 0x5555;      // pattern 0101 ...
```

```
    unsigned short b = 0xFFFF;      // pattern 1111 ...

    cout  << hex << ( a ^ b ) << endl;   // prints "aaaa" pattern 1010 ...
}
```

Voir aussi

Opérateurs intégrés C++ , précédence et associativité

Opérateur OR inclusif au niveau du bit : `|`

Article • 16/06/2023

Syntaxe

> *expression1* `|` *expression2*

Notes

L'opérateur OR inclusif au niveau du bit (`|`) compare chaque bit de son premier opérande au bit correspondant de son deuxième opérande. Si l'un des deux bits a pour valeur 1, le bit obtenu correspondant a pour valeur 1. Sinon, il a pour valeur 0.

Les deux opérandes de l'opérateur doivent avoir des types intégraux. Les conversions arithmétiques habituelles couvertes dans les conversions standard sont appliquées aux opérandes.

Opérateur mot clé pour `|`

C++ spécifie **`bitor`** comme autre orthographe pour `|`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'orthographe alternative est une mot clé ; l'utilisation de <iso646.h> ou de l'équivalent <C++ ciso646> est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
   unsigned short a = 0x5555;      // pattern 0101 ...
   unsigned short b = 0xAAAA;      // pattern 1010 ...
```

```
    cout  << hex << ( a | b ) << endl;   // prints "ffff" pattern 1111 ...
}
```

Voir aussi

Opérateurs intégrés C++ , précédence et associativité

Opérateurs au niveau du bit C

Opérateur cast : `()`

Article • 16/06/2023

Un cast de type fournit une méthode pour la conversion explicite du type d'un objet dans une situation spécifique.

Syntaxe

`cast-expression`:

`unary-expression`

`(` *`type-name`* `)` *`cast-expression`*

Notes

Toute expression unaire est considérée comme une expression de cast.

Le compilateur traite *`cast-expression`* comme un type *`type-name`* après la réalisation d'un cast de type. Les casts peuvent être utilisés pour convertir des objets de tout type scalaire vers ou depuis tout autre type scalaire. Les casts de types explicites sont contraints par les mêmes règles qui déterminent les effets des conversions implicites. D'autres restrictions sur les casts peuvent résulter de la taille réelle ou de la représentation de types spécifiques.

Exemples

Conversion de cast standard entre les types intégrés :

C++

```
// expre_CastOperator.cpp
// compile with: /EHsc
// Demonstrate cast operator
#include <iostream>

using namespace std;

int main()
{
    double x = 3.1;
    int i;
    cout << "x = " << x << endl;
    i = (int)x;   // assign i the integer part of x
```

```
    cout << "i = " << i << endl;
}
```

Opérateur de cast défini dans un type défini par l'utilisateur :

C++

```
// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
                      m_nSize(nSize)
    {
        m_pStr = new char[sizeOfBuffer];

        strncpy_s(m_pStr, sizeOfBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
        m_pStr[m_nSize] = '\0';
        return(m_pStr);
    }

    enum
    {
        sizeOfBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
```

```
    const char *kStr = "Excitinggg";
    CountedAnsiString myStr(kStr, 8);

    const char *pRaw = myStr.GetRawBytes();
    printf_s("RawBytes truncated to 10 chars:   %.10s\n", pRaw);

    const char *pCast = myStr; // or (const char *)myStr;
    printf_s("Casted Bytes:   %s\n", pCast);

    puts("Note that the cast changed the raw internal string");
    printf_s("Raw Bytes after cast:   %s\n", pRaw);
}
```

Output

```
RawBytes truncated to 10 chars:   Exciting!!
Casted Bytes:   Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast:   Exciting
```

Voir aussi

Expressions avec opérateurs unaires

Opérateurs intégrés C++ , précédence et associativité

Opérateur de conversion de type explicite : ()

Opérateurs de cast (C++)

Opérateurs de cast (C)

Opérateur virgule : ,

Article • 03/04/2023

Permet le regroupement de deux instructions lorsque l'une d'elles est attendue.

Syntaxe

```
expression , expression
```

Notes

L'opérateur virgule présente une associativité de gauche à droite. Deux expressions séparées par une virgule sont évaluées de gauche à droite. L'opérande gauche est toujours évalué et tous les effets secondaires sont terminés avant l'évaluation de l'opérande droite.

Les virgules peuvent être utilisées comme séparateurs dans certains contextes, par exemple les listes d'arguments de fonction. Ne confondez pas l'utilisation de la virgule comme séparateur avec son utilisation comme opérateur. Ces deux utilisations sont entièrement différentes.

Prenons l'exemple de l'expression `e1, e2`. Le type et la valeur de l'expression sont le type et la valeur de *e2* ; le résultat de l'évaluation *d'e1* est ignoré. Le résultat est une l-value si l'opérande droite est une l-value.

Alors que la virgule est normalement utilisée comme séparateur (par exemple dans les arguments réels des fonctions ou les initialiseurs d'agrégat), l'opérateur virgule et ses opérandes doivent être placés entre parenthèses. Par exemple :

C++

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

Dans l'appel de fonction à `func_one` ci-dessus, trois arguments, séparés par des virgules, sont passés : `x`, `y + 2` et `z`. Dans l'appel de fonction à `func_two`, les parenthèses forcent le compilateur à interpréter la première virgule comme l'opérateur d'évaluation séquentielle. Cet appel de fonction passe deux arguments à `func_two`. Le premier

argument est le résultat de l'opération d'évaluation séquentielle `(x--, y + 2)`, qui a la valeur et le type de l'expression `y + 2`. Le second argument est `z`.

Exemple

C++

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
   int i = 10, b = 20, c= 30;
   i = b, c;
   printf("%i\n", i);

   i = (b, c);
   printf("%i\n", i);
}
```

Output

```
20
30
```

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés, priorité et associativité C++
Opérateur d'évaluation séquentielle

Opérateur conditionnel : `?:`

Article • 16/06/2023

Syntaxe

```
expression ? expression : expression
```

Notes

L'opérateur conditionnel (**? :**) est un opérateur ternaire (il prend trois opérandes). L'opérateur conditionnel fonctionne comme suit :

- Le premier opérande est implicitement converti `bool` en . Il est évalué et tous les effets secondaires sont résolus avant de continuer.
- Si le premier opérande est évalué à `true` (1), le deuxième opérande est évalué.
- Si le premier opérande est évalué à `false` (0), le troisième opérande est évalué.

Le résultat de l'opérateur conditionnel est que le résultat de tout opérande est évalué — le deuxième ou le troisième. Seul l'un des deux derniers opérandes est évalué dans une expression conditionnelle.

Les expressions conditionnelles ont une associativité de droite à gauche. Le premier opérande doit être de type intégral ou de type pointeur. Les règles suivantes s'appliquent au deuxième et au troisième opérandes :

- Si les deux opérandes sont du même type, le résultat est de ce type.
- Si les deux opérandes sont de type arithmétique ou énumération, les conversions arithmétiques habituelles (couvertes dans les conversions standard) sont effectuées pour les convertir en un type commun.
- Si les deux opérandes sont des types pointeur ou si l'un est de type pointeur et l'autre une expression constante qui correspond à 0, les conversions de pointeur sont exécutées pour les convertir en un type commun.
- Si les deux opérandes sont des types référence, les conversions de référence sont exécutées pour les convertir en un type commun.

- Si les deux opérandes sont de type void, le type commun est le type void.
- Si les deux opérandes sont du même type défini par l'utilisateur, le type commun correspond à ce type.
- Si les opérandes ont des types différents et qu'au moins un des opérandes a un type défini par l'utilisateur, alors les règles de langage sont utilisées pour déterminer le type commun. (Voir l'avertissement ci-dessous.)

Toutes les combinaisons des deuxième et troisième opérandes qui ne sont pas dans la liste précédente ne sont pas conformes. Le type du résultat est le type commun, et c'est une l-value si le deuxième et le troisième opérande sont du même type et si les deux sont des valeurs l-value.

> ⚠ **Avertissement**
>
> Si les types des deuxième et troisième opérandes ne sont pas identiques, les règles de conversion de type complexe, telles que spécifiées par la norme C++, sont appelées. Ces conversions peuvent entraîner un comportement inattendu, notamment la construction et la destruction d'objets temporaires. Pour cette raison, il est fortement conseillé de soit (1) éviter d'utiliser les types définis par l'utilisateur comme des opérandes avec l'opérateur conditionnel ou (2) si vous utilisez des types définis par l'utilisateur, d'effectuer une conversion de type explicite pour chaque opérande vers un type commun.

Exemple

C++

```
// expre_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
   int i = 1, j = 2;
   cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

Voir aussi

Opérateurs intégrés, priorité et associativité C++
Opérateur d'expression conditionnelle

delete, opérateur (C++)

Article • 03/04/2023

Libère un bloc de mémoire.

Syntaxe

> [`::`] `delete` *cast-expression*
>
> [`::`] `delete []` *cast-expression*

Notes

L'argument cast-expression doit être un pointeur vers un bloc de mémoire précédemment alloué à un objet créé avec le nouvel opérateur. L'opérateur **`delete`** a un résultat de type **`void`** et ne retourne donc pas de valeur. Par exemple :

```
C++

CDialog* MyDialog = new CDialog;
// use MyDialog
delete MyDialog;
```

L'utilisation **`delete`** de sur un pointeur vers un objet non alloué avec **`new`** donne des résultats imprévisibles. Vous pouvez toutefois utiliser **`delete`** sur un pointeur avec la valeur 0. Cette disposition signifie que, lorsque **`new`** retourne 0 en cas d'échec, la suppression du résultat d'une opération ayant échoué **`new`** est sans danger. Pour plus d'informations, consultez Opérateurs nouveaux et supprimés.

Les **`new`** opérateurs et **`delete`** peuvent également être utilisés pour les types intégrés, y compris les tableaux. Si `pointer` fait référence à un tableau, placez des crochets vides (`[]`) avant `pointer`:

```
C++

int* set = new int[100];
//use set[]
delete [] set;
```

L'utilisation de l'opérateur **`delete`** sur un objet libère sa mémoire. Un programme qui déréférence un pointeur après la suppression de l'objet peut avoir des résultats

imprévisibles ou se bloquer.

Quand `delete` est utilisé pour libérer la mémoire d'un objet de classe C++, le destructeur de l'objet est appelé avant que la mémoire de l'objet soit désallouée (si l'objet a un destructeur).

Si l'opérande de l'opérateur `delete` est une valeur l modifiable, sa valeur n'est pas définie après la suppression de l'objet.

Si l'option du compilateur /sdl (Activer des vérifications de sécurité supplémentaires) est spécifiée, l'opérande de l'opérateur `delete` est défini sur une valeur non valide après la suppression de l'objet.

Utilisation de delete

Il existe deux variantes syntaxiques pour l'opérateur delete : l'une pour les objets uniques et l'autre pour les tableaux d'objets. Le fragment de code suivant montre comment ils diffèrent :

C++

```
// expre_Using_delete.cpp
struct UDType
{
};

int main()
{
   // Allocate a user-defined object, UDObject, and an object
   //  of type double on the free store using the
   //  new operator.
   UDType *UDObject = new UDType;
   double *dObject = new double;
   // Delete the two objects.
   delete UDObject;
   delete dObject;
   // Allocate an array of user-defined objects on the
   // free store using the new operator.
   UDType (*UDArr)[7] = new UDType[5][7];
   // Use the array syntax to delete the array of objects.
   delete [] UDArr;
}
```

Les deux cas suivants produisent des résultats non définis : l'utilisation de la forme matricielle de la suppression (`delete []`) sur un objet et l'utilisation de la forme non-array de la suppression sur un tableau.

Exemple

Pour obtenir des exemples d'utilisation `delete` de , consultez nouvel opérateur.

Fonctionnement de delete

L'opérateur delete appelle **l'opérateur de fonction delete**.

Pour les objets qui ne sont pas de type classe (classe, struct ou union), l'opérateur de suppression globale est appelé. Pour les objets de type classe, le nom de la fonction de désallocation est résolu dans l'étendue globale si l'expression delete commence par l'opérateur de résolution d'étendue unaire (`::`). Sinon, l'opérateur delete appelle le destructeur pour un objet avant de désallouer la mémoire (si le pointeur n'est pas null). L'opérateur delete peut être défini pour chaque classe ; si cette définition n'existe pas pour une classe donnée, l'opérateur de suppression global est appelé. Si l'expression de suppression est utilisée pour désallouer un objet de classe dont le type statique a un destructeur virtuel, la fonction de désallocation est résolue par l'intermédiaire du destructeur virtuel du type dynamique de l'objet.

Voir aussi

Expressions avec opérateurs unaires
Mots clés
opérateurs nouveaux et supprimés

Opérateurs d’égalité : == et !=

Article • 16/06/2023

Syntaxe

> *Expression* `==` *Expression*
> *Expression* `!=` *Expression*

Notes

Les opérateurs d'égalité binaires comparent l'égalité ou l'inégalité stricte de leurs opérandes.

Les opérateurs d'égalité, égal à (`==`) et différent de (`!=`), ont une priorité inférieure aux opérateurs relationnels, mais ils se comportent de la même manière. Le type de résultat pour ces opérateurs est `bool`.

L’opérateur égal à (`==`) retourne `true` si les deux opérandes ont la même valeur ; sinon, il retourne `false`. L’opérateur non égal à (`!=`) retourne `true` si les opérandes n’ont pas la même valeur ; sinon, il retourne `false`.

Opérateur mot clé pour !=

C++ spécifie `not_eq` comme une autre orthographe pour `!=`. (Il n’existe pas d’autre orthographe pour `==`.) En C, l’orthographe alternative est fournie sous forme de macro dans l’en-tête <iso646.h> . En C++, l’autre orthographe est un mot clé ; l’utilisation d’iso646.h><ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l’option du /permissive- compilateur ou /Za est requise pour activer l’orthographe alternative.

Exemple

C++

```
// expre_Equality_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
```

```
int main() {
   cout  << boolalpha
         << "The true expression 3 != 2 yields: "
         << (3 != 2) << endl
         << "The false expression 20 == 10 yields: "
         << (20 == 10) << endl;
}
```

Les opérateurs d'égalité permettent de comparer les pointeurs à des membres du même type. Dans une telle comparaison, des conversions de pointeur à membre sont effectuées. Les conversions de pointeur vers membre peuvent également être comparées à une expression constante qui a pour valeur 0.

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés C++, priorité ; et l'associativité
Opérateurs relationnels et d'égalité C

Opérateur de conversions de type explicite : ()

Article • 03/04/2023

C++ permet de convertir en type explicite à l'aide d'une syntaxe similaire à la syntaxe de l'appel de fonction.

Syntaxe

```
simple-type-name ( expression-list )
```

Notes

Un *simple-type-name* suivi d'une *expression-list* entre parenthèses construit un objet du type spécifié à l'aide des expressions spécifiées. L'exemple suivant montre une conversion de type explicite en type int :

C++

```
int i = int( d );
```

L'exemple suivant montre une `Point` classe .

Exemple

C++

```
// expre_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }
```

```
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show()   { cout << "x = " << _x << ", "
                         << "y = " << _y << "\n"; }
private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    //  of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    //  conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}
```

Sortie

Output

```
x = 20, y = 10
x = 0, y = 0
```

Bien que l'exemple précédent montre la conversion de type explicite à l'aide de constantes, la même technique fonctionne pour exécuter ces conversions sur des objets. Le fragment de code suivant illustre ceci :

C++

```
int i = 7;
float d;

d = float( i );
```

Les conversions de types explicites peuvent également être spécifiées en utilisant la syntaxe du cast. L'exemple précédent, réécrit à l'aide de la syntaxe du cast, est :

```
C++

d = (float)i;
```

Les conversions de style cast et fonction ont les mêmes résultats en convertissant à partir de valeurs uniques. Toutefois, dans la syntaxe de style fonction, vous pouvez spécifier plusieurs arguments pour la conversion. Cette différence est importante pour les types définis par l'utilisateur. Prenons une classe `Point` et ses conversions :

```
C++

struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};
...
Point pt = Point( 3, 10 );
```

L'exemple précédent, qui utilise la conversion de style de fonction, montre comment convertir deux valeurs (une pour *x* et une pour *y*) en type `Point` défini par l'utilisateur .

> **⊗ Attention**
>
> Utilisez les conversions de type explicites avec précaution, car elles remplacent la vérification de type intégrée du compilateur C++.

La notation de cast doit être utilisée pour les conversions vers des types qui n'ont pas de *nom-type simple* (types pointeur ou référence, par exemple). La conversion en types qui peuvent être exprimés avec un *nom de type simple* peut être écrite sous l'une ou l'autre des formes.

La définition de type avec des casts n'est pas conforme.

Voir aussi

Expressions postfixées
Opérateurs intégrés, priorité et associativité C++

Opérateur d'appel de fonction : ()

Article • 16/06/2023

Un appel de fonction est une sorte de *postfix-expression*, formé par une expression qui s'évalue à une fonction ou à un objet callable suivi de l'opérateur d'appel de fonction, `()`. Un objet peut déclarer une `operator ()` fonction, ce qui fournit la sémantique de l'appel de fonction pour l'objet.

Syntaxe

> *postfix-expression*:
>
> *postfix-expression* `(` *argument-expression-list* opt `)`

Notes

Les arguments de l'opérateur d'appel de fonction proviennent d'une liste d'expressions *argument-expression-list* séparée par des virgules. Les valeurs de ces expressions sont transmises à la fonction en tant qu'arguments. *L'argument-expression-list* peut être vide. Avant C++ 17, l'ordre d'évaluation de l'expression de fonction et des expressions d'argument n'est pas spécifié et peut se produire dans n'importe quel ordre. En C++17 et versions ultérieures, l'expression de fonction est évaluée avant toute expression d'argument ou argument par défaut. Les expressions d'argument sont évaluées dans une séquence indéterminée.

correspond *postfix-expression* à la fonction à appeler. Il peut prendre plusieurs formes :

- identificateur de fonction, visible dans l'étendue actuelle ou dans l'étendue de l'un des arguments de fonction fournis,
- expression qui prend la valeur d'une fonction, d'un pointeur de fonction, d'un objet pouvant être appelé ou d'une référence à un,
- un accesseur de fonction membre, explicite ou implicite,
- pointeur déréférencé vers une fonction membre.

Il peut s'agir *postfix-expression* d'un identificateur de fonction surchargé ou d'un accesseur de fonction membre surchargé. Les règles de résolution de surcharge déterminent la fonction réelle à appeler. Si la fonction membre est virtuelle, la fonction à appeler est déterminée au moment de l'exécution.

Voici quelques exemples de déclarations :

- Fonction retournant le type `T`. Voici un exemple de déclaration

```
C++

T func( int i );
```

- Pointeur d'une fonction qui retourne le type `T`. Voici un exemple de déclaration

```
C++

T (*func)( int i );
```

- Référence à une fonction qui retourne le type `T`. Voici un exemple de déclaration

```
C++

T (&func)(int i);
```

- Déréférencement de fonction de pointeur de membre qui retourne le type `T`. Voici des exemples d'appels de fonction

```
C++

(pObject->*pmf)();
(Object.*pmf)();
```

Exemple

L'exemple suivant appelle la fonction de bibliothèque standard `strcat_s` avec trois arguments :

```
C++

// expre_Function_Call_Operator.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library name space
using namespace std;

int main()
{
    enum
```

```
    {
        sizeOfBuffer = 20
    };

    char s1[ sizeOfBuffer ] = "Welcome to ";
    char s2[ ] = "C++";

    strcat_s( s1, sizeOfBuffer, s2 );

    cout << s1 << endl;
}
```

Output

```
Welcome to C++
```

Résultats de l'appel de fonction

Un appel de fonction prend la valeur rvalue, sauf si la fonction est déclarée en tant que type de référence. Les fonctions avec des types de retour de référence sont évaluées sur lvalues. Ces fonctions peuvent être utilisées sur le côté gauche d'une instruction d'affectation, comme indiqué ici :

C++

```
// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y();  // Use y() as an r-value.

    // Use x() and y() as r-values.
```

```
    cout << "x = " << ThePoint.x() << "\n"
         << "y = " << ThePoint.y() << "\n";
}
```

Le code précédent définit une classe appelée `Point`, qui contient des objets de données privés qui représentent des coordonnées *x* et *y* . Ces objets de données doivent être modifiés et leurs valeurs, récupérées. Ce programme n'est qu'une des nombreuses conceptions de ce type de classe ; l'utilisation des fonctions `GetX` et `SetX` ou `GetY` et `SetY` est une autre conception possible.

Les fonctions qui retournent des types de classe, des pointeurs vers des types de classe ou des références à des types de classe peuvent être utilisées comme opérande gauche pour les opérateurs de sélection de membres. Le code suivant est légal :

C++

```
// expre_Function_Results2.cpp
class A {
public:
   A() {}
   A(int i) {}
   int SetA( int i ) {
      return (I = i);
   }

   int GetA() {
      return I;
   }

private:
   int I;
};

A func1() {
   A a = 0;
   return a;
}

A* func2() {
   A *a = new A();
   return a;
}

A& func3() {
   A *a = new A();
   A &b = *a;
   return b;
}

int main() {
   int iResult = func1().GetA();
```

```
    func2()->SetA( 3 );
    func3().SetA( 7 );
}
```

Les fonctions peuvent être appelées de manière récursive. Pour plus d'informations sur les déclarations de fonction, consultez Fonctions. Le matériel connexe se trouve dans unités de traduction et liaison.

Voir aussi

Expressions suffixées
Opérateurs intégrés C++, priorité et associativité
Appel de fonction

Opérateur d'indirection : *

Article • 16/06/2023

Syntaxe

```
* cast-expression
```

Notes

L'opérateur indirection unaire (*) déréférence un pointeur ; autrement dit, il convertit une valeur de pointeur en valeur l. L'opérande de l'opérateur d'indirection doit être un pointeur vers un type. Le résultat de l'expression d'indirection est le type à partir duquel le type pointeur est dérivé. L'utilisation de l'opérateur * dans ce contexte est différente de sa signification en tant qu'opérateur binaire, qui est la multiplication.

Si l'opérande pointe vers une fonction, le résultat est un désignateur de fonction. S'il pointe vers un emplacement de stockage, le résultat est une l-value désignant l'emplacement de stockage.

L'opérateur d'indirection peut être utilisé cumulativement pour déréférencer les pointeurs vers des pointeurs. Par exemple :

C++

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
   int n = 5;
   int *pn = &n;
   int **ppn = &pn;

   cout  << "Value of n:\n"
         << "direct value: " << n << endl
         << "indirect value: " << *pn << endl
         << "doubly indirect value: " << **ppn << endl
         << "address of n: " << pn << endl
         << "address of n via indirection: " << *ppn << endl;
}
```

Si la valeur du pointeur n'est pas valide, le résultat n'est pas défini. La liste ci-dessous inclut certaines des conditions les plus courantes qui invalident une valeur de pointeur.

- Le pointeur est un pointeur null.
- Le pointeur spécifie l'adresse d'un élément local qui n'est pas visible au moment de la référence.
- Le pointeur spécifie une adresse alignée de façon inappropriée pour le type de l'objet désigné.
- Le pointeur spécifie une adresse non utilisée par le programme en cours d'exécution.

Voir aussi

Expressions avec opérateurs unaires
Opérateurs intégrés, priorité et associativité C++
Adresse de l’opérateur : &
Indirection et adresse des opérateurs

Opérateurs maj gauche et décalage droit (`<<` et `>>`)

Article • 14/03/2023

Les opérateurs de décalage au niveau du bit sont l'opérateur de décalage vers la droite (`>>`), qui déplace les bits d'une expression de type entier ou de type énumération vers la droite, et l'opérateur de décalage de gauche (`<<`), qui déplace les bits vers la gauche. [1]

Syntaxe

`shift-expression`:

`additive-expression`

`shift-expression` `<<` *`additive-expression`*

`shift-expression` `>>` *`additive-expression`*

Notes

> **ⓘ Important**
>
> Les descriptions et exemples suivants sont valides sur Windows pour les architectures x86 et x64. L'implémentation des opérateurs de décalage gauche et droit est sensiblement différente sur Windows pour les appareils ARM. Pour plus d'informations, consultez la section « Opérateurs de décalage » du billet de blog Hello ARM .

Décalages vers la gauche

L'opérateur de décalage de gauche entraîne le décalage des bits vers *`shift-expression`* la gauche par le nombre de positions spécifié par *`additive-expression`*. Les positions de bits qui ont été libérées par l'opération de décalage sont remplies de zéros. Un décalage vers la gauche est un décalage logique (les bits qui sont déplacés après la fin sont ignorés, y compris le bit de signe). Pour plus d'informations sur les types de décalages au niveau du bit, consultez Décalages au niveau du bit .

L'exemple suivant illustre les opérations de décalage vers la gauche à l'aide de nombres non signés. L'exemple montre ce qui arrive aux bits en représentant la valeur en tant que bitset. Pour plus d'informations, consultez Bitset Class.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};   // the bitset representation of 4
    cout << bitset1 << endl;  // 0b00000000'00000100

    unsigned short short2 = short1 << 1;     // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;  // 0b00000000'00001000

    unsigned short short3 = short1 << 2;     // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;  // 0b00000000'00010000
}
```

Si vous décalez vers la gauche un nombre signé de manière à affecter le bit de signe, le résultat n'est pas défini. L'exemple suivant montre ce qui se passe lorsqu'un bit 1 est déplacé vers la gauche dans la position de bit de signe.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;  // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3);  // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl;  // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4);  // 4 left-shifted by 14 = 0
    cout << bitset4 << endl;  // 0b00000000'00000000
}
```

Décalages vers la droite

L’opérateur de décalage droit entraîne le décalage du modèle de bits dans *shift-expression* vers la droite par le nombre de positions spécifié par *additive-expression*. Pour les nombres non signés, les positions de bit qui ont été libérées par l'opération de décalage sont remplies de zéros. Pour les nombres signés, le bit de signe est utilisé pour remplir les positions de bit libérées. En d'autres termes, si le nombre est positif, 0 est utilisé, et si le nombre est négatif, 1 est utilisé.

> **ⓘ Important**
>
> Le résultat d'un décalage vers la droite d'une nombre négatif signé dépend de l'implémentation. Bien que le compilateur Microsoft C++ utilise le bit de signe pour remplir les positions de bits libérées, il n’est pas garanti que d’autres implémentations le fassent également.

Cet exemple illustre les opérations de décalage vers la droite à l'aide de nombres non signés :

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl;     // 0b00000100'00000000

    unsigned short short12 = short11 >> 1;  // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl;     // 0b00000010'00000000

    unsigned short short13 = short11 >> 10;  // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl;     // 0b00000000'00000001

    unsigned short short14 = short11 >> 11;  // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl;     // 0b00000000'00000000
}
```

L'exemple suivant illustre les opérations de décalage vers la droite avec des nombres positifs signés.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;     // 0b00000100'00000000

    short short2 = short1 >> 1;  // 512
    bitset<16> bitset2(short2);
    cout << bitset2 << endl;     // 0b00000010'00000000

    short short3 = short1 >> 11;  // 0
    bitset<16> bitset3(short3);
    cout << bitset3 << endl;     // 0b00000000'00000000
}
```

L'exemple suivant illustre les opérations de décalage vers la droite avec des entiers négatifs signés.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl;  // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl;  // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl;  // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
    cout << bn4 << endl;  // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl;  // 0b11111111'11111111
}
```

Shifts et promotions

Les expressions des deux côtés de l'opérateur de décalage doivent être de type intégral. Les promotions intégrales sont effectuées conformément aux règles décrites dans Conversions standard. Le type du résultat est le même que le type du promu *`shift-expression`*.

Dans l'exemple suivant, une variable de type **`char`** est promue en **`int`**.

C++

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1;   // 194
    cout << typeid(promoted1).name() << endl;  // int

    auto promoted2 = char1 << 10;  // 99328
    cout << typeid(promoted2).name() << endl;  // int
}
```

Détails

Le résultat d'une opération de décalage n'est pas défini si *`additive-expression`* est négatif ou si *`additive-expression`* est supérieur ou égal au nombre de bits dans (promu) *`shift-expression`*. Aucune opération de shift n'a lieu si *`additive-expression`* est 0.

C++

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl;    // 0b00000000'00000000'00000000'00000100

    unsigned int int2 = int1 << -3;  // C4293: '<<' : shift count negative
or too big, undefined behavior
    unsigned int int3 = int1 >> -3;  // C4293: '>>' : shift count negative
```

```
or too big, undefined behavior
    unsigned int int4 = int1 << 32;  // C4293: '<<' : shift count negative
or too big, undefined behavior
    unsigned int int5 = int1 >> 32;  // C4293: '>>' : shift count negative
or too big, undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl;    // 0b00000000'00000000'00000000'00000100 (no
change)
}
```

Notes de bas de page

[1] Voici la description des opérateurs de décalage dans la spécification ISO C++11 (INCITS/ISO/IEC 14882-2011[2012]), sections 5.8.2 et 5.8.3.

La valeur de `E1 << E2` est `E1` décalée vers la gauche de positions de bit `E2` ; les bits libérés sont remplis de zéros. Si `E1` a un type non signé, la valeur du résultat est **E1 × 2^{E2}**, réduit modulo un de plus que la valeur maximale pouvant être représentée dans le type de résultat. Sinon, si `E1` a un type signé et une valeur non négative, et **si E1 ×$^{2\ E2}$** est représentable dans le type non signé correspondant du type de résultat, cette valeur, convertie en type de résultat, est la valeur résultante ; sinon, le comportement n'est pas défini.

La valeur de `E1 >> E2` est `E1` décalée vers la droite de positions de bit `E2`. Si `E1` a un type non signé ou si `E1` a un type signé et une valeur non négative, la valeur du résultat fait partie intégrante du quotient de **E1/2^{E2}**. Si `E1` a un type signé et une valeur négative, la valeur résultante est définie par l'implémentation.

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés C++, priorité et associativité

Opérateur AND logique : `&&`

Article • 16/06/2023

Syntaxe

`logical-and-expression`:

`equality-expression`

`logical-and-expression` **`&&`** *`equality-expression`*

Notes

L'opérateur AND logique (**`&&`**) retourne **`true`** si les deux opérandes sont **`true`** et retourne **`false`** sinon. Les opérandes sont implicitement convertis en type **`bool`** avant l'évaluation, et le résultat est de type **`bool`**. L'opérateur logique présente une associativité de gauche à droite.

Les opérandes de l'opérateur AND logique n'ont pas besoin d'avoir le même type, mais ils doivent avoir un type booléen, intégral ou pointeur. Les opérandes sont souvent des expressions relationnelles ou d'égalité.

Le premier opérande est entièrement évalué et tous les effets secondaires sont terminés avant que l'évaluation de l'expression AND logique se poursuive.

Le deuxième opérande n'est évalué que si le premier opérande est évalué à **`true`** (non nul). Cette évaluation élimine l'évaluation inutile du deuxième opérande lorsque l'expression AND logique est **`false`**. Vous pouvez utiliser cette évaluation de court-circuit pour empêcher le déréférencement du pointeur NULL, comme indiqué dans l'exemple suivant :

C++

```
char *pch = 0;
// ...
(pch) && (*pch = 'a');
```

Si `pch` est null (0), le côté droit de l'expression n'est jamais évalué. Cette évaluation de court-circuit rend l'affectation par le biais d'un pointeur null impossible.

Opérateur mot clé pour `&&`

C++ spécifie `and` comme une autre orthographe pour `&&`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'autre orthographe est un mot clé ; l'utilisation d'iso646.h><ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Logical_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate logical AND
#include <iostream>

using namespace std;

int main() {
   int a = 5, b = 10, c = 15;
   cout  << boolalpha
         << "The true expression "
         << "a < b && b < c yields "
         << (a < b && b < c) << endl
         << "The false expression "
         << "a > b && b < c yields "
         << (a > b && b < c) << endl;
}
```

Voir aussi

Opérateurs intégrés C++, priorité et associativité
Opérateurs logiques C

Opérateur de négation logique : !

Article • 16/06/2023

Syntaxe

> `!` *cast-expression*

Notes

L'opérateur de négation logique (`!`) inverse la signification de son opérande. L'opérande doit être de type arithmétique ou pointeur (ou une expression qui a pour valeur le type arithmétique ou pointeur). L'opérande est implicitement converti en type `bool`. Le résultat est `true` si l'opérande converti est `false`; le résultat est `false` si l'opérande converti est `true`. Le résultat est de type `bool`.

Pour une expression `e`, l'expression `!e` unaire équivaut à l'expression `(e == 0)`, sauf lorsque des opérateurs surchargés sont impliqués.

Opérateur mot clé pour !

C++ spécifie `not` comme une autre orthographe pour `!`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'autre orthographe est un mot clé ; l'utilisation d'iso646.h><ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

Voir aussi

Expressions avec opérateurs unaires
Opérateurs intégrés C++, priorité et associativité
Opérateurs arithmétiques unaires

Opérateur OR logique : `||`

Article • 16/06/2023

Syntaxe

> *logical-or-expression* `||` *logical-and-expression*

Notes

L'opérateur OR logique (`||`) retourne la valeur `true` booléenne si l'un ou les deux opérandes est `true` et retourne `false` dans le cas contraire. Les opérandes sont implicitement convertis en type `bool` avant l'évaluation, et le résultat est de type `bool`. L'opérateur OR logique présente une associativité de gauche à droite.

Les opérandes de l'opérateur OR logique n'ont pas besoin d'avoir le même type, mais ils doivent être de type booléen, intégral ou pointeur. Les opérandes sont souvent des expressions relationnelles ou d'égalité.

Le premier opérande doit être complètement évalué et tous les effets secondaires terminés pour que l'évaluation de l'expression OR logique se poursuive.

Le deuxième opérande est évalué uniquement si le premier opérande prend la valeur `false`, car l'évaluation n'est pas nécessaire lorsque l'expression OR logique est `true`. C'est ce qu'on appelle l'évaluation *de court-circuit* .

```
C++
```

```
printf( "%d" , (x == w || x == y || x == z) );
```

Dans l'exemple ci-dessus, si `x` est égal à `w`, `y` ou `z`, le deuxième argument de la `printf` fonction est évalué à `true`, qui est ensuite promu en entier, et la valeur 1 est imprimée. Sinon, elle correspond à `false` et la valeur 0 est imprimée. Dès que l'une des conditions est évaluée à , l'évaluation s'arrête `true`.

Opérateur mot clé pour `||`

C++ spécifie `or` comme une autre orthographe pour `||`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'autre orthographe est un mot clé ; l'utilisation d'iso646.h><ou de ciso646> équivalent <C++

est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

C++

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
   int a = 5, b = 10, c = 15;
   cout  << boolalpha
         << "The true expression "
         << "a < b || b > c yields "
         << (a < b || b > c) << endl
         << "The false expression "
         << "a > b || b > c yields "
         << (a > b || b > c) << endl;
}
```

Voir aussi

Opérateurs intégrés C++, priorité et associativité

Opérateurs logiques C

Opérateurs d’accès aux membres : `.` et `->`

Article • 16/06/2023

Syntaxe

`postfix-expression`:

`postfix-expression` **`.`** **`template`**$_{opt}$ *`id-expression`*

`postfix-expression` **`->`** **`template`**$_{opt}$ *`id-expression`*

Notes

Les opérateurs `.` d’accès membres et `->` sont utilisés pour faire référence aux membres des `struct` types , `union` et `class` . Les expressions d’accès au membre ont la valeur et le type du membre sélectionné.

Il existe deux formes d’expressions d’accès au membre :

1. Dans le premier formulaire, *`postfix-expression`* représente une valeur de `struct` type , `class` ou `union` , et *`id-expression`* nomme un membre du spécifié `struct`, `union` ou `class`. La valeur de l’opération est celle de *`id-expression`*. Il s’agit d’une l-value si *`postfix-expression`* est une l-value.

2. Dans le deuxième formulaire, *`postfix-expression`* représente un pointeur vers un `struct`, `union` ou `class`, et *`id-expression`* nomme un membre du spécifié `struct`, `union` ou `class`. La valeur est celle de `id-expression`. Il s’agit d’une l-value. L’opérateur `->` déréférence le pointeur. Les expressions `e->member` et `(*(e)).member` (où *`e`* représente un pointeur) produisent des résultats identiques (sauf lorsque les opérateurs `->` ou `*` sont surchargés).

Exemple

L'exemple suivant illustre les deux formes de l'opérateur d'accès aux membres.

C++

```
// expre_Selection_Operator.cpp
// compile with: /EHsc
```

```
#include <iostream>
using namespace std;

struct Date {
   Date(int i, int j, int k) : day(i), month(j), year(k){}
   int month;
   int day;
   int year;
};

int main() {
   Date mydate(1,1,1900);
   mydate.month = 2;
   cout  << mydate.month << "/" << mydate.day
         << "/" << mydate.year << endl;

   Date *mydate2 = new Date(1,1,2000);
   mydate2->month = 2;
   cout  << mydate2->month << "/" << mydate2->day
         << "/" << mydate2->year << endl;
   delete mydate2;
}
```

Output

```
2/1/1900
2/1/2000
```

Voir aussi

Expressions suffixées

Opérateurs intégrés C++, priorité et associativité

Classes et structs

Membres de la structure et de l'union

Opérateurs de multiplication et opérateur modulo

Article • 16/06/2023

Syntaxe

```
expression * expression
expression / expression
expression % expression
```

Notes

Les opérateurs de multiplication sont :

- Multiplication (*)
- Division (/)
- Module (reste de la division) (%)

Ces opérateurs binaires ont une associativité de droite à gauche.

Les opérateurs de multiplication prennent des opérandes de types arithmétiques. L'opérateur de module (%) a une exigence plus stricte en ce que ses opérandes doivent être de type intégral. (Pour obtenir le reste d'une division à virgule flottante, utilisez la fonction d'exécution, fmod.) Les conversions couvertes dans Conversions standard sont appliquées aux opérandes, et le résultat est du type converti.

L'opérateur de multiplication montre le résultat de la multiplication du premier opérande par le second.

L'opérateur de division montre le résultat de la division du premier opérande par le second.

L'opérateur de module génère le reste donné par l'expression suivante, où *e1* est le premier opérande et *e2* le second : *e1* - (*e1* / *e2*) * *e2*, où les deux opérandes sont de types intégraux.

La division par 0 dans une division ou une expression de modulo est éliminée et provoque une erreur d'exécution. Par conséquent, les expressions suivantes génèrent des résultats indéterminés et erronés :

```
C++
```

```
i % 0
f / 0.0
```

Si les deux opérandes pour une expression de multiplication, de division ou de modulo ont le même signe, le résultat est positif. Sinon, le résultat est négative. Le résultat du signe d'une opération modulo est défini par l'implémentation.

> ⓘ **Notes**
>
> Étant donné que les conversions exécutées par les opérateurs de multiplication ne fournissent pas de conditions de dépassement de capacité positif ou de dépassement de capacité négatif, les informations peuvent être perdues si le résultat d'une opération de multiplication ne peut pas être représenté dans le type des opérandes après conversion.

Section spécifique à Microsoft

Dans Microsoft C++, le résultat d'une expression de modulo est toujours le même que le signe du premier opérande.

FIN de la section spécifique à Microsoft

Si la division calculée de deux entiers est incorrecte et qu'un seul opérande est négatif, le résultat est le plus grand entier (en amplitude, sans tenir compte du signe) qui est inférieur à la valeur exacte que l'opérateur de division produirait. Par exemple, la valeur calculée de -11 / 3 est -3,6666666666. Le résultat de cette division intégrale est -3.

La relation entre les opérateurs multiplicatifs est donnée par l'identité (*e1* / *e2*) * *e2* + *e1* % *e2* == *e1* *e1*.

Exemple

Le programme suivant montre les opérateurs de multiplication. Notez que l'un ou l'autre opérande de `10 / 3` doit être explicitement casté en type **`float`** pour éviter la troncation afin que les deux opérandes soient de type **`float`** avant la division.

```
```

C++

```
// expre_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
   int x = 3, y = 6, z = 10;
   cout  << "3 * 6 is " << x * y << endl
         << "6 / 3 is " << y / x << endl
         << "10 % 3 is " << z % x << endl
         << "10 / 3 is " << (float) z / x << endl;
}
```

Voir aussi

Expressions avec opérateurs binaires

Opérateurs intégrés, priorité et associativité C++

Opérateurs de multiplication C

`new` opérateur (C++)

Article • 16/06/2023

Tente d'allouer et d'initialiser un objet ou un tableau d'objets d'un type spécifié ou d'espace réservé, et retourne un pointeur de type différent de zéro vers l'objet (ou vers l'objet initial du tableau).

Syntaxe

new-expression:

- `::`opter `new` *new-placement*opter *new-type-id* *new-initializer*opter
- `::`opter `new` *new-placement*opter `(` *type-id* `)` *new-initializer*opter

new-placement:

- `(` *expression-list* `)`

new-type-id:

- *type-specifier-seq* *new-declarator*opt

new-declarator:

- *ptr-operator* *new-declarator*opt
- *noptr-new-declarator*

noptr-new-declarator:

- `[` *expression* `]` *attribute-specifier-seq*opt
- *noptr-new-declarator* `[` *constant-expression* `]` *attribute-specifier-seq*opter

new-initializer:

- `(` *expression-list*opt `)`
- *braced-init-list*

Notes

En cas d'échec, `new` retourne zéro ou lève une exception. Pour plus d'informations, consultez Opérateurs new et delete. Vous pouvez modifier ce comportement par défaut en écrivant une routine de gestion des exceptions personnalisée et en appelant la _set_new_handler fonction de bibliothèque d'exécution avec le nom de votre fonction comme argument.

Pour plus d'informations sur la création d'un objet sur le tas managé dans C++/CLI et C++/CX, consultez gcnew.

> ⓘ **Notes**
>
> Les extensions de composant Microsoft C++ (C++/CX) prennent en charge la `new` mot clé d'ajouter des entrées d'emplacement de table virtuelle. Pour plus d'informations, consultez **new (nouvel emplacement dans vtable)**

Quand `new` est utilisé pour allouer de la mémoire pour un objet de classe C++, le constructeur de l'objet est appelé après l'allocation de la mémoire.

Utilisez l'opérateur delete pour libérer la mémoire allouée par l'opérateur `new` . Utilisez l'opérateur `delete[]` pour supprimer un tableau alloué par l'opérateur `new` .

L'exemple suivant alloue, puis libère un tableau de caractères à deux dimensions de taille `dim` par 10. Lors de l'allocation d'un tableau multidimensionnel, toutes les dimensions à l'exception de la première doivent être des expressions constantes qui s'évaluent à des valeurs positives. La dimension de tableau la plus à gauche peut être n'importe quelle expression qui est évaluée à une valeur positive. Lors de l'allocation d'un tableau à l'aide de l'opérateur `new` , la première dimension peut être égale à zéro ; l'opérateur `new` retourne un pointeur unique.

```
C++
```

```
char (*pchar)[10] = new char[dim][10];
delete [] pchar;
```

Ne *`type-id`* peut pas contenir `const`, `volatile`, les déclarations de classe ou les déclarations d'énumération. L'expression suivante est mal formée :

```
C++
```

```
volatile char *vch = new volatile char[20];
```

L'opérateur `new` n'alloue pas de types de référence, car il ne s'agit pas d'objets.

L'opérateur `new` ne peut pas être utilisé pour allouer une fonction, mais il peut être utilisé pour allouer des pointeurs à des fonctions. L'exemple suivant alloue, puis libère un tableau de sept pointeurs vers des fonctions qui retournent des entiers.

```
C++
```

```
int (**p) () = new (int (*[7]) ());
delete p;
```

Si vous utilisez l'opérateur **`new`** sans arguments supplémentaires et que vous compilez avec l'option /GX, /EHaou /EHs , le compilateur génère du code pour appeler l'opérateur **`delete`** si le constructeur lève une exception.

La liste suivante décrit les éléments de grammaire de **`new`**:

`new-placement`
Fournit un moyen de passer des arguments supplémentaires si vous surchargez **`new`**.

`type-id`
Spécifie le type à allouer ; il peut s'agir d'un type intégré ou défini par l'utilisateur. Si la spécification de type est compliquée, elle peut être placée entre parenthèses pour forcer l'ordre de liaison. Le type peut être un espace réservé (**`auto`**) dont le type est déterminé par le compilateur.

`new-initializer`
Fournit une valeur pour l'objet initialisé. Les initialiseurs ne peuvent pas être spécifiés pour les tableaux. L'opérateur **`new`** crée des tableaux d'objets uniquement si la classe a un constructeur par défaut.

`noptr-new-declarator`
Spécifie les limites d'un tableau. Lors de l'allocation d'un tableau multidimensionnel, toutes les dimensions à l'exception de la première doivent être des expressions constantes qui s'évaluent en valeurs positives convertibles en `std::size_t`. La dimension de tableau la plus à gauche peut être n'importe quelle expression qui est évaluée à une valeur positive. s'applique *`attribute-specifier-seq`* au type de tableau associé.

Exemple : Allouer et libérer un tableau de caractères

L'exemple de code suivant alloue un tableau de caractères et un objet de classe `CName`, puis les libère.

C++

```
// expre_new_Operator.cpp
// compile with: /EHsc
#include <string.h>
```

```
class CName {
public:
   enum {
      sizeOfBuffer = 256
   };

   char m_szFirst[sizeOfBuffer];
   char m_szLast[sizeOfBuffer];

public:
   void SetName(char* pszFirst, char* pszLast) {
     strcpy_s(m_szFirst, sizeOfBuffer, pszFirst);
     strcpy_s(m_szLast, sizeOfBuffer, pszLast);
   }

};

int main() {
   // Allocate memory for the array
   char* pCharArray = new char[CName::sizeOfBuffer];
   strcpy_s(pCharArray, CName::sizeOfBuffer, "Array of characters");

   // Deallocate memory for the array
   delete [] pCharArray;
   pCharArray = NULL;

   // Allocate memory for the object
   CName* pName = new CName;
   pName->SetName("Firstname", "Lastname");

   // Deallocate memory for the object
   delete pName;
   pName = NULL;
}
```

Exemple : `new` opérateur

Si vous utilisez le formulaire de placement de l'opérateur **`new`** (le formulaire avec plus d'arguments que la taille), le compilateur ne prend pas en charge un formulaire de placement de l'opérateur **`delete`** si le constructeur lève une exception. Par exemple :

C++

```
// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
   A(int) { throw "Fail!"; }
};
void F(void) {
   try {
```

```
      // heap memory pointed to by pa1 will be deallocated
      // by calling ::operator delete(void*).
      A* pa1 = new A(10);
   } catch (...) {
   }
   try {
      // This will call ::operator new(size_t, char*, int).
      // When A::A(int) does a throw, we should call
      // ::operator delete(void*, char*, int) to deallocate
      // the memory pointed to by pa2.  Since
      // ::operator delete(void*, char*, int) has not been implemented,
      // memory will be leaked when the deallocation can't occur.

      A* pa2 = new(__FILE__, __LINE__) A(20);
   } catch (...) {
   }
}

int main() {
   A a;
}
```

Initialisation des objets alloués avec `new`

Un champ facultatif *`new-initializer`* est inclus dans la grammaire de l'opérateur **`new`** . Ce champ permet d'initialiser de nouveaux objets avec des constructeurs définis par l'utilisateur. Pour plus d'informations sur la façon dont l'initialisation est effectuée, consultez Initialiseurs. L'exemple suivant montre comment utiliser une expression d'initialisation avec l'opérateur **`new`** :

C++

```
// expre_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    //  an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }
private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double { 43.0 };
```

```
    // ...
}
```

Dans cet exemple, l'objet `CheckingAcct` est alloué à l'aide de l'opérateur **`new`** , mais aucune initialisation par défaut n'est spécifiée. Par conséquent, le constructeur par défaut de la classe , `Acct()`, est appelé. Ensuite, l'objet `SavingsAcct` est alloué de la même façon, sauf qu'il est explicitement initialisé à 34.98. Étant donné que 34.98 est de type **`double`**, le constructeur qui prend un argument de ce type est appelé pour gérer l'initialisation. Enfin, le type `HowMuch` non-classe est initialisé à 43.0.

Si un objet est d'un type de classe et que cette classe possède des constructeurs (comme dans l'exemple précédent), l'objet ne peut être initialisé par l'opérateur que si l'une **`new`** de ces conditions est remplie :

- Les arguments fournis dans l'initialiseur correspondent aux arguments d'un constructeur.
- La classe possède un constructeur par défaut (un constructeur pouvant être appelé sans argument).

L'initialisation explicite par élément ne peut pas être effectuée lors de l'allocation de tableaux à l'aide de l'opérateur **`new`** ; seul le constructeur par défaut, le cas échéant, est appelé. Pour plus d'informations, consultez Arguments par défaut.

Si l'allocation de mémoire échoue (**`operator new`** retourne la valeur 0), aucune initialisation n'est effectuée. Ce comportement protège contre les tentatives d'initialisation des données qui n'existent pas.

Comme pour les appels de fonction, l'ordre dans lequel les expressions initialisées sont évaluées n'est pas défini. En outre, vous ne devez pas compter sur l'évaluation complète de ces expressions avant que l'allocation de mémoire ait lieu. Si l'allocation de mémoire échoue et que l'opérateur **`new`** retourne zéro, certaines expressions de l'initialiseur peuvent ne pas être évaluées complètement.

Durée de vie des objets alloués avec **`new`**

Les objets alloués avec l'opérateur **`new`** ne sont pas détruits lorsque l'étendue dans laquelle ils sont définis est quittée. Étant donné que l'opérateur **`new`** retourne un pointeur vers les objets qu'il alloue, le programme doit définir un pointeur avec une étendue appropriée pour accéder à ces objets et les supprimer. Par exemple :

```
C++
```

```
// expre_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        //  another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
    }

    delete [] AnotherArray; // Error: pointer out of scope.
    delete [] AnArray;      // OK: pointer still in scope.
}
```

Une fois que le pointeur `AnotherArray` sort de la portée de l'exemple, l'objet ne peut plus être supprimé.

Fonctionnement `new`

Le *`new-expression`* (l'expression contenant l'opérateur **`new`**) effectue trois opérations :

- Localise et réserve le stockage pour les objets à allouer. Une fois cette étape terminée, la quantité de stockage appropriée est allouée, mais il ne s'agit pas encore d'un objet.
- Initialise les objets. Une fois que l'initialisation est terminée, les informations disponibles sont suffisantes pour que le stockage alloué soit un objet.
- Retourne un pointeur vers les objets d'un type de pointeur dérivé de *`new-type-id`* ou *`type-id`*. Le programme utilise ce pointeur pour accéder au nouvel objet alloué.

L'opérateur **`new`** appelle la fonction **`operator new`**. Pour les tableaux de tout type et pour les objets qui ne sont pas **`class`** des types , **`struct`** ou **`union`** , une fonction globale, **`::operator new`**, est appelée pour allouer du stockage. Les objets de type classe peuvent définir leur propre **`operator new`** fonction membre statique par classe.

Lorsque le compilateur rencontre l'opérateur **`new`** pour allouer un objet de type `T`, il émet un appel à `T::operator new( sizeof(T) )` ou, si aucun paramètre défini par

`operator new` l'utilisateur n'est défini, `::operator new( sizeof(T) )`. C'est la façon dont l'opérateur `new` peut allouer la quantité de mémoire correcte pour l'objet.

> ⓘ Notes
>
> L'argument à `operator new` est de type `std::size_t`. Ce type est défini dans <direct.h>, <malloc.h>, <memory.h>, <search.h>, <stddef.h>, <stdio.h>, <stdlib.h>, <string.h> et <time.h>.

Une option de la grammaire autorise la spécification de *`new-placement`* (voir grammaire pour new opérateur). Le *`new-placement`* paramètre ne peut être utilisé que pour les implémentations définies par l'utilisateur de `operator new`; il permet de passer des informations supplémentaires à `operator new`. Une expression avec un *`new-placement`* champ tel que `T *TObject = new ( 0x0040 ) T;` est traduit en `T *TObject = T::operator new( sizeof( T ), 0x0040 );` si la classe T a le membre `operator new`, sinon en `T *TObject = ::operator new( sizeof( T ), 0x0040 );`.

L'intention initiale du champ était d'autoriser l'allocation *`new-placement`* d'objets dépendants du matériel aux adresses spécifiées par l'utilisateur.

> ⓘ Notes
>
> Bien que l'exemple précédent n'affiche qu'un seul argument dans le *`new-placement`* champ, il n'existe aucune restriction quant au nombre d'arguments supplémentaires qui peuvent être passés de `operator new` cette façon.

Même si `operator new` a été défini pour un type `T`de classe , vous pouvez utiliser l'opérateur `new` global explicitement, comme dans cet exemple :

```
C++

T *TObject = ::new TObject;
```

L'opérateur de résolution d'étendue (`::`) force l'utilisation de l'opérateur global `new` .

Voir aussi

Expressions avec opérateurs unaires
Mots clés

newopérateurs et delete

Opérateur de complément à 1 : ~

Article • 16/06/2023

Syntaxe

```
C++
```

```
~ cast-expression
```

Notes

L'opérateur de complément d'un (`~`), parfois appelé opérateur de *complément au niveau du bit* , donne un complément au niveau du bit de son opérande. Autrement dit, chaque bit qui est 1 dans l'opérande est 0 dans le résultat. Inversement, chaque bit qui est 0 dans l'opérande est 1 dans le résultat. L'opérande de l'opérateur de complément à un doit être un type intégral.

Opérateur mot clé pour ~

C++ spécifie **`compl`** comme une autre orthographe pour `~`. En C, l'orthographe alternative est fournie sous forme de macro dans l'en-tête <iso646.h> . En C++, l'autre orthographe est un mot clé ; l'utilisation d'iso646.h><ou de ciso646> équivalent <C++ est déconseillée. Dans Microsoft C++, l'option du /permissive- compilateur ou /Za est requise pour activer l'orthographe alternative.

Exemple

```
C++
```

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
   unsigned short y = 0xFFFF;
   cout << hex << y << endl;
   y = ~y;   // Take one's complement
```

```
    cout << hex << y << endl;
}
```

Dans cet exemple, la nouvelle valeur assignée à `y` est le complément à 1 de la valeur non signée 0xFFFF ou 0x0000.

La promotion d'un intégral est exécutée sur des opérandes intégraux. Le type vers lequel l'opérande est promu est le type résultant. Pour plus d'informations sur la promotion intégrale, consultez Conversions standard.

Voir aussi

Expressions avec opérateurs unaires
Opérateurs intégrés C++, priorité et associativité
Opérateurs arithmétiques unaires

Opérateurs pointeur vers membre : `.*` et `->*`

Article • 16/06/2023

Syntaxe

`pm-expression`:

`cast-expression`

`pm-expression` `.*` *`cast-expression`*

`pm-expression` `->*` *`cast-expression`*

Notes

Opérateurs pointeur vers membre `.*` et `->*` retournent la valeur d'un membre de classe spécifique pour l'objet spécifié sur le côté gauche de l'expression. Le côté droit doit spécifier un membre de la classe. L'exemple ci-dessous illustre l'utilisation de ces opérateurs :

C++

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
   void m_func1() { cout << "m_func1\n"; }
   int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
   Testpm ATestpm;
   Testpm *pTestpm = new Testpm;

// Access the member function
   (ATestpm.*pmfn)();
```

```
    (pTestpm->*pmfn)();   // Parentheses required since * binds
                          // less tightly than the function call.

// Access the member data
    ATestpm.*pmd = 1;
    pTestpm->*pmd = 2;

    cout  << ATestpm.*pmd << endl
          << pTestpm->*pmd << endl;
    delete pTestpm;
}
```

Output

Output

```
m_func1
m_func1
1
2
```

Dans l'exemple précédent, un pointeur vers un membre, `pmfn`, est utilisé pour appeler la fonction membre `m_func1`. Un autre pointeur vers un membre, `pmd`, est utilisé pour accéder au membre `m_num`.

L'opérateur `.*` binaire combine son premier opérande, qui doit être un objet de type classe, avec son deuxième opérande, qui doit être un type pointeur vers membre.

L'opérateur `->*` binaire combine son premier opérande, qui doit être un pointeur vers un objet de type classe, avec son deuxième opérande, qui doit être un type pointeur vers membre.

Dans une expression contenant l'opérateur `.*` , le premier opérande doit être du type de classe du pointeur vers le membre spécifié dans le deuxième opérande ou d'un type accessible sans ambiguïté dérivé de et accessible à cette classe.

Dans une expression contenant l'opérateur `->*` , le premier opérande doit être du type « pointeur vers le type de classe » du type spécifié dans le deuxième opérande, ou il doit être d'un type dérivé sans ambiguïté de cette classe.

Exemple

Considérez les classes et le fragment de programme suivants :

C++

```cpp
// expre_Expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
   BaseClass(); // Base class constructor.
   void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
   Derived();  // Derived class constructor.
   void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
   BaseClass ABase;
   Derived ADerived;

   (ABase.*pmfnFunc1)();   // OK: defined for BaseClass.
   (ABase.*pmfnFunc2)();   // Error: cannot use base class to
                           // access pointers to members of
                           // derived classes.

   (ADerived.*pmfnFunc1)();   // OK: Derived is unambiguously
                              // derived from BaseClass.
   (ADerived.*pmfnFunc2)();   // OK: defined for Derived.
}
```

Le résultat des `.*` opérateurs ou `->*` pointeur vers membre est un objet ou une fonction du type spécifié dans la déclaration du pointeur vers le membre. Par conséquent, dans l'exemple précédent, le résultat de l'expression `ADerived.*pmfnFunc1()` est un pointeur vers une fonction qui retourne `void`. Ce résultat est une l-value si le second opérande est une l-value.

ⓘ Notes

Si le résultat d'un des opérateurs de pointeur vers membre est une fonction, le résultat peut être utilisé uniquement comme opérande pour l'opérateur d'appel de fonction.

Voir aussi

Opérateurs intégrés C++, priorité et associativité

Opérateurs suffixés d'incrémentation et de décrémentation : ++ et --

Article • 03/04/2023

Syntaxe

```
postfix-expression ++
postfix-expression --
```

Notes

C++ fournit des opérateurs d'incrémentation et de décrémentation préfixés et suffixés. Cette section décrit uniquement les opérateurs d'incrémentation et de décrémentation suffixés. (Pour plus d'informations, consultez Opérateurs d'incrémentation et de décrémentation de préfixe.) La différence entre les deux est que dans la notation postfix, l'opérateur apparaît après *postfix-expression*, alors que dans la notation de préfixe, l'opérateur apparaît avant *l'expression*. L'exemple suivant montre un opérateur postfix-incrément :

```
C++

i++;
```

L'application de l'opérateur d'incrément postfix (++) a pour effet que la valeur de l'opérande est augmentée d'une unité du type approprié. De même, l'effet de l'application de l'opérateur de décrémentation postfix (--) est que la valeur de l'opérande est réduite d'une unité du type approprié.

Il est important de noter qu'une expression d'incrément ou de décrémentation postfixe correspond à la valeur de l'expression *avant* l'application de l'opérateur respectif. L'opération d'incrémentation ou de décrémentation se produit *après* l'évaluation de l'opérande. Ce problème survient uniquement lorsque l'opération d'incrémentation ou de décrémentation suffixée intervient dans le contexte d'une plus grande expression.

Lorsqu'un opérateur suffixé est appliqué à un argument de fonction, il n'est pas garanti que la valeur de cet argument sera incrémentée ou décrémentée avant d'être passée à la fonction. Pour plus d'informations, reportez-vous à la section 1.9.17 de la norme C++.

L'application de l'opérateur d'incrément postfix à un pointeur vers un tableau d'objets de type `long` en ajoute en fait quatre à la représentation interne du pointeur. Ce comportement amène le pointeur, qui faisait précédemment référence au *nième*élément du tableau, à faire référence à l'élément (*n*+1)th.

Les opérandes à incrémenter postfix et les opérateurs de décrémentation postfix doivent être modifiables (et non `const`) valeurs l de type arithmétique ou pointeur. Le type du résultat est le même que celui de la *postfix-expression*, mais il ne s'agit plus d'une valeur l.

Visual Studio 2017 version 15.3 et ultérieure (disponible en mode et versions /std:c++17 ultérieures) : L'opérande d'un opérateur d'incrémentation ou de décrémentation postfix peut ne pas être de type `bool`.

Le code suivant illustre l'opérateur d'incrémentation suffixé :

C++

```
// expre_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
   int i = 10;
   cout << i++ << endl;
   cout << i << endl;
}
```

Les opérations d'incrémentation et de décrémentation suffixées sur les types énumérés ne sont pas prises en charge :

C++

```
enum Compass { North, South, East, West );
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

Voir aussi

Postfix Expressions
Opérateurs intégrés, priorité et associativité C++
Opérateurs suffixés d'incrémentation et de décrémentation C

Opérateurs préfixés d'incrémentation et de décrémentation : ++ et --

Article • 16/06/2023

Syntaxe

```
++ unary-expression
-- unary-expression
```

Notes

L'opérateur d'incrément de préfixe (++) en ajoute un à son opérande ; cette valeur incrémentée est le résultat de l'expression. L'opérande doit être une valeur l non de type `const`. Le résultat est une l-value du même type que l'opérande.

L'opérateur de décrémentation de préfixe (--) est analogue à l'opérateur d'incrémentation de préfixe, sauf que l'opérande est décrémenté par un et que le résultat est cette valeur décrémentée.

Visual Studio 2017 version 15.3 et ultérieures (disponible en /std:c++17 mode et versions ultérieures) : l'opérande d'un opérateur d'incrémentation ou de décrémentation peut ne pas être de type `bool`.

Les opérateurs préfixés et suffixés d'incrémentation et de décrémentation affectent leurs opérandes. La différence principale qui les distingue est lorsque l'incrémentation ou la décrémentation se produit dans l'évaluation d'une expression. (Pour plus d'informations, consultez Opérateurs d'incrémentation et de décrémentation de postfix.) Dans la forme de préfixe, l'incrément ou la décrémentation a lieu avant que la valeur ne soit utilisée dans l'évaluation de l'expression, de sorte que la valeur de l'expression est différente de la valeur de l'opérande. Dans leur forme suffixée, l'incrémentation ou la décrémentation ont lieu après que la valeur ne soit utilisée dans l'évaluation de l'expression. Par conséquent, la valeur de l'expression est identique à la valeur de l'opérande. Par exemple, le programme suivant affiche `++i = 6` :

C++

```
// expre_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
```

```
#include <iostream>

using namespace std;

int main() {
   int i = 5;
   cout << "++i = " << ++i << endl;
}
```

Un opérande de type intégral ou virgule flottante est incrémenté ou décrémenté par la valeur 1 entière. Le type du résultat est identique au type d'opérande. Un opérande de type pointeur est incrémenté ou décrémenté par la taille de l'objet qu'il adresse. Un pointeur incrémenté pointe vers l'objet suivant. Un pointeur décrémenté pointe vers l'objet précédent.

Étant donné que les opérateurs d'incrémentation et de décrémentation ont des effets secondaires, l'utilisation d'expressions avec des opérateurs d'incrémentation ou de décrémentation dans une macro de préprocesseur peut avoir des résultats indésirables. Examinez cet exemple :

C++

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
   int i = 0, j = 0, k;
   k = max( ++i, j );
}
```

La macro se développe pour donner :

C++

```
k = ((++i)<(j))?(j):(++i);
```

Si `i` est supérieur ou égal à `j` ou inférieur à `j` par 1, il est incrémenté deux fois.

> ⓘ **Notes**
>
> Les fonctions inline C++ sont préférables aux macros dans de nombreux cas car elles éliminent des effets secondaires tels que ceux décrits ici et permettent au langage d'effectuer une vérification de type plus complète.

Voir aussi

Expressions avec opérateurs unaires
Opérateurs intégrés, priorité et associativité C++
Opérateurs d'incrémentation et de décrémentation de préfixe

Opérateurs relationnels : `<`, `>`, `<=` et `>=`

Article • 03/04/2023

Syntaxe

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Notes

Les opérateurs relationnels binaires déterminent les relations suivantes :

- Inférieur à (<)
- Supérieur à (>)
- Inférieur ou égal à (<=)
- Supérieur ou égal à (>=)

Les opérateurs relationnels ont une associativité de gauche à droite. Les deux opérandes des opérateurs relationnels doivent être de type arithmétique ou pointeur. Ils produisent des valeurs de type **`bool`**. La valeur retournée est **`false`** (0) si la relation dans l'expression est false ; sinon, la valeur retournée est **`true`** (1).

Exemple

C++

```
// expre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
   cout  << "The true expression 3 > 2 yields: "
         << (3 > 2) << endl
         << "The false expression 20 < 10 yields: "
```

```
          << (20 < 10) << endl;
}
```

Les expressions de l'exemple précédent doivent être placées entre parenthèses, car l'opérateur d'insertion de flux (<<) a une priorité plus élevée que les opérateurs relationnels. Par conséquent, la première expression sans parenthèses est évaluée comme suit :

```
C++

(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

Les conversions arithmétiques habituelles couvertes dans les conversions standard sont appliquées aux opérandes de types arithmétiques.

Comparaison des pointeurs

Lorsque deux pointeurs vers des objets du même type sont comparés, le résultat est déterminé par l'emplacement des objets pointés figurant dans l'espace d'adressage du programme. Les pointeurs peuvent également être comparés à une expression constante qui prend la valeur 0 ou à un pointeur de type `void *`. Si une comparaison de pointeur est effectuée par rapport à un pointeur de type `void *`, l'autre pointeur est implicitement converti en type `void *`. Ensuite la comparaison est effectuée.

Deux pointeurs de types différents ne peuvent pas être comparés à moins que :

- Un type est un type de classe dérivé de l'autre type.
- Au moins l'un des pointeurs est explicitement converti (cast) en type `void *`. (L'autre pointeur est implicitement converti en type `void *` pour la conversion.)

Deux pointeurs du même type qui pointent vers le même objet ont la garantie d'être de comparer une valeur égale. Si deux pointeurs vers des membres non statique d'un objet sont comparés, les règles suivantes s'appliquent :

- Si le type de classe n'est pas un **`union`**, et si les deux membres ne sont pas séparés par un *spécificateur d'accès*, tel que **`public`**, **`protected`** ou **`private`**, le pointeur vers le membre déclaré en dernier sera comparé au pointeur vers le membre déclaré précédemment.
- Si les deux membres sont séparés par un *spécificateur d'accès*, les résultats ne sont pas définis.

- Si le type de classe est un `union`, les pointeurs vers différents membres de données dans qui comparent `union` égaux.

Si les deux pointeurs pointent vers des éléments du même tableau ou vers l'avant dernier élément du tableau, le pointeur vers l'objet avec l'indice le plus élevé compare une valeur supérieure. La comparaison des pointeurs est garantie comme étant valide uniquement lorsque les pointeurs font référence à des objets du même tableau ou à l'emplacement situé après la fin du tableau.

Voir aussi

Expressions avec opérateurs binaires
Opérateurs intégrés, priorité et associativité C++
Opérateurs relationnels et d'égalité C

Opérateur de résolution d'étendue : `::`

Article • 16/06/2023

L'opérateur `::` de résolution d'étendue est utilisé pour identifier et lever l'ambiguïté des identificateurs utilisés dans différentes étendues. Pour plus d'informations sur l'étendue, consultez Étendue.

Syntaxe

> *qualified-id*:
>
> *nested-name-specifier* **template**$_{opt}$ *unqualified-id*

> *nested-name-specifier*:
>
> `::`
>
> *type-name* `::`
>
> *namespace-name* `::`
>
> *decltype-specifier* `::`
>
> *nested-name-specifier* *identifier* `::`
>
> *nested-name-specifier* **template**$_{opt}$ *simple-template-id* `::`

> *unqualified-id*:
>
> *identifier*
>
> *operator-function-id*
>
> *conversion-function-id*
>
> *literal-operator-id*
>
> `~` *type-name*
>
> `~` *decltype-specifier*
>
> *template-id*

Notes

`identifier` peut être une variable, une fonction ou une valeur d'énumération.

Utiliser `::` pour les classes et les espaces de noms

L'exemple suivant montre comment l'opérateur de résolution de portée est utilisé avec les espaces de noms et les classes :

C++

```
namespace NamespaceA{
    int x;
    class ClassA {
    public:
        int x;
    };
}

int main() {

    // A namespace name used to disambiguate
    NamespaceA::x = 1;

    // A class name used to disambiguate
    NamespaceA::ClassA a1;
    a1.x = 2;
}
```

Un opérateur de résolution de portée sans qualificateur de portée fait référence à l'espace de noms global.

C++

```
namespace NamespaceA{
    int x;
}

int x;

int main() {
    int x;

    // the x in main()
    x = 0;
    // The x in the global namespace
    ::x = 1;

    // The x in the A namespace
    NamespaceA::x = 2;
}
```

Vous pouvez utiliser l'opérateur de résolution d'étendue pour identifier un membre d'un `namespace` ou pour identifier un espace de noms qui nomme l'espace de noms du membre dans une `using` directive. Dans l'exemple ci-dessous, vous pouvez utiliser

`NamespaceC` pour qualifier `ClassB`, même si `ClassB` a été déclaré dans l'espace de noms `NamespaceB`, car `NamespaceB` a été nommé dans `NamespaceC` par une **`using`** directive.

C++

```
namespace NamespaceB {
    class ClassB {
    public:
        int x;
    };
}

namespace NamespaceC{
    using namespace NamespaceB;
}

int main() {
    NamespaceB::ClassB b_b;
    NamespaceC::ClassB c_b;

    b_b.x = 3;
    c_b.x = 4;
}
```

Vous pouvez utiliser des chaînes d'opérateurs de résolution de portée. Dans l'exemple suivant, `NamespaceD::NamespaceD1` identifie l'espace de noms imbriqué `NamespaceD1` et `NamespaceE::ClassE::ClassE1` identifie la classe imbriquée `ClassE1`.

C++

```
namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{
    public:
        class ClassE1{
        public:
            int x;
        };
    };
}

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7  ;
}
```

Utiliser pour les `::` membres statiques

Vous devez utiliser l'opérateur de résolution de portée pour appeler des membres statiques de classes.

C++

```
class ClassG {
public:
    static int get_x() { return x;}
    static int x;
};

int ClassG::x = 6;

int main() {

    int gx1 = ClassG::x;
    int gx2 = ClassG::get_x();
}
```

Utiliser `::` pour les énumérations délimitées

L'opérateur de résolution délimitée est également utilisé avec les valeurs d'une énumération délimitée déclarations d'énumération, comme dans l'exemple suivant :

C++

```
enum class EnumA{
    First,
    Second,
    Third
};

int main() {
    EnumA enum_value = EnumA::First;
}
```

Voir aussi

Opérateurs intégrés C++ , précédence et associativité
Espaces de noms

sizeof, opérateur

Article • 03/04/2023

Génère la taille de son opérande par rapport à la taille de type `char`.

> ⓘ **Notes**
>
> Pour plus d'informations sur l'opérateur `sizeof ...` , consultez **Points de suspension et modèles variadiques**.

Syntaxe

```
sizeof unary-expression
sizeof  ( type-name )
```

Notes

Le résultat de l'opérateur `sizeof` est de type `size_t`, un type intégral défini dans le fichier <include stddef.h>. Cet opérateur vous permet d'éviter de spécifier les tailles de données dépendantes de l'ordinateur dans vos programmes.

L'opérande à `sizeof` peut être l'un des suivants :

- Un nom de type. Pour utiliser `sizeof` avec un nom de type, le nom doit être placé entre parenthèses.
- Expression. Lorsqu'il est utilisé avec une expression, `sizeof` peut être spécifié avec ou sans parenthèses. L'expression n'est pas évaluée.

Lorsque l'opérateur `sizeof` est appliqué à un objet de type `char`, il génère 1. Lorsque l'opérateur `sizeof` est appliqué à un tableau, il génère le nombre total d'octets dans ce tableau, et non la taille du pointeur représenté par l'identificateur du tableau. Pour obtenir la taille du pointeur représenté par l'identificateur de tableau, passez-le en tant que paramètre à une fonction qui utilise `sizeof`. Par exemple :

Exemple

C++

```
#include <iostream>
using namespace std;

size_t getPtrSize( char *ptr )
{
   return sizeof( ptr );
}

int main()
{
   char szHello[] = "Hello, world!";

   cout  << "The size of a char is: "
         << sizeof( char )
         << "\nThe length of " << szHello << " is: "
         << sizeof szHello
         << "\nThe size of the pointer is "
         << getPtrSize( szHello ) << endl;
}
```

Exemple de sortie

Output

```
The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4
```

Lorsque l'opérateur `sizeof` est appliqué à un `class` type , `struct` ou `union` , le résultat est le nombre d'octets dans un objet de ce type, ainsi que tout remplissage ajouté pour aligner les membres sur les limites des mots. Le résultat ne correspond pas forcément à la taille calculée en additionnant les exigences en matière de stockage des membres individuels. L'option de compilateur /Zp et le pragma pack affectent les limites d'alignement pour les membres.

L'opérateur `sizeof` ne génère jamais 0, même pour une classe vide.

L'opérateur `sizeof` ne peut pas être utilisé avec les opérandes suivants :

- Fonctions. (Toutefois, `sizeof` peut être appliqué aux pointeurs vers des fonctions.)
- Champ de bits.
- Classes indéfinies.

- Le type `void`.
- Tableaux alloués de manière dynamique.
- Tableaux externes.
- Types incomplets.
- Noms entre parenthèses de types incomplets.

Lorsque l'opérateur `sizeof` est appliqué à une référence, le résultat est le même que si `sizeof` avait été appliqué à l'objet lui-même.

Si un tableau non dimensionné est le dernier élément d'une structure, l'opérateur `sizeof` retourne la taille de la structure sans le tableau.

L'opérateur `sizeof` est souvent utilisé pour calculer le nombre d'éléments d'un tableau à l'aide d'une expression de la forme :

```
C++

sizeof array / sizeof array[0]
```

Voir aussi

Expressions avec opérateurs unaires
Mots clés

Opérateur d'indice []

Article • 16/06/2023

Syntaxe

```
postfix-expression [ expression ]
```

Notes

Une expression postfix (qui peut également être une expression primaire) suivie de l'opérateur subscript **, [ ]**, spécifie l'indexation de tableau.

Pour plus d'informations sur les tableaux managés dans C++/CLI, consultez Tableaux.

En règle générale, la valeur représentée par *postfix-expression* est une valeur de pointeur, telle qu'un identificateur de tableau, et *expression* est une valeur intégrale (y compris les types énumérés). Toutefois, sur le plan de la syntaxe, il est requis que l'une des expressions soit de type pointeur et l'autre de type intégral. Ainsi, la valeur intégrale peut se trouver dans la position *postfix-expression* et la valeur du pointeur peut se trouver entre crochets dans la position *de l'expression* ou de l'indice. Prenons le fragment de code suivant :

C++

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;            // prints "2"
cout << 2[nArray] << endl;            // prints "2"
```

Dans l'exemple précédent, l'expression `nArray[2]` est identique à `2[nArray]`. La raison en est que le résultat d'une expression d'indice `e1[e2]` est donné par :

`*((e2) + (e1))`

L'adresse obtenue par l'expression n'est pas *e2* octets de l'adresse *e1*. Au lieu de cela, l'adresse est mise à l'échelle pour générer l'objet suivant dans le tableau *e2*. Par exemple :

C++

```
double aDbl[2];
```

Les adresses de et `aDb[1]` sont séparées de `aDb[0]` 8 octets , la taille d'un objet de type `double`. Cette mise à l'échelle en fonction du type d'objet est effectuée automatiquement par le langage C++ et est définie dans les opérateurs additifs où l'ajout et la soustraction d'opérandes de type pointeur sont abordés.

Une expression d'indice peut également avoir plusieurs indices, comme suit :

expression1[*expression2*] [*expression3*] ...

Les expressions d'indice s'associent de gauche à droite. L'expression d'indice extrême gauche, *expression1*[*expression2*], est évaluée en premier. L'adresse qui résulte de l'ajout *d'expression1* et *expression2* forme une expression de pointeur. Ensuite, *expression3* est ajouté à cette expression de pointeur pour former une nouvelle expression de pointeur, et ainsi de suite jusqu'à ce que la dernière expression d'indice ait été ajoutée. L'opérateur d'indirection (*) est appliqué après l'évaluation de la dernière expression d'indice, à moins que la valeur de pointeur finale traite un type tableau.

Les expressions à indices multiples font référence à des éléments de tableaux multidimensionnels. Un tableau multidimensionnel est un tableau dont les éléments sont des tableaux. Par exemple, le premier élément d'un tableau tridimensionnel est un tableau à deux dimensions. L'exemple suivant déclare et initialise un tableau de caractères simple à deux dimensions :

C++

```
// expre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
  char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
  for ( int i = 0; i < MAX_ROWS; i++ )
     for ( int j = 0; j < MAX_COLS; j++ )
        cout << c[ i ][ j ] << endl;
}
```

Indices positifs et négatifs

Le premier élément d'un tableau est l'élément 0. La plage d'un tableau C++ va du *tableau*[0] au *tableau*[*taille* - 1]. Toutefois, C++ prend en charge les indices positifs et négatifs. Les indices négatifs doivent rester dans les limites du tableau. Sinon, les résultats sont imprévisibles. Le code suivant montre des indices de tableau positifs et négatifs :

C++

```
#include <iostream>
using namespace std;

int main() {
    int intArray[1024];
    for (int i = 0, j = 0; i < 1024; i++)
    {
        intArray[i] = j++;
    }

    cout << intArray[512] << endl;   // 512

    cout << 257[intArray] << endl;   // 257

    int *midArray = &intArray[512];  // pointer to the middle of the array

    cout << midArray[-256] << endl;  // 256

    cout << intArray[-256] << endl;  // unpredictable, may crash
}
```

L'indice négatif de la dernière ligne peut produire une erreur d'exécution, car il pointe vers une adresse 256 `int` positions inférieure en mémoire à l'origine du tableau. Le pointeur `midArray` est initialisé au milieu de `intArray`; il est donc possible (mais dangereux) d'utiliser des index tableaux positifs et négatifs sur celui-ci. Les erreurs d'indice de tableau ne génèrent pas d'erreurs au moment de la compilation, mais ils produisent des résultats imprévisibles.

L'opérateur d'indice est commutatif. Par conséquent, les expressions *array*[*index*] et *index*[*array*] sont garanties comme équivalentes tant que l'opérateur d'indice n'est pas surchargé (voir Opérateurs surchargés). La première forme constitue la pratique de codage la plus courante, mais les deux fonctionnent.

Voir aussi

Postfix Expressions
Opérateurs intégrés, priorité et associativité C++
Tableaux

Tableaux unidimensionnels
Tableaux multidimensionnels

typeid, opérateur

Article • 16/06/2023

Syntaxe

```
typeid(type-id)
typeid(expression)
```

Notes

L'opérateur **`typeid`** permet de déterminer le type d'un objet au moment de l'exécution.

Le résultat de **`typeid`** est un `const type_info&`. La valeur est une référence à un `type_info` objet qui représente *l'id de type* ou le type de l'expression, selon la forme de **`typeid`** utilisée. Pour plus d'informations, consultez type_info Classe.

L'opérateur **`typeid`** ne fonctionne pas avec les types managés (déclarateurs ou instances abstraits). Pour plus d'informations sur l'obtention Type du d'un type spécifié, consultez typeid.

L'opérateur **`typeid`** effectue une case activée d'exécution lorsqu'il est appliqué à une valeur l d'un type de classe polymorphe, où le type vrai de l'objet ne peut pas être déterminé par les informations statiques fournies. Dans les cas suivants par exemple :

- Une référence à une classe
- Pointeur déréférencé avec `*`
- Pointeur en indice (`[ ]`). (Il n'est pas sûr d'utiliser un indice avec un pointeur vers un type polymorphe.)

Si *l'expression* pointe vers un type de classe de base, mais que l'objet est en fait d'un type dérivé de cette classe de base, une `type_info` référence pour la classe dérivée est le résultat. *L'expression* doit pointer vers un type polymorphe (classe avec des fonctions virtuelles). Sinon, le résultat est le `type_info` pour la classe statique référencée dans *l'expression* . En outre, le pointeur doit être déréférencé afin que l'objet utilisé soit celui vers lequel il pointe. Sans déréférencement du pointeur, le résultat sera le `type_info` pour le pointeur, et non ce vers quoi il pointe. Par exemple :

C++

```
// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
   virtual void vvfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
   Derived* pd = new Derived;
   Base* pb = pd;
   cout << typeid( pb ).name() << endl;   //prints "class Base *"
   cout << typeid( *pb ).name() << endl;   //prints "class Derived"
   cout << typeid( pd ).name() << endl;   //prints "class Derived *"
   cout << typeid( *pd ).name() << endl;   //prints "class Derived"
   delete pd;
}
```

Si *l'expression* déréférence un pointeur et que sa valeur est égale à zéro, **`typeid`** lève une exception bad_typeid. Si le pointeur ne pointe pas vers un objet valide, une `__non_rtti_object` exception est levée. Il indique une tentative d'analyse du RTTI qui a déclenché une erreur, car l'objet n'est pas valide en quelque sorte. (Par exemple, il s'agit d'un pointeur incorrect, ou le code n'a pas été compilé avec /GR).

Si *l'expression* n'est pas un pointeur et non une référence à une classe de base de l'objet, le résultat est une `type_info` référence représentant le type statique de l'expression. Le *type statique* d'une expression fait référence au type d'une expression telle qu'elle est connue au moment de la compilation. Les sémantiques d'exécution sont ignorées en évaluant le type statique d'une expression. De plus, les références sont ignorées si possible lors de la détermination du type statique d'une expression :

C++

```
// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
   typeid(int) == typeid(int&); // evaluates to true
}
```

`typeid` peut également être utilisé dans les modèles pour déterminer le type d'un paramètre de modèle :

C++

```
// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
   cout << typeid( T ).name() << "s compared." << endl;
   return ( arg1 > arg2 ? arg1 : arg2 );
}
```

Voir aussi

Informations de type au moment de l'exécution
Mots clés

Opérateurs plus et de négation unaires : + et -

Article • 16/06/2023

Syntaxe

```
+ cast-expression
- cast-expression
```

+ (opérateur)

Le résultat de l'opérateur plus unaire (+) est la valeur de son opérande. L'opérande de l'opérateur plus unaire doit être d'un type arithmétique.

La promotion d'un intégral est exécutée sur des opérandes intégraux. Le type résultant est le type vers lequel l'opérande est promu. Ainsi, l'expression `+ch`, où `ch` est de type `char`, génère un type `int`; la valeur n'est pas modifiée. Pour plus d'informations sur la façon dont la promotion est effectuée, consultez Conversions standard .

- (opérateur)

L'opérateur de négation unaire (-) produit le négatif de son opérande. L'opérande de l'opérateur de négation unaire doit être un type arithmétique.

La promotion intégrale est exécutée sur les opérandes intégraux et le type résultant est le type vers lequel l'opérande est promu. Pour plus d'informations sur la façon dont la promotion est effectuée, consultez Conversions standard .

Section spécifique à Microsoft

La négation unaire des quantités non signées est exécutée en soustrayant la valeur de l'opérande de 2^n, n correspondant au nombre de bits dans un objet du type non signé donné.

FIN de la section spécifique à Microsoft

Voir aussi

Expressions avec opérateurs unaires

Opérateurs intégrés, priorité et associativité C++

Expressions (C++)

Article • 16/06/2023

Cette section décrit les expressions C++. Les expressions sont des séquences d'opérateurs et d'opérandes utilisés à plusieurs fins :

- Calcul d'une valeur à partir des opérandes.
- Désignation d'objets ou de fonctions.
- Générant des « effets secondaires ». (Les effets secondaires sont des actions autres que l'évaluation de l'expression, par exemple, la modification de la valeur d'un objet.)

En C++, les opérateurs peuvent être surchargés et leurs significations peuvent être définies par l'utilisateur. Toutefois, leur priorité et le nombre d'opérandes qu'ils supportent ne peuvent pas être modifiés. Cette section décrit la syntaxe et la sémantique des opérateurs tels qu'ils sont fournis avec le langage, non surchargés. Outre les types d'expressions et la sémantique des expressions, les rubriques suivantes sont traitées :

- Expressions principales
- Opérateur de résolution d'étendue
- Expressions suffixées
- Expressions avec opérateurs unaires
- Expressions avec opérateurs binaires
- Opérateur conditionnel
- Expressions constantes
- Opérateurs de casting
- Informations sur le type d'exécution

Rubriques sur les opérateurs dans d'autres sections :

- Opérateurs intégrés, priorité et associativité C++
- Opérateurs surchargés
- typeid (C++/CLI)

> **Notes**
>
> Les opérateurs pour les types intégrés ne peuvent pas être surchargés ; leur comportement est prédéfini.

Voir aussi

Informations de référence sur le langage C++

Types d'expressions

Article • 16/06/2023

Les expressions C++ sont classées en plusieurs catégories :

- Expressions principales. Ce sont les blocs de construction à partir desquels toutes les autres expressions sont formées.
- Expressions postfixées. Ce sont des expressions primaires suivies d'un opérateur, telles que l'indice de tableau ou l'opérateur d'incrémentation suffixé.
- Expressions formées avec des opérateurs unaires. Les opérateurs unaires agissent sur un seul opérande dans une expression.
- Expressions formées avec des opérateurs binaires. Les opérateurs binaires agissent sur deux opérandes dans une expression.
- Expressions avec l'opérateur conditionnel. L'opérateur conditionnel est un opérateur ternaire (le seul opérateur de ce type en langage C++) et il accepte trois opérandes.
- Expressions constantes. Les expressions constantes sont constituées entièrement de données constantes.
- Expressions avec conversions de type explicites. Des conversions de type explicite, ou « casts », peuvent être utilisées dans les expressions.
- Expressions avec des opérateurs pointeur vers membre.
- Casting. Des « casts » de type sécurisé peuvent être utilisés dans les expressions.
- Informations sur le type d'exécution. Déterminez le type d'un objet pendant l'exécution du programme.

Voir aussi

Expressions

Expressions primaires

Article • 16/06/2023

Les expressions primaires sont des blocs de construction d'expressions plus complexes. Il peut s'agir de littéraux, de noms et de noms qualifiés par l'opérateur de résolution d'étendue (`::`). Une expression primaire peut prendre chacune des formes suivantes :

primary-expression

- *literal*
- **this**
- *name*
- :: *name*
- (*expression*)

A *literal* est une expression primaire constante. Son type dépend de la forme de sa spécification. Pour obtenir des informations complètes sur la spécification de littéraux, consultez Littérals .

Le **this** mot clé est un pointeur vers un objet de classe. Il est disponible dans les fonctions membres non statiques. Il pointe vers le instance de la classe pour laquelle la fonction a été appelée. Le **this** mot clé ne peut pas être utilisé en dehors du corps d'une fonction membre de classe.

Le type du **this** pointeur est `type * const` (où `type` est le nom de la classe) dans les fonctions qui ne modifient pas spécifiquement le **this** pointeur. L'exemple suivant montre les déclarations de fonction membre et les types de **this**:

C++

```
// expre_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();          //  * const this
    void Func() const;    //  const * const this
    void Func() volatile; //  volatile * const this
};
```

Pour plus d'informations sur la modification du type du **this** pointeur, consultez this pointeur.

L'opérateur de résolution d'étendue (`::`) suivi d'un nom est une expression principale. Ces noms doivent être des noms au niveau de la portée globale, pas des noms de

membres. Le type de l'expression est déterminé par la déclaration du nom. Il s'agit d'une valeur l (autrement dit, elle peut apparaître sur le côté gauche d'une expression d'affectation) si le nom de déclaration est une valeur l. L'opérateur résolution-portée permet de faire référence à un nom global, même si ce nom est masqué dans la portée actuelle. Consultez Étendue pour obtenir un exemple d'utilisation de l'opérateur de résolution d'étendue.

Une expression placée entre parenthèses est une expression primaire. Son type et sa valeur sont identiques au type et à la valeur de l'expression nonparenthésisée. Il s'agit d'une valeur l si l'expression nonparenthesized est une valeur l.

Les exemples d'expressions principaux incluent :

C++

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // a global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

Ces exemples sont tous considérés comme *des noms* et, par conséquent, des expressions primaires, sous différentes formes :

C++

```
MyClass // an identifier
MyClass::f // a qualified name
operator = // an operator function name
operator char* // a conversion operator function name
~MyClass // a destructor name
A::B   // a qualified name
A<int> // a template id
```

Voir aussi

Types d'expressions

Ellipses et modèles variadiques

Article • 16/06/2023

Cet article montre comment utiliser les points de suspension (`...`) avec des modèles variadiques C++. Les points de suspension ont eu de nombreuses utilisations en C et C++. Il s'agit notamment des listes d'arguments de variables pour les fonctions. La `printf()` fonction de la bibliothèque runtime C est l'un des exemples les plus connus.

Un *modèle variadic* est un modèle de classe ou de fonction qui prend en charge un nombre arbitraire d'arguments. Ce mécanisme est particulièrement utile pour les développeurs de bibliothèqueS C++ : vous pouvez l'appliquer aux modèles de classe et aux modèles de fonction, et ainsi fournir un large éventail de fonctionnalités et de flexibilité non triviales et sécurisées de type.

Syntaxe

Les points de suspension sont utilisés de deux façons par les modèles variadiques. À gauche du nom du paramètre, cela signifie un pack de *paramètres*, et à droite du nom du paramètre, il développe les packs de paramètres dans des noms distincts.

Voici un exemple de base de syntaxe de définition de *modèle de classe variadique* :

```
C++
```

```
template<typename... Arguments> class classname;
```

Pour les packs de paramètres et les extensions, vous pouvez ajouter des espaces blancs autour des points de suspension, en fonction de votre préférence, comme illustré dans cet exemple :

```
C++
```

```
template<typename ...Arguments> class classname;
```

Ou cet exemple :

```
C++
```

```
template<typename ... Arguments> class classname;
```

Cet article utilise la convention indiquée dans le premier exemple (les points de suspension sont attachés à `typename`).

Dans les exemples précédents, `Arguments` est un pack de paramètres. La classe `classname` peut accepter un nombre variable d'arguments, comme dans ces exemples :

```
C++

template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

En utilisant une définition de modèle de classe variadicique, vous pouvez également exiger au moins un paramètre :

```
C++

template <typename First, typename... Rest> class classname;
```

Voici un exemple de base de syntaxe de *modèle de fonction variadique* :

```
C++

template <typename... Arguments> returntype functionname(Arguments... args);
```

Le `Arguments` pack de paramètres est ensuite développé pour utilisation, comme indiqué dans la section suivante.

D'autres formes de syntaxe de modèle de fonction variadique sont possibles, y compris, mais sans s'y limiter, les exemples suivants :

```
C++

template <typename... Arguments> returntype functionname(Arguments&...
args);
template <typename... Arguments> returntype functionname(Arguments&&...
args);
template <typename... Arguments> returntype functionname(Arguments*...
args);
```

Les spécificateurs tels que `const` sont également autorisés :

```
```

C++

```
template <typename... Arguments> returntype functionname(const Arguments&...
args);
```

Comme avec les définitions de classe de modèle variadic, vous pouvez créer des fonctions qui nécessitent au moins un paramètre :

C++

```
template <typename First, typename... Rest> returntype functionname(const
First& first, const Rest&... args);
```

Les modèles variadiciques utilisent l'opérateur `sizeof...()` (sans rapport avec l'opérateur plus ancien `sizeof()`) :

C++

```
template<typename... Arguments>
void tfunc(const Arguments&... args)
{
    constexpr auto numargs{ sizeof...(Arguments) };

    X xobj[numargs]; // array of some previously defined type X

    helper_func(xobj, args...);
}
```

En savoir plus sur le placement des points de suspension

Auparavant, cet article décrivait le placement des points de suspension qui définit les packs de paramètres et les extensions sous la forme suivante : « à gauche du nom du paramètre, cela signifie un pack de paramètres et, à droite du nom du paramètre, il développe les packs de paramètres dans des noms distincts ». Bien que techniquement vrai, il peut prêter à confusion dans la traduction en code. Considérez les aspects suivants :

- Dans un modèle-parameter-list (`template <parameter-list>`), `typename...` introduit un pack de paramètres de modèle.
- Dans une clause de déclaration de paramètre (`func(parameter-list)`), un point de suspension « de niveau supérieur » introduit un pack de paramètres de fonction, et

le positionnement des points de suspension est important :

```
C++

// v1 is NOT a function parameter pack:
template <typename... Types> void func1(std::vector<Types...> v1);

// v2 IS a function parameter pack:
template <typename... Types> void func2(std::vector<Types>... v2);
```

- Lorsque les points de suspension s'affichent immédiatement après un nom de paramètre, vous disposez d'une extension de pack de paramètres.

Exemple

Un bon moyen d'illustrer le mécanisme de modèle de fonction variadique consiste à l'utiliser dans une réécriture de certaines des fonctionnalités de `printf`:

```
C++

#include <iostream>

using namespace std;

void print() {
    cout << endl;
}

template <typename T> void print(const T& t) {
    cout << t << endl;
}

template <typename First, typename... Rest> void print(const First& first,
const Rest&... rest) {
    cout << first << ", ";
    print(rest...); // recursive call using pack expansion syntax
}

int main()
{
    print(); // calls first overload, outputting only a newline
    print(1); // calls second overload

    // these call the third overload, the variadic template,
    // which uses recursion as needed.
    print(10, 20);
    print(100, 200, 300);
    print("first", 2, "third", 3.14159);
}
```

Output

Output

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```

Notes

La plupart des implémentations qui incorporent des modèles de fonction variadique utilisent une récursivité d'une forme quelconque, mais elle est légèrement différente de la récursivité traditionnelle. La récursivité traditionnelle implique une fonction qui s'appelle elle-même à l'aide de la même signature. (Il peut être surchargé ou modèle, mais la même signature est choisie à chaque fois.) La récursion variadicique implique l'appel d'un modèle de fonction variadique en utilisant des nombres d'arguments différents (presque toujours décroissants), ce qui permet d'tamponner une signature différente à chaque fois. Un « cas de base » est toujours requis, mais la nature de la récursivité est différente.

Expressions suffixées

Article • 03/04/2023

Les expressions suffixées se composent d'expressions primaires ou d'expressions dans lesquelles les opérateurs suffixés suivent une expression primaire. Les opérateurs suffixés sont répertoriées dans le tableau suivant.

Opérateurs suffixés

Nom de l'opérateur	Notation de l'opérateur
Opérateur Indice	[ ]
Opérateur d'appel de fonction	( )
Opérateur de conversion de type explicite	*type-name*( )
Opérateur d'accès aux membres	. ou ->
Opérateur d'incrément suffixé	++
Opérateur de décrémentation suffixé	--

La syntaxe suivante décrit les expressions suffixées possibles :

```
primary-expression
postfix-expression[expression]postfix-expression(expression-list)simple-
type-name(expression-list)postfix-expression.namepostfix-expression-
>namepostfix-expression++postfix-expression--cast-keyword < typename >
(expression )typeid ( typename )
```

La *postfix-expression ci-dessus* peut être une expression primaire ou une autre expression de postfix. Les expressions suffixées sont regroupées de gauche à droite. Elles peuvent ainsi être chaînées les unes aux autres comme suit :

```
C++

func(1)->GetValue()++
```

Dans l'expression ci-dessus, `func` est une expression primaire, `func(1)` est une expression postfixe de fonction, `func(1)->GetValue` est une expression de postfix spécifiant un membre de la classe, `func(1)->GetValue()` est une autre expression postfix

de fonction, et l'expression entière est une expression postfix qui incrémente la valeur de retour de GetValue. L'expression entière signifie : Appeler fonction passant 1 comme argument et obtenir un pointeur vers une classe comme valeur de retour. Appelez `GetValue()` ensuite cette classe, puis incrémentez la valeur retournée.

Les expressions répertoriées ci-dessus sont des expressions d'assignation, ce qui signifie que le résultat de ces expressions doit être une r-value.

Forme de l'expression suffixée

```
C++

simple-type-name ( expression-list )
```

indique l'appel du constructeur. Si le nom-type-simple est un type fondamental, la liste d'expressions doit être une expression unique, et cette expression montre un cast de la valeur de l'expression au type fondamental. Ce type d'expression de cast simule un constructeur. Cette forme permet aux classes et aux types fondamentaux d'être construits à l'aide de la même syntaxe. Elle est donc particulièrement utile lorsque vous définissez des classes de modèles.

Le *cast-mot clé* est l'un des **`dynamic_cast`**, **`static_cast`** ou **`reinterpret_cast`**. Vous trouverez plus d'informations dans dynamic_cast, static_cast et reinterpet_cast.

L'opérateur **`typeid`** est considéré comme une expression postfixée. Consultez **l'opérateur typeid**.

Arguments formels et arguments réels

L'appel de programmes transmet des informations aux fonctions appelées dans les « arguments réels ». Les fonctions appelées accèdent aux informations à l'aide des « arguments formels » correspondants.

Lorsqu'une fonction est appelée, les tâches suivantes sont effectuées :

- Tous les arguments réels (ceux fournis par l'appelant) sont évalués. Il n'y a pas d'ordre implicite dans lequel ces arguments sont évalués, mais ils sont tous évalués et les effets secondaires sont résolus avant l'entrée dans la fonction.

- Chaque argument formel est initialisé avec son argument réel correspondant dans la liste d'expressions. (Un argument formel est un argument déclaré dans l'en-tête de fonction et utilisé dans le corps d'une fonction.) Les conversions sont effectuées comme si par initialisation : les conversions standard et définies par l'utilisateur

sont effectuées lors de la conversion d'un argument réel en type correct. L'initialisation exécutée est illustrée conceptuellement par le code suivant :

```
C++

void Func( int i ); // Function prototype
...
Func( 7 );          // Execute function call
```

Les initialisations conceptuelles avant l'appel sont :

```
C++

int Temp_i = 7;
Func( Temp_i );
```

Notez que l'initialisation est exécutée comme si la syntaxe de signe égal était utilisée au lieu de la syntaxe de parenthèses. Une copie de `i` est effectuée avant de passer la valeur à la fonction. (Pour plus d'informations, consultez Initialiseurs et conversions).

Par conséquent, si le prototype de fonction (déclaration) appelle un argument de type **`long`** et si le programme appelant fournit un argument réel de type **`int`**, l'argument réel est promu à l'aide d'une conversion de type standard en type **`long`** (voir Conversions standard).

C'est une erreur de fournir un argument réel pour lequel il n'existe aucune conversion standard ou définie par l'utilisateur au type de l'argument formel.

Pour les arguments réels du type classe, l'argument formel est initialisé en appelant le constructeur de la classe. (Pour plus d'informations sur ces fonctions membres de classe spéciales, consultez Constructeurs .)

- L'appel de fonction est exécuté.

Le fragment de programme suivant montre un appel de fonction :

```
C++

// expre_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;
```

```
    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
}
```

Quand `func` est appelé à partir de main, le paramètre `param1` formel est initialisé avec la valeur de `i` (`i` est converti en type **`long`** pour correspondre au type correct à l'aide d'une conversion standard) et le paramètre `param2` formel est initialisé avec la valeur de `j` (`j` est converti en type **`double`** à l'aide d'une conversion standard).

Traitement des types d'arguments

Les arguments formels déclarés en tant que **`const`** types ne peuvent pas être modifiés dans le corps d'une fonction. Les fonctions peuvent modifier n'importe quel argument qui n'est pas de type **`const`**. Toutefois, la modification est locale à la fonction et n'affecte pas la valeur de l'argument réel, sauf si l'argument réel était une référence à un objet non de type **`const`**.

Les fonctions suivantes illustrent certains de ces concepts :

C++

```
// expre_Treatment_of_Argument_Types.cpp
int func1( const int i, int j, char *c ) {
   i = 7;   // C3892 i is const.
   j = i;   // value of j is lost at return
   *c = 'a' + j;   // changes value of c in calling function
   return i;
}

double& func2( double& d, const char *c ) {
   d = 14.387;   // changes value of d in calling function.
   *c = 'a';   // C3892 c is a pointer to a const object.
    return d;
}
```

Points de suspension et arguments par défaut

Les fonctions peuvent être déclarées pour accepter moins d'arguments que le nombre spécifié dans la définition de fonction, en utilisant l'une des deux méthodes suivantes :

points de suspension (`...`) ou arguments par défaut.

Les points de suspension indiquent que des arguments peuvent être requis, mais que le nombre et les types ne sont pas spécifiés dans la déclaration. C'est habituellement une mauvaise pratique de programmation C++ car elle occulte l'un des avantages du C++ : la sécurité de type. Différentes conversions sont appliquées aux fonctions déclarées avec des points de suspension de celles pour lesquelles les types d'arguments formels et réels sont connus :

- Si l'argument réel est de type **`float`**, il est promu en type **`double`** avant l'appel de fonction.
- Tout **`signed char`** type énuméré ou **`unsigned char`**, **`signed short`** ou **`unsigned short`** champ bit est converti en un ou un à l'aide d'une **`signed int`** **`unsigned int`** promotion intégrale.
- Tout argument de type classe est transmis par valeur comme une structure de données. La copie est créée par copie binaire plutôt que par appel d'un constructeur de copie de la classe (le cas échéant).

Les points de suspension, s'ils sont utilisés, doivent être déclarés en dernier dans la liste d'arguments. Pour plus d'informations sur la transmission d'un nombre variable d'arguments, consultez la discussion sur les va_arg, les va_start et les va_list dans la *référence de la bibliothèque d'exécution*.

Pour plus d'informations sur les arguments par défaut dans la programmation CLR, consultez Listes d'arguments variables (...) (C++/CLI).

Les arguments par défaut vous permettent de spécifier la valeur qu'un argument doit prendre si aucune n'est fournie dans l'appel de fonction. Le fragment de code suivant montre le fonctionnement des arguments par défaut. Pour plus d'informations sur les restrictions relatives à la spécification des arguments par défaut, consultez Arguments par défaut.

C++

```
// expre_Ellipsis_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
```

```
{
    print( "hello," );
    print( "world!" );

    print( "good morning", ", " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

Le programme précédent déclare une fonction, `print`, qui accepte deux arguments. Toutefois, le deuxième argument, *terminateur*, a une valeur par défaut, `"\n"`. Dans `main`, les deux premiers appels permettent au `print` deuxième argument par défaut de fournir une nouvelle ligne pour terminer la chaîne imprimée. Le troisième appel indique une valeur explicite pour le deuxième argument. Le résultat généré par le programme est

Output

```
hello,
world!
good morning, sunshine.
```

Voir aussi

Types d'expressions

Expressions avec opérateurs unaires

Article • 16/06/2023

Les opérateurs unaires agissent sur un seul opérande dans une expression. Opérateurs unaires :

- Opérateur d'indirection (*)
- Opérateur d'adresse (&)
- Opérateur plus unaire (+)
- Opérateur de négation unaire (-)
- Opérateur de négation logique (!)
- Opérateur de complément d'un (~)
- Opérateur d'incrémentation de préfixe (++)
- Opérateur de décrémentation de préfixe (--)
- Opérateur cast ()
- sizeof, opérateur
- alignof, opérateur
- noexcept, expression
- new, opérateur
- delete, opérateur

Ces opérateurs ont une associativité de droite à gauche. Les expressions unaires impliquent en général une syntaxe qui précède une expression primaire ou suffixée.

Syntaxe

unary-expression:

postfix-expression

++ *cast-expression*

-- *cast-expression*

unary-operator *cast-expression*

- `sizeof` *unary-expression*
- `sizeof` `(` *type-id* `)`
- `sizeof` `...` `(` *identifier* `)`
- `alignof` `(` *type-id* `)`
- *noexcept-expression*
- *new-expression*
- *delete-expression*

unary-operator : l'un des éléments suivants :

- `*` `&` `+` `-` `!` `~`

Notes

N'importe quel *postfix-expression* est considéré comme un *unary-expression*, et comme tout *primary-expression* est considéré comme un *postfix-expression*, tout *primary-expression* est également considéré comme un *unary-expression* . Pour plus d'informations, consultez Expressions postfix et Expressions primaires.

cast-expression est un *unary-expression* avec un cast facultatif pour modifier le type. Pour plus d'informations, consultez Opérateur cast : ().

est *noexcept-expression* un *noexcept-specifier* avec un *constant-expression* argument . Pour plus d'informations, consultez noexcept.

fait *new-expression* référence à l'opérateur `new` . fait *delete-expression* référence à l'opérateur `delete` . Pour plus d'informations, consultez new Opérateur et delete opérateur.

Voir aussi

Types d'expressions

Expressions avec opérateurs binaires

Article • 16/06/2023

Les opérateurs binaires agissent sur deux opérandes dans une expression. Les opérateurs binaires sont :

- Opérateurs multiplicatifs
  - Multiplication (*)
  - Division (/)
  - Modulo (%)
- Opérateurs additifs
  - Addition (+)
  - Soustraction (-)
- Opérateurs de décalage
  - Maj droit (>>)
  - Maj gauche (<<)
- Opérateurs relationnels et d'égalité
  - Inférieur à (<)
  - Supérieur à (>)
  - Inférieur ou égal à (<=)
  - Supérieur ou égal à (>=)
  - Égal à (==)
  - Différent de (!=)
- Opérateurs de bits
  - Au niveau du bit AND (&)
  - OR exclusif au niveau du bit (^)
  - OR inclusive au niveau du bit (|)

- Opérateurs logiques
  - Logique AND (&&)
  - OR logique (||)
- Opérateurs d’affectation
  - Assignation (=)
  - Assignation d'addition (+=)
  - Affectation de soustraction (-=)
  - Assignation de multiplication (*=)
  - Assignation de division (/=)
  - Assignation de modulo (%=)
  - Affectation de décalage gauche (<<=)
  - Affectation de décalage droit (>>=)
  - Affectation AND au niveau du bit (&=)
  - Assignation d'opération OR exclusive au niveau du bit (^=)
  - Assignation d'opération OR inclusive au niveau du bit (|=)
- Opérateur virgule (,)

Voir aussi

Types d’expressions

Expressions constantes C++

Article • 16/06/2023

Une valeur *constante* est une valeur qui ne change pas. C++ fournit deux mots clés qui vous permettent d'exprimer l'intention qu'un objet n'est pas destiné à être modifié, et d'appliquer cette intention.

C++ requiert des expressions constantes (expressions qui ont pour valeur une constante) pour les déclarations des éléments suivants :

- Limites d'index de tableau
- Sélecteurs dans les instructions case
- Spécification de longueur de champ de bits
- Initialiseurs d'énumération

Les seuls opérandes autorisés dans les expressions constantes sont :

- Littéraux
- Constantes d'énumération
- Valeurs déclarées comme const qui sont initialisées avec des expressions constantes
- Expressions `sizeof`

Les constantes non intégrales doivent être converties (explicitement ou implicitement) en types intégraux pour être autorisées dans une expression constante. Par conséquent, le code suivant est conforme :

```
C++
```

```
const double Size = 11.0;
char chArray[(int)Size];
```

Les conversions explicites en types intégraux sont légales dans les expressions constantes ; tous les autres types et types dérivés ne sont pas conformes, sauf lorsqu'ils sont utilisés en tant qu'opérandes pour l'opérateur `sizeof` .

L'opérateur virgule et les opérateurs d'assignation de virgule ne peuvent pas être utilisés dans les expressions constantes.

Voir aussi

Types d’expressions

Sémantique des expressions

Article • 03/04/2023

Les expressions sont évaluées selon la priorité et le regroupement de leurs opérateurs. (Priorité et associativité de l'opérateur dans les conventions lexicales, montre les relations que les opérateurs C++ imposent aux expressions.)

Ordre d'évaluation

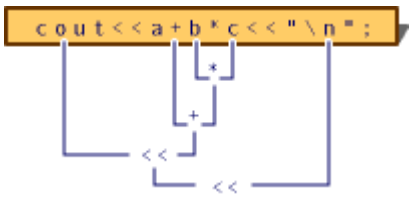
Examinez cet exemple :

C++

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

Output

```
38
38
54
```



Ordre d'évaluation des expressions

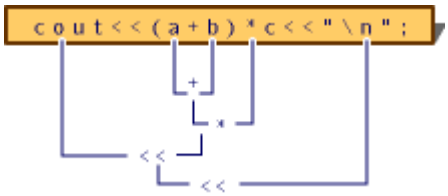
L'ordre dans lequel l'expression affichée dans l'illustration ci-dessus est évaluée est déterminé par la priorité et l'associativité des opérateurs :

1. La multiplication (*) a la priorité la plus élevée dans cette expression. Donc, la sous-expression `b * c` est évaluée en premier.

2. L'ajout (+) a la priorité la plus élevée suivante, donc `a` est ajouté au produit de `b` et `c`.

3. Le décalage gauche (<<) a la priorité la plus faible dans l'expression, mais il existe deux occurrences. Étant donné que l'opérateur de décalage vers la gauche groupe de gauche à droite, la sous-expression gauche est évaluée en premier avant la sous-expression droite.

Lorsque les parenthèses sont utilisées pour grouper les sous-expressions, elles modifient la priorité et également l'ordre dans lequel l'expression est évaluée, comme indiqué dans l'illustration suivante.



Ordre d'évaluation des expressions entre parenthèses

Les expressions telles qu'elles existent dans l'illustration ci-dessus sont purement évaluées pour leurs effets secondaires, dans ce cas pour transférer des informations au périphérique de sortie standard.

Notation dans les expressions

Le langage C++ spécifie certaines compatibilités lors de la spécification des opérandes. Le tableau suivant présente les types d'opérandes acceptables pour les opérateurs qui nécessitent des opérandes de *type*.

Types d'opérande acceptables pour les opérateurs

Type attendu	Types autorisés
type	`const` *type* `volatile` *type* *Type*& `const` *type*& `volatile` *type*& `volatile const` *type* `volatile const` *type*&

Type attendu	Types autorisés
Type *	*Type* * `const` *type* * `volatile` *type* * `volatile const` *type* *
`const` *type*	*type* `const` *type* `const` *type*&
`volatile` *type*	*type* `volatile` *type* `volatile` *type*&

Comme les règles précédentes peuvent toujours être utilisées de manière combinée, un pointeur const vers un objet volatile peut être fourni là où un pointeur est attendu.

Expressions ambiguës

La signification de certaines expressions est ambiguë. Ces expressions se rencontrent généralement lorsque la valeur d'un objet change plusieurs fois dans la même expression. Elles reposent sur un ordre particulier d'évaluation lorsque le langage n'en définit aucun. Prenons l'exemple suivant :

```
int i = 7;

func( i, ++i );
```

Le langage C++ ne garantit pas l'ordre dans lequel les arguments d'un appel de fonction sont évalués. Par conséquent, dans l'exemple précédent, `func` peut recevoir comme paramètres les valeurs 7 et 8, ou 8 et 8, selon que les paramètres sont évalués de gauche à droite ou de droite à gauche.

Points de séquence C++ (spécifiques à Microsoft)

Une expression peut modifier la valeur d'un objet une seule fois entre des points de séquence consécutifs.

La définition de langage C++ ne spécifie pas actuellement les points de séquence. Microsoft C++ utilise les mêmes points de séquence que C ANSI pour toute expression utilisant des opérateurs C et n'impliquent pas les opérateurs surchargés. Lorsque des opérateurs sont surchargés, la sémantique passe de la séquence par l'opérateur à la séquence par l'appel de fonction. Microsoft C++ utilise les points de séquence ci-dessous.

- Opérande gauche de l'opérateur AND logique (`&&`). L'opérande gauche de l'opérateur AND logique est complètement évalué et tous les effets secondaires sont terminés avant de continuer. Il n'existe aucune garantie que l'opérande droit de l'opérateur AND logique sera évalué.

- Opérande gauche de l'opérateur LOGIQUE OR (`||`). L'opérande gauche de l'opérateur OR logique est complètement évalué et tous les effets secondaires sont terminés avant de continuer. Il n'existe aucune garantie que l'opérande approprié de l'opérateur OU logique sera évalué.

- Opérande gauche de l'opérateur virgule. L'opérande gauche de l'opérateur virgule est complètement évalué et tous les effets secondaires sont terminés avant de continuer. Les deux opérandes de l'opérateur virgule sont toujours évalués.

- Opérateur d'appel de fonction. L'expression appel-fonction et tous les arguments d'une fonction, notamment les arguments par défaut, sont évalués et tous les effets secondaires sont terminés avant l'entrée dans la fonction. Il n'y a aucun ordre spécifié d'évaluation entre les arguments ou l'expression appel-fonction.

- Premier opérande de l'opérateur conditionnel. Le premier opérande de l'opérateur conditionnel est complètement évalué et tous les effets secondaires sont terminés avant de continuer.

- La fin d'une expression complète d'initialisation, comme la fin d'une initialisation dans une instruction de déclaration.

- L'expression dans une instruction d'expression. Les instructions Expression sont composées d'une expression facultative suivie d'un point-virgule (;). L'expression est complètement évaluée pour ses effets secondaires.

- L'expression de contrôle d'une instruction de sélection (if ou switch). L'expression est complètement évaluée et tous les effets secondaires sont terminés avant que le code dépendant de la sélection soit exécuté.

- Expression de contrôle d'une instruction while ou do. L'expression est complètement évaluée et tous les effets secondaires sont terminés avant que

toutes les instructions de l'itération suivante de la boucle while ne soient exécutées.

- Chacune des trois expressions d'une instruction for. Chaque expression est complètement évaluée et tous les effets secondaires sont terminés avant de passer à l'expression suivante.
- Expression dans une instruction return. L'expression est complètement évaluée et tous les effets secondaires sont terminés avant que le contrôle ne retourne à la fonction appelante.

Voir aussi

Expressions

Cast

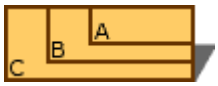
Article • 03/04/2023

Le langage C++ prévoit que si une classe est dérivée d'une classe de base contenant des fonctions virtuelles, un pointeur vers ce type de classe de base peut être utilisé pour appeler les implémentations des fonctions virtuelles résidant dans l'objet classe dérivé. Une classe contenant des fonctions virtuelles est parfois appelée une classe polymorphe.

Étant donné qu'une classe dérivée contient les définitions de toutes les classes de base dont elle est dérivée, il est possible de convertir un pointeur qui va jusqu'en haut de la hiérarchie de classes en l'une de ces classes de base. S'il existe un pointeur vers une classe de base, il est possible de le convertit jusqu'en bas de la hiérarchie. Cela est possible si l'objet qui est pointé est réellement d'un type dérivé de la classe de base. Dans ce cas, l'objet réel est dit comme « l'objet complet ». Le pointeur vers la classe de base pointe vers un « sous-objet » de l'objet complet. Considérons, par exemple, la hiérarchie de classe représentée dans l'illustration ci-dessous.

Hiérarchie de classes

Un objet de type `C` peut être visualisé comme indiqué dans l'illustration ci-dessous.



Classe C avec les sous-objets B et A

Avec une instance de classe `C`, il y a un sous-objet `B` et un sous-objet `A`. L'instance de `C`, avec les sous-objets `A` et `B` forme l'objet complet.

En utilisant les informations de type au moment de l'exécution, il est possible de vérifier si un pointeur pointe réellement vers un objet complet et peut être casté sans risque vers un autre objet de sa hiérarchie. L'opérateur dynamic_cast peut être utilisé pour effectuer ces types de casts. Il exécute également le contrôle à l'exécution nécessaire pour sécuriser l'opération.

Pour la conversion de types nonpolymorphes, vous pouvez utiliser l'opérateur static_cast (cette rubrique explique la différence entre les conversions de cast statique et dynamique, et quand il convient d'utiliser chacune d'elles).

Cette section couvre les sujets suivants :

- Opérateurs de casting

- Informations sur le type d'exécution

Voir aussi

Expressions

Opérateurs de casting

Article • 03/04/2023

Il existe plusieurs opérateurs de cast spécifiques au langage C++. Ces opérateurs ont pour but de supprimer une partie de l'ambiguïté et du risque inhérents aux casts en langage C de style ancien. Ces opérateurs sont :

- dynamic_cast Utilisé pour la conversion de types polymorphes.
- static_cast Utilisé pour la conversion de types nonpolymorphes.
- const_cast Utilisé pour supprimer les `const` attributs , `volatile` et `__unaligned` .
- reinterpret_cast Utilisé pour une réinterprétation simple des bits.
- safe_cast Utilisé en C++/CLI pour produire des MSIL vérifiables.

Utilisez `const_cast` et `reinterpret_cast` en dernier recours, puisque ces opérateurs présentent les mêmes dangers que les casts de style ancien. Ils sont néanmoins encore nécessaires pour remplacer complètement les casts de style ancien.

Voir aussi

Cast

dynamic_cast, opérateur

Article • 06/04/2023

Convertit l'opérande `expression` en objet de type `type-id`.

Syntaxe

```
dynamic_cast < type-id > ( expression )
```

Notes

Doit `type-id` être un pointeur ou une référence à un type de classe défini précédemment ou un « pointeur vers void ». Le type de `expression` doit être un pointeur si `type-id` est un pointeur ou une l-value si `type-id` est une référence.

Consultez static_cast pour une explication de la différence entre les conversions de casting statiques et dynamiques, et pour savoir quand il est approprié d'utiliser chacune d'elles.

Il existe deux changements cassants dans le comportement de **`dynamic_cast`** dans le code managé :

- **`dynamic_cast`** vers un pointeur vers le type sous-jacent d'une énumération boxée échoue au moment de l'exécution, renvoyant 0 au lieu du pointeur converti.
- **`dynamic_cast`** ne lève plus d'exception quand `type-id` est un pointeur intérieur vers un type valeur ; au lieu de cela, le cast échoue au moment de l'exécution. Le cast retourne la valeur 0 pointeur au lieu de lever.

Si `type-id` est un pointeur vers une classe de base directe ou indirecte accessible sans ambiguïté de `expression`, alors un pointeur vers le sous-objet unique de type `type-id` est le résultat. Par exemple :

C++

```
// dynamic_cast_1.cpp
// compile with: /c
class B { };
class C : public B { };
```

```
class D : public C { };

void f(D* pd) {
   C* pc = dynamic_cast<C*>(pd);   // ok: C is a direct base class
                                   // pc points to C subobject of pd
   B* pb = dynamic_cast<B*>(pd);   // ok: B is an indirect base class
                                   // pb points to B subobject of pd
}
```

Ce type de conversion est appelé « upcast », car il déplace un pointeur vers le haut d'une hiérarchie de classes, d'une classe dérivée à une classe dont elle est dérivée. Une upcast est une conversion implicite.

Si `type-id` est void*, une case activée d'exécution est effectuée pour déterminer le type réel de `expression`. Le résultat est un pointeur vers l'objet complet pointé vers `expression`. Par exemple :

C++

```
// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
   A* pa = new A;
   B* pb = new B;
   void* pv = dynamic_cast<void*>(pa);
   // pv now points to an object of type A

   pv = dynamic_cast<void*>(pb);
   // pv now points to an object of type B
}
```

Si `type-id` n'est pas `void*`, une case activée d'exécution est effectuée pour voir si l'objet pointé par `expression` peut être converti en type pointé par `type-id`.

Si le type de `expression` est une classe de base du type de `type-id`, une case activée d'exécution est effectuée pour voir si `expression` pointe réellement vers un objet complet du type de `type-id`. Si cette valeur est vraie, le résultat est un pointeur vers un objet complet du type .`type-id` Par exemple :

C++

```
// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};
```

```
void f() {
   B* pb = new D;   // unclear but ok
   B* pb2 = new B;

   D* pd = dynamic_cast<D*>(pb);   // ok: pb actually points to a D
   D* pd2 = dynamic_cast<D*>(pb2);   // pb2 points to a B not a D
}
```

Ce type de conversion est appelé « downcast », car il déplace un pointeur vers le bas d'une hiérarchie de classes, d'une classe donnée à une classe dérivée de celle-ci.

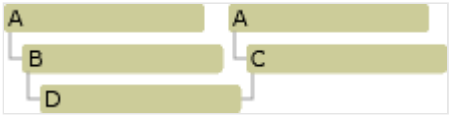
En cas d'héritage multiple, des possibilités d'ambiguïté sont introduites. Considérez la hiérarchie de classes illustrée dans la figure suivante.

Pour les types CLR, `dynamic_cast` aboutit soit à un no-op si la conversion peut être effectuée implicitement, soit à une instruction MSIL `isinst`, qui effectue une case activée dynamique et retourne `nullptr` si la conversion échoue.

L'exemple suivant utilise `dynamic_cast` pour déterminer si une classe est une instance de type particulier :

C++

```
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

void PrintObjectType( Object^o ) {
   if( dynamic_cast<String^>(o) )
      Console::WriteLine("Object is a String");
   else if( dynamic_cast<int^>(o) )
      Console::WriteLine("Object is an int");
}

int main() {
   Object^o1 = "hello";
   Object^o2 = 10;

   PrintObjectType(o1);
   PrintObjectType(o2);
}
```



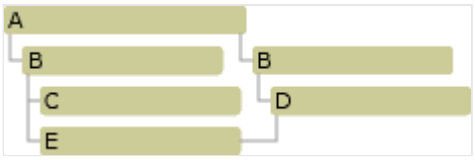
Un pointeur vers un objet de type `D` peut être converti en toute sécurité vers `B` ou `C`. Toutefois, si `D` est casté pour pointer vers un `A` objet, quelle instance de `A` résulterait ?

Cela entraînerait une erreur de casting ambiguë. Pour contourner ce problème, vous pouvez effectuer deux casts non ambigus. Par exemple :

C++

```
// dynamic_cast_4.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {virtual void f();};
class D : public B, public C {virtual void f();};

void f() {
   D* pd = new D;
   A* pa = dynamic_cast<A*>(pd);   // C4540, ambiguous cast fails at runtime
   B* pb = dynamic_cast<B*>(pd);   // first cast to B
   A* pa2 = dynamic_cast<A*>(pb);   // ok: unambiguous
}
```

D'autres ambiguïtés peuvent être introduites lorsque vous utilisez des classes de base virtuelles. Considérez la hiérarchie de classes illustrée dans la figure suivante.



Hiérarchie de classes montrant des classes de base virtuelles

Dans cette hiérarchie, `A` est une classe de base virtuelle. Étant donné une instance de classe `E` et un pointeur vers le `A` sous-objet, un **`dynamic_cast`** pointeur vers un pointeur vers `B` échoue en raison d'une ambiguïté. Vous devez d'abord revenir à l'objet complet `E` , puis remonter la hiérarchie, de manière non ambiguë, pour atteindre l'objet correct `B` .

Considérez la hiérarchie de classes illustrée dans la figure suivante.

Hiérarchie de classes montrant des classes de base en double

Étant donné un objet de type `E` et un pointeur vers le `D` sous-objet, pour naviguer du `D` sous-objet vers le sous-objet le plus `A` à gauche, trois conversions peuvent être effectuées. Vous pouvez effectuer une **`dynamic_cast`** conversion à partir du `D` pointeur

vers un `E` pointeur, puis une conversion (`dynamic_cast` soit une conversion implicite) de `E` vers `B`, et enfin une conversion implicite de `B` vers `A`. Par exemple :

C++

```
// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
   E* pe = dynamic_cast<E*>(pd);
   B* pb = pe;   // upcast, implicit conversion
   A* pa = pb;   // upcast, implicit conversion
}
```

L'opérateur `dynamic_cast` peut également être utilisé pour effectuer un « cast croisé ». À l'aide de la même hiérarchie de classes, il est possible de caster un pointeur, par exemple, du `B` sous-objet vers le `D` sous-objet, tant que l'objet complet est de type `E`.

Compte tenu des casts croisés, il est possible d'effectuer la conversion d'un pointeur vers `D` un pointeur vers le sous-objet le plus `A` à gauche en seulement deux étapes. Vous pouvez effectuer une conversion croisée de `D` vers `B`, puis une conversion implicite de `B` en `A`. Par exemple :

C++

```
// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A { };
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
   B* pb = dynamic_cast<B*>(pd);   // cross cast
   A* pa = pb;   // upcast, implicit conversion
}
```

Une valeur de pointeur null est convertie en valeur de pointeur null du type de destination par `dynamic_cast`.

Lorsque vous utilisez `dynamic_cast < type-id > ( expression )`, si `expression` ne peut pas être converti en toute sécurité en type `type-id`, le case activée d'exécution entraîne l'échec de la conversion. Par exemple :

C++

```
// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
   A* pa = new A;
   B* pb = dynamic_cast<B*>(pa);   // fails at runtime, not safe;
   // B not derived from A
}
```

La valeur d'un type de pointeur ayant échoué est le pointeur null. Un cast ayant échoué en type référence lève une exception bad_cast. Si `expression` ne pointe pas vers ou ne référence pas un objet valide, une `__non_rtti_object` exception est levée.

Pour obtenir une explication de l'exception `__non_rtti_object`, consultez typeid.

Exemple

L'exemple suivant crée le pointeur de classe de base (struct A) vers un objet (struct C). Cela, en plus du fait qu'il existe des fonctions virtuelles, permet le polymorphisme d'exécution.

L'exemple appelle également une fonction non virtuelle dans la hiérarchie.

C++

```
// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
   }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }
```

```
    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

   // fails because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}
```

Output

```
in C
test2 in B
```

```
in GlobalTest
Can't cast to C
```

Voir aussi

Opérateurs de casting

Mots clés

bad_cast, exception

Article • 03/04/2023

L'exception bad_cast est levée par l'opérateur `dynamic_cast` à la suite d'un échec de conversion en type référence.

Syntaxe

```
catch (bad_cast)
   statement
```

Notes

L'interface pour **bad_cast** est la suivante :

C++

```
class bad_cast : public exception
```

Le code suivant contient un exemple d'échec `dynamic_cast` qui lève l'exception **bad_cast** .

C++

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class Shape {
public:
   virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
   virtual void virtualfunc() const {}
};

using namespace std;
int main() {
   Shape shape_instance;
```

```
    Shape& ref_shape = shape_instance;
    try {
       Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
    }
    catch (bad_cast b) {
       cout << "Caught: " << b.what();
    }
}
```

L'exception est levée, car l'objet en cours de cast (une forme) n'est pas dérivé du type de cast spécifié (Circle). Pour éviter l'exception, ajoutez les déclarations ci-dessous à `main` :

C++

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

Ensuite, inversez le sens du cast dans le `try` bloc comme suit :

C++

```
Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

Membres

Constructeurs

Constructeur	Description
bad_cast	Constructeur des objets de type `bad_cast`.

Fonctions

Fonction	Description
Quel	TBD

Opérateurs

Opérateur	Description
operator=	Opérateur d'affectation qui affecte un `bad_cast` objet à un autre.

bad_cast

Constructeur des objets de type `bad_cast`.

```
C++
```

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

opérateur =

Opérateur d'affectation qui affecte un `bad_cast` objet à un autre.

```
C++
```

```
bad_cast& operator=(const bad_cast&) noexcept;
```

quoi

```
C++
```

```
const char* what() const noexcept override;
```

Voir aussi

opérateur dynamic_cast
Mots clés
Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs

static_cast, opérateur

Article • 16/06/2023

Convertit une *expression* en type *type-id,* en fonction uniquement des types présents dans l'expression.

Syntaxe

```
static_cast <type-id> ( expression )
```

Notes

Dans C++ standard, aucun contrôle de type à l'exécution n'est fait pour garantir la sécurité de la conversion. Dans C++/CX, il est procédé à un calcul du temps de compilation et à un contrôle à l'exécution. Pour plus d'informations, consultez Cast.

L'opérateur `static_cast` peut être utilisé pour des opérations telles que la conversion d'un pointeur en classe de base en pointeur vers une classe dérivée. Ces conversions ne sont pas toujours sûres.

En général, vous utilisez `static_cast` quand vous souhaitez convertir des types de données numériques tels que les énumérations en ints ou ints en floats, et vous êtes certain des types de données impliqués dans la conversion. `static_cast`les conversions ne sont pas aussi sécurisées que `dynamic_cast` les conversions, car `static_cast` ne fait pas de type d'exécution case activée, alors que `dynamic_cast` fait. A `dynamic_cast` vers un pointeur ambigu échouera, tandis qu'un `static_cast` retourne comme si rien n'était mal ; cela peut être dangereux. Bien que `dynamic_cast` les conversions soient plus sécurisées, `dynamic_cast` ne fonctionne que sur des pointeurs ou des références, et le type d'exécution case activée est une surcharge. Pour plus d'informations, consultez Opérateur dynamic_cast.

Dans l'exemple qui suit, la ligne `D* pd2 = static_cast<D*>(pb);` n'est pas sécurisée car `D` peut avoir des champs et des méthodes qui ne sont pas dans `B`. Toutefois, la ligne `B* pb2 = static_cast<B*>(pd);` est une conversion sûre car `D` contient toujours tout le `B`.

```
C++
```

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
   D* pd2 = static_cast<D*>(pb);   // Not safe, D can have fields
                                   // and methods that are not in B.

   B* pb2 = static_cast<B*>(pd);   // Safe conversion, D always
                                   // contains all of B.
}
```

Contrairement à dynamic_cast, aucune case activée d'exécution n'est effectuée sur la `static_cast` conversion de `pb`. L'objet pointé par `pb` peut ne pas être un objet de type `D`, auquel cas l'utilisation de `*pd2` peut être désastreuse. Par exemple, appeler une fonction membre de la classe `D`, mais pas de la classe `B`, peut entraîner une violation d'accès.

Les `dynamic_cast` opérateurs et `static_cast` déplacent un pointeur dans une hiérarchie de classes. Toutefois, `static_cast` s'appuie exclusivement sur les informations fournies dans l'instruction cast et peut donc être dangereux. Par exemple :

C++

```
// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
   virtual void Test(){}
};
class D : public B {};

void f(B* pb) {
   D* pd1 = dynamic_cast<D*>(pb);
   D* pd2 = static_cast<D*>(pb);
}
```

Si `pb` pointe vraiment sur un objet de type `D`, alors `pd1` et `pd2` reçoivent la même valeur. Ils auront également la même valeur si `pb == 0`.

Si `pb` pointe vers un objet de type `B` et non vers la classe complète `D` , vous `dynamic_cast` en saurez suffisamment pour retourner zéro. Toutefois, `static_cast` s'appuie sur l'assertion du programmeur qui `pb` pointe vers un objet de type `D` et retourne simplement un pointeur vers cet objet supposé `D` .

Par conséquent, `static_cast` peut effectuer l'inverse des conversions implicites, auquel cas les résultats ne sont pas définis. Il appartient au programmeur de vérifier que les résultats d'une `static_cast` conversion sont sécurisés.

Ce comportement s'applique également aux types autres que les types de classe. Par instance, `static_cast` peut être utilisé pour convertir d'un int en . `char` Toutefois, le résultat `char` peut ne pas avoir suffisamment de bits pour contenir la valeur entière `int` . Là encore, il appartient au programmeur de vérifier que les résultats d'une `static_cast` conversion sont sécurisés.

L'opérateur `static_cast` peut également être utilisé pour effectuer toute conversion implicite, y compris les conversions standard et les conversions définies par l'utilisateur. Par exemple :

C++

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
   char ch;
   int i = 65;
   float f = 2.5;
   double dbl;

   ch = static_cast<char>(i);   // int to char
   dbl = static_cast<double>(f);   // float to double
   i = static_cast<BYTE>(ch);
}
```

L'opérateur `static_cast` peut convertir explicitement une valeur intégrale en type d'énumération. Si la valeur du type entier ne fait pas partie de la plage des valeurs d'énumération, la valeur d'énumération résultante n'est pas définie.

L'opérateur `static_cast` convertit une valeur de pointeur null en valeur de pointeur null du type de destination.

N'importe quelle expression peut être explicitement convertie en type void par l'opérateur `static_cast` . Le type void de destination peut éventuellement inclure l'attribut `const`, `volatile` ou `__unaligned` .

L'opérateur `static_cast` ne peut pas caster les `const` attributs , `volatile` ou `__unaligned` . Pour plus d'informations sur la suppression de ces attributs , consultez opérateur const_cast .

C++/CLI : En raison du risque d'effectuer des casts non vérifiés au-dessus d'un garbage collector en cours de déplacement, l'utilisation de `static_cast` ne doit être dans le code critique que lorsque vous êtes certain qu'il fonctionnera correctement. Si vous devez utiliser `static_cast` en mode mise en production, remplacez-le par safe_cast dans vos builds de débogage pour garantir la réussite.

Voir aussi

Opérateurs de casting
Mots clés

const_cast, opérateur

Article • 16/06/2023

Supprime le `const` ou les attributs , `volatile` et `__unaligned` d'une classe.

Syntaxe

```
const_cast <type-id> (expression)
```

Notes

Un pointeur vers n'importe quel type d'objet ou un pointeur vers un membre de données peut être explicitement converti en un type identique à l'exception des `const` qualificateurs , `volatile` et `__unaligned` . Pour les pointeurs et des références, le résultat fait référence à l'objet d'origine. Pour les pointeurs vers des données membres, le résultat fait référence au même membre que le pointeur d'origine (sans cast) vers les données membres. Selon le type de l'objet référencé, une opération d'écriture via le pointeur résultant, la référence, ou le pointeur vers les données membres peut entraîner un comportement non défini.

Vous ne pouvez pas utiliser l'opérateur pour remplacer directement la `const_cast` constante status d'une variable constante.

L'opérateur `const_cast` convertit une valeur de pointeur null en valeur de pointeur null du type de destination.

Exemple

C++

```
// expre_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
   void setNumber( int );
   void printNumber() const;
```

```
private:
   int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
   cout << "\nBefore: " << number;
   const_cast< CCTest * >( this )->number--;
   cout << "\nAfter: " << number;
}

int main() {
   CCTest X;
   X.setNumber( 8 );
   X.printNumber();
}
```

Sur la ligne contenant le **`const_cast`**, le type de données du **`this`** pointeur est `const CCTest *`. L'opérateur **`const_cast`** modifie le type de données du **`this`** pointeur en `CCTest *`, ce qui permet de modifier le membre `number` . Le cast dure uniquement pendant le reste de l'instruction dans laquelle il apparaît.

Voir aussi

Opérateurs de casting
Mots clés

reinterpret_cast, opérateur

Article • 03/04/2023

Autorise la conversion de tout pointeur en tout autre type pointeur. Autorise également la conversion de tout type entier en tout type pointeur et vice versa.

Syntaxe

```
reinterpret_cast < type-id > ( expression )
```

Notes

Une mauvaise utilisation de l'opérateur `reinterpret_cast` peut facilement être dangereuse. À moins que la conversion souhaitée soit fondamentalement de bas niveau, vous devez utiliser l'un des autres opérateurs de cast.

L'opérateur `reinterpret_cast` peut être utilisé pour les conversions telles que `char*` `int*` en , ou `One_class*` en `Unrelated_class*`, qui sont intrinsèquement dangereuses.

Le résultat d'un `reinterpret_cast` ne peut pas être utilisé en toute sécurité pour autre chose que d'être converti en son type d'origine. Les autres utilisations sont, au mieux, non portables.

L'opérateur `reinterpret_cast` ne peut pas caster les `const` attributs , `volatile` ou `__unaligned` . Pour plus d'informations sur la suppression de ces attributs , consultez opérateur const_cast .

L'opérateur `reinterpret_cast` convertit une valeur de pointeur null en valeur de pointeur null du type de destination.

Une utilisation pratique de est dans une fonction de `reinterpret_cast` hachage, qui mappe une valeur à un index de telle sorte que deux valeurs distinctes se retrouvent rarement avec le même index.

C++

```
#include <iostream>
using namespace std;
```

```
// Returns a hash code based on an address
unsigned short Hash( void *p ) {
   unsigned int val = reinterpret_cast<unsigned int>( p );
   return ( unsigned short )( val ^ (val >> 16));
}

using namespace std;
int main() {
   int a[20];
   for ( int i = 0; i < 20; i++ )
      cout << Hash( a + i ) << endl;
}

Output:
64641
64645
64889
64893
64881
64885
64873
64877
64865
64869
64857
64861
64849
64853
64841
64845
64833
64837
64825
64829
```

permet `reinterpret_cast` au pointeur d'être traité comme un type intégral. Le résultat binaire est alors déplacé et soumis à une opération XOR avec lui-même pour produire un index unique (à un niveau de probabilité élevé). L'index est ensuite tronqué par un cast de style C standard en type de retour de la fonction.

Voir aussi

Opérateurs de casting

Mots clés

Informations de type au moment de l'exécution

Article • 03/04/2023

Les informations de type au moment de l'exécution (RTTI) sont un mécanisme qui permet de déterminer le type d'un objet pendant l'exécution du programme. RTTI a été ajouté au langage C++ car de nombreux fournisseurs de bibliothèques de classes implémentaient cette fonctionnalité eux-mêmes. Cela provoquait des incompatibilités entre les bibliothèques. Par conséquent, il est devenu évident que la prise en charge des informations de type au moment de l'exécution s'avérait nécessaire au niveau du langage.

Pour des raisons de simplicité, cette discussion sur RTTI se limite presque exclusivement aux pointeurs. Toutefois, les concepts présentés également s'appliquent également aux références.

Il existe trois principaux éléments de langage C++ se rapportant aux informations de type au moment de l'exécution :

- Opérateur dynamic_cast .

  Utilisé pour la conversion des types polymorphes.

- Opérateur typeid .

  Utilisé pour identifier le type exact d'un objet.

- Classe type_info .

  Utilisé pour contenir les informations de type retournées par l'opérateur `typeid` .

Voir aussi

Cast

bad_typeid, exception

Article • 03/04/2023

L'exception bad_typeid est levée par l'opérateur typeid lorsque l'opérande pour `typeid` est un pointeur NULL.

Syntaxe

```
catch (bad_typeid)
   statement
```

Notes

L'interface pour **bad_typeid** est la suivante :

C++

```
class bad_typeid : public exception
{
public:
   bad_typeid();
   bad_typeid(const char * _Message = "bad typeid");
   bad_typeid(const bad_typeid &);
   virtual ~bad_typeid();

   bad_typeid& operator=(const bad_typeid&);
   const char* what() const;
};
```

L'exemple suivant montre l'opérateur `typeid` qui lève une exception **bad_typeid** .

C++

```
// expre_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class A{
public:
   // object for class needs vtable
   // for RTTI
   virtual ~A();
```

```
};

using namespace std;
int main() {
A* a = NULL;

try {
   cout << typeid(*a).name() << endl;  // Error condition
   }
catch (bad_typeid){
   cout << "Object is NULL" << endl;
   }
}
```

Sortie

Output

```
Object is NULL
```

Voir aussi

Informations de type au moment de l'exécution
Mots clés

type_info, classe

Article • 14/07/2023

La **classe type_info** décrit les informations de type générées dans le programme par le compilateur. Les objets de cette classe stockent efficacement un pointeur dans un nom pour le type. La classe **type_info** stocke également une valeur encodée adaptée à la comparaison de deux types à des fins d'égalité ou de classement. Les règles d'encodage et la séquence de classement pour les types ne sont pas spécifiées et peuvent différer entre les programmes.

Le `<typeinfo>` fichier d'en-tête doit être inclus pour pouvoir utiliser la classe **type_info** . L'interface de la classe **type_info** est la suivante :

C++

```
class type_info {
public:
    type_info(const type_info& rhs) = delete; // cannot be copied
    virtual ~type_info();
    size_t hash_code() const;
    _CRTIMP_PURE bool operator==(const type_info& rhs) const;
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    _CRTIMP_PURE bool operator!=(const type_info& rhs) const;
    _CRTIMP_PURE int before(const type_info& rhs) const;
    size_t hash_code() const noexcept;
    _CRTIMP_PURE const char* name() const;
    _CRTIMP_PURE const char* raw_name() const;
};
```

Vous ne pouvez pas instancier directement des objets de la classe **type_info** , car la classe n'a qu'un constructeur de copie privée. La seule façon de construire un objet **type_info** (temporaire) consiste à utiliser l'opérateur typeid . Étant donné que l'opérateur d'affectation est également privé, vous ne pouvez pas copier ou affecter des objets de **la classe type_info**.

`type_info::hash_code` définit une fonction de hachage adaptée au mappage de valeurs de type **typeinfo** à une distribution de valeurs d'index.

Les opérateurs `==` et `!=` peuvent être utilisés pour comparer l'égalité et l'inégalité avec d'autres objets **type_info** , respectivement.

Il n'existe aucun lien entre l'ordre de classement des types et les relations d'héritage. Utilisez la `type_info::before` fonction membre pour déterminer la séquence de classement des types. Il n'y a aucune garantie que `type_info::before` le résultat sera le

même dans des programmes différents ou même des exécutions différentes du même programme. De cette façon, `type_info::before` est similaire à l'opérateur address-of `(&)` .

La `type_info::name` fonction membre retourne un `const char*` à une chaîne terminée par null représentant le nom lisible par l'homme du type. La mémoire désignée est mise en cache et ne doit jamais être directement libérée.

La `type_info::raw_name` fonction membre est spécifique à Microsoft. Elle retourne un `const char*` à une chaîne terminée par null représentant le nom décoré du type d'objet. Le nom est stocké dans sa forme décorée pour économiser de l'espace. Par conséquent, cette fonction est plus rapide que `type_info::name` parce qu'elle n'a pas besoin de désécoérer le nom. La chaîne retournée par la fonction est utile dans les `type_info::raw_name` opérations de comparaison, mais n'est pas lisible. Si vous avez besoin d'une chaîne lisible par l'homme, utilisez `type_info::name` à la place.

Les informations de type sont générées pour les classes polymorphes uniquement si l'option du compilateur /GR (Enable Run-Time Type Information) est spécifiée.

Voir aussi

Informations de type au moment de l'exécution

Instructions (C++)

Article • 16/06/2023

Les instructions C++ sont les éléments de programme qui contrôlent comment et dans quel ordre les objets sont manipulés. Cette section comprend :

- Vue d'ensemble
- Instructions étiquetées
- Catégories d'instructions
  - Instructions d'expression. Ces instructions évaluent une expression pour ses effets secondaires ou pour sa valeur de retour.
  - Instructions Null. Ces instructions peuvent être fournies lorsqu'une instruction est requise par la syntaxe C++, mais où aucune action ne doit être effectuée.
  - Instructions composées. Ces instructions sont des groupes d'instructions placés dans des accolades ({ }). Ils peuvent être utilisés partout où une seule instruction peut être utilisée.
  - Instructions de sélection. Ces instructions effectuent un test ; ils exécutent ensuite une section de code si le test prend la valeur true (différente de zéro). Ils peuvent exécuter une autre section de code si le test prend la valeur false.
  - Instructions d'itération. Ces instructions permettent l'exécution répétée d'un bloc de code jusqu'à ce qu'un critère d'arrêt spécifié soit satisfait.
  - Instructions de saut. Ces instructions transfèrent immédiatement le contrôle à un autre emplacement de la fonction ou retournent le contrôle de la fonction.
  - Instructions de déclaration. Les déclarations introduisent un nom dans un programme.

Pour plus d'informations sur les instructions de gestion des exceptions, consultez Gestion des exceptions.

Voir aussi

Informations de référence sur le langage C++

Vue d'ensemble des instructions C++

Article • 03/04/2023

Les instructions C++ sont exécutées séquentiellement, sauf lorsqu'une instruction d'expression, une instruction de sélection, une instruction d'itération ou une instruction de saut modifie spécifiquement cette séquence.

Les instructions peuvent être des types suivants :

> *labeled-statement*
> *expression-statement*
> *compound-statement*
> *selection-statement*
> *iteration-statement*
> *jump-statement*
> *declaration-statement*
> *try-throw-catch*

Dans la plupart des cas, la syntaxe de l'instruction C++ est identique à celle d'ANSI C89. La principale différence entre les deux est que dans C89, les déclarations ne sont autorisées qu'au début d'un bloc ; C++ ajoute le *declaration-statement*, ce qui supprime efficacement cette restriction. Cela vous permet d'entrer des variables à un point du programme où une valeur d'initialisation précalculée peut être calculée.

La déclaration de variables à l'intérieur de blocs vous permet également d'exercer un contrôle précis sur la portée et la durée de vie de ces variables.

Les articles sur les instructions décrivent les mots clés C++ suivants :

break
case
catch
continue
default
do

else
__except
__finally

for
goto

if
__if_exists
__if_not_exists
__leave
return

switch
throw
__try
try
while

Voir aussi

Instructions

Instructions étiquetées

Article • 16/06/2023

Les étiquettes sont utilisées pour transférer le contrôle du programme directement à l'instruction spécifiée.

Syntaxe

> *`labeled-statement`*:
>
> *`identifier`* **`:`** *`statement`*
>
> **`case`** *`constant-expression`* **`:`** *`statement`*
>
> **`default`** **`:`** *`statement`*

L'étendue d'une étiquette est la fonction entière dans laquelle elle est déclarée.

Notes

Il existe trois types d'instructions étiquetées. Tous utilisent un signe deux-points (`:`) pour séparer un type d'étiquette de l'instruction. Les **`case`** étiquettes et **`default`** sont spécifiques aux instructions case.

C++

```
#include <iostream>
using namespace std;

void test_label(int x) {

    if (x == 1){
        goto label1;
    }
    goto label2;

label1:
    cout << "in label1" << endl;
    return;

label2:
    cout << "in label2" << endl;
    return;
}

int main() {
    test_label(1);  // in label1
```

```
    test_label(2);  // in label2
}
```

Étiquettes et instruction `goto`

L'apparence d'une *`identifier`* étiquette dans le programme source déclare une étiquette. Seule une goto instruction peut transférer le contrôle vers une *`identifier`* étiquette. Le fragment de code suivant illustre l'utilisation de l'instruction **`goto`** et d'une *`identifier`* étiquette :

Une étiquette ne peut pas apparaître d'elle-même, mais doit toujours être jointe à une instruction. Si une étiquette doit apparaître seule, placez une instruction null après l'étiquette.

L'étiquette a une étendue de fonction et ne peut pas être redeclaré dans la fonction. Toutefois, le même nom peut être utilisé en tant qu'étiquette dans différentes fonctions.

C++

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
   using namespace std;
   goto Test2;

   cout << "testing" << endl;

   Test2:
      cerr << "At Test2 label." << endl;
}

//Output: At Test2 label.
```

Étiquettes dans l'instruction `case`

Les étiquettes qui apparaissent après l'mot clé **`case`** ne peuvent pas également apparaître en dehors d'une **`switch`** instruction. (Cette restriction s'applique également aux **`default`** mot clé.) Le fragment de code suivant montre l'utilisation correcte des **`case`** étiquettes :

C++

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
   case WM_TIMER:    // Process timer event.
      SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
      ShowWindow( hWnd, SW_SHOWNA );
      nIcon %= 14;
      Yield();
      break;

   case WM_PAINT:
      memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
      hDC = BeginPaint( hWnd, &ps );
      EndPaint( hWnd, &ps );
      break;

   case WM_CLOSE:
      KillTimer( hWnd, TIMER1 );
      DestroyWindow( hWnd );
      if ( hWnd == hWndMain )
         PostQuitMessage( 0 );  // Quit the application.
      break;

   default:
      // This choice is taken for all messages not specifically
      //  covered by a case statement.
      return DefWindowProc( hWnd, Message, wParam, lParam );
      break;
}
```

Voir aussi

Vue d'ensemble des instructions C++

Instruction switch (C++)

Instruction d'expression

Article • 03/04/2023

Les instructions d'expression entraînent des expressions à évaluer. Aucun transfert de contrôle ou d'itération n'a lieu à la suite d'une instruction d'expression.

La syntaxe de l'instruction d'expression est simplement

Syntaxe

```
[expression ] ;
```

Notes

Toutes les expressions appartenant à une instruction d'expression sont évaluées et tous les effets secondaires sont terminés avant l'exécution de l'instruction suivante. Les instructions d'expression les plus courantes sont les assignations et les appels de fonction. Étant donné que l'expression est facultative, un point-virgule seul est considéré comme une instruction d'expression vide, appelée instruction null .

Voir aussi

Vue d'ensemble des instructions C++

null, instruction

Article • 16/06/2023

L'instruction « null » est une instruction d'expression avec *l'expression* manquante. Elle s'avère utile lorsque la syntaxe du langage demande une instruction, mais pas d'évaluation d'expression. Elle se compose d'un point-virgule.

Les instructions null sont couramment utilisées comme espaces réservés dans les instructions d'itération ou comme instructions sur lesquelles placer des étiquettes à la fin d'instructions composées ou de fonctions.

Le fragment de code suivant montre comment copier une chaîne vers une autre et comporte l'instruction null :

C++

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ;   // Null statement.

    return DestStart;
}

int main()
{
}
```

Voir aussi

Expression, instruction

Instructions composées (blocs)

Article • 16/06/2023

Une instruction composée se compose de zéro ou plus d'instructions placées dans des accolades ({ }). Une instruction composée peut être utilisée partout où une instruction est attendue. Les instructions composées sont généralement appelées « blocs ».

Syntaxe

```
{ [ statement-list ] }
```

Notes

L'exemple suivant utilise une instruction composée comme partie *d'instruction* de l'instruction `if` (consultez l'instruction if pour plus d'informations sur la syntaxe) :

C++

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

> ⓘ Notes
>
> Étant donné qu'une déclaration est une instruction, une déclaration peut être l'une des instructions de la *liste des instructions*. Par conséquent, les noms déclarés au sein d'une instruction composée, mais pas explicitement déclarés comme statiques, ont une portée locale et (pour les objets) une durée de vie locale. Pour plus d'informations sur le traitement des noms avec une étendue locale, consultez **Étendue** .

Voir aussi

Vue d’ensemble des instructions C++

Instructions de sélection (C++)

Article • 03/04/2023

Les instructions de sélection C++, si et switch, fournissent un moyen d'exécuter de manière conditionnelle des sections de code.

Les instructions __if_exists et __if_not_exists vous permettent d'inclure du code de manière conditionnelle en fonction de l'existence d'un symbole.

Consultez les différentes rubriques concernant la syntaxe de ces instructions.

Voir aussi

Vue d'ensemble des instructions C++

if-else, instruction (C++)

Article • 03/04/2023

Une instruction if-else contrôle la branche conditionnelle. Les instructions dans le *`if-branch`* ne sont exécutées que si le *`condition`* est évalué à une valeur différente de zéro (ou **`true`**). Si la valeur de est différente de *`condition`* zéro, l'instruction suivante est exécutée et l'instruction qui suit l'option est **`else`** ignorée. Sinon, l'instruction suivante est ignorée et, s'il existe une **`else`** instruction, l'instruction qui suit est **`else`** exécutée.

`condition` les expressions qui n'ont pas la valeur zéro sont les suivantes :

- **`true`**
- pointeur non null,
- toute valeur arithmétique autre que zéro, ou
- type de classe qui définit une conversion non ambiguë en type arithmétique, booléen ou pointeur. (Pour plus d'informations sur les conversions, consultez Conversions standard.)

Syntaxe

`init-statement`:

`expression-statement`

`simple-declaration`

`condition`:

`expression`

`attribute-specifier-seq`$_{opt}$ *`decl-specifier-seq`* *`declarator`* *`brace-or-equal-initializer`*

`statement`:

`expression-statement`

`compound-statement`

`expression-statement`:

`expression`$_{opt}$ **`;`**

`compound-statement`:

`{` *`statement-seq`*$_{opt}$ **`}`**

statement-seq:

statement

statement-seq *statement*

if-branch:

statement

else-branch:

statement

selection-statement:

if **constexpr**$_{opt}$[17] **(** *init-statement*$_{opt}$[17] *condition* **)** *if-branch*

if **constexpr**$_{opt}$[17] **(** *init-statement*$_{opt}$[17] *condition* **)** *if-branch* **else** *else-branch*

[17] Cet élément facultatif est disponible à partir de C++17.

instructions if-else

Dans toutes les formes de l'instruction **if** , *condition*, qui peut avoir n'importe quelle valeur à l'exception d'une structure, est évalué, y compris tous les effets secondaires. Le contrôle passe de l'instruction **if** à l'instruction suivante dans le programme, sauf si le exécuté *if-branch* ou *else-branch* contient un break, continueou goto.

La **else** clause d'une **if...else** instruction est associée à l'instruction précédente **if** la plus proche dans la même étendue que celle qui n'a pas d'instruction correspondante **else** .

Exemple

Cet exemple de code montre plusieurs **if** instructions en cours d'utilisation, à la fois avec et sans **else**:

C++

```
// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
    public:
    void do_something(){}
};
```

```
void init(C){}
bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n";  // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}
```

instruction if avec un initialiseur

À partir de C++17, une `if` instruction peut également contenir une *`init-statement`* expression qui déclare et initialise une variable nommée. Utilisez cette forme de l'instruction if lorsque la variable est uniquement nécessaire dans l'étendue de l'instruction if. **Spécifique à Microsoft** : ce formulaire est disponible à partir de Visual Studio 2017 version 15.3 et nécessite au moins l'option du /std:c++17 compilateur.

Exemple

C++

```
#include <iostream>
#include <mutex>
#include <map>
#include <string>
```

```
#include <algorithm>

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{

    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(),
keywords.end(), [&s](const char* kw) { return s == kw; }))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}
```

instructions si constexpr

À partir de C++17, vous pouvez utiliser une `if constexpr` instruction dans les modèles de fonction pour prendre des décisions de branchement au moment de la compilation sans avoir à recourir à plusieurs surcharges de fonction. **Spécifique à Microsoft** : ce formulaire est disponible à partir de Visual Studio 2017 version 15.3 et nécessite au moins l'option du /std:c++17 compilateur.

Exemple

Cet exemple montre comment écrire une seule fonction qui gère le décompressage des paramètres. Aucune surcharge de paramètre zéro n'est nécessaire :

C++

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

Voir aussi

Instructions de sélection

Mots clés

Instruction switch (C++)

__if_exists, instruction

Article • 03/04/2023

L'instruction `__if_exists` teste si l'identificateur spécifié existe. Si l'identificateur existe, le bloc d'instructions spécifié est exécuté.

Syntaxe

```
__if_exists ( identifier ) {
statements
};
```

Paramètres

identifier
Identificateur dont vous voulez tester l'existence.

Déclarations
Une ou plusieurs instructions à exécuter si *l'identificateur* existe.

Notes

⊗ Attention

Pour obtenir les résultats les plus fiables, utilisez l'instruction `__if_exists` sous les contraintes suivantes.

- Appliquez l'instruction `__if_exists` uniquement aux types simples, pas aux modèles.
- Appliquez l'instruction `__if_exists` aux identificateurs à l'intérieur ou à l'extérieur d'une classe. N'appliquez pas l'instruction `__if_exists` aux variables locales.
- Utilisez l'instruction `__if_exists` uniquement dans le corps d'une fonction. En dehors du corps d'une fonction, l'instruction `__if_exists` peut tester uniquement les types entièrement définis.

- Lorsque vous vérifiez la présence de fonctions surchargées, vous ne pouvez pas effectuer le test sur une forme spécifique de la surcharge.

Le complément de l'instruction `__if_exists` est l'instruction __if_not_exists .

Exemple

Notez que cet exemple utilise des modèles, ce qui n'est pas recommandé.

C++

```
// the__if_exists_statement.cpp
// compile with: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
   void Dump() {
      std::cout << "In X<T>::Dump()" << std::endl;

      __if_exists(T::Dump) {
         T::Dump();
      }

      __if_not_exists(T::Dump) {
         std::cout << "T::Dump does not exist" << std::endl;
      }
   }
};

class A {
public:
   void Dump() {
      std::cout << "In A::Dump()" << std::endl;
   }
};

class B {};

bool g_bFlag = true;

class C {
public:
   void f(int);
   void f(double);
};

int main() {
   X<A> x1;
   X<B> x2;
```

```
    x1.Dump();
    x2.Dump();

    __if_exists(::g_bFlag) {
        std::cout << "g_bFlag = " << g_bFlag << std::endl;
    }

    __if_exists(C::f) {
        std::cout << "C::f exists" << std::endl;
    }

    return 0;
}
```

Sortie

Output

```
In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists
```

Voir aussi

Instructions de sélection

Mots clés

instruction __if_not_exists

__if_not_exists, instruction

Article • 03/04/2023

L'instruction `__if_not_exists` teste si l'identificateur spécifié existe. S'il n'existe pas, le bloc d'instructions spécifié est exécuté.

Syntaxe

```
__if_not_exists ( identifier ) {
statements
};
```

Paramètres

identifier
Identificateur dont vous voulez tester l'existence.

Déclarations
Une ou plusieurs instructions à exécuter si *l'identificateur* n'existe pas.

Notes

> ⊗ **Attention**
>
> Pour obtenir les résultats les plus fiables, utilisez l'instruction `__if_not_exists` sous les contraintes suivantes.

- Appliquez l'instruction `__if_not_exists` uniquement aux types simples, pas aux modèles.
- Appliquez l'instruction `__if_not_exists` aux identificateurs à l'intérieur ou à l'extérieur d'une classe. N'appliquez pas l'instruction `__if_not_exists` aux variables locales.
- Utilisez l'instruction `__if_not_exists` uniquement dans le corps d'une fonction. En dehors du corps d'une fonction, l'instruction `__if_not_exists` peut tester uniquement les types entièrement définis.

- Lorsque vous vérifiez la présence de fonctions surchargées, vous ne pouvez pas effectuer le test sur une forme spécifique de la surcharge.

Le complément de l'instruction **`__if_not_exists`** est l'instruction __if_exists .

Exemple

Pour obtenir un exemple d'utilisation **`__if_not_exists`**, consultez instruction __if_exists.

Voir aussi

Instructions de sélection
Mots clés
instruction __if_exists

`switch` instruction (C++)

Article • 03/04/2023

Autorise la sélection parmi plusieurs sections de code, selon la valeur d'une expression intégrale.

Syntaxe

> *selection-statement*:
>
> **switch** **(** *init-statement*$_{opt}$ ^C++17^ *condition* **)** *statement*

> *init-statement*:
>
> *expression-statement*
>
> *simple-declaration*

> *condition*:
>
> *expression*
>
> *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq* *declarator* *brace-or-equal-initializer*

> *labeled-statement*:
>
> **case** *constant-expression* **:** *statement*
>
> **default** **:** *statement*

Notes

Une instruction **switch** entraîne le transfert du contrôle à une *labeled-statement* dans son corps d'instruction, en fonction de la valeur de *condition*.

Le *condition* doit avoir un type intégral, ou être un type de classe qui a une conversion non ambiguë en type intégral. La promotion intégrale a lieu comme décrit dans Conversions standard.

Le **switch** corps de l'instruction se compose d'une série d'étiquettes **case** et d'une optétiquette ional **default** . A *labeled-statement* est l'une de ces étiquettes et les instructions qui suivent. Les instructions étiquetées ne sont pas des exigences syntaxiques, mais l'instruction **switch** n'a aucun sens sans elles. *constant-expression*

Deux valeurs dans `case` les instructions ne peuvent pas être évaluées à la même valeur. L'étiquette `default` ne peut apparaître qu'une seule fois. L'instruction `default` est souvent placée à la fin, mais elle peut apparaître n'importe où dans le corps de l'instruction `switch` . Une étiquette `case` ou `default` ne peut apparaître qu'à l'intérieur d'une instruction `switch`.

Le *`constant-expression`* dans chaque `case` étiquette est converti en une valeur constante du même type que *`condition`*. Ensuite, il est comparé à *`condition`* pour l'égalité. Le contrôle passe à la première instruction après la `case` *`constant-expression`* valeur qui correspond à la valeur de *`condition`*. Le comportement résultant est indiqué dans le tableau suivant.

`switch` comportement de l'instruction

Condition	Action
La valeur convertie correspond à celle de l'expression de contrôle promue.	Le contrôle est transféré à l'instruction qui suit cette étiquette.
Aucune des constantes ne correspond aux constantes dans les `case` étiquettes ; une `default` étiquette est présente.	Le contrôle est transféré à l'étiquette `default` .
Aucune des constantes ne correspond aux constantes dans les `case` étiquettes ; aucune étiquette n'est `default` présente.	Le contrôle est transféré à l'instruction après l'instruction `switch` .

Si une expression correspondante est trouvée, l'exécution peut se poursuivre via des étiquettes ultérieures `case` ou `default` . L'instruction break est utilisée pour arrêter l'exécution et transférer le contrôle à l'instruction après l'instruction `switch` . `break` Sans instruction, chaque instruction de l'étiquette correspondante `case` à la fin de , `switch`y compris le `default`, est exécutée. Par exemple :

C++

```
// switch_statement1.cpp
#include <stdio.h>

int main() {
   const char *buffer = "Any character stream";
   int uppercase_A, lowercase_a, other;
   char c;
   uppercase_A = lowercase_a = other = 0;

   while ( c = *buffer++ )   // Walks buffer until NULL
```

```
    {
       switch ( c )
       {
          case 'A':
             uppercase_A++;
             break;
          case 'a':
             lowercase_a++;
             break;
          default:
             other++;
       }
    }
    printf_s( "\nUppercase A: %d\nLowercase a: %d\nTotal: %d\n",
       uppercase_A, lowercase_a, (uppercase_A + lowercase_a + other) );
}
```

Dans l'exemple ci-dessus, `uppercase_A` est incrémenté si `c` est une valeur supérieurecase `'A'`. L'instruction **`break`** après `uppercase_A++` met fin à l'exécution du corps de l'instruction **`switch`** et le contrôle passe à la **`while`** boucle. Sans l'instruction **`break`** , l'exécution serait « passée » à l'instruction étiquetée suivante, de sorte que `lowercase_a` et `other` seraient également incrémentés. L'instruction **`break`** pour `case 'a'`. Si `c` est inférieurcase `'a'`, `lowercase_a` est incrémenté et l'instruction **`break`** termine le corps de l'instruction **`switch`** . Si `c` n'est pas un `'a'` ou `'A'`, l'instruction **`default`** est exécutée.

Visual Studio 2017 et versions ultérieures (disponible en /std:c++17 mode et versions ultérieures) : l'attribut `[[fallthrough]]` est spécifié dans la norme C++17. Vous pouvez l'utiliser dans une **`switch`** instruction . Il s'agit d'un indice pour le compilateur, ou toute personne qui lit le code, que le comportement de basculement est intentionnel. Actuellement, le compilateur Microsoft C++ n'avertit pas du comportement de basculement. Cet attribut n'a donc aucun effet sur le comportement du compilateur. Dans l'exemple, l'attribut est appliqué à une instruction vide dans l'instruction étiquetée non déterminée. En d'autres termes, le point-virgule est nécessaire.

C++

```
int main()
{
    int n = 5;
    switch (n)
    {

    case 1:
        a();
        break;
    case 2:
```

```
        b();
        d();
        [[fallthrough]]; // I meant to do this!
    case 3:
        c();
        break;
    default:
        d();
        break;
    }

    return 0;
}
```

Visual Studio 2017 version 15.3 et ultérieures (disponible en /std:c++17 mode et versions ultérieures) : une `switch` instruction peut avoir une *`init-statement`* clause, qui se termine par un point-virgule. Il introduit et initialise une variable dont l'étendue est limitée au bloc de l'instruction `switch` :

C++

```
    switch (Gadget gadget(args); auto s = gadget.get_status())
    {
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
    };
```

Un bloc interne d'une `switch` instruction peut contenir des définitions avec des initialiseurs tant qu'ils sont *accessibles*, c'est-à-dire qu'ils ne sont pas contournés par tous les chemins d'exécution possibles. Les noms présentés à l'aide de ces déclarations ont une portée locale. Par exemple :

C++

```
// switch_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    switch( tolower( *argv[1] ) )
    {
        // Error. Unreachable declaration.
        char szChEntered[] = "Character entered was: ";

    case 'a' :
```

```
        {
        // Declaration of szChEntered OK. Local scope.
        char szChEntered[] = "Character entered was: ";
        cout << szChEntered << "a\n";
        }
        break;

    case 'b' :
        // Value of szChEntered undefined.
        cout << szChEntered << "b\n";
        break;

    default:
        // Value of szChEntered undefined.
        cout << szChEntered << "neither a nor b\n";
        break;
    }
}
```

Une `switch` instruction peut être imbriquée. Lorsqu'elles sont imbriquées, les `case` étiquettes ou s'associent `default` à l'instruction la plus `switch` proche qui les entoure.

Comportement spécifique à Microsoft

Microsoft C++ ne limite pas le nombre de `case` valeurs dans une `switch` instruction. Le nombre est limité uniquement par la mémoire disponible.

Voir aussi

Instructions de sélection
Mots clés

Instructions d'itération (C++)

Article • 16/06/2023

Les instructions d'itération entraînent une exécution des instructions (ou des instructions composées) zéro ou plusieurs fois, compte tenu de certains critères de terminaison des boucles. Lorsque ces instructions sont des instructions composées, elles sont exécutées dans l'ordre, sauf lorsque l'instruction break ou l'instruction continue est rencontrée.

C++ fournit quatre instructions d'itération : while, do, for et range-based for. Chacune de ces itérations jusqu'à ce que son expression de terminaison soit évaluée à zéro (false), ou jusqu'à ce que l'arrêt de boucle soit forcé avec une `break` instruction. Le tableau suivant résume ces instructions et leurs actions ; chacune est décrit en détail dans les sections suivantes.

Instructions d'itération

.	Évaluée en	Initialisation	Incrément
`while`	Début de boucle	Non	Non
`do`	Fin de boucle	Non	Non
`for`	Début de boucle	Oui	Oui
range-based for	Début de boucle	Oui	Oui

La partie instruction d'une instruction d'itération ne peut pas être une déclaration. Toutefois, il peut s'agir d'une instruction composée contenant une déclaration.

Voir aussi

Vue d'ensemble des instructions C++

while, instruction (C++)

Article • 03/04/2023

Exécute *l'instruction* à plusieurs reprises jusqu'à ce que *l'expression* soit évaluée à zéro.

Syntaxe

```
while ( expression )
   statement
```

Notes

Le test de *l'expression* a lieu avant chaque exécution de la boucle ; par conséquent, une `while` boucle s'exécute zéro ou plusieurs fois. *expression* doit être d'un type intégral, d'un type pointeur ou d'un type de classe avec une conversion non ambiguë en type intégral ou pointeur.

Une `while` boucle peut également se terminer lorsqu'un break, un goto ou un retour dans le corps de l'instruction est exécuté. Utilisez continuer pour terminer l'itération actuelle sans quitter la `while` boucle. `continue` passe le contrôle à l'itération suivante de la `while` boucle.

Le code suivant utilise une `while` boucle pour réduire les traits de soulignement de fin d'une chaîne :

C++

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    //  Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    //  iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';
```

```
    return szSource;
}
int main()
{
    char szbuf[] = "12345_____";

    printf_s("\nBefore trim: %s", szbuf);
    printf_s("\nAfter trim: %s\n", trim(szbuf));
}
```

La condition d'arrêt est évaluée en haut de la boucle. S'il n'y a pas de traits de soulignement de fin, la boucle ne s'exécute jamais.

Voir aussi

Instructions d'itération
Mots clés
do-while, instruction (C++)
for Statement (C++)
Basé sur une plage, instruction (C++)

do-while, instruction (C++)

Article • 03/04/2023

Exécute une *instruction* à plusieurs reprises jusqu'à ce que la condition d'arrêt spécifiée (*l'expression*) soit évaluée à zéro.

Syntaxe

```
do
   statement
while ( expression ) ;
```

Notes

Le test de la condition d'arrêt est effectué après chaque exécution de la boucle ; par conséquent, une boucle **do-while** s'exécute une ou plusieurs fois, en fonction de la valeur de l'expression de terminaison. L'instruction **do-while** peut également se terminer lorsqu'une instruction break, goto ou return est exécutée dans le corps de l'instruction.

L'élément *expression* doit être de type arithmétique ou pointeur. L'exécution se déroule comme suit :

1. Le corps de l'instruction est exécuté.
2. Ensuite, l'élément *expression* est évalué. Si l'élément *expression* est false, l'instruction **do-while** se termine et le contrôle passe à l'instruction suivante du programme. Si l'élément *expression* est true (différent de zéro), le processus se répète, en commençant à l'étape 1.

Exemple

L'exemple suivant illustre l'instruction **do-while** :

C++

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
```

```
    do
    {
        printf_s("\n%d",i++);
    } while (i < 3);
}
```

Voir aussi

Instructions d'itération
Mots clés
while, instruction (C++)
for Statement (C++)
Basé sur une plage, instruction (C++)

`for` instruction (C++)

Article • 03/04/2023

Exécute une instruction à plusieurs reprises jusqu'à ce que la condition soit false. Pour plus d'informations sur l'instruction basée sur `for` la plage, consultez Instruction basée sur la for plage (C++) . Pour plus d'informations sur l'instruction C++/CLI `for each` , consultez for each, in.

Syntaxe

> **`for (`** *`init-expression`* **`;`** *`cond-expression`* **`;`** *`loop-expression`* **`)`**
> *`statement`*

Notes

Utilisez l'instruction `for` pour construire des boucles qui doivent exécuter un nombre spécifié de fois.

L'instruction `for` se compose de trois parties facultatives, comme indiqué dans le tableau suivant.

pour les éléments de boucle

Nom de la syntaxe	En cas d'exécution	Description
`init-expression`	Avant tout autre élément de l'instruction `for` , *`init-expression`* n'est exécuté qu'une seule fois. Le contrôle passe alors à *`cond-expression`*.	Souvent employé pour initialiser des index de boucle. Elle peut contenir des expressions ou des déclarations.
`cond-expression`	Avant l'exécution de chaque itération de *`statement`*, y compris la première itération. *`statement`* est exécuté uniquement si *`cond-expression`* a la valeur true (valeur différente de zéro).	Expression qui correspond à un type intégral ou à un type de classe avec une conversion non ambiguë en type intégral. Normalement utilisée pour déterminer les critères d'arrêts de boucles.
`loop-expression`	À la fin de chaque itération de *`statement`*. *`loop-expression`* est évalué après l'exécution de *`cond-expression`*.	Normalement utilisée pour incrémenter les index de boucle.

Les exemples suivants montrent différentes façons d'utiliser l'instruction `for` .

C++

```
#include <iostream>
using namespace std;

int main() {
    // The counter variable can be declared in the init-expression.
    for (int i = 0; i < 2; i++ ){
       cout << i;
    }
    // Output: 01
    // The counter variable can be declared outside the for loop.
    int i;
    for (i = 0; i < 2; i++){
        cout << i;
    }
    // Output: 01
    // These for loops are the equivalent of a while loop.
    i = 0;
    while (i < 2){
        cout << i++;
    }
    // Output: 01
}
```

`init-expression` et *`loop-expression`* peuvent contenir plusieurs instructions séparées par des virgules. Par exemple :

C++

```
#include <iostream>
using namespace std;

int main(){
    int i, j;
    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {
        cout << "i + j = " << (i + j) << '\n';
    }
}
/* Output:
    i + j = 15
    i + j = 17
    i + j = 19
*/
```

`loop-expression` peut être incrémenté ou décrémenté, ou autrement modifié.

C++

```
#include <iostream>
using namespace std;

int main(){
for (int i = 10; i > 0; i--) {
        cout << i << ' ';
    }
    // Output: 10 9 8 7 6 5 4 3 2 1
    for (int i = 10; i < 20; i = i+2) {
        cout << i << ' ';
    }
}
// Output: 10 12 14 16 18
```

Une `for` boucle se termine lorsqu'une breakinstruction , retourne ou goto (à une instruction étiquetée en dehors de la `for` boucle) dans *`statement`* est exécutée. Une continue instruction d'une `for` boucle met fin uniquement à l'itération actuelle.

Si *`cond-expression`* est omis, il est considéré comme `true`et la `for` boucle ne se termine pas sans , `break` `return`ou `goto` dans *`statement`*.

Bien que les trois champs de l'instruction soient normalement utilisés pour l'initialisation `for` , le test d'arrêt et l'incrémentation, ils ne sont pas limités à ces utilisations. Par exemple, le code suivant affiche les nombres 0 à 4. Dans ce cas, *`statement`* est l'instruction null :

C++

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

`for` boucles et C++ Standard

La norme C++ indique qu'une variable déclarée dans une `for` boucle doit sortir de l'étendue une fois la `for` boucle terminée. Par exemple :

C++

```
for (int i = 0 ; i < 5 ; i++) {
   // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

Par défaut, sous /Ze, une variable déclarée dans une `for` boucle reste dans l'étendue jusqu'à ce que l'étendue englobante de la `for` boucle se termine.

/Zc:forScope active le comportement standard des variables déclarées dans pour les boucles sans avoir à spécifier `/Za`.

Il est également possible d'utiliser les différences d'étendue de la `for` boucle pour redéclarer les variables sous `/Ze` les étapes suivantes :

C++

```
// for_statement5.cpp
int main(){
   int i = 0;   // hidden by var with same name declared in for loop
   for ( int i = 0 ; i < 3; i++ ) {}

   for ( int i = 0 ; i < 3; i++ ) {}
}
```

Ce comportement imite plus étroitement le comportement standard d'une variable déclarée dans une `for` boucle, ce qui nécessite que les variables déclarées dans une `for` boucle sortent de l'étendue une fois la boucle terminée. Lorsqu'une variable est déclarée dans une `for` boucle, le compilateur la promeut en interne en tant que variable locale dans l'étendue englobante de la `for` boucle. Il est promu même s'il existe déjà une variable locale portant le même nom.

Voir aussi

Instructions d'itération
Mots clés
Instruction while (C++)
Instruction do-while (C++)
Instruction basée sur la plage for (C++)

Basé sur une plage, instruction (C++)

Article • 16/06/2023

S'exécute `statement` de manière répétée et séquentielle pour chaque élément dans `expression`.

Syntaxe

> **`for (`** *for-range-declaration* **`:`** *Expression* **`)`**
>
> *instruction*

Notes

Utilisez l'instruction basée sur **`for`** une plage pour construire des boucles qui doivent s'exécuter via une *plage*, qui est définie comme tout ce que vous pouvez itérer, par exemple, `std::vector` ou toute autre séquence de bibliothèque standard C++ dont la plage est définie par un `begin()` et `end()`. Le nom déclaré dans la `for-range-declaration` partie est local de l'instruction **`for`** et ne peut pas être déclaré à nouveau dans `expression` ou `statement`. Notez que la auto mot clé est préférée dans la `for-range-declaration` partie de l'instruction .

Nouveautés de Visual Studio 2017 : Les boucles basées sur **`for`** des plages n'ont plus besoin de ce `begin()` et `end()` retournent des objets du même type. Cela permet `end()` de retourner un objet sentinel tel qu'utilisé par les plages, comme défini dans la proposition Ranges-V3. Pour plus d'informations, consultez Généralisation de la boucle Range-Based For et de la bibliothèque range-v3 sur GitHub .

Ce code montre comment utiliser des boucles basées sur **`for`** une plage pour itérer au sein d'un tableau et d'un vecteur :

C++

```
// range-based-for.cpp
// compile by using: cl /EHsc /nologo /W4
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
    // Range-based for loop to iterate through the array.
    for( int y : x ) { // Access by value using a copy declared as a
specific type.
                       // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is
needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-
place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}
```

Voici la sortie :

Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

```
1 2 3 4 5 6 7 8 9 10
end of integer array test

0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159
9.14159
end of vector test
```

Une boucle basée sur `for` une plage se termine quand l'un de ces éléments dans `statement` est exécuté : une breakinstruction , returnou goto à une instruction étiquetée en dehors de la boucle basée sur `for` la plage. Une continue instruction dans une boucle basée sur `for` une plage termine uniquement l'itération actuelle.

Gardez à l'esprit ces faits sur la plage :`for`

- Reconnaît automatiquement les tableaux.
- Reconnaît les conteneurs qui ont `.begin()` et `.end()`.
- Utilise la recherche dépendante des arguments `begin()` et `end()` pour tout autre chose.

Voir aussi

auto
Instructions d'itération
Mots clés
Instruction while (C++)
Instruction do-while (C++)
Instruction for (C++)

Instructions de saut (C++)

Article • 16/06/2023

Une instruction de saut C++ exécute un transfert de contrôle local immédiat.

Syntaxe

```
break;
continue;
return [expression];
goto identifier;
```

Notes

Consultez les rubriques suivantes pour une description des instructions de saut C++.

- Instruction break
- continue, instruction
- return, instruction
- goto, instruction

Voir aussi

Vue d'ensemble des instructions C++

break, instruction (C++)

Article • 16/06/2023

L'instruction `break` met fin à l'exécution de la boucle englobante ou de l'instruction conditionnelle la plus proche dans laquelle elle apparaît. Le contrôle est passé à l'instruction qui suit l'instruction terminée, le cas échéant.

Syntaxe

```
break;
```

Notes

L'instruction `break` est utilisée avec l'instruction de commutateur conditionnel et avec les instructions de boucle do, for et while .

Dans une `switch` instruction, l'instruction `break` amène le programme à exécuter l'instruction suivante en dehors de l'instruction `switch` . `break` Sans instruction, chaque instruction de l'étiquette correspondante `case` à la fin de l'instruction `switch`, y compris la `default` clause, est exécutée.

Dans les boucles, l'instruction `break` met fin à l'exécution de l'instruction , `for` ou `while` englobante `do` la plus proche. Le contrôle est passé à l'instruction qui suit l'instruction terminée, le cas échéant.

Dans les instructions imbriquées, l'instruction `break` termine uniquement l'instruction `do`, `for`, `switch` ou `while` qui l'entoure immédiatement. Vous pouvez utiliser une `return` instruction ou `goto` pour transférer le contrôle à partir de structures imbriquées plus profondément.

Exemple

Le code suivant montre comment utiliser l'instruction `break` dans une `for` boucle.

```
C++
```

```
#include <iostream>
using namespace std;

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }

    // An example of a range-based for loop
int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i : nums) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }
}
```

Output

```
In each case:
1
2
3
```

Le code suivant montre comment utiliser `break` dans une `while` boucle et une `do` boucle.

C++

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }
```

```
    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}
```

Output

```
In each case:
0123
```

Le code suivant montre comment utiliser `break` dans une instruction switch. Vous devez utiliser `break` dans tous les cas si vous souhaitez gérer chaque cas séparément ; si vous n'utilisez `break` pas , l'exécution du code passe au cas suivant.

C++

```
#include <iostream>
using namespace std;

enum Suit{ Diamonds, Hearts, Clubs, Spades };

int main() {

    Suit hand;
    . . .
    // Assume that some enum value is set for hand
    // In this example, each case is handled separately
    switch (hand)
    {
    case Diamonds:
        cout << "got Diamonds \n";
        break;
    case Hearts:
        cout << "got Hearts \n";
        break;
    case Clubs:
        cout << "got Clubs \n";
        break;
    case Spades:
        cout << "got Spades \n";
        break;
    default:
          cout << "didn't get card \n";
    }
    // In this example, Diamonds and Hearts are handled one way, and
```

```
    // Clubs, Spades, and the default value are handled another way
    switch (hand)
    {
    case Diamonds:
    case Hearts:
        cout << "got a red card \n";
        break;
    case Clubs:
    case Spades:
    default:
        cout << "didn't get a red card \n";
    }
}
```

Voir aussi

Instructions de saut

Mots clés

continue, instruction

continue, instruction (C++)

Article • 16/06/2023

Force le transfert de contrôle à l'expression de contrôle de la plus petite boucle englobante do, for ou while .

Syntaxe

```
continue;
```

Notes

Toutes les instructions restantes dans l'itération actuelle ne sont pas exécutées. L'itération suivante de la boucle est déterminée comme suit :

- Dans une `do` boucle ou `while` , l'itération suivante commence par réévaluer l'expression de contrôle de l'instruction `do` ou `while` .
- Dans une `for` boucle (à l'aide de la syntaxe `for( <init-expr> ; <cond-expr> ; <loop-expr> )`), la `<loop-expr>` clause est exécutée. Ensuite la clause `<cond-expr>` est réévaluée et, selon le résultat, la boucle se termine ou une autre itération a lieu.

L'exemple suivant montre comment l'instruction `continue` peut être utilisée pour contourner des sections de code et commencer l'itération suivante d'une boucle.

Exemple

C++

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
```

```
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

Output

```
before the continue
before the continue
before the continue
after the do loop
```

Voir aussi

Instructions jump

Mots clés

return, instruction (C++)

Article • 16/06/2023

Termine l'exécution d'une fonction et retourne le contrôle à la fonction d'appel (ou au système d'exploitation si vous transférez le contrôle à partir de la fonction `main`). L'exécution reprend dans la fonction d'appel au point immédiatement après l'appel.

Syntaxe

```
return [expression];
```

Notes

La clause `expression`, si elle est présente, est convertie dans le type spécifié dans la déclaration de fonction, comme si une initialisation était exécutée. La conversion du type de l'expression en **`return`** type de la fonction peut créer des objets temporaires. Pour plus d'informations sur la façon et le moment où les temporaires sont créés, consultez Objets temporaires.

La valeur de la clause `expression` est retournée à la fonction d'appel. Si l'expression est omise, la valeur de retour de la fonction n'est pas définie. Les constructeurs et les destructeurs, ainsi que les fonctions de type **`void`**, ne peuvent pas spécifier une expression dans l'instruction **`return`** . Les fonctions de tous les autres types doivent spécifier une expression dans l'instruction **`return`** .

Lorsque le flux de contrôle quitte le bloc englobant la définition de fonction, le résultat est le même qu'il serait si une **`return`** instruction sans expression avait été exécutée. Ceci n'est pas valable pour les fonctions déclarées comme retournant une valeur.

Une fonction peut avoir n'importe quel nombre d'instructions **`return`** .

L'exemple suivant utilise une expression avec une **`return`** instruction pour obtenir le plus grand des deux entiers.

Exemple

C++

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
   return ( a > b ? a : b );
}

int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ));
}
```

Voir aussi

Instructions jump

Mots clés

goto, instruction (C++)

Article • 03/04/2023

L'instruction `goto` transfère de manière inconditionnelle le contrôle vers l'instruction étiquetée par l'identificateur spécifié.

Syntaxe

```
goto identifier;
```

Notes

L'instruction étiquetée indiquée par `identifier` doit se trouver dans la fonction actuelle. Tous les noms `identifier` sont membres d'un espace de noms interne et, par conséquent, n'interfèrent pas avec d'autres identificateurs.

Une étiquette d'instruction n'a de sens que pour une `goto` instruction ; sinon, les étiquettes d'instruction sont ignorées. Les étiquettes ne peuvent pas être redéclarées.

Une `goto` instruction n'est pas autorisée à transférer le contrôle vers un emplacement qui ignore l'initialisation d'une variable qui est dans l'étendue dans cet emplacement. L'exemple suivant déclenche L'erreur C2362 :

C++

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit;  // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

Dans la mesure du possible, il est judicieux d'utiliser les `break` instructions , `continue` et `return` au lieu de l'instruction `goto` . Toutefois, étant donné que l'instruction `break` ne quitte qu'un seul niveau d'une boucle, vous devrez peut-être utiliser une `goto` instruction pour quitter une boucle profondément imbriquée.

Pour plus d'informations sur les étiquettes et l'instruction `goto` , consultez Instructions étiquetées.

Exemple

Dans cet exemple, une `goto` instruction transfère le contrôle au point étiqueté `stop` quand `i` est égal à 3.

C++

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

    stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

Output

```
Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
```

```
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3
```

Voir aussi

Instructions jump

Mots clés

Transferts de contrôle

Article • 03/04/2023

Vous pouvez utiliser l'instruction `goto` ou une `case` étiquette dans une `switch` instruction pour spécifier un programme qui se branche au-delà d'un initialiseur. Ce code est conforme sauf si la déclaration qui contient l'initialiseur figure dans un bloc se trouvant lui-même dans le bloc dans lequel l'instruction de saut s'exécute.

L'exemple suivant montre une boucle qui déclare et initialise les objets `total`, `ch` et `i`. Il existe également une instruction erronée `goto` qui transfère le contrôle au-delà d'un initialiseur.

C++

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
   char MyArray[5] = {'2','2','a','c'};
   int i = 0;
   while( 1 )
   {
      int total = 0;

      char ch = MyArray[i++];

      if ( ch >= '0' && ch <= '9' )
      {
         goto Label1;

         int i = ch - '0';
      Label1:
         total += i;   // C4700: transfers past initialization of i.
      } // i would be destroyed here if  goto error were not present
   else
      // Break statement transfers control out of loop,
      //  destroying total and ch.
      break;
   }
}
```

Dans l'exemple précédent, l'instruction tente de transférer le `goto` contrôle après l'initialisation de `i`. Toutefois, si `i` était déclaré mais non initialisé, le transfert serait conforme.

Les objets `total` et `ch`, déclarés dans le bloc qui sert *d'instruction* de l'instruction **`while`** , sont détruits lorsque ce bloc est supprimé à l'aide de l'instruction **`break`** .

Espaces de noms (C++)

Article • 16/06/2023

Un espace de noms est une région déclarative qui fournit une portée aux identificateurs (noms de types, fonctions, variables, etc.) à l'intérieur. Les espaces de noms sont utilisés pour organiser le code en groupes logiques et pour éviter les conflits de noms qui peuvent se produire en particulier lorsque votre base de code inclut plusieurs bibliothèques. Tous les identificateurs de portée espace de noms sont visibles les uns pour les autres sans qualification. Les identificateurs en dehors de l'espace de noms peuvent accéder aux membres en utilisant le nom complet de chaque identificateur, par exemple `std::vector<std::string> vec;`, ou bien par une déclaration using pour un identificateur unique (`using std::string`), ou une directive using pour tous les identificateurs de l'espace de noms (`using namespace std;`). Le code dans les fichiers d'en-tête doit toujours utiliser le nom de l'espace de noms qualifié complet.

L'exemple suivant montre une déclaration d'espace de noms et trois façons pour le code en dehors de l'espace de noms d'accéder à leurs membres.

C++

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Utiliser le nom qualifié complet :

C++

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Utiliser une déclaration using pour placer un identificateur dans la portée :

C++

```
using ContosoData::ObjectManager;
ObjectManager mgr;
```

```
mgr.DoSomething();
```

Utiliser une directive using pour placer tous les éléments de l'espace de noms dans la portée :

```
C++

using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

Directives using

La `using` directive permet d'utiliser tous les noms d'un `namespace` sans le *nom-espace de noms* comme qualificateur explicite. Utilisez une directive using dans un fichier d'implémentation (c'est-à-dire *.cpp) si vous utilisez plusieurs identificateurs différents dans un espace de noms ; Si vous n'utilisez qu'un ou deux identificateurs, envisagez une déclaration using pour placer uniquement ces identificateurs dans l'étendue et non tous les identificateurs de l'espace de noms. Si une variable locale a le même nom qu'une variable d'espace de noms, la variable d'espace de noms est masquée. Le fait qu'une variable d'espace de noms porte le même nom qu'une variable globale est une erreur.

> ⓘ **Notes**
>
> Une directive using peut être placée en haut d'un fichier .cpp (au niveau de la portée de fichier) ou à l'intérieur d'une définition de classe ou de fonction.
>
> En règle générale, évitez de placer des directives using dans les fichiers d'en-tête (*.h), car n'importe quel fichier qui inclut cet en-tête placera tous les éléments de l'espace de noms dans la portée, ce qui peut provoquer des problèmes de masquage de noms et de conflits de noms très difficiles à déboguer. Utilisez toujours des noms qualifiés complets dans un fichier d'en-tête. Si ces noms sont trop longs, vous pouvez utiliser un alias d'espace de noms pour les raccourcir. (Voir ci-dessous.)

Déclaration d'espaces de noms et de membres d'espaces de noms

En général, vous déclarez un espace de noms dans un fichier d'en-tête. Si les implémentations de vos fonctions se trouvent dans un fichier distinct, qualifiez les noms des fonctions, comme dans cet exemple.

C++

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Les implémentations de fonction dans contosodata.cpp doivent utiliser le nom complet, même si vous placez une `using` directive en haut du fichier :

C++

```
#include "contosodata.h"
using namespace ContosoDataServer;

void ContosoDataServer::Foo() // use fully-qualified name here
{
   // no qualification needed for Bar()
   Bar();
}

int ContosoDataServer::Bar(){return 0;}
```

Un espace de noms peut être déclaré dans plusieurs blocs, dans un seul fichier et dans plusieurs fichiers. Le compilateur joint les parties pendant le prétraitement et l'espace de noms obtenu contient tous les membres déclarés dans toutes les parties. Un exemple de ceci est l'espace de noms std qui est déclaré dans chacun des fichiers d'en-tête de la bibliothèque standard.

Les membres d'un espace de noms nommé peuvent être définis en dehors de l'espace de noms dans lequel ils sont déclarés par une qualification explicite du nom actuellement défini. Toutefois, la définition doit figurer après le point de déclaration dans un espace de noms qui englobe l'espace de noms de la déclaration. Par exemple :

C++

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {
    void f();
```

```
}

void V::f() { }        // ok
void V::g() { }        // C2039, g() is not yet a member of V

namespace V {
    void g();
}
```

Cette erreur peut se produire quand les membres de l'espace de noms sont déclarés dans plusieurs fichiers d'en-tête et que vous n'avez pas inclus ces en-têtes dans le bon ordre.

Espace de noms global

Si un identificateur n'est pas déclaré dans un espace de noms explicite, il fait partie de l'espace de noms global implicite. En général, essayez d'éviter d'effectuer des déclarations à l'étendue globale lorsque cela est possible, à l'exception du point d'entrée main Function, qui doit se trouver dans l'espace de noms global. Pour qualifier explicitement un identificateur global, utilisez l'opérateur de résolution de portée sans nom, comme dans `::SomeFunction(x);`. Vous différenciez ainsi l'identificateur de tout élément portant le même nom dans un autre espace de noms et votre code devient également plus facile à comprendre.

Espace de noms std

Tous les types et fonctions de bibliothèque standard C++ sont déclarés dans l'espace `std` de noms ou les espaces de noms imbriqués dans `std`.

Espaces de noms imbriqués

Les espaces de noms peuvent être imbriqués. Un espace de noms imbriqué ordinaire a un accès non qualifié aux membres de son parent, mais les membres parents n'ont pas d'accès non qualifié à l'espace de noms imbriqué (sauf s'il est déclaré comme inline), comme illustré dans l'exemple suivant :

C++

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
```

```
    {
        int CountImpl;
        void Ban() { return Foo(); }
    }

    int Bar(){...};
    int Baz(int i) { return Details::CountImpl; }
}
```

Les espaces de noms imbriqués ordinaires permettent d'encapsuler les détails d'implémentation interne qui ne font pas partie de l'interface publique de l'espace de noms parent.

Espaces de noms inline (C++11)

Contrairement à un espace de noms imbriqué ordinaire, les membres d'un espace de noms inline sont traités en tant que membres de l'espace de noms parent. Cette caractéristique permet à une recherche dépendante d'un argument sur les fonctions surchargées de fonctionner sur les fonctions qui ont des surcharges dans un parent et un espace de noms inline imbriqué. Elle vous permet également de déclarer une spécialisation dans un espace de noms parent pour un modèle qui est déclaré dans l'espace de noms inline. L'exemple suivant montre comment le code externe est lié à l'espace de noms inline par défaut :

C++

```
//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
```

```
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}
```

L'exemple suivant vous montre comment déclarer une spécialisation dans un parent d'un modèle qui est déclaré dans un espace de noms inline :

C++

```
namespace Parent
{
    inline namespace new_ns
    {
         template <typename T>
         struct C
         {
             T member;
         };
    }
     template<>
     class C<int> {};
}
```

Vous pouvez utiliser des espaces de noms inline comme mécanisme de contrôle de version pour gérer les modifications apportées à l'interface publique d'une bibliothèque. Par exemple, vous pouvez créer un espace de noms parent unique et encapsuler chaque version de l'interface dans son propre espace de noms imbriqué dans le parent. L'espace de noms qui contient la version la plus récente ou préférée est qualifié comme étant inline et est par conséquent exposé comme s'il s'agissait d'un membre direct de l'espace de noms parent. Le code client qui appelle Parent::Class se liera automatiquement au nouveau code. Les clients qui préfèrent utiliser l'ancienne version peuvent toujours y accéder en utilisant le chemin d'accès qualifié complet à l'espace de noms imbriqué qui possède ce code.

Le mot clé inline doit être appliqué à la première déclaration de l'espace de noms dans une unité de compilation.

L'exemple suivant montre deux versions d'une interface, chacune dans un espace de noms imbriqué. L'espace de noms `v_20` a été modifié à partir de l'interface `v_10` et est marqué comme inline. Le code client qui utilise la nouvelle bibliothèque et appelle `Contoso::Funcs::Add` appelle la version v_20. Le code qui tente d'appeler `Contoso::Funcs::Divide` reçoit désormais une erreur de compilation. S'ils ont vraiment

besoin de cette fonction, ils peuvent toujours accéder à la version `v_10` en appelant explicitement `Contoso::v_10::Funcs::Divide`.

C++

```
namespace Contoso
{
    namespace v_10
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            T Divide(T a, T b);
        };
    }

    inline namespace v_20
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            std::vector<double> Log(double);
            T Accumulate(std::vector<T> nums);
      };
    }
}
```

Alias d'espaces de noms

Les noms des espaces de noms doivent être uniques, ce qui signifie qu'ils ne doivent pas être trop courts. Si la longueur d'un nom rend le code difficile à lire ou s'il est fastidieux de taper dans un fichier d'en-tête où les directives d'utilisation ne peuvent pas être utilisées, vous pouvez créer un alias d'espace de noms qui sert d'abréviation pour le nom réel. Par exemple :

C++

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
```

```
void Bar(AVLNN::Foo foo){ }
```

espaces de noms anonymes ou sans nom

Vous pouvez créer un espace de noms explicite, mais sans lui donner de nom :

C++

```
namespace
{
    int MyFunc(){}
}
```

Il s'agit d'un espace de noms anonyme ou sans nom, et il est utile lorsque vous souhaitez rendre les déclarations de variables invisibles pour le code dans d'autres fichiers (c'est-à-dire leur donner une liaison interne) sans avoir à créer un espace de noms nommé. Tout le code du même fichier peut voir les identificateurs dans un espace de noms sans nom, mais les identificateurs, ainsi que l'espace de noms lui-même, ne sont pas visibles en dehors de ce fichier ou, plus précisément, à l'extérieur de l'unité de traduction.

Voir aussi

Déclarations et définitions

Énumérations (C++)

Article • 03/04/2023

Une énumération est un type défini par l'utilisateur qui se compose d'un ensemble de constantes intégrales nommées *appelées énumérateurs*.

> **Notes**
>
> Cet article couvre le type de langage `enum` C++ ISO Standard et le type délimité (ou fortement typé) `enum class` qui est introduit dans C++11. Pour plus d'informations sur les types ou `private enum class` dans C++/CLI et C++/CX, consultez **enum class (C++/CLI et C++/CX).** `public enum class`

Syntaxe

enum-name:

identifier

enum-specifier:

enum-head **{** *enumerator-list*$_{opt}$ **}**

enum-head **{** *enumerator-list* **,** **}**

enum-head:

enum-key *attribute-specifier-seq*$_{opt}$ *enum-head-name*$_{opt}$ *enum-base*$_{opt}$

enum-head-name:

nested-name-specifier$_{opt}$ *identifier*

opaque-enum-declaration:

enum-key *attribute-specifier-seq*$_{Opter}$ *enum-head-name* *enum-base*$_{Opter}$ **;**

enum-key:

enum

enum class

enum struct

enum-base:

: *type-specifier-seq*

enumerator-list:

enumerator-definition

enumerator-list , *enumerator-definition*

enumerator-definition:

enumerator

enumerator = *constant-expression*

enumerator:

identifier *attribute-specifier-seq*$_{\text{opt}}$

Usage

C++

```
// unscoped enum:
// enum [identifier] [: type] {enum-list};

// scoped enum:
// enum [class|struct] [identifier] [: type] {enum-list};

// Forward declaration of enumerations  (C++11):
enum A : int;          // non-scoped enum must have type specified
enum class B;          // scoped enum defaults to int but ...
enum class C : short;  // ... may have any integral underlying type
```

Paramètres

identifier

Nom de type donné à l'énumération.

type

Type sous-jacent des énumérateurs ; tous les énumérateurs ont le même type sous-jacent. Peut être tout type intégral.

enum-list

Liste délimitée par des virgules des énumérateurs dans l'énumération. Chaque nom d'énumérateur ou de variable dans la portée doit être unique. Toutefois, les valeurs peuvent être dupliquées. Dans une énumération non étendue, l'étendue est l'étendue environnante ; dans une énumération délimitée, l'étendue est l'étendue *enum-list* elle-même. Dans une énumération délimitée, la liste peut être vide, ce qui définit en effet un nouveau type intégral.

class

En utilisant cette mot clé dans la déclaration, vous spécifiez que l'énumération est délimitée et *identifier* doit être fournie. Vous pouvez également utiliser les **struct** mot clé à la place de **class**, car elles sont sémantiquement équivalentes dans ce contexte.

Étendue de l'énumérateur

Une énumération fournit un contexte pour décrire une plage de valeurs représentées sous forme de constantes nommées. Ces constantes nommées sont également *appelées énumérateurs*. Dans les types C et C++ `enum` d'origine, les énumérateurs non qualifiés sont visibles dans toute l'étendue dans laquelle est `enum` déclaré. Dans les énumérations délimitées, le nom de l'énumérateur doit être qualifié par le nom de `enum` type.
L'exemple suivant illustre cette différence fondamentale entre les deux genres d'enums :

C++

```
namespace CardGame_Scoped
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Suit::Clubs) // Enumerator must be qualified by enum
type
        { /*...*/}
    }
}

namespace CardGame_NonScoped
{
    enum Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Clubs) // Enumerator is visible without qualification
        { /*...*/
        }
    }
}
```

À chaque nom d'une énumération est assignée une valeur entière qui correspond à sa place dans l'ordre des valeurs de l'énumération. Par défaut, la première valeur est 0, la suivante 1, et ainsi de suite, mais vous pouvez définir explicitement la valeur d'un énumérateur, comme indiqué ci-dessous :

C++

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

L'énumérateur `Diamonds` reçoit la valeur `1`. Les énumérateurs suivants, s'ils ne reçoivent pas de valeur explicite, reçoivent la valeur de l'énumérateur précédent plus un. Dans l'exemple précédent, `Hearts` aurait la valeur 2, `Clubs` aurait la valeur 3, et ainsi de suite.

Chaque énumérateur est traité comme une constante et doit avoir un nom unique dans l'étendue où est `enum` défini (pour les énumérations non étendues) ou dans le `enum` lui-même (pour les énumérations délimitées). Les valeurs données aux noms ne doivent pas être uniques. Par exemple, considérez cette déclaration d'une énumération `Suit` non étendue :

C++

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

Les valeurs de `Diamonds`, `Hearts`, `Clubs` et `Spades` sont respectivement 5, 6, 4 et 5. Notez que 5 est utilisé plusieurs fois ; elle est autorisée même si elle n'est peut-être pas prévue. Ces règles sont les mêmes pour les enums délimités.

Règles de transtypage

Les constantes d'énumération non étendues peuvent être converties `int` implicitement en , mais un `int` n'est jamais implicitement convertible en valeur d'énumération. L'exemple suivant montre ce qui se passe si vous essayez d'attribuer `hand` une valeur qui n'est pas un `Suit` :

C++

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to
'Suit'
```

Un cast est nécessaire pour convertir un en `int` énumérateur délimité ou non. Toutefois, vous pouvez promouvoir un énumérateur sans étendue en valeur entière sans cast.

C++

```
int account_num = Hearts; //OK if Hearts is in a unscoped enum
```

L'utilisation des conversions implicites de cette façon peut provoquer des effets secondaires inattendus. Pour aider à éliminer les erreurs de programmation associées aux enums non délimités, les valeurs d'enums délimités sont fortement typées. Les énumérateurs délimités doivent être qualifiés par le nom du type d'énumération (identificateur) et ne peuvent pas être convertis implicitement, comme illustré dans l'exemple suivant :

C++

```
namespace ScopedEnumConversions
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void AttemptConversions()
    {
        Suit hand;
        hand = Clubs; // error C2065: 'Clubs' : undeclared identifier
        hand = Suit::Clubs; //Correct.
        int account_num = 135692;
        hand = account_num; // error C2440: '=' : cannot convert from 'int'
to 'Suit'
        hand = static_cast<Suit>(account_num); // OK, but probably a bug!!!

        account_num = Suit::Hearts; // error C2440: '=' : cannot convert
from 'Suit' to 'int'
        account_num = static_cast<int>(Suit::Hearts); // OK
    }
}
```

Notez que la ligne `hand = account_num;` provoque toujours l'erreur avec les enums non délimités, comme indiqué précédemment. Elle est autorisée avec un cast explicite. Toutefois, avec les enums délimités, la tentative de conversion dans l'instruction suivante, `account_num = Suit::Hearts;`, n'est plus autorisée sans transtypage explicite.

Énumérations sans énumérateur

Visual Studio 2017 version 15.3 et ultérieures (disponible avec /std:c++17 et versions ultérieures) : en définissant une énumération (régulière ou étendue) avec un type sous-jacent explicite et aucun énumérateur, vous pouvez en effet introduire un nouveau type intégral qui n'a pas de conversion implicite vers un autre type. En utilisant ce type au lieu de son type sous-jacent intégré, vous pouvez éliminer le risque d'erreurs subtiles causées par des conversions implicites par inadvertance.

C++

```
enum class byte : unsigned char { };
```

Le nouveau type est une copie exacte du type sous-jacent, et a donc la même convention d'appel, ce qui signifie qu'il peut être utilisé entre les abis sans aucune pénalité de performances. Aucun cast n'est requis lorsque les variables du type sont initialisées à l'aide de l'initialisation de liste directe. L'exemple suivant montre comment initialiser des énumérations sans énumérateur dans différents contextes :

C++

```
enum class byte : unsigned char { };

enum class E : int { };
E e1{ 0 };
E e2 = E{ 0 };

struct X
{
    E e{ 0 };
    X() : e{ 0 } { }
};

E* p = new E{ 0 };

void f(E e) {};

int main()
{
    f(E{ 0 });
    byte i{ 42 };
    byte j = byte{ 42 };

    // unsigned char c = j; // C2440: 'initializing': cannot convert from
'byte' to 'unsigned char'
    return 0;
}
```

Voir aussi

Déclarations d'énumération C

Mots clés

`union`

Article • 17/06/2023

> ⓘ **Notes**
>
> Dans C++17 et versions ultérieures, le `std::variant` class est une alternative de type sécurisé pour un union.

Un **`union`** est un type défini par l'utilisateur dans lequel tous les membres partagent le même emplacement de mémoire. Cette définition signifie qu'à un moment donné, un union ne peut pas contenir plus d'un objet de sa liste de membres. Cela signifie également que, quel que soit le nombre de membres qu'un union a, il utilise toujours suffisamment de mémoire pour stocker le membre le plus grand.

Un union peut être utile pour conserver la mémoire lorsque vous avez un grand nombre d'objets et une mémoire limitée. Toutefois, un union nécessite un soin supplémentaire pour l'utiliser correctement. Il vous incombe de vous assurer que vous accédez toujours au même membre que celui que vous avez affecté. Si des types de membres ont un constructnontrivial ou, vous devez écrire du code pour con explicitementstruct et détruire ce membre. Avant d'utiliser un union, déterminez si le problème que vous essayez de résoudre peut être mieux exprimé à l'aide d'une base class et de types dérivés class .

Syntaxe

> **`union`** *`tag`*_{opt} **`{`** *`member-list`* **`};`**

Paramètres

`tag`
Nom de type donné au union.

`member-list`
Membres que le union peut contenir.

Déclarer un union

Commencez la déclaration d'un union en utilisant le `union` mot clé et placez la liste des membres dans des accolades :

C++

```
// declaring_a_union.cpp
union RecordType    // Declare a simple union type
{
    char   ch;
    int    i;
    long   l;
    float  f;
    double d;
    int *int_ptr;
};

int main()
{
    RecordType t;
    t.i = 5; // t holds an int
    t.f = 7.25; // t now holds a float
}
```

Utiliser un union

Dans l'exemple précédent, tout code qui accède union au doit savoir quel membre contient les données. La solution la plus courante à ce problème est *appelée discriminationunion*. Il inclut le dans un struct, et inclut un enum membre qui indique le type de membre actuellement stocké dans le union.union L'exemple suivant illustre le modèle de base :

C++

```
#include <queue>

using namespace std;

enum class WeatherDataType
{
    Temperature, Wind
};

struct TempData
{
    int StationId;
    time_t time;
    double current;
    double max;
    double min;
```

```
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
    short direction;
};

struct Input
{
    WeatherDataType type;
    union
    {
        TempData temp;
        WindData wind;
    };
};

// Functions that are specific to data types
void Process_Temp(TempData t) {}
void Process_Wind(WindData w) {}

void Initialize(std::queue<Input>& inputs)
{
    Input first;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    Input second;
    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

int main(int argc, char* argv[])
{
    // Container for all the data records
    queue<Input> inputs;
    Initialize(inputs);
    while (!inputs.empty())
    {
        Input const i = inputs.front();
        switch (i.type)
        {
        case WeatherDataType::Temperature:
            Process_Temp(i.temp);
            break;
        case WeatherDataType::Wind:
            Process_Wind(i.wind);
            break;
        default:
            break;
```

```
        }
        inputs.pop();

    }
    return 0;
}
```

Dans l'exemple précédent, le union dans n'a `Input` struct pas de nom. Il est donc appelé *anonyme* union. Ses membres sont accessibles directement comme s'ils sont membres du struct. Pour plus d'informations sur l'utilisation d'un anonyme union, consultez la section Anonyme union .

L'exemple précédent montre un problème que vous pouvez également résoudre à l'aide class de types qui dérivent d'une base classcommune . Vous pouvez brancher votre code en fonction du type d'exécution de chaque objet dans le conteneur. Votre code peut être plus facile à gérer et à comprendre, mais il peut également être plus lent que d'utiliser un union. En outre, avec un union, vous pouvez stocker des types non liés. Un union vous permet de modifier dynamiquement le type de la valeur stockée sans modifier le type de la union variable elle-même. Par exemple, vous pouvez créer un tableau hétérogène de `MyUnionType`, dont les éléments stockent différentes valeurs de types différents.

Il est facile d'utiliser à mauvais escient le `Input` struct dans l'exemple. Il appartient à l'utilisateur d'utiliser correctement le discriminateur pour accéder au membre qui contient les données. Vous pouvez vous protéger contre toute utilisation incorrecte en mettant en place et union `private` en fournissant des fonctions d'accès spéciales, comme illustré dans l'exemple suivant.

Non restreint union (C++11)

Dans C++03 et versions antérieures, un peut contenir des union membres nonstatic de données qui ont un class type, tant que le type n'a aucun opérateur de constructors, destructors ou d'affectation fourni par l'utilisateur. En C++11, ces restrictions sont supprimées. Si vous incluez un tel membre dans votre union, le compilateur marque automatiquement toutes les fonctions membres spéciales qui ne sont pas fournies par l'utilisateur comme `deleted`. Si est union un anonyme union à l'intérieur d'un class ou struct, toutes les fonctions membres spéciales du class ou struct qui ne sont pas fournies par l'utilisateur sont marquées comme `deleted`. L'exemple suivant montre comment gérer ce cas. L'un des membres du union a un membre qui nécessite ce traitement spécial :

```
C++
```

```
// for MyVariant
#include <crtdbg.h>
#include <new>
#include <utility>

// for sample objects and output
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct A
{
    A() = default;
    A(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    //...
};

struct B
{
    B() = default;
    B(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    vector<int> vec;
    // ...
};

enum class Kind { None, A, B, Integer };

#pragma warning (push)
#pragma warning(disable:4624)
class MyVariant
{
public:
    MyVariant()
        : kind_(Kind::None)
    {
    }

    MyVariant(Kind kind)
        : kind_(kind)
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A();
```

```
            break;
        case Kind::B:
            new (&b_) B();
            break;
        case Kind::Integer:
            i_ = 0;
            break;
        default:
            _ASSERT(false);
            break;
        }
    }

    ~MyVariant()
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            a_.~A();
            break;
        case Kind::B:
            b_.~B();
            break;
        case Kind::Integer:
            break;
        default:
            _ASSERT(false);
            break;
        }
        kind_ = Kind::None;
    }

    MyVariant(const MyVariant& other)
        : kind_(other.kind_)
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(other.a_);
            break;
        case Kind::B:
            new (&b_) B(other.b_);
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
        }
    }
```

```
    MyVariant(MyVariant&& other)
        : kind_(other.kind_)
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A(move(other.a_));
            break;
        case Kind::B:
            new (&b_) B(move(other.b_));
            break;
        case Kind::Integer:
            i_ = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
        }
        other.kind_ = Kind::None;
    }

    MyVariant& operator=(const MyVariant& other)
    {
        if (&other != this)
        {
            switch (other.kind_)
            {
            case Kind::None:
                this->~MyVariant();
                break;
            case Kind::A:
                *this = other.a_;
                break;
            case Kind::B:
                *this = other.b_;
                break;
            case Kind::Integer:
                *this = other.i_;
                break;
            default:
                _ASSERT(false);
                break;
            }
        }
        return *this;
    }

    MyVariant& operator=(MyVariant&& other)
    {
        _ASSERT(this != &other);
        switch (other.kind_)
        {
```

```
        case Kind::None:
            this->~MyVariant();
            break;
        case Kind::A:
            *this = move(other.a_);
            break;
        case Kind::B:
            *this = move(other.b_);
            break;
        case Kind::Integer:
            *this = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
        }
        other.kind_ = Kind::None;
        return *this;
    }

    MyVariant(const A& a)
        : kind_(Kind::A), a_(a)
    {
    }

    MyVariant(A&& a)
        : kind_(Kind::A), a_(move(a))
    {
    }

    MyVariant& operator=(const A& a)
    {
        if (kind_ != Kind::A)
        {
            this->~MyVariant();
            new (this) MyVariant(a);
        }
        else
        {
            a_ = a;
        }
        return *this;
    }

    MyVariant& operator=(A&& a)
    {
        if (kind_ != Kind::A)
        {
            this->~MyVariant();
            new (this) MyVariant(move(a));
        }
        else
        {
            a_ = move(a);
        }
```

```
        return *this;
    }

    MyVariant(const B& b)
        : kind_(Kind::B), b_(b)
    {
    }

    MyVariant(B&& b)
        : kind_(Kind::B), b_(move(b))
    {
    }

    MyVariant& operator=(const B& b)
    {
        if (kind_ != Kind::B)
        {
            this->~MyVariant();
            new (this) MyVariant(b);
        }
        else
        {
            b_ = b;
        }
        return *this;
    }

    MyVariant& operator=(B&& b)
    {
        if (kind_ != Kind::B)
        {
            this->~MyVariant();
            new (this) MyVariant(move(b));
        }
        else
        {
            b_ = move(b);
        }
        return *this;
    }

    MyVariant(int i)
        : kind_(Kind::Integer), i_(i)
    {
    }

    MyVariant& operator=(int i)
    {
        if (kind_ != Kind::Integer)
        {
            this->~MyVariant();
            new (this) MyVariant(i);
        }
        else
        {
```

```
            i_ = i;
        }
        return *this;
    }

    Kind GetKind() const
    {
        return kind_;
    }

    A& GetA()
    {
        _ASSERT(kind_ == Kind::A);
        return a_;
    }

    const A& GetA() const
    {
        _ASSERT(kind_ == Kind::A);
        return a_;
    }

    B& GetB()
    {
        _ASSERT(kind_ == Kind::B);
        return b_;
    }

    const B& GetB() const
    {
        _ASSERT(kind_ == Kind::B);
        return b_;
    }

    int& GetInteger()
    {
        _ASSERT(kind_ == Kind::Integer);
        return i_;
    }

    const int& GetInteger() const
    {
        _ASSERT(kind_ == Kind::Integer);
        return i_;
    }

private:
    Kind kind_;
    union
    {
        A a_;
        B b_;
        int i_;
    };
};
```

```
#pragma warning (pop)

int main()
{
    A a(1, "Hello from A");
    B b(2, "Hello from B");

    MyVariant mv_1 = a;

    cout << "mv_1 = a: " << mv_1.GetA().name << endl;
    mv_1 = b;
    cout << "mv_1 = b: " << mv_1.GetB().name << endl;
    mv_1 = A(3, "hello again from A");
    cout << R"aaa(mv_1 = A(3, "hello again from A"): )aaa" <<
mv_1.GetA().name << endl;
    mv_1 = 42;
    cout << "mv_1 = 42: " << mv_1.GetInteger() << endl;

    b.vec = { 10,20,30,40,50 };

    mv_1 = move(b);
    cout << "After move, mv_1 = b: vec.size = " << mv_1.GetB().vec.size() <<
endl;

    cout << endl << "Press a letter" << endl;
    char c;
    cin >> c;
}
```

Un union ne peut pas stocker une référence. Un union ne prend pas non plus en charge l'héritage. Cela signifie que vous ne pouvez pas utiliser un union comme base class, ou hériter d'un autre class, ou avoir des fonctions virtuelles.

Initialise un union

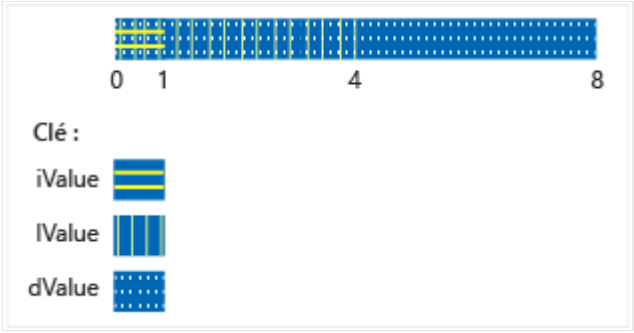
Vous pouvez déclarer et initialiser un union dans la même instruction en affectant une expression entourée d'accolades. L'expression est évaluée et affectée au premier champ du union.

C++

```
#include <iostream>
using namespace std;

union NumericType
{
    short       iValue;
    long        lValue;
    double      dValue;
};
```

```
int main()
{
    union NumericType Values = { 10 };   // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}
/* Output:
10
3.141600
*/
```

Le `NumericType` union est organisé en mémoire (conceptuellement) comme illustré dans la figure suivante :



Anonyme union

Un anonyme union est un déclaré sans *`class-name`* ou *`declarator-list`*.

> **`union {`** *`member-list`* **`}`**

Les noms déclarés dans un anonyme union sont utilisés directement, comme les variables non membres. Cela implique que les noms déclarés dans un anonyme union doivent être uniques dans l'étendue environnante.

Un anonyme union est soumis à ces restrictions :

- S'il est déclaré dans l'étendue du fichier ou de l'espace de noms, il doit également être déclaré en tant que **`static`**.
- Il ne peut avoir que **`public`** des membres ; avoir **`private`** des membres et **`protected`** dans un anonyme union génère des erreurs.
- Il ne peut pas avoir de fonctions membres.

Voir aussi

Classes et structs

Mots clés

class

struct

Fonctions (C++)

Article • 03/04/2023

Une fonction est un bloc de code qui effectue une opération. Une fonction peut éventuellement définir les paramètres d'entrée qui permettent aux appelants de passer des arguments dans la fonction. Une fonction peut éventuellement retourner une valeur en tant que sortie. Les fonctions sont utiles pour encapsuler les opérations courantes dans un seul bloc réutilisable, idéalement avec un nom qui décrit clairement ce que fait la fonction. La fonction suivante accepte deux entiers d'un appelant et retourne leur somme ; *a* et *b* sont des *paramètres* de type `int`.

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

La fonction peut être appelée, ou *appelée*, à partir d'un nombre quelconque d'emplacements dans le programme. Les valeurs passées à la fonction sont *les arguments*, dont les types doivent être compatibles avec les types de paramètres dans la définition de la fonction.

C++

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

Il n'existe aucune limite pratique à la longueur des fonctions, mais une bonne conception vise les fonctions qui exécutent une seule tâche bien définie. Les algorithmes complexes doivent être divisés en fonctions plus simples à comprendre chaque fois que possible.

Les fonctions qui sont définies au niveau de la portée de classe sont appelées fonctions membres. En C++, contrairement à d'autres langages, une fonction peut également être définie au niveau de la portée d'espace de noms (y compris l'espace de noms global implicite). Ces fonctions sont appelées *fonctions libres* ou *fonctions non membres* ; ils sont largement utilisés dans la bibliothèque Standard.

Les fonctions peuvent être *surchargées*, ce qui signifie que différentes versions d'une fonction peuvent partager le même nom si elles diffèrent par le nombre et/ou le type de paramètres formels. Pour plus d'informations, consultez Surcharge des fonctions.

Éléments d'une déclaration de fonction

Une *déclaration* de fonction minimale se compose du type de retour, du nom de la fonction et de la liste de paramètres (qui peuvent être vides), ainsi que de mots clés facultatifs qui fournissent des instructions supplémentaires au compilateur. L'exemple suivant est une déclaration de fonction :

C++

```
int sum(int a, int b);
```

Une définition de fonction se compose d'une déclaration, plus le *corps*, qui est tout le code entre les accolades :

C++

```
int sum(int a, int b)
{
    return a + b;
}
```

Une déclaration de fonction suivie d'un point-virgule peut apparaître à plusieurs endroits dans un programme. Elle doit apparaître avant tout appel à cette fonction dans chaque unité de traduction. La définition de fonction doit apparaître une seule fois dans le programme, en fonction de la règle de définitions.

Les éléments requis d'une déclaration de fonction sont les suivants :

1. Type de retour, qui spécifie le type de la valeur retournée par la fonction, ou **`void`** si aucune valeur n'est retournée. En C++11, **`auto`** est un type de retour valide qui indique au compilateur de déduire le type de l'instruction return. En C++14, `decltype(auto)` est également autorisé. Pour plus d'informations, consultez Déduction de type dans les types de retours ci-dessous.

2. Nom de la fonction, qui doit commencer par une lettre ou un trait de soulignement et ne peut pas contenir d'espaces. En général, les traits de soulignement de début dans les noms de fonctions de la bibliothèque standard indiquent des fonctions membres privées ou des fonctions non membres qui ne sont pas destinées à être utilisées par votre code.

3. La liste de paramètres, un ensemble de zéro ou plusieurs paramètres séparés par des virgules et délimités par des accolades qui spécifient le type et éventuellement un nom local selon lequel les valeurs sont accessibles à l'intérieur du corps de la fonction.

Les éléments facultatifs d'une déclaration de fonction sont les suivants :

1. **`constexpr`**, qui indique que la valeur de retour de la fonction est une valeur de constante pouvant être calculée au moment de la compilation.

```
C++

constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

2. Sa spécification de liaison, **`extern`** ou **`static`**.

```
C++

//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

   Pour plus d'informations, consultez Unités de traduction et liaison.

3. **`inline`**, qui indique au compilateur de remplacer chaque appel à la fonction par le code de la fonction. L'incorporation peut améliorer les performances dans les scénarios où une fonction s'exécute rapidement et est appelée à plusieurs reprises dans une section du code critique pour les performances.

```
C++

inline double Account::GetBalance()
{
    return balance;
}
```

   Pour plus d'informations, consultez Fonctions inline.

4. Expression **`noexcept`** qui spécifie si la fonction peut lever une exception ou non. Dans l'exemple suivant, la fonction ne lève pas d'exception si l'expression prend

`true` la `is_pod` valeur .

```
C++

#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

Pour plus d'informations, consultez noexcept.

5. (Fonctions membres uniquement) Qualificateurs cv, qui spécifient si la fonction est `const` ou `volatile`.

6. (Fonctions membres uniquement) `virtual`, `override` ou `final`. `virtual` spécifie qu'une fonction peut être substituée dans une classe dérivée. `override` signifie qu'une fonction dans une classe dérivée remplace une fonction virtuelle. `final` signifie qu'une fonction ne peut pas être remplacée dans une classe dérivée supplémentaire. Pour plus d'informations, consultez Fonctions virtuelles.

7. (fonctions membres uniquement) `static` appliqué à une fonction membre signifie que la fonction n'est associée à aucune instance d'objet de la classe .

8. (Fonctions membres non statiques uniquement) Ref-qualifier, qui spécifie au compilateur la surcharge d'une fonction à choisir lorsque le paramètre d'objet implicite (`*this`) est une référence rvalue par rapport à une référence lvalue. Pour plus d'informations, consultez Surcharge des fonctions.

Définitions de fonction

Une *définition de fonction* se compose de la déclaration et du corps de la fonction, entourés d'accolades, qui contiennent des déclarations de variables, des instructions et des expressions. L'exemple suivant montre une définition de fonction complète :

```
C++

    int foo(int i, std::string s)
    {
       int value {i};
       MyClass mc;
       if(strcmp(s, "default") != 0)
       {
            value = mc.do_something(i);
       }
```

```
    return value;
}
```

Les variables déclarées à l'intérieur du corps sont appelées variables locales. Elles sortent de la portée lors de la fermeture de la fonction ; par conséquent, une fonction ne doit jamais retourner une référence à une variable locale !

C++

```
MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

fonctions const et constexpr

Vous pouvez déclarer une fonction membre comme `const` pour spécifier que la fonction n'est pas autorisée à modifier les valeurs des membres de données de la classe . En déclarant une fonction membre comme `const`, vous aidez le compilateur à appliquer *const-correctness*. Si quelqu'un tente par erreur de modifier l'objet à l'aide d'une fonction déclarée comme `const`, une erreur du compilateur est générée. Pour plus d'informations, consultez const.

Déclarez une fonction comme `constexpr` lorsque la valeur qu'elle produit peut éventuellement être déterminée au moment de la compilation. Une fonction constexpr s'exécute généralement plus rapidement qu'une fonction normale. Pour plus d'informations, consultez constexpr.

Modèles de fonctions

Un modèle de fonction est similaire à un modèle de classe ; il génère des fonctions concrètes basées sur les arguments template. Dans de nombreux cas, le modèle est capable de déduire les arguments de type et, par conséquent, il n'est pas nécessaire de les spécifier explicitement.

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
```

```
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895); // a is a double
auto b = Add2(string{ "Hello" }, string{ " World" }); // b is a std::string
```

Pour plus d'informations, consultez Modèles de fonction.

Paramètres et arguments de fonction

Une fonction possède une liste de paramètres séparés par des virgules de zéro, un ou plusieurs types, chacun d'eux ayant un nom selon lequel il est accessible à l'intérieur du corps de la fonction. Un modèle de fonction peut spécifier d'autres paramètres de type ou de valeur. L'appelant transmet les arguments, qui sont des valeurs concrètes dont les types sont compatibles avec la liste de paramètres.

Par défaut, les arguments sont passés à la fonction par valeur, ce qui signifie que la fonction reçoit une copie de l'objet passé. Pour les objets volumineux, la réalisation d'une copie peut être coûteuse et n'est pas toujours nécessaire. Pour que les arguments soient passés par référence (plus précisément la référence lvalue), ajoutez un quantificateur de référence au paramètre :

```
C++

void DoSomething(std::string& input){...}
```

Lorsqu'une fonction modifie un argument passé par référence, elle modifie l'objet d'origine, pas une copie locale. Pour empêcher une fonction de modifier un tel argument, qualifiez le paramètre comme const& :

```
C++

void DoSomething(const std::string& input){...}
```

C++ 11 : Pour gérer explicitement les arguments passés par rvalue-reference ou lvalue-reference, utilisez une double-esperluette sur le paramètre pour indiquer une référence universelle :

```
C++

void DoSomething(const std::string&& input){...}
```

Une fonction déclarée avec le mot clé `void` unique dans la liste de déclaration de paramètre ne prend aucun argument, tant que le mot clé `void` est le premier et le seul membre de la liste de déclaration d'arguments. Les arguments de type `void` ailleurs dans la liste génèrent des erreurs. Par exemple :

C++

```
// OK same as GetTickCount()
long GetTickCount( void );
```

Bien qu'il ne soit pas possible de spécifier un `void` argument, sauf comme indiqué ici, les types dérivés du type `void` (tels que les pointeurs vers `void` et les tableaux de) peuvent apparaître n'importe où dans la liste de déclarations d'arguments `void`.

Arguments par défaut

Le ou les derniers paramètres dans une signature de fonction peuvent recevoir un argument par défaut, ce qui signifie que l'appelant peut omettre l'argument lors de l'appel de la fonction, à moins de vouloir spécifier une autre valeur.

C++

```
int DoSomething(int num,
    string str,
    Allocator& alloc = defaultAllocator)
{ ... }

// OK both parameters are at end
int DoSomethingElse(int num,
    string str = string{ "Working" },
    Allocator& alloc = defaultAllocator)
{ ... }

// C2548: 'DoMore': missing default parameter for parameter 2
int DoMore(int num = 5, // Not a trailing parameter!
    string str,
    Allocator& = defaultAllocator)
{...}
```

Pour plus d'informations, consultez Arguments par défaut.

Types de retour de fonction

Une fonction ne peut pas retourner une autre fonction ou un tableau intégré ; Toutefois, il peut retourner des pointeurs vers ces types, ou un *lambda*, qui produit un objet de

fonction. Sauf dans ces cas, une fonction peut retourner une valeur de n'importe quel type qui est dans l'étendue, ou elle peut retourner aucune valeur, auquel cas le type de retour est `void`.

Types de retours de fin

Un type de retour « ordinaire » se trouve sur le côté gauche de la signature de fonction. Un *type de retour de fin* se trouve sur le côté le plus à droite de la signature et est précédé de l'opérateur `->` . Les types de retours de fin sont particulièrement utiles dans les modèles de fonction quand le type de la valeur de retour dépend des paramètres de modèle.

```
C++

template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

Quand `auto` est utilisé conjointement avec un type de retour de fin, il sert simplement d'espace réservé pour tout ce que l'expression decltype produit, et n'effectue pas elle-même la déduction de type.

Variables locales de fonction

Une variable déclarée à l'intérieur d'un corps de fonction est appelée *variable locale* ou simplement variable *locale*. Les locaux non statiques ne sont visibles qu'à l'intérieur du corps de la fonction et, s'ils sont déclarés sur la pile, sortent de l'étendue lorsque la fonction se ferme. Lorsque vous construisez une variable locale et que vous la retournez par valeur, le compilateur peut généralement effectuer *l'optimisation de la valeur de retour nommée* pour éviter les opérations de copie inutiles. Si vous retournez une variable locale par référence, le compilateur émet un avertissement, car toute tentative d'utiliser cette référence par l'appelant se produit après la destruction de la variable locale.

En C++, une variable locale peut être déclarée comme statique. La variable n'est visible que dans le corps de fonction, mais une seule copie de la variable existe pour toutes les instances de la fonction. Les objets statiques locaux sont détruits durant l'arrêt spécifié par `atexit`. Si un objet statique n'a pas été construit parce que le flux de contrôle du programme a contourné sa déclaration, aucune tentative de destruction de cet objet n'est effectuée.

Déduction de type dans les types de retours (C++14)

Dans C++14, vous pouvez utiliser `auto` pour indiquer au compilateur de déduire le type de retour à partir du corps de la fonction sans avoir à fournir un type de retour de fin. Notez que `auto` déduisent toujours à un retour par valeur. Utilisez `auto&&` pour indiquer au compilateur de déduire une référence.

Dans cet exemple, `auto` est déduit comme une copie de valeur non const de la somme de lhs et rhs.

C++

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

Notez que `auto` ne conserve pas la const-ness du type qu'il déduit. Pour les fonctions de transfert dont la valeur de retour doit conserver la const-ness ou la ref-ness de ses arguments, vous pouvez utiliser le `decltype(auto)` mot clé, qui utilise les règles d'inférence de `decltype` type et conserve toutes les informations de type. `decltype(auto)` peut être utilisé comme valeur de retour ordinaire sur le côté gauche, ou en tant que valeur de retour de fin.

L'exemple suivant (basé sur le code N3493) montre `decltype(auto)` comment être utilisé pour activer le transfert parfait des arguments de fonction dans un type de retour qui n'est pas connu tant que le modèle n'est pas instancié.

C++

```
template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);
}

template<typename F, typename Tuple = tuple<T...>,
    typename Indices = make_index_sequence<tuple_size<Tuple>::value >>
   decltype( auto)
    apply(F&& f, Tuple&& args)
{
    return apply_(std::forward<F>(f), std::forward<Tuple>(args), Indices());
}
```

Retour de plusieurs valeurs à partir d'une fonction

Il existe plusieurs façons de retourner plusieurs valeurs à partir d'une fonction :

1. Encapsulez les valeurs dans une classe ou un objet struct nommé. Nécessite que la définition de classe ou de struct soit visible par l'appelant :

```
C++
```

```cpp
#include <string>
#include <iostream>

using namespace std;

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    S s = g();
    cout << s.name << " " << s.num << endl;
    return 0;
}
```

2. Retourne un objet std::tuple ou std::p air :

```
C++
```

```cpp
#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
```

```
    return { i,s,d };
}

int main()
{
    auto t = f();
    cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << endl;

    // --or--

    int myval;
    string myname;
    double mydecimal;
    tie(myval, myname, mydecimal) = f();
    cout << myval << " " << myname << " " << mydecimal << endl;

    return 0;
}
```

3. **Visual Studio 2017 version 15.3 et ultérieures** (disponible en /std:c++17 mode et versions ultérieures) : utilisez des liaisons structurées. L'avantage des liaisons structurées est que les variables qui stockent les valeurs de retour sont initialisées en même temps qu'elles sont déclarées, ce qui peut être beaucoup plus efficace dans certains cas. Dans l'instruction `auto[x, y, z] = f();` , les crochets introduisent et initialisent les noms qui sont dans l'étendue du bloc de fonction entier.

C++

```
#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}
struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
```

```
    int u{ 42 };
    return { t, u };
}

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
    return 0;
}
```

4. En plus d'utiliser la valeur de retour elle-même, vous pouvez « retourner » des valeurs en définissant n'importe quel nombre de paramètres à utiliser par référence directe afin que la fonction puisse modifier ou initialiser les valeurs des objets que l'appelant fournit. Pour plus d'informations, consultez Arguments de fonction reference-type.

Pointeurs fonction

C++ prend en charge les pointeurs de fonction de la même manière que le langage C. Toutefois, une alternative de type plus sécurisé consiste généralement à utiliser un objet de fonction.

Il est recommandé d'utiliser `typedef` pour déclarer un alias pour le type de pointeur de fonction si vous déclarez une fonction qui retourne un type de pointeur de fonction. Par exemple

C++

```
typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer
```

Si ce n'est pas le cas, la syntaxe appropriée pour la déclaration de fonction peut être déduite de la syntaxe du déclarateur pour le pointeur de fonction en remplaçant l'identificateur (`fp` dans l'exemple ci-dessus) par le nom de la fonction et la liste d'arguments, comme suit :

C++

```
int (*myFunction(char* s))(int);
```

La déclaration précédente est équivalente à la déclaration utilisée `typedef` précédemment.

Voir aussi

Surcharge des fonctions
Fonctions avec listes d'arguments de variables
Fonctions explicitement par défaut et supprimées
Recherche de nom dépendant de l'argument (Koenig) sur les fonctions
Arguments par défaut
Fonctions inline

Fonctions avec listes d'arguments variables (C++)

Article • 03/04/2023

Les déclarations de fonction dans lesquelles le dernier membre de est les points de suspension (...) peuvent prendre un nombre variable d'arguments. Dans ces cas-là, C++ fournit une vérification du type uniquement pour les arguments déclarés explicitement. Vous pouvez utiliser des listes d'arguments variables lorsque vous devez rendre une fonction si générale que même le nombre et les types des arguments peuvent varier. La famille de fonctions est un exemple de fonctions qui utilisent des listes d'arguments de variables. `printf` *argument-declaration-list*

Fonctions avec des arguments de variables

Pour accéder aux arguments après ceux déclarés, utilisez les macros contenues dans le fichier <include standard stdarg.h> , comme décrit ci-dessous.

Section spécifique à Microsoft

Microsoft C++ autorise la spécification de l'ellipse comme argument s'il s'agit du dernier argument et qu'elle est précédée d'une virgule. Par conséquent, la déclaration `int Func( int i, ... );` est conforme, contrairement à `int Func( int i ... );`.

FIN de la section spécifique à Microsoft

La déclaration d'une fonction qui accepte une quantité variable d'arguments nécessite au moins un argument d'espace réservé, même si elle n'est pas utilisée. Si cet argument d'espace réservé n'est pas fourni, il est impossible d'accéder aux arguments restants.

Lorsque des arguments de type `char` sont passés en tant qu'arguments variables, ils sont convertis en type `int`. De même, lorsque des arguments de type `float` sont passés en tant qu'arguments variables, ils sont convertis en type `double`. Les arguments d'autres types sont soumis aux promotions intégrales et à virgule flottante classiques. Pour plus d'informations, consultez Conversions standard .

Les fonctions qui requièrent des listes de variables sont déclarées avec des points de suspension (...) dans la liste d'arguments. Utilisez les types et macros décrits dans le <fichier include stdarg.h> pour accéder aux arguments passés par une liste de variables. Pour plus d'informations sur ces macros, consultez va_arg, va_copy, va_end va_start. dans la documentation de la bibliothèque Runtime C.

L'exemple suivant montre comment les macros fonctionnent avec le type (déclaré dans <stdarg.h>) :

C++

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

//  Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
   ShowVar( "fcsi", 32.4f, 'a', "Test string", 4 );
}

//  ShowVar takes a format string of the form
//   "ifcs", where each character specifies the
//   type of the argument in that position.
//
//  i = int
//  f = float
//  c = char
//  s = string (char *)
//
//  Following the format specification is a variable
//  list of arguments. Each argument corresponds to
//  a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
   va_list vl;
   int i;

   //  szTypes is the last argument specified; you must access
   //  all others using the variable-argument macros.
   va_start( vl, szTypes );

   // Step through the list.
   for( i = 0; szTypes[i] != '\0'; ++i ) {
      union Printable_t {
         int     i;
         float   f;
         char    c;
         char   *s;
      } Printable;

      switch( szTypes[i] ) {   // Type to expect.
         case 'i':
            Printable.i = va_arg( vl, int );
            printf_s( "%i\n", Printable.i );
         break;

         case 'f':
             Printable.f = va_arg( vl, double );
             printf_s( "%f\n", Printable.f );
```

```
            break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
            break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
            break;

            default:
            break;
        }
    }
    va_end( vl );
}
//Output:
// 32.400002
// a
// Test string
```

L'exemple ci-dessus illustre les concepts importants suivants :

1. Vous devez établir un marqueur de liste en tant que variable de type `va_list` avant d'accéder à tout argument variable. Dans l'exemple précédent, le marqueur est appelé `vl`.

2. L'accès aux arguments individuels s'effectue à l'aide de la macro `va_arg`. Vous devez indiquer à la macro `va_arg` le type d'argument à extraire afin qu'elle puisse transférer le nombre correct d'octets de la pile. Si vous spécifiez un type incorrect d'une taille différente de celle fournie par le programme appelant à `va_arg`, les résultats sont imprévisibles.

3. Vous devez effectuer un cast explicite du résultat obtenu à l'aide de la macro `va_arg` vers le type souhaité.

Vous devez appeler la macro pour arrêter le traitement des arguments variables.`va_end`

Surcharge de fonction

Article • 03/04/2023

C++ vous permet de spécifier plusieurs fonctions du même nom dans la même étendue. Ces fonctions sont appelées fonctions *surchargées* , ou *surcharges*. Les fonctions surchargées vous permettent de fournir une sémantique différente pour une fonction, en fonction des types et du nombre de ses arguments.

Par exemple, considérez une `print` fonction qui prend un `std::string` argument. Cette fonction peut effectuer des tâches très différentes d'une fonction qui prend un argument de type `double`. La surcharge vous évite d'avoir à utiliser des noms tels que `print_string` ou `print_double`. Au moment de la compilation, le compilateur choisit la surcharge à utiliser en fonction des types et du nombre d'arguments transmis par l'appelant. Si vous appelez `print(42.0)`, la `void print(double d)` fonction est appelée. Si vous appelez `print("hello world")`, la `void print(std::string)` surcharge est appelée.

Vous pouvez surcharger les fonctions membres et les fonctions libres. Le tableau suivant montre quelles parties d'une déclaration de fonction C++ utilise pour différencier les groupes de fonctions portant le même nom dans la même étendue.

Considérations en matière de surcharge

Élément de déclaration de fonction	**Utilisé pour la surcharge ?**
Type de retour de fonction	Non
Nombre d'arguments	Oui
Type d'arguments	Oui
Présence ou absence de points de suspension	Oui
Utilisation de `typedef` noms	Non
Limites de tableau non spécifiées	Non
`const` ou `volatile`	Oui, en cas d'application à la fonction entière
Qualificateurs de référence (`&` et `&&`)	Oui

Exemple

L'exemple suivant montre comment utiliser les surcharges de fonction :

C++

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>
#include <string>

// Prototype three print functions.
int print(std::string s);             // Print a string.
int print(double dvalue);             // Print a double.
int print(double dvalue, int prec);   // Print a double with a
                                      //  given precision.
using namespace std;
int main(int argc, char *argv[])
{
    const double d = 893094.2987;
    if (argc < 2)
    {
        // These calls to print invoke print( char *s ).
        print("This program requires one argument.");
        print("The argument specifies the number of");
        print("digits precision for the second number");
        print("printed.");
        exit(0);
    }

    // Invoke print( double dvalue ).
    print(d);

    // Invoke print( double dvalue, int prec ).
    print(d, atoi(argv[1]));
}

// Print a string.
int print(string s)
{
    cout << s << endl;
    return cout.good();
}

// Print a double in default precision.
int print(double dvalue)
{
    cout << dvalue << endl;
    return cout.good();
}

//  Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
```

```
//  to the left of the decimal point.
int print(double dvalue, int prec)
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1,
        10E0, 10E1,  10E2,  10E3,  10E4, 10E5,  10E6 };
    const int iPowZero = 6;

    // If precision out of range, just print the number.
    if (prec < -6 || prec > 7)
    {
        return print(dvalue);
    }
    // Scale, truncate, then rescale.
    dvalue = floor(dvalue / rgPow10[iPowZero - prec]) *
        rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}
```

Le code précédent montre les surcharges de la fonction dans l'étendue `print` du fichier.

L'argument par défaut n'est pas considéré comme faisant partie du type de fonction. Par conséquent, il n'est pas utilisé pour sélectionner des fonctions surchargées. Deux fonctions qui diffèrent uniquement de par leurs arguments par défaut sont considérées comme des définitions distinctes plutôt que comme des fonctions surchargées.

Les arguments par défaut ne peuvent pas être fournis pour les opérateurs surchargés.

Correspondance d'argument

Le compilateur sélectionne la fonction surchargée à appeler en fonction de la meilleure correspondance entre les déclarations de fonction dans l'étendue actuelle et les arguments fournis dans l'appel de fonction. Si une fonction appropriée est trouvée, elle est appelée. « Approprié » dans ce contexte signifie :

- Une correspondance exacte a été trouvée.
- Une conversion ordinaire a été exécutée.
- Une promotion intégrale a été exécutée.
- Une conversion standard vers le type d'argument souhaité existe.
- Il existe une conversion définie par l'utilisateur (opérateur de conversion ou constructeur) vers le type d'argument souhaité.

- Des arguments représentés par une ellipse ont été détectés.

Le compilateur crée un ensemble de fonctions candidates pour chaque argument. Les fonctions candidates sont des fonctions dans lesquelles l'argument réel dans cette position peut être converti vers le type de l'argument formel.

Un ensemble des « meilleures fonctions correspondantes » est généré pour chaque argument, et la fonction sélectionnée correspond à l'intersection de tous les ensembles. Si l'intersection contient plusieurs fonctions, la surcharge est ambiguë et génère une erreur. La fonction qui est finalement sélectionnée correspond toujours mieux que toutes les autres fonctions du groupe pour au moins un argument. S'il n'y a pas de gagnant clair, l'appel de fonction génère une erreur du compilateur.

Prenons les déclarations suivantes (les fonctions sont marquées `Variant 1`, `Variant 2` et `Variant 3`, afin de les identifier dans la discussion suivante) :

```
C++

Fraction &Add( Fraction &f, long l );       // Variant 1
Fraction &Add( long l, Fraction &f );       // Variant 2
Fraction &Add( Fraction &f, Fraction &f );  // Variant 3

Fraction F1, F2;
```

Prenons l'instruction suivante :

```
C++

F1 = Add( F2, 23 );
```

L'instruction précédente génère deux ensembles :

Jeu 1 : fonctions candidates qui ont le premier argument de type `Fraction`	**Set 2 : Fonctions candidates dont le deuxième argument peut être converti en type `int`**
Variant 1	Variante 1 (**`int`** peut être converti en à l'aide d'une **`long`** conversion standard)
Variant 3	

Les fonctions de l'ensemble 2 sont des fonctions qui ont des conversions implicites du type de paramètre réel au type de paramètre formel. L'une de ces fonctions a le plus petit « coût » pour convertir le type de paramètre réel en son type de paramètre formel correspondant.

L'intersection de ces deux ensembles est Variant 1. Voici un exemple d'appel de fonction ambigu :

```
C++

F1 = Add( 3, 6 );
```

L'appel de fonction précédent génère les ensembles suivants :

Jeu 1 : Fonctions candidates qui ont le premier argument de type `int`	**Jeu 2 : fonctions candidates qui ont un deuxième argument de type `int`**
Variante 2 (`int` peut être converti en à l'aide d'une `long` conversion standard)	Variante 1 (`int` peut être converti en à l'aide d'une `long` conversion standard)

Étant donné que l'intersection de ces deux ensembles est vide, le compilateur génère un message d'erreur.

Pour la correspondance d'arguments, une fonction avec *n* arguments par défaut est traitée comme *n*+1 fonctions distinctes, chacune avec un nombre différent d'arguments.

Les points de suspension (`...`) agissent comme un caractère générique ; ils correspondent à n'importe quel argument réel. Cela peut entraîner de nombreux ensembles ambigus, si vous ne concevez pas vos jeux de fonctions surchargés avec un soin extrême.

> ⓘ **Notes**
>
> L'ambiguïté des fonctions surchargées ne peut pas être déterminée tant qu'un appel de fonction n'est pas rencontré. À ce stade, les ensembles sont générés pour chaque argument dans l'appel de fonction, et vous pouvez déterminer si une surcharge non équivoque existe. Cela signifie que les ambiguïtés peuvent rester dans votre code jusqu'à ce qu'elles soient évoquées par un appel de fonction particulier.

Différences de type d'argument

Les fonctions surchargées distinguent les types d'arguments qui acceptent différents initialiseurs. Par conséquent, un argument d'un type donné et une référence à ce type sont considérés comme identiques pour les besoins de la surcharge. Ils sont considérés comme les mêmes, car ils prennent les mêmes initialiseurs. Par exemple, `max( double,`

`double )` est considéré comme identique à `max( double &, double & )`. La déclaration de deux de ces fonctions provoque une erreur.

Pour la même raison, les arguments de fonction d'un type modifié par **`const`** ou **`volatile`** ne sont pas traités différemment du type de base à des fins de surcharge.

Toutefois, le mécanisme de surcharge de fonction peut faire la distinction entre les références qualifiées par **`const`** et **`volatile`** et les références au type de base. Il rend possible le code tel que le suivant :

C++

```
// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
   Over() { cout << "Over default constructor\n"; }
   Over( Over &o ) { cout << "Over&\n"; }
   Over( const Over &co ) { cout << "const Over&\n"; }
   Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

int main() {
   Over o1;            // Calls default constructor.
   Over o2( o1 );      // Calls Over( Over& ).
   const Over o3;      // Calls default constructor.
   Over o4( o3 );      // Calls Over( const Over& ).
   volatile Over o5;   // Calls default constructor.
   Over o6( o5 );      // Calls Over( volatile Over& ).
}
```

Output

Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

Les pointeurs vers **`const`** les objets et **`volatile`** sont également considérés comme différents des pointeurs vers le type de base à des fins de surcharge.

Correspondance et conversions d'argument

Lorsque le compilateur tente de mettre en correspondance les arguments réels et les arguments des déclarations de fonction, il peut fournir des conversions standard ou définies par l'utilisateur pour obtenir le type correct si aucune correspondance exacte n'est trouvée. L'application des conversions est soumise aux règles ci-dessous.

- Les séquences de conversions qui contiennent plusieurs conversions définies par l'utilisateur ne sont pas prises en compte.
- Les séquences de conversions qui peuvent être raccourcies en supprimant les conversions intermédiaires ne sont pas prises en compte.

La séquence de conversions résultante, le cas échéant, est appelée *séquence de correspondance* optimale. Il existe plusieurs façons de convertir un objet de type `int` en type `unsigned long` à l'aide de conversions standard (décrites dans Conversions standard) :

- Convertissez de `int` vers `long` , puis de `long` en `unsigned long`.
- Convertissez de `int` en `unsigned long`.

Bien que la première séquence atteigne l'objectif souhaité, elle n'est pas la meilleure séquence correspondante, car il existe une séquence plus courte.

Le tableau suivant présente un groupe de conversions *appelées conversions triviales*. Les conversions triviales ont un effet limité sur la séquence que le compilateur choisit comme meilleure correspondance. L'effet des conversions triviales est décrit après le tableau.

Conversions triviales

Type d'argument	Type converti
`type-name`	`type-name&`
`type-name&`	`type-name`
`type-name[]`	`type-name*`
`type-name(argument-list)`	`(*type-name)(argument-list)`
`type-name`	`const type-name`
`type-name`	`volatile type-name`

Type d’argument	Type converti
`type-name*`	`const type-name*`
`type-name*`	`volatile type-name*`

La séquence dans laquelle les conversions sont tentées est la suivante :

1. Correspondance exacte. Une correspondance exacte entre les types avec lesquels la fonction est appelée et les types déclarés dans le prototype de fonction est toujours la meilleure correspondance. Les séquences de conversions ordinaires sont classées comme correspondances exactes. Toutefois, les séquences qui n’effectuent aucune de ces conversions sont considérées comme meilleures que les séquences qui convertissent :

    - Du pointeur à pointeur vers **`const`** (`type-name*` vers `const type-name*`).
    - Du pointeur à pointeur vers **`volatile`** (`type-name*` vers `volatile type-name*`).
    - De référence, à référence à **`const`** (`type-name&` à `const type-name&`).
    - De référence, à référence à **`volatile`** (`type-name&` à `volatile type&`).

2. Correspondance avec des promotions. Toute séquence non classée comme une correspondance exacte qui contient uniquement des promotions intégrales, des conversions de **`float`** à **`double`** et des conversions triviales est classée comme une correspondance à l’aide de promotions. Bien qu'elle ne soit pas aussi appropriée qu'une correspondance exacte, une correspondance avec des promotions est meilleure qu'une correspondance avec des conversions standard.

3. Correspondance avec des conversions standard. Toute séquence non classée comme correspondance exacte ou correspondance avec des promotions et qui contient uniquement des conversions standard et des conversions ordinaires est classée comme correspondance avec des conversions standard. Dans cette catégorie, les règles ci-dessous s'appliquent.

    - La conversion d’un pointeur vers une classe dérivée en pointeur vers une classe de base directe ou indirecte est préférable à la conversion en `void *` ou `const void *`.
    - La conversion d'un pointeur vers une classe dérivée, vers un pointeur vers une classe de base génère une correspondance d'autant meilleure que la classe de base est proche d'une classe de base directe. Supposons que la hiérarchie de classes soit comme illustré dans la figure suivante :

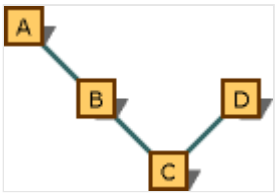
Graphique montrant les conversions préférées.

La conversion du type `D*` vers le type `C*` est préférable à une conversion du type `D*` vers le type `B*`. De même, la conversion du type `D*` vers le type `B*` est préférable à une conversion du type `D*` vers le type `A*`.

Cette même règle s'applique aux conversions de référence. La conversion du type `D&` vers le type `C&` est préférable à une conversion du type `D&` vers le type `B&`, et ainsi de suite.

Cette même règle s'applique aux conversions de pointeur vers membre. La conversion du type `T D::*` vers le type `T C::*` est préférable à une conversion du type `T D::*` vers le type `T B::*`, et ainsi de suite (où `T` est le type du membre).

La règle précédente s'applique uniquement à un chemin de dérivation donné. Examinez le graphique présenté dans l'illustration ci-dessous.



Graphique à héritage multiple qui montre les conversions préférées.

La conversion du type `C*` vers le type `B*` est préférable à une conversion du type `C*` vers le type `A*`. Cela provient du fait qu'ils se trouvent dans le même chemin et que `B*` est plus proche. Toutefois, la conversion d'un type `C*` à un type `D*` n'est pas préférable à la conversion en type `A*`; il n'y a pas de préférence, car les conversions suivent des chemins différents.

1. Correspondance avec des conversions définies par l'utilisateur. Cette séquence ne peut pas être classée comme une correspondance exacte, une correspondance à l'aide de promotions ou une correspondance à l'aide de conversions standard. Pour être classée comme une correspondance avec des conversions définies par l'utilisateur, la séquence doit uniquement contenir des conversions définies par l'utilisateur, des conversions standard ou des conversions triviales. Une correspondance avec des conversions définies par l'utilisateur est considérée comme une meilleure correspondance qu'une correspondance avec des points de

suspension (`...`), mais pas une correspondance aussi bonne qu'une correspondance avec des conversions standard.

2. Correspondance avec des points de suspension. Toute séquence qui correspond à des points de suspension dans la déclaration est classée comme correspondance avec des points de suspension. C'est considéré comme le match le plus faible.

Les conversions définies par l'utilisateur sont appliquées s'il n'existe aucune promotion ou conversion intégrée. Ces conversions sont sélectionnées en fonction du type de l'argument mis en correspondance. Prenez le code suivant :

C++

```
// argument_matching1.cpp
class UDC
{
public:
   operator int()
   {
      return 0;
   }
   operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
   Print( udc );
}
```

Les conversions définies par l'utilisateur disponibles pour la classe `UDC` sont de type **`int`** et de type **`long`**. Par conséquent, le compilateur considère les conversions pour le type de l'objet qui est mis en correspondance : `UDC`. Une conversion en **`int`** existe et elle est sélectionnée.

Pendant le processus de mise en correspondance d'arguments, les conversions standard peuvent être appliquées à l'argument et au résultat d'une conversion définie par l'utilisateur. Par conséquent, le code suivant fonctionne :

C++

```
void LogToFile( long l );
...
```

```
UDC udc;
LogToFile( udc );
```

Dans cet exemple, le compilateur appelle une conversion définie par l'utilisateur, `operator long`, pour convertir en `udc` type **`long`**. Si aucune conversion en type **`long`** définie par l'utilisateur n'a été définie, le compilateur convertit d'abord le type `UDC` en type **`int`** à l'aide de la conversion définie par `operator int` l'utilisateur. Ensuite, il applique la conversion standard de type **`int`** en type **`long`** pour correspondre à l'argument dans la déclaration.

Si des conversions définies par l'utilisateur sont requises pour faire correspondre un argument, les conversions standard ne sont pas utilisées lors de l'évaluation de la meilleure correspondance. Même si plusieurs fonctions candidates nécessitent une conversion définie par l'utilisateur, les fonctions sont considérées comme égales. Par exemple :

C++

```
// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
   UDC1( int );  // User-defined conversion from int.
};

class UDC2
{
public:
   UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
   Func( 1 );
}
```

Les deux versions de `Func` nécessitent une conversion définie par l'utilisateur pour convertir le type **`int`** en argument de type de classe. Les conversions possibles sont indiquées ci-dessous.

- Convertir de type **`int`** en type `UDC1` (conversion définie par l'utilisateur).

- Convertir de type `int` en type `long`, puis convertir en type `UDC2` (conversion en deux étapes).

Même si la seconde nécessite à la fois une conversion standard et une conversion définie par l'utilisateur, les deux conversions sont toujours considérées comme égales.

> ⓘ Notes
>
> Les conversions définies par l'utilisateur sont considérées comme une conversion par construction ou par initialisation. Le compilateur considère les deux méthodes comme égales lorsqu'il détermine la meilleure correspondance.

Correspondance d'argument et pointeur `this`

Les fonctions membres de classe sont traitées différemment, selon qu'elles sont déclarées en tant que `static`. `static` Les fonctions n'ayant pas d'argument implicite qui fournit le `this` pointeur, elles sont considérées comme ayant un argument de moins que les fonctions membres normales. Sinon, elles sont déclarées identiques.

Les fonctions membres qui ne nécessitent pas `static` que le pointeur implicite `this` corresponde au type d'objet via lequel la fonction est appelée. Ou, pour les opérateurs surchargés, ils nécessitent le premier argument pour correspondre à l'objet auquel l'opérateur est appliqué. Pour plus d'informations sur les opérateurs surchargés, consultez Opérateurs surchargés.

Contrairement à d'autres arguments dans les fonctions surchargées, le compilateur n'introduit aucun objet temporaire et ne tente aucune conversion lors de la tentative de correspondance avec l'argument `this` pointeur.

Lorsque l'opérateur `->` de sélection de membre est utilisé pour accéder à une fonction membre de la classe `class_name`, l'argument `this` pointeur a un type .`class_name * const` Si les membres sont déclarés comme `const` ou `volatile`, les types sont `const class_name * const` et `volatile class_name * const`, respectivement.

L'opérateur de sélection de membres `.` fonctionne exactement de la même façon, sauf qu'un opérateur (d'adresse) `&` implicite est préfixé au nom de l'objet. L'exemple suivant illustre cela :

```
C++

// Expression encountered in code
obj.name
```

```
// How the compiler treats it
(&obj)->name
```

L'opérande gauche des opérateurs (pointeur vers membre) `->*` et `.*` est traité de la même façon que les opérateurs (sélection de membres) `.` et `->` en ce qui concerne la correspondance d'arguments.

Qualificateurs de référence sur les fonctions membres

Les qualificateurs de référence permettent de surcharger une fonction membre selon que l'objet pointé par **`this`** est une rvalue ou une lvalue. Utilisez cette fonctionnalité pour éviter les opérations de copie inutiles dans les scénarios où vous choisissez de ne pas fournir d'accès pointeur aux données. Par exemple, supposons que la classe `C` initialise certaines données dans son constructeur et retourne une copie de ces données dans la fonction `get_data()` membre . Si un objet de type `C` est une rvalue sur le point d'être détruite, le compilateur choisit la `get_data() &&` surcharge, qui se déplace au lieu de copier les données.

C++

```
#include <iostream>
#include <vector>

using namespace std;

class C
{

public:
    C() {/*expensive initialization*/}
    vector<unsigned> get_data() &
    {
        cout << "lvalue\n";
        return _data;
    }
    vector<unsigned> get_data() &&
    {
        cout << "rvalue\n";
        return std::move(_data);
    }

private:
    vector<unsigned> _data;
};
```

```
int main()
{
    C c;
    auto v = c.get_data(); // get a copy. prints "lvalue".
    auto v2 = C().get_data(); // get the original. prints "rvalue"
    return 0;
}
```

Restrictions sur la surcharge

Plusieurs restrictions régissent un ensemble acceptable de fonctions surchargées :

- Deux fonctions d'un ensemble de fonctions surchargées doivent avoir des listes d'arguments différentes.
- La surcharge de fonctions qui ont des listes d'arguments des mêmes types, basées uniquement sur le type de retour, est une erreur.

  Spécifique à Microsoft

  Vous pouvez surcharger **`operator new`** en fonction du type de retour, en particulier, en fonction du modificateur de modèle mémoire spécifié.

  END Spécifique à Microsoft

- Les fonctions membres ne peuvent pas être surchargées uniquement parce que l'une est **`static`** et l'autre n'est pas **`static`**.
- **`typedef`** les déclarations ne définissent pas de nouveaux types ; ils introduisent des synonymes pour les types existants. Elles n'affectent pas le mécanisme de surcharge. Prenez le code suivant :

  C++

  ```
  typedef char * PSTR;

  void Print( char *szToPrint );
  void Print( PSTR szToPrint );
  ```

  Les deux fonctions précédentes ont des listes d'arguments identiques. `PSTR` est un synonyme de type `char *`. Dans la portée du membre, ce code génère une erreur.

- Les types énumérés sont des types distincts et peuvent être utilisés pour établir une distinction entre les fonctions surchargées.

- Les types « tableau de » et « pointeur vers » sont considérés comme identiques pour faire la distinction entre les fonctions surchargées, mais uniquement pour les tableaux unidimensionnels. Ces fonctions surchargées sont en conflit et génèrent un message d'erreur :

  ```
  C++
  ```

  ```
  void Print( char *szToPrint );
  void Print( char szToPrint[] );
  ```

  Pour les tableaux de dimension plus élevés, les deuxièmes dimensions et les dimensions ultérieures sont considérées comme faisant partie du type . Ils sont utilisés pour distinguer les fonctions surchargées :

  ```
  C++
  ```

  ```
  void Print( char szToPrint[] );
  void Print( char szToPrint[][7] );
  void Print( char szToPrint[][9][42] );
  ```

Surcharge, substitution et masquage

Deux déclarations de fonction portant le même nom dans la même étendue peuvent faire référence à la même fonction ou à deux fonctions surchargées discrètes. Si les listes d'arguments des déclarations contiennent des arguments de types équivalents (comme décrit dans la section précédente), les déclarations de fonction font référence à la même fonction. Sinon, elles font référence à deux fonctions différentes qui sont sélectionnées à l'aide de la surcharge.

L'étendue de la classe est strictement respectée. Une fonction déclarée dans une classe de base n'est pas dans la même étendue qu'une fonction déclarée dans une classe dérivée. Si une fonction dans une classe dérivée est déclarée avec le même nom qu'une `virtual` fonction dans la classe de base, la fonction derived-class *remplace* la fonction de classe de base. Pour plus d'informations, consultez Fonctions virtuelles.

Si la fonction de classe de base n'est pas déclarée comme `virtual`, la fonction de classe dérivée est censée *la masquer* . Le remplacement et le masquage sont distincts de la surcharge.

L'étendue du bloc est strictement respectée. Une fonction déclarée dans l'étendue de fichier n'est pas dans la même étendue qu'une fonction déclarée localement. Si une fonction déclarée localement a le même nom qu'une fonction déclarée avec portée de

fichier, la fonction déclarée localement masque la fonction déclarée avec portée de fichier au lieu de provoquer la surcharge. Par exemple :

C++

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally declared func : " << sz << endl;
}

int main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );   // C2664 Error. func( int ) is hidden.
    func( "s" );
}
```

Le code précédent montre deux définitions de la fonction `func`. La définition qui prend un argument de type `char *` est locale en `main` raison de l'instruction **`extern`** . Par conséquent, la définition qui accepte un argument de type **`int`** est masquée et le premier appel à `func` est en erreur.

Pour les fonctions membres surchargées, différents privilèges d'accès peuvent être accordés à différentes versions de la fonction. Elles sont toujours considérées comme étant dans l'étendue de la classe englobante et sont donc des fonctions surchargées. Prenons le code suivant, dans lequel la fonction membre `Deposit` est surchargée ; une version est publique, l'autre privée.

Cet exemple sert à fournir une classe `Account` dans laquelle un mot de passe correct est requis pour effectuer des dépôts. Cela se fait à l'aide de la surcharge.

L'appel à `Deposit` dans `Account::Deposit` appelle la fonction membre privée. Cet appel est correct, car `Account::Deposit` est une fonction membre et a accès aux membres privés de la classe.

C++

```
// declaration_matching2.cpp
class Account
{
public:
   Account()
   {
   }
   double Deposit( double dAmount, char *szPassword );

private:
   double Deposit( double dAmount )
   {
      return 0.0;
   }
   int Validate( char *szPassword )
   {
      return 0;
   }

};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    //  public function.
    pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
   if ( Validate( szPassword ) )
      return Deposit( dAmount );
   else
      return 0.0;
}
```

Voir aussi

Fonctions (C++)

Fonctions utilisées par défaut et supprimées explicitement

Article • 16/06/2023

En C++11, les fonctions utilisées par défaut et supprimées vous permettent de contrôler de façon explicite si les fonctions membres spéciales sont générées automatiquement. Les fonctions supprimées vous fournissent également un langage simple pour empêcher les promotions de type problématiques dans les arguments de fonctions de tous les types (les fonctions membres spéciales, ainsi que les fonctions membres normales et les fonctions non membres) qui provoqueraient autrement un appel de fonction indésirable.

Avantages des fonctions utilisées par défaut et supprimées explicitement

En C++, le compilateur génère automatiquement le constructeur par défaut, le constructeur de copie, l'opérateur d'assignation de copie et le destructeur pour un type s'il ne déclare pas le sien. Ces fonctions sont connues sous le nom de *fonctions membres spéciales*, et elles font en sorte que les types simples définis par l'utilisateur en C++ se comportent comme le font les structures en C. Autrement dit, vous pouvez les créer, les copier et les détruire sans effort de codage supplémentaire. C++11 fournit la sémantique de déplacement au langage et ajoute le constructeur de déplacement et l'opérateur d'assignation de mouvement à la liste des fonctions membres spéciales que le compilateur peut générer automatiquement.

Cela est pratique pour les types simples, mais les types complexes définissent eux-mêmes souvent une ou plusieurs des fonctions membres spéciales, et cela peut empêcher d'autres fonctions membres spéciales d'être générées automatiquement. Dans la pratique :

- Si un constructeur est déclaré explicitement, aucun constructeur par défaut n'est automatiquement généré.
- Si un destructeur virtuel est déclaré explicitement, aucun destructeur par défaut n'est automatiquement généré.
- Si un constructeur de déplacement ou un opérateur d'assignation de déplacement est déclaré explicitement :
    - Aucun constructeur de copie n'est généré automatiquement.

    - Aucun opérateur d'assignation de copie n'est généré automatiquement.

- Si un constructeur de copie, un opérateur d'assignation de copie, un constructeur de déplacement, un opérateur d'assignation de mouvement ou un destructeur est déclaré explicitement :
    - Aucun constructeur de déplacement n'est généré automatiquement.
    - Aucun opérateur d'assignation de déplacement n'est généré automatiquement.

> ⓘ Notes
>
> En outre, la norme C++11 spécifie les règles supplémentaires suivantes :
>
> - Si un constructeur de copie ou un destructeur est déclaré explicitement, la génération automatique de l'opérateur d'assignation de copie est déconseillée.
> - Si un opérateur d'assignation de copie ou un destructeur est déclaré explicitement, la génération automatique du constructeur de copie est déconseillée.
>
> Dans les deux cas, Visual Studio continue à générer automatiquement les fonctions nécessaires implicitement, sans émettre d'avertissement.

Les conséquences de ces règles peuvent également se répercuter dans la hiérarchie des objets. Par exemple, si, pour une raison quelconque, une classe de base ne parvient pas à avoir un constructeur par défaut pouvant être appelé à partir d'une classe dérivée, c'est-à-dire un `public` constructeur ou `protected` qui ne prend aucun paramètre, une classe qui en dérive ne peut pas générer automatiquement son propre constructeur par défaut.

Ces règles peuvent compliquer l'implémentation d'idiomes C++ qui devraient pourtant être simples, courants et de types définis par l'utilisateur (par exemple, rendre un type défini par l'utilisateur impossible à copier en déclarant le constructeur de copie et l'opérateur d'assignation de copie en privé et en ne les définissant pas).

C++

```
struct noncopyable
{
  noncopyable() {};

private:
  noncopyable(const noncopyable&);
```

```
  noncopyable& operator=(const noncopyable&);
};
```

Avant C++11, cet extrait de code était la forme idiomatique des types qu'il était impossible de copier. Toutefois, il présente plusieurs problèmes :

- Le constructeur de copie doit être déclaré en privé pour le masquer, mais comme il est déclaré du tout, la génération automatique du constructeur par défaut est empêchée. Vous devez définir explicitement le constructeur par défaut si vous en souhaitez un, même s'il ne fait rien.
- Même si le constructeur par défaut défini explicitement ne fait rien, il est considéré comme étant non trivial par le compilateur. Il est moins efficace qu'un constructeur par défaut généré automatiquement et empêche un type `noncopyable` d'être un type POD réel.
- Bien que le constructeur de copie et l'opérateur d'assignation de copie soient masqués vis-à-vis du code externe, les fonctions membres et les amis de `noncopyable` peuvent tout de même les voir et les appeler. S'ils sont déclarés mais non définis, leur appel entraîne une erreur de l'éditeur de liens.
- Bien qu'il s'agisse d'un idiome couramment accepté, l'intention n'est pas claire tant que vous ne comprenez pas toutes les règles de génération automatique des fonctions membres spéciales.

En C++11, l'idiome non copiable peut être implémenté de façon plus simple.

C++

```
struct noncopyable
{
  noncopyable() =default;
  noncopyable(const noncopyable&) =delete;
  noncopyable& operator=(const noncopyable&) =delete;
};
```

Notez comment les problèmes d'idiome pré-C++11 sont résolus :

- La génération du constructeur par défaut est toujours empêchée en déclarant le constructeur de copie, mais vous pouvez la réactiver en la définissant comme valeur par défaut explicite.
- Les fonctions membres spéciales utilisées par défaut explicitement sont toujours considérées comme triviales ; par conséquent, il n'y a aucune altération des performances et un type `noncopyable` peut être un type POD réel.

- Le constructeur de copie et l'opérateur d'assignation de copie sont publics, mais supprimés. La définition ou l'appel d'une fonction supprimée constitue une erreur de compilation.
- L'intention est claire pour toute personne qui comprend `=default` et `=delete`. Vous ne devez pas comprendre les règles pour la génération automatique des fonctions membres spéciales.

Des idiomes similaires existent pour créer des types définis par l'utilisateur qui ne peuvent pas être déplacés, qui ne peuvent qu'être alloués dynamiquement ou qui ne peuvent pas être alloués dynamiquement. Chacun de ces idiomes possède des implémentations pré-C++11 qui connaissent des problèmes similaires, et qui sont résolus de façon identique en C++11 en les implémentant en termes de fonctions membres spéciales utilisées par défaut et supprimées.

Fonctions utilisées par défaut explicitement

Vous pouvez définir par défaut n'importe laquelle des fonctions membres spéciales : pour établir explicitement que la fonction membre spéciale utilise l'implémentation par défaut, pour définir la fonction membre spéciale avec un qualificateur d'accès non public ou pour rétablir une fonction membre spéciale dont la génération automatique a été empêchée par d'autres circonstances.

Vous définissez par défaut une fonction membre spéciale en la déclarant comme dans cet exemple :

C++

```
struct widget
{
  widget()=default;

  inline widget& operator=(const widget&);
};

inline widget& widget::operator=(const widget&) =default;
```

Notez que vous pouvez par défaut une fonction membre spéciale en dehors du corps d'une classe tant qu'elle est inlinable.

Étant donné les avantages des fonctions membres spéciales ordinaires en termes de performance, nous vous recommandons de préférer les fonctions membres spéciales générées automatiquement aux corps de fonction vides lorsque vous souhaitez rétablir le comportement par défaut. Vous pouvez effectuer cette opération en définissant

explicitement par défaut la fonction membre spéciale, ou en ne la déclarant pas (et en ne déclarant pas non plus d'autres fonctions membres spéciales qui l'empêcheraient d'être générée automatiquement.)

Fonctions supprimées

Vous pouvez supprimer des fonctions membres spéciales ainsi que des fonctions membres normales et des fonctions non-membres pour empêcher qu'elles ne soient définies ou appelées. La suppression de fonctions membres spéciales offre un moyen plus propre d'empêcher le compilateur de générer des fonctions membres spéciales que vous ne souhaitez pas. La fonction doit être supprimée lorsqu'elle est déclarée ; elle ne peut pas être supprimée après de la façon dont une fonction peut-être déclarée et utilisée ensuite par défaut.

C++

```
struct widget
{
  // deleted operator new prevents widget from being dynamically allocated.
  void* operator new(std::size_t) = delete;
};
```

La suppression des fonctions membres ou non-membres normales empêche les promotions de type problématiques de provoquer l'appel d'une fonction non voulue. Cela fonctionne, car les fonctions supprimées participent toujours à la résolution de surcharge et fournissent une meilleure correspondance que la fonction qui peut être appelée une fois que les types sont promus. L'appel de fonction résout la fonction la plus spécifique mais supprimée et provoque une erreur du compilateur.

C++

```
// deleted overload prevents call through type promotion of float to double
from succeeding.
void call_with_true_double_only(float) =delete;
void call_with_true_double_only(double param) { return; }
```

Notez dans l'exemple précédent que l'appel `call_with_true_double_only` à l'aide d'un **`float`** argument entraînerait une erreur du compilateur, mais que l'appel `call_with_true_double_only` à l'aide d'un **`int`** argument ne le ferait pas . Dans ce **`int`** cas, l'argument sera promu de **`int`** à **`double`** et appellera correctement la **`double`** version de la fonction, même si ce n'est peut-être pas ce qui est prévu. Pour vous assurer que tout appel à cette fonction à l'aide d'un argument non double provoque

une erreur du compilateur, vous pouvez déclarer une version de modèle de la fonction supprimée.

C++

```
template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type
promotion of any T to double from succeeding.

void call_with_true_double_only(double param) { return; } // also define for
const double, double&, etc. as needed.
```

Recherche de nom qui dépend de l'argument (Koenig) sur les fonctions

Article • 03/04/2023

Le compilateur peut utiliser la recherche de nom dépendante d'un argument pour rechercher la définition d'un appel de fonction non qualifié. La recherche de nom dépendante d'un argument est également appelée recherche Koenig. Le type de chaque argument dans un appel de fonction est défini dans une hiérarchie d'espaces de noms, de classes, de structures, d'unions ou de modèles. Lorsque vous spécifiez un appel de fonction postfix non qualifié, le compilateur recherche la définition de fonction dans la hiérarchie associée à chaque type d'argument.

Exemple

Dans l'exemple, le compilateur note que la fonction `f()` prend un argument `x`. L'argument `x` est de type `A::X`, qui est défini dans l'espace de noms `A`. Le compilateur recherche l'espace de noms `A` et trouve une définition pour la fonction `f()` qui prend un argument de type `A::X`.

C++

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
   struct X
   {
   };
   void f(const X&)
   {
   }
}
int main()
{
// The compiler finds A::f() in namespace A, which is where
// the type of argument x is defined. The type of x is A::X.
   A::X x;
   f(x);
}
```

Arguments par défaut

Article • 16/06/2023

Dans de nombreux cas, les fonctions ont des arguments utilisés si rarement qu'une valeur par défaut suffirait. La fonctionnalité d'argument par défaut permet de spécifier uniquement les arguments d'une fonction qui sont significatifs dans un appel donné. Pour illustrer ce concept, considérez l'exemple présenté dans Surcharge de fonction.

C++

```
// Prototype three print functions.
int print( char *s );                  // Print a string.
int print( double dvalue );            // Print a double.
int print( double dvalue, int prec );  // Print a double with a
//  given precision.
```

Dans de nombreuses applications, une valeur par défaut raisonnable peut être fournie pour `prec`, éliminant ainsi la nécessité de faire appel à deux fonctions :

C++

```
// Prototype two print functions.
int print( char *s );                    // Print a string.
int print( double dvalue, int prec=2 );  // Print a double with a
//  given precision.
```

L'implémentation de la `print` fonction est légèrement modifiée pour refléter le fait qu'une seule fonction de ce type existe pour le type **`double`**:

C++

```
// default_arguments.cpp
// compile with: /EHsc /c

// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.

#include <iostream>
#include <math.h>
using namespace std;

int print( double dvalue, int prec ) {
   // Use table-lookup for rounding/truncation.
```

```
   static const double rgPow10[] = {
      10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
         10E1,  10E2,  10E3,  10E4, 10E5,  10E6
   };
   const int iPowZero = 6;
   // If precision out of range, just print the number.
   if( prec >= -6 && prec <= 7 )
      // Scale, truncate, then rescale.
      dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
      rgPow10[iPowZero - prec];
   cout << dvalue << endl;
   return cout.good();
}
```

Pour appeler la nouvelle fonction `print`, utilisez du code comme semblable au suivant :

```
C++

print( d );    // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
//  results.
```

Tenez compte des points suivants lors de l'utilisation des arguments par défaut :

- Les arguments par défaut sont utilisés uniquement dans les appels de fonction où les arguments de fin sont omis. Il doit s'agir des derniers arguments. Par conséquent, le code suivant n'est pas valide :

  ```
  C++

  int print( double dvalue = 0.0, int prec );
  ```

- Un argument par défaut ne peut pas être redéfini dans des déclarations ultérieures, même si la redéfinition est identique à celle d'origine. Par conséquent, le code suivant génère une erreur :

  ```
  C++

  // Prototype for print function.
  int print( double dvalue, int prec = 2 );

  ...

  // Definition for print function.
  int print( double dvalue, int prec = 2 )
  {
  ...
  }
  ```

Le problème avec ce code est que la déclaration de fonction dans la définition redéfinit l'argument par défaut pour `prec`.

- Des arguments par défaut supplémentaires peuvent être ajoutés par des déclarations ultérieures.
- Des arguments par défaut peuvent être fournis pour des pointeurs vers des fonctions. Par exemple :

```
C++
int (*pShowIntVal)( int i = 0 );
```

Fonctions inline (C++)

Article • 03/04/2023

Le mot clé `inline` indique au compilateur de substituer le code dans la définition de fonction pour chaque instance d'un appel de fonction.

L'utilisation des fonctions inline permet accélérer l'exécution de votre programme, car celles-ci éliminent la surcharge associée aux appels de fonction. Le compilateur peut optimiser les fonctions développées inline de manière non disponible pour les fonctions normales.

La substitution de code inline se produit à la discrétion du compilateur. Par exemple, le compilateur n'inclut pas une fonction si son adresse est prise ou si elle trop volumineuse pour être incluse.

Une fonction définie dans le corps d'une déclaration de classe est implicitement une fonction inline.

Exemple

Dans la déclaration de classe suivante, le constructeur `Account` est une fonction inline. Les fonctions `GetBalance` membres , `Deposit` et `Withdraw` ne sont pas spécifiées en tant que `inline` mais peuvent être implémentées en tant que fonctions inline.

C++

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
```

```
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}
int main()
{
}
```

> ⓘ Notes
>
> Dans la déclaration de classe, les fonctions ont été déclarées sans le `inline` mot clé. Le `inline` mot clé peut être spécifié dans la déclaration de classe ; le résultat est le même.

Une fonction membre inline donnée doit être déclarée de la même manière dans chaque unité de compilation. Cette contrainte entraîne les fonctions inline à se comporter comme si elles étaient des fonctions instanciées. En outre, il doit exister une seule définition d'une fonction inline.

Une fonction membre de classe utilise par défaut une liaison externe, sauf si une définition de cette fonction contient le `inline` spécificateur. L'exemple précédent montre que vous n'avez pas besoin de déclarer ces fonctions explicitement avec le `inline` spécificateur. L'utilisation `inline` dans la définition de la fonction fait qu'il s'agit d'une fonction inline. Toutefois, vous ne pouvez pas redéfinir une fonction comme `inline` après un appel à cette fonction.

`inline`, `__inline` et `__forceinline`

Les `inline` spécificateurs et `__inline` indiquent au compilateur d'insérer une copie du corps de la fonction à chaque emplacement où la fonction est appelée.

L'insertion, appelée *extension inline* ou *inline*, se produit uniquement si l'analyse coûts-avantages du compilateur montre qu'elle en vaut la peine. L'extension inline réduit la surcharge d'appel de fonction au coût potentiel d'une plus grande taille de code.

Le `__forceinline` mot clé remplace l'analyse coûts-avantages et s'appuie plutôt sur le jugement du programmeur. Soyez prudent lors de l'utilisation de `__forceinline`. L'utilisation indiscriminée de `__forceinline` peut entraîner un code plus volumineux avec seulement des gains de performances marginaux ou, dans certains cas, des pertes

de performances (en raison de la pagination accrue d'un fichier exécutable plus volumineux, par exemple).

Le compilateur traite les options d'expansion inline et les mots clés comme des suggestions. Il n'y a aucune garantie que les fonctions seront incorporées. Vous ne pouvez pas forcer le compilateur à inliner une fonction particulière, même avec le `__forceinline` mot clé. Lorsque vous compilez avec `/clr`, le compilateur n'intègre pas une fonction si des attributs de sécurité sont appliqués à la fonction.

Pour la compatibilité avec les versions précédentes, `_inline` et `_forceinline` sont synonymes respectivement de `__inline` et `__forceinline`, sauf si l'option du compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Le `inline` mot clé indique au compilateur que l'extension inline est préférable. Toutefois, le compilateur peut créer une instance séparée de la fonction (instanciation) et créer des liaisons d'appel standard au lieu d'insérer le code inline. Deux cas dans lesquels ce comportement peut se produire sont :

- Fonctions récursives.
- Fonctions référencées par l'intermédiaire d'un pointeur ailleurs dans l'unité de traduction.

Ces motifs peuvent interférer avec l'inline, *comme d'autres*, à la discrétion du compilateur; vous ne devez pas dépendre du `inline` spécificateur pour provoquer l'inline d'une fonction.

Au lieu de développer une fonction inline définie dans un fichier d'en-tête, le compilateur peut la créer en tant que fonction pouvant être appelée dans plusieurs unités de traduction. Le compilateur marque la fonction générée pour l'éditeur de liens afin d'empêcher les violations d'une règle de définition (ODR).

Comme pour les fonctions normales, il n'existe aucun ordre défini pour l'évaluation des arguments dans une fonction inline. En fait, il peut être différent de l'ordre d'évaluation de l'argument lorsqu'il est passé à l'aide du protocole d'appel de fonction normal.

L'option /Ob d'optimisation du compilateur permet de déterminer si l'expansion de la fonction inline se produit réellement.

/LTCG inline plusieurs modules, qu'il soit demandé dans le code source ou non.

Exemple 1

```
C++
```

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
   if( a > b )
      return a;
   return b;
}
```

Les fonctions membres d'une classe peuvent être déclarées inline, soit à l'aide de la `inline` mot clé, soit en plaçant la définition de fonction dans la définition de classe.

Exemple 2

C++

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
   void print() { cout << i << ' '; }   // Implicitly inline
private:
   int i;
};
```

Spécifique à Microsoft

Le `__inline` mot clé équivaut à `inline`.

Même avec `__forceinline`, le compilateur ne peut pas inline code dans toutes les circonstances. Le compilateur ne peut pas inliner une fonction si :

- La fonction ou son appelant est compilé avec `/Ob0` (option par défaut pour les builds de débogage).
- La fonction et l'appelant utilisent différents types de gestion des exceptions (gestion des exceptions C++ dans l'un, gestion structurée des exceptions dans l'autre).
- La fonction a une liste d'arguments variable.
- La fonction utilise un assembly inline, sauf si elle est compilée avec `/Ox`, `/O1` ou `/O2`.

- La fonction est récursive et n'a `#pragma inline_recursion(on)` pas défini. Avec le pragma, les fonctions récursives sont incorporées à une profondeur par défaut de 16 appels. Pour réduire la profondeur de l'inlining, utilisez inline_depth pragma.
- La fonction est virtuelle et est appelée virtuellement. Les appels directs aux fonctions virtuelles peuvent être incorporés.
- Le programme prend l'adresse de la fonction et l'appel est effectué via le pointeur vers la fonction. Les appels directs aux fonctions dont l'adresse est acceptée peuvent être incorporés.
- La fonction est également marquée avec le naked__declspec modificateur.

Si le compilateur ne peut pas inline une fonction déclarée avec `__forceinline`, il génère un avertissement de niveau 1, sauf dans les cas suivants :

- La fonction est compilée à l'aide de /Od ou /Ob0. Aucune inlining n'est attendue dans ces cas.
- La fonction est définie en externe, dans une bibliothèque incluse ou une autre unité de traduction, ou est une cible d'appel virtuel ou une cible d'appel indirect. Le compilateur ne peut pas identifier le code non incorporé qu'il ne trouve pas dans l'unité de traduction actuelle.

Les fonctions récursives peuvent être remplacées par du code inline à une profondeur spécifiée par le inline_depth pragma, jusqu'à un maximum de 16 appels. Au-delà de cette profondeur, les appels de fonction récursive sont traités comme des appels à une instance de la fonction. La profondeur à laquelle les fonctions récursives sont examinées par l'heuristique inline ne peut pas dépasser 16. Le inline_recursion pragma contrôle l'expansion inline d'une fonction actuellement en cours d'expansion. Pour obtenir des informations connexes, consultez l'option du compilateur Extension de fonction inline (/Ob).

FIN de la section spécifique à Microsoft

Pour plus d'informations sur l'utilisation du `inline` spécificateur, consultez :

- Fonctions membres des classes inline
- Définition de fonctions C++ inline avec dllexport et dllimport

Quand utiliser les fonctions inline

Les fonctions inline sont utilisées de façon optimale pour les petites fonctions telles que l'accès aux membres de données privées. L'objectif main de ces fonctions « accesseur » à une ou deux lignes est de retourner des informations d'état sur les objets. Les fonctions courtes sont sensibles à la surcharge des appels de fonction. Les fonctions plus longues passent proportionnellement moins de temps dans la séquence d'appel et de retour et bénéficient moins de l'inlining.

Une `Point` classe peut être définie comme suit :

C++

```
// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    //  reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}
inline unsigned& Point::y()
{
    return _y;
}
int main()
{
}
```

En supposant que la manipulation de coordonnées est une opération relativement courante dans un client d'une telle classe, la spécification des deux fonctions d'accesseur (`x` et `y` dans l'exemple précédent) comme **`inline`** permet généralement d'économiser la surcharge sur :

- Appels de fonction (notamment le paramètre passe et place l'adresse de l'objet dans la pile)
- Conservation du frame de pile de l'appelant
- Nouvelle configuration du cadre de pile
- Communication de la valeur de retour

- Restauration de l'ancien cadre de pile
- Renvoie

Fonctions inline et macros

Les fonctions inline sont similaires aux macros, car le code de fonction est développé au point de l'appel au moment de la compilation. Toutefois, les fonctions inline sont analysées par le compilateur et les macros sont développées par le préprocesseur. Par conséquent, il existe plusieurs différences importantes :

- Les fonctions inline suivent tous les protocoles de sécurité de type appliqués sur les fonctions normales.
- Les fonctions inline sont spécifiées à l'aide de la même syntaxe que toute autre fonction, sauf qu'elles incluent le `inline` mot clé dans la déclaration de fonction.
- Les expressions passées comme arguments aux fonctions inline sont évaluées une fois. Dans certains cas, les expressions passées comme arguments aux macros peuvent être évaluées plusieurs fois.

L'exemple suivant montre une macro qui convertit les minuscules en majuscules :

C++

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
   char ch;
   printf_s("Enter a character: ");
   ch = toupper( getc(stdin) );
   printf_s( "%c", ch );
}
//  Sample Input:  xyz
// Sample Output:  Z
```

L'objectif de l'expression `toupper(getc(stdin))` est qu'un caractère doit être lu à partir de l'appareil console (`stdin`) et, si nécessaire, converti en majuscules.

En raison de l'implémentation de la macro, `getc` est exécuté une fois pour déterminer si le caractère est supérieur ou égal à « a », et une fois pour déterminer s'il est inférieur ou égal à « z ». S'il se trouve dans cette plage, `getc` est réexécuté pour convertir le

caractère en majuscules. Cela signifie que le programme attend deux ou trois caractères quand, dans l'idéal, il ne devrait attendre qu'un seul.

Les fonctions inline remédient au problème décrit auparavant :

C++

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
   return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );
}

int main() {
   printf_s("Enter a character: ");
   char ch = toupper( getc(stdin) );
   printf_s( "%c", ch );
}
```

Output

```
Sample Input: a
Sample Output: A
```

Voir aussi

noinline
auto_inline

Surcharge d'opérateur

Article • 16/06/2023

Le `operator` mot clé déclare une fonction spécifiant ce que signifie *opérateur-symbole* lorsqu'il est appliqué aux instances d'une classe. Cela donne à l'opérateur plusieurs significations ou le « surcharge ». Le compilateur fait la distinction entre les différentes significations d'un opérateur en examinant les types de ses opérandes.

Syntaxe

> *Type* `operator` *operator-symbol*(*parameter-list*)

Notes

Vous pouvez redéfinir la fonction de la plupart des opérateurs intégrés de manière globale ou classe par classe. Les opérateurs surchargés sont implémentés en tant que fonctions.

Le nom d'un opérateur surchargé est `operator`*x*, où *x* est l'opérateur tel qu'il apparaît dans le tableau suivant. Par exemple, pour surcharger l'opérateur d'ajout, vous définissez une fonction appelée **operator+**. De même, pour surcharger l'opérateur d'ajout/affectation, **+=**définissez une fonction appelée **operator+=**.

Opérateurs redéfinissables

Opérateur	Nom	Type
,	Comma	Binary
!	NOT logique	Unaire
!=	Inégalité	Binary
%	Modulo	Binary
%=	Assignation de modulo	Binary
&	ET au niveau du bit	Binary
&	Adresse-de	Unaire
&&	ET logique	Binary

Opérateur	Nom	Type
&=	Assignation d'opération AND au niveau du bit	Binary
( )	Appel de fonction	—
( )	Opérateur de conversion	Unaire
*	Multiplication	Binary
*	Déréférencement du pointeur	Unaire
*=	Assignation de multiplication	Binary
+	Addition	Binary
+	Plus unaire	Unaire
++	Incrémenter 1	Unaire
+=	Assignation d'addition	Binary
-	Soustraction	Binary
-	Négation unaire	Unaire
--	Décrémentation [1]	Unaire
-=	Assignation de soustraction	Binary
->	Sélection de membres	Binary
->*	Sélection de pointeur de membre	Binary
/	Division	Binary
/=	Assignation de division	Binary
<	Inférieur à	Binary
<<	Décalage vers la gauche	Binary
<<=	Assignation de décalage vers la gauche	Binary
<=	Inférieur ou égal à	Binary
=	Affectation	Binary
==	Égalité	Binary
>	Supérieur à	Binary
>=	Supérieur ou égal à	Binary

Opérateur	Nom	Type
>>	Décalage vers la droite	Binary
>>=	Assignation de décalage vers la droite	Binary
[ ]	Indice de tableau	—
^	OR exclusive	Binary
^=	Assignation OR exclusive	Binary
|	Opération OR inclusive au niveau du bit	Binary
|=	Assignation d'opération OR inclusive au niveau du bit	Binary
||	OU logique	Binary
~	Complément à 1	Unaire
`delete`	Supprimer	—
`new`	Nouveau	—
opérateurs de conversion	opérateurs de conversion	Unaire

[1] Il existe deux versions des opérateurs d'incrément et de décrémentation unaire : préincrement et postincrement.

Pour plus d'informations, consultez Règles générales relatives à la surcharge des opérateurs. Les contraintes sur les différentes catégories d'opérateurs surchargés sont décrites dans les rubriques suivantes :

- Opérateurs unaires
- Opérateurs binaires
- Cession
- Appel de fonction
- Indices
- Accès aux membres de la classe
- Incrémenter et décrémenter.
- Conversions de types définis par l'utilisateur

Les opérateurs indiqués dans le tableau suivant ne peuvent pas être surchargés. Le tableau inclut les symboles # de préprocesseur et ##.

Opérateurs non redéfinissables

Opérateur	Nom
.	Sélection de membres
`.*`	Sélection de pointeur de membre
::	Résolution de portée
?:	Logique conditionnelle
#	Préprocesseur convertir en type string
##	Préprocesseur concaténer

Bien que les opérateurs surchargés soient généralement appelés implicitement par le compilateur quand ils sont trouvés dans le code, ils peuvent être invoqués explicitement de la même façon que n'importe quelle fonction membre ou non membre :

C++

```
Point pt;
pt.operator+( 3 );  // Call addition operator to add 3 to pt.
```

Exemple

L'exemple suivant surcharge l'opérateur + pour ajouter deux nombres complexes et retourne le résultat.

C++

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};
```

```
// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display();
}
```

Output

```
6.8, 11.2
```

Contenu de cette section

- Règles générales pour la surcharge des opérateurs
- Surcharge des opérateurs unaires
- Opérateurs binaires
- Cession
- Appel de fonction
- Indices
- Accès aux membres

Voir aussi

Opérateurs intégrés, priorité et associativité C++
Mots clés

Règles générales de surcharge d'opérateur

Article • 03/04/2023

Les règles suivantes limitent le mode d'implémentation des opérateurs surchargés. Toutefois, elles ne s'appliquent pas aux opérateurs nouveaux et supprimés , qui sont couverts séparément.

- Vous ne pouvez pas définir de nouveaux opérateurs, tels que ..
- Vous ne pouvez pas redéfinir la signification d'opérateurs lorsqu'ils sont appliqués à des types de données intégrés.
- Les opérateurs surchargés doivent être soit une fonction de membre de classe non statique, soit une fonction globale. Une fonction globale qui nécessite un accès à des membres de classe privés ou protégés doit être déclarée comme amie de cette classe. Une fonction globale doit prendre au moins un argument qui est de type classe ou énuméré ou qui est une référence à un type énuméré ou classe. Par exemple :

  C++

  ```
  // rules_for_operator_overloading.cpp
  class Point
  {
  public:
      Point operator<( Point & );  // Declare a member operator
                                   //  overload.
      // Declare addition operators.
      friend Point operator+( Point&, int );
      friend Point operator+( int, Point& );
  };

  int main()
  {
  }
  ```

  L'exemple de code précédent déclare l'opérateur Inférieur à comme fonction membre ; toutefois, les opérateurs d'addition sont déclarés comme fonctions globales qui ont un accès ami. Notez que plusieurs implémentations peuvent être fournies pour un opérateur donné. Dans le cas de l'opérateur d'addition précédent, les deux implémentations sont fournies pour faciliter la commutativité. Il est tout aussi probable que les opérateurs qui ajoutent un `Point` à un `Point`, `int` à un `Point`, et ainsi de suite, puissent être implémentés.

- Les opérateurs obéissent aux règles de priorité, de regroupement et de nombre d'opérandes dictées par leur utilisation classique avec les types intégrés. Par conséquent, il n'existe aucun moyen d'exprimer le concept « ajouter 2 et 3 à un objet de type `Point` », en s'attendant à ce que 2 soit ajouté à la coordonnée *x* et 3 à la coordonnée *y* .

- Les opérateurs unaires déclarés comme fonctions membres n'acceptent pas d'argument ; s'ils sont déclarés comme fonctions globales, ils en prennent un.

- Les opérateurs binaires déclarés comme fonctions membres acceptent un argument ; s'ils sont déclarés comme fonctions globales, ils en prennent deux.

- Si un opérateur peut être utilisé comme opérateur unaire ou binaire (**&**, *****, **+**et **-**), vous pouvez surcharger chaque utilisation séparément.

- Les opérateurs surchargés ne peuvent pas avoir d'arguments par défaut.

- Tous les opérateurs surchargés à l'exception de **l'affectation (operator=**) sont hérités par les classes dérivées.

- Le premier argument pour les opérateurs surchargés déclarés comme fonctions membres est toujours le type de classe de l'objet pour lequel l'opérateur est appelé (la classe dans laquelle l'opérateur est déclaré ou une classe dérivée de cette classe). Aucune conversion n'est fournie pour le premier argument.

Notez que la signification de n'importe quel opérateur peut être modifiée complètement. Cela inclut la signification des opérateurs d'adresse de (**&**), d'affectation (=) et d'appel de fonction. En outre, les identités sur lesquelles on peut se reposer pour les types intégrés peuvent être modifiées à l'aide de la surcharge d'opérateur. Par exemple, les quatre instructions suivantes sont généralement équivalentes une fois leur évaluation terminée :

C++

```
var = var + 1;
var += 1;
var++;
++var;
```

On ne peut pas se reposer sur cette identité pour les types de classe qui surchargent des opérateurs. De plus, certaines des exigences implicites dans l'utilisation de ces opérateurs pour les types de base sont allégées pour les opérateurs surchargés. Par exemple, l'opérateur d'ajout/affectation, **+=**, exige que l'opérande gauche soit une

valeur l lorsqu'il est appliqué aux types de base ; il n'existe aucune exigence de ce type lorsque l'opérateur est surchargé.

> **Notes**
>
> Pour des raisons de cohérence, il est souvent préférable de suivre le modèle des types intégrés lors de la définition des opérateurs surchargés. Si la sémantique d'un opérateur surchargé diffère sensiblement de sa signification dans d'autres contextes, il peut être plus perturbant qu'utile.

Voir aussi

Surcharge d'opérateur

Surcharge des opérateurs unaires

Article • 03/04/2023

Les opérateurs unaires produisent un résultat à partir d'un opérande unique. Vous pouvez définir des surcharges d'un ensemble standard d'opérateurs unaires pour travailler sur des types définis par l'utilisateur.

Opérateurs unaires surchargés

Vous pouvez surcharger les opérateurs unaires suivants sur des types définis par l'utilisateur :

- `!` (not logique)
- `&` (adresse de)
- `~` (complément)
- `*` (déréférencement de pointeur)
- `+` (plus unaire)
- `-` (négation unaire)
- `++`(incrément de préfixe) ou (incrément de postfix)
- `--` (décrémentation du préfixe) ou (décrémentation postfixée)
- Opérateurs de conversion

Déclarations de surcharge d'opérateur unaire

Vous pouvez déclarer des opérateurs unaires surchargés en tant que fonctions membres non statiques ou en tant que fonctions non membres. Les fonctions membres unaires surchargées ne prennent aucun argument, car elles fonctionnent implicitement sur `this`. Les fonctions non membres sont déclarées avec un seul argument. Lorsque les deux formulaires sont déclarés, le compilateur suit les règles de résolution de surcharge pour déterminer la fonction à utiliser, le cas échéant.

Les règles suivantes s'appliquent à tous les opérateurs unaires de préfixe. Pour déclarer une fonction d'opérateur unaire en tant que fonction membre non statique, utilisez ce formulaire de déclaration :

```
return-type operator op ();
```

Dans ce formulaire, `return-type` est le type de retour et `op` est l'un des opérateurs répertoriés dans le tableau précédent.

Pour déclarer une fonction d'opérateur unaire en tant que fonction non membre, utilisez ce formulaire de déclaration :

```
return-type operator op ( class-type );
```

Dans cette forme, `return-type` est le type de retour, `op` est l'un des opérateurs répertoriés dans le tableau précédent et `class-type` est le type de classe de l'argument sur lequel opérer.

Les formes postfixes de `++` et `--` prennent un argument supplémentaire `int` pour les distinguer des formes de préfixe. Pour plus d'informations sur les formes préfixe et postfix de `++` et `--`, consultez Surcharge des opérateurs incrémenter et décrémenter.

> ⓘ **Notes**
>
> Il n'existe aucune restriction sur les types de retour des opérateurs unaires. Par exemple, il est logique que la logique NOT (`!`) retourne une `bool` valeur, mais ce comportement n'est pas appliqué.

Voir aussi

Surcharge d'opérateur

Surcharge d'opérateurs d'incrémentation et de décrémentation (C++)

Article • 03/04/2023

Les opérateurs d'incrémentation et de décrémentation appartiennent à une catégorie spéciale car chacun comporte deux variantes :

- Incrémentation préfixée et incrémentation suffixée
- Décrémentation préfixée et décrémentation suffixée

Lorsque vous écrivez des fonctions d'opérateur surchargées, il peut être utile d'implémenter des versions distinctes des versions préfixées et suffixées de ces opérateurs. Pour faire la distinction entre les deux, la règle suivante est observée : la forme de préfixe de l'opérateur est déclarée exactement de la même façon que tout autre opérateur unaire ; le formulaire postfix accepte un argument supplémentaire de type `int`.

> ⓘ **Notes**
>
> Lorsque vous spécifiez un opérateur surchargé pour la forme postfixée de l'opérateur d'incrémentation ou de décrémentation, l'argument supplémentaire doit être de type `int`; la spécification de tout autre type génère une erreur.

L'exemple suivant indique comment définir les opérateurs d'incrémentation et de décrémentation préfixés et suffixés pour la classe `Point` :

C++

```
// increment_and_decrement1.cpp
class Point
{
public:
   // Declare prefix and postfix increment operators.
   Point& operator++();       // Prefix increment operator.
   Point operator++(int);     // Postfix increment operator.

   // Declare prefix and postfix decrement operators.
   Point& operator--();       // Prefix decrement operator.
   Point operator--(int);     // Postfix decrement operator.

   // Define default constructor.
```

```
   Point() { _x = _y = 0; }

   // Define accessor functions.
   int x() { return _x; }
   int y() { return _y; }
private:
   int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
   _x++;
   _y++;
   return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
   Point temp = *this;
   ++*this;
   return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
   _x--;
   _y--;
   return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
   Point temp = *this;
   --*this;
   return temp;
}

int main()
{
}
```

Les mêmes opérateurs peuvent être définis dans l'étendue de fichier (globalement) à l'aide des prototypes de fonction suivants :

C++

```
friend Point& operator++( Point& );      // Prefix increment
friend Point operator++( Point&, int );  // Postfix increment
```

```
friend Point& operator--( Point& );      // Prefix decrement
friend Point operator--( Point&, int );  // Postfix decrement
```

L'argument de type `int` qui indique la forme postfixée de l'opérateur d'incrémentation ou de décrémentation n'est pas couramment utilisé pour passer des arguments. Il contient généralement la valeur 0. Toutefois, il peut être utilisé comme suit :

C++

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int operator++( int n ); // Postfix increment operator
private:
    int _i;
};

Int Int::operator++( int n )
{
    Int result = *this;
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;       // Handle case where no argument is passed.
    return result;
}

int main()
{
   Int i;
   i.operator++( 25 ); // Increment by 25.
}
```

Il n'existe aucune syntaxe permettant d'utiliser les opérateurs d'incrémentation ou de décrémentation pour passer ces valeurs autres que l'appel explicite, comme indiqué dans le code précédent. Un moyen plus simple d'implémenter cette fonctionnalité consiste à surcharger l'opérateur d'ajout/affectation (+=).

Voir aussi

Surcharge d'opérateur

Opérateurs binaires

Article • 16/06/2023

Le tableau suivant affiche une liste des opérateurs qui peuvent être surchargés.

Opérateurs binaires redéfinissables

Opérateur	Nom
,	Comma
!=	Inégalité
%	Modulo
%=	Modulo/assignation
&	ET au niveau du bit
&&	ET logique
&=	Opération de bits AND/assignation
`*`	Multiplication
`*=`	Multiplication/assignation
+	Addition
+=	Addition/assignation
-	Soustraction
-=	Soustraction/assignation
->	Sélection de membres
`->*`	Sélection de pointeur de membre
/	Division
/=	Division/assignation
<	Inférieur à
<<	Décalage vers la gauche
<<=	Décalage vers la gauche/assignation

Opérateur	Nom
<=	Inférieur ou égal à
=	Affectation
==	Égalité
>	Supérieur à
>=	Supérieur ou égal à
>>	Décalage vers la droite
>>=	Décalage vers la droite/assignation
^	OR exclusive
^=	OR exclusive/assignation
|	Opération OR inclusive au niveau du bit
|=	OR inclusive au niveau du bit/assignation
||	OU logique

Pour déclarer une fonction d'opérateur binaire en tant que membre non statique, vous devez la déclarer comme suit :

> *ret-type* `operator` *op*(*arg*)

où *ret-type* est le type de retour, *op* est l'un des opérateurs répertoriés dans le tableau précédent et *arg* est un argument de n'importe quel type.

Pour déclarer une fonction d'opérateur binaire en tant que fonction globale, vous devez la déclarer comme suit :

> *ret-type* `operator` *op*(*arg1*,*arg2*)

où *ret-type* et *op* sont décrits pour les fonctions d'opérateur membre et *arg1* et *arg2* sont des arguments. Au moins un des arguments doit être de type classe.

> ⓘ **Notes**
>
> Il n'existe aucune restriction sur les types de retour des opérateurs binaires. En revanche, la plupart des opérateurs binaires définis par l'utilisateur retournent un type de classe ou une référence à un type de classe.

Voir aussi

Surcharge d'opérateur

Affectation

Article • 03/04/2023

L'opérateur d'assignation (=) est, à proprement parler, un opérateur binaire. Sa déclaration est identique à celle de tout autre opérateur binaire, avec les exceptions suivantes :

- Il doit s'agir d'une fonction membre non statique. Aucun **opérateur=** ne peut être déclaré en tant que fonction non membre.
- Il n'est pas hérité par les classes dérivées.
- Une fonction **operator=** par défaut peut être générée par le compilateur pour les types de classes, si aucun n'existe.

L'exemple suivant montre comment déclarer un opérateur d'assignation :

C++

```
class Point
{
public:
    int _x, _y;

    // Right side of copy assignment is the argument.
    Point& operator=(const Point&);
};

// Define copy assignment operator.
Point& Point::operator=(const Point& otherPoint)
{
    _x = otherPoint._x;
    _y = otherPoint._y;

    // Assignment operator returns left side of assignment.
    return *this;
}

int main()
{
    Point pt1, pt2;
    pt1 = pt2;
}
```

L'argument fourni est le côté droit de l'expression. L'opérateur retourne l'objet pour conserver le comportement de l'opérateur d'assignation, qui retourne la valeur du côté gauche une fois l'assignation terminée. Cela permet le chaînage des affectations, telles que :

C++

```
pt1 = pt2 = pt3;
```

L'opérateur d'assignation de copie ne doit pas être confondu avec le constructeur de copie. Ce dernier est appelé lors de la construction d'un nouvel objet à partir d'un objet existant :

C++

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```

> **Notes**
>
> Il est recommandé de suivre la **règle de trois** selon laquelle une classe qui définit un opérateur d'assignation de copie doit également définir explicitement le constructeur de copie, le destructeur et, à compter de C++11, déplacer le constructeur et l'opérateur d'assignation de déplacement.

Voir aussi

- Surcharge d'opérateur
- Constructeurs de copie et opérateurs d'assignation de copie (C++)

Appel de fonction (C++)

Article • 16/06/2023

L'opérateur d'appel de fonction, invoqué à l'aide de parenthèses, est un opérateur binaire.

Syntaxe

```
primary-expression ( expression-list )
```

Notes

Dans ce contexte, `primary-expression` est le premier opérande et `expression-list`, une liste d'arguments éventuellement vide, est le deuxième opérande. L'opérateur d'appel de fonction est utilisé pour les opérations nécessitant un certain nombre de paramètres. Cela fonctionne car `expression-list` est une liste et non un opérande unique. L'opérateur d'appel de fonction doit être une fonction membre non statique.

L'opérateur d'appel de fonction, quand il est surchargé, ne modifie pas la façon dont les fonctions sont appelées, mais la façon dont l'opérateur doit être interprété quand il est appliqué aux objets d'un type de classe donné. Par exemple, le code suivant n'a généralement pas de signification :

C++

```
Point pt;
pt( 3, 2 );
```

Toutefois, pour un opérateur d'appel de fonction surchargé approprié donné, cette syntaxe peut être utilisée pour décaler de 3 unités la coordonnée `x` et de 2 unités la coordonnée `y`. Le code suivant illustre ce type de définition :

C++

```
// function_call.cpp
class Point
{
public:
    Point() { _x = _y = 0; }
```

```
    Point &operator()( int dx, int dy )
        { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
};

int main()
{
   Point pt;
   pt( 3, 2 );
}
```

Notez que l'opérateur d'appel de fonction est appliqué au nom d'un objet et non à celui d'une fonction.

Vous pouvez aussi surcharger l'opérateur d'appel de fonction à l'aide d'un pointeur vers une fonction (et non la fonction elle-même).

C++

```
typedef void(*ptf)();
void func()
{
}
struct S
{
   operator ptf()
   {
      return func;
   }
};

int main()
{
   S s;
   s();//operates as s.operator ptf()()
}
```

Voir aussi

Surcharge d'opérateur

Indices

Article • 16/06/2023

L'opérateur d'indice ([ ]), comme l'opérateur d'appel de fonction, est considéré comme un opérateur binaire. L'opérateur d'indice doit être une fonction membre non statique qui accepte un seul argument. Cet argument peut être de tout type et désigne l'indice de tableau souhaité.

Exemple

L'exemple suivant montre comment créer un vecteur de type `int` qui implémente la vérification des limites :

C++

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
   IntVector( int cElements );
   ~IntVector() { delete [] _iElements; }
   int& operator[](int nSubscript);
private:
   int *_iElements;
   int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
   _iElements = new int[cElements];
   _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
   static int iErr = -1;

   if( nSubscript >= 0 && nSubscript < _iUpperBound )
      return _iElements[nSubscript];
   else {
      clog << "Array bounds violation." << endl;
      return iErr;
   }
}
```

```
// Test the IntVector class.
int main() {
   IntVector v( 10 );
   int i;

   for( i = 0; i <= 10; ++i )
      v[i] = i;

   v[3] = v[9];

   for ( i = 0; i <= 10; ++i )
      cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

Output

```
Array bounds violation.
Element: [0] = 0
Element: [1] = 1
Element: [2] = 2
Element: [3] = 9
Element: [4] = 4
Element: [5] = 5
Element: [6] = 6
Element: [7] = 7
Element: [8] = 8
Element: [9] = 9
Array bounds violation.
Element: [10] = 10
```

Commentaires

Quand `i` atteint 10 dans le programme précédent, **operator[]** détecte qu'un indice hors limites est utilisé et émet un message d'erreur.

Notez que l'opérateur de fonction **[]** retourne un type référence. Il s'agit par conséquent d'une l-value, ce qui vous permet d'utiliser des expressions indicées de chaque côté des opérateurs d'assignation.

Voir aussi

Surcharge d'opérateur

Accès aux membres

Article • 16/06/2023

L’accès aux membres de classe peut être contrôlé en surchargeant l’opérateur d’accès membre (->). Cet opérateur est considéré comme un opérateur unaire dans cette utilisation, et la fonction surchargée d'opérateur doit être une fonction membre de classe. Par conséquent, la déclaration pour une telle fonction est :

Syntaxe

```
class-type *operator->()
```

Notes

où *class-type* est le nom de la classe à laquelle cet opérateur appartient. La fonction opérateur d'accès au membre doit être une fonction membre non statique.

Cet opérateur est utilisé (souvent avec l'opérateur pointer-dereference) pour implémenter des « pointeurs intelligents » qui valident les pointeurs avant de déréférencer ou de compter l'utilisation.

L’opérateur d’accès membre . ne peut pas être surchargé.

Voir aussi

Surcharge d'opérateur

Classes et structs (C++)

Article • 03/04/2023

Cette section présente les classes et structs C++. Les deux constructions sont identiques en C++ sauf que l'accessibilité par défaut est publique dans les structs, alors qu'elle est privée dans les classes.

Les classes et structs sont les constructions par lesquels vous définissez vos propres types. Les classes et structs peuvent tous deux contenir des données membres et des fonctions membres, qui vous permettent de décrire l'état et le comportement du type.

Cet article contient les rubriques suivantes :

- class
- struct
- Vue d'ensemble du membre de classe
- Access Control membres
- Héritage
- Membres statiques
- Conversions de types définis par l'utilisateur
- Membres de données mutables (spécificateur mutable)
- Déclarations de classes imbriquées
- Types de classes anonymes
- Pointeurs vers les membres
- ce pointeur
- Champs de bits C++

Les trois types de classes sont la structure, la classe et l'union. Ils sont déclarés à l'aide des mots clés struct, class et union . Le tableau suivant montre les différences entre les trois types de classes.

Pour plus d'informations sur les syndicats, consultez Unions. Pour plus d'informations sur les classes et les structs dans C++/CLI et C++/CX, consultez Classes et Structs.

Contrôle d'accès et contraintes de structures, de classes et d'unions

Structures	Classes	Unions
la clé de la classe est `struct`	la clé de la classe est `class`	la clé de la classe est `union`
L'accès par défaut est public	L'accès par défaut est privé	L'accès par défaut est public
Aucune contrainte d'utilisation	Aucune contrainte d'utilisation	Utilisation d'un seul membre à la fois

Voir aussi

Informations de référence sur le langage C++

class (C++)

Article • 16/06/2023

Le `class` mot clé déclare un type de classe ou définit un objet d'un type de classe.

Syntaxe

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
   member-list
} [declarators];
[ class ] tag declarators;
```

Paramètres

template-spec
Spécifications de modèle facultatives. Pour plus d'informations, consultez Modèles.

class
Le mot clé `class`.

ms-decl-spec
Spécification de classe de stockage facultative. Pour plus d'informations, reportez-vous au mot clé __declspec.

étiquette
Nom de type donné à la classe. La balise devient un mot réservé dans l'étendue de la classe. La balise est facultative. En cas d'omission, une classe anonyme est définie. Pour plus d'informations, consultez Types de classes anonymes.

base-list
Liste facultative de classes ou de structures dont cette classe dérivera ses membres. Pour plus d'informations, consultez Classes de base . Chaque nom de classe ou de structure de base peut être précédé d'un spécificateur d'accès (public, privé, protégé) et du mot clé virtuel. Pour plus d'informations, consultez le tableau accès aux membres dans Contrôle de l'accès aux membres de la classe .

member-list
Liste des membres de la classe. Pour plus d'informations, consultez Vue d'ensemble des

membres de la classe.

declarators
Liste de déclarateurs spécifiant les noms d'une ou plusieurs instances du type de classe. Les déclarateurs peuvent inclure des listes d'initialiseurs si tous les membres de données de la classe sont `public`. Cela est plus courant dans les structures, dont les membres de données sont `public` par défaut, que dans les classes. Pour plus d'informations, consultez Vue d'ensemble des déclarateurs .

Notes

Pour plus d'informations sur les classes en général, reportez-vous à l'une des rubriques suivantes :

- struct
- union
- __multiple_inheritance
- __single_inheritance
- __virtual_inheritance

Pour plus d'informations sur les classes et les structs managés dans C++/CLI et C++/CX, consultez Classes et Structs

Exemple

C++

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
using namespace std;

class dog
{
public:
   dog()
   {
      _legs = 4;
```

```
      _bark = true;
   }

   void setDogSize(string dogSize)
   {
      _dogSize = dogSize;
   }
   virtual void setEars(string type)      // virtual function
   {
      _earType = type;
   }

private:
   string _dogSize, _earType;
   int _legs;
   bool _bark;

};

class breed : public dog
{
public:
   breed( string color, string size)
   {
      _color = color;
      setDogSize(size);
   }

   string getColor()
   {
      return _color;
   }

   // virtual function redefined
   void setEars(string length, string type)
   {
      _earLength = length;
      _earType = type;
   }

protected:
   string _color, _earLength, _earType;
};

int main()
{
   dog mongrel;
   breed labrador("yellow", "large");
   mongrel.setEars("pointy");
   labrador.setEars("long", "floppy");
   cout << "Cody is a " << labrador.getColor() << " labrador" << endl;
}
```

Voir aussi

Mots clés

Classes et structs

struct (C++)

Article • 16/06/2023

Le `struct` mot clé définit un type de structure et/ou une variable d'un type de structure.

Syntaxe

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
   member-list
} [declarators];
[struct] tag declarators;
```

Paramètres

template-spec
Spécifications de modèle facultatives. Pour plus d'informations, consultez Spécifications de modèle.

struct
Le mot clé `struct`.

ms-decl-spec
Spécification de classe de stockage facultative. Pour plus d'informations, reportez-vous au mot clé __declspec.

étiquette
Nom de type donné à la structure. La balise devient un mot réservé dans la portée de la structure. La balise est facultative. Si elle est omise, une structure anonyme est définie. Pour plus d'informations, consultez Types de classes anonymes.

base-list
Liste facultative des classes ou structures dont dérivent les membres de cette structure. Pour plus d'informations, consultez Classes de base . Chaque nom de classe ou de structure de base peut être précédé d'un spécificateur d'accès (public, privé, protégé) et du mot clé virtuel. Pour plus d'informations, consultez le tableau accès aux membres dans Contrôle de l'accès aux membres de la classe .

member-list
Liste des membres de structure. Pour plus d'informations, consultez Vue d'ensemble des membres de la classe. La seule différence ici est que `struct` est utilisé à la place de `class`.

declarators
Liste de déclarateurs spécifiant les noms de la structure. Les listes des déclarateurs déclarent une ou plusieurs instances du type structure. Les déclarateurs peuvent inclure des listes d'initialiseurs si tous les membres de données de la structure sont `public`. Les listes d'initialiseurs sont courantes dans les structures, car les membres de données le sont `public` par défaut. Pour plus d'informations, consultez Vue d'ensemble des déclarateurs .

Notes

Un type structure est un type composite défini par l'utilisateur. Il comprend des champs ou des membres qui peuvent avoir différents types.

En C++, une structure est identique à une classe, sauf que ses membres sont `public` par défaut.

Pour plus d'informations sur les classes managées et les structs dans C++/CLI, consultez Classes et Structs.

Utilisation d'une structure

Dans C, vous devez utiliser explicitement le `struct` mot clé pour déclarer une structure. En C++, vous n'avez pas besoin d'utiliser le `struct` mot clé une fois le type défini.

Vous avez la possibilité de déclarer des variables lorsque le type de structure est défini en plaçant un ou plusieurs noms de variables séparés par des virgules entre l'accolade fermante et le point-virgule.

Les variables de structure peuvent être initialisées. L'initialisation de chaque variable doit être placée entre accolades.

Pour plus d'informations, consultez classe, union et énumération.

Exemple

```
C++
```

```cpp
#include <iostream>
using namespace std;

struct PERSON {   // Declare PERSON struct type
    int age;   // Declare member types
    long ss;
    float weight;
    char name[25];
} family_member;   // Define object of type PERSON

struct CELL {   // Declare CELL bit field
    unsigned short character  : 8;  // 00000000 ????????
    unsigned short foreground : 3;  // 00000??? 00000000
    unsigned short intensity  : 1;  // 0000?000 00000000
    unsigned short background : 3;  // 0???0000 00000000
    unsigned short blink      : 1;  // ?0000000 00000000
} screen[25][80];       // Array of bit fields

int main() {
    struct PERSON sister;   // C style structure declaration
    PERSON brother;   // C++ style structure declaration
    sister.age = 13;   // assign values to members
    brother.age = 7;
    cout << "sister.age = " << sister.age << '\n';
    cout << "brother.age = " << brother.age << '\n';

    CELL my_cell;
    my_cell.character = 1;
    cout << "my_cell.character = " << my_cell.character;
}
// Output:
// sister.age = 13
// brother.age = 7
// my_cell.character = 1
```

Vue d’ensemble des membres de classe

Article • 16/06/2023

ou `class` `struct` se compose de ses membres. Le travail exécuté par une classe est effectué par ses fonctions membres. L'état géré est stocké dans ses données membres. L’initialisation des membres est effectuée par des constructeurs, et le travail de nettoyage tel que la libération de la mémoire et la libération des ressources est effectué par les destructeurs. En C++11 et versions ultérieures, les données membres peuvent (et généralement doivent) être initialisées au point de déclaration.

Genres de membres de classe

La liste complète des catégories de membres est la suivante :

- Fonctions membres spéciales.
- Vue d’ensemble des fonctions membres.
- Membres de données mutables et statiques, y compris les types intégrés et d’autres types définis par l’utilisateur.
- Opérateurs
- Déclarations de classe imbriquées et.)
- Unions
- Énumérations.
- Champs de bits.
- Amis.
- Alias et typedefs.

> ⓘ **Notes**
>
> Les friends sont inclus dans la liste précédente parce qu'ils figurent dans la déclaration de classe. Toutefois, ils ne sont pas des membres de classe True, car ils ne sont pas dans la portée de la classe.

Exemple de déclaration de classe

L'exemple suivant illustre une déclaration de classe simple :

C++

```
// TestRun.h

class TestRun
{
    // Start member list.

    // The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
    TestRun(const TestRun&) = delete;
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun();
    enum class State { Active, Suspended };

    // Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

    // Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };
```

Accessibilité des membres

Les membres d'une classe sont déclarés dans la liste des membres. La liste des membres d'une classe peut être divisée en un nombre quelconque de sections et `public` à l'aide de `protected` `private`mots clés appelés spécificateurs d'accès. Un signe deux-points `:` doit suivre le spécificateur d'accès. Ces sections n'ont pas besoin d'être contiguës ; autrement dit, l'un de ces mots clés peut apparaître plusieurs fois dans la

liste des membres. Le mot clé désigne l'accès de tous les membres jusqu'au spécificateur d'accès suivant ou l'accolade fermante. Pour plus d'informations, consultez Contrôle d'accès aux membres (C++).

Membres static

Des données membres peuvent être déclarées comme statiques, ce qui signifie que tous les objets de la classe ont accès à la même copie de celles-ci. Une fonction membre peut être déclarée statique, auquel cas elle peut uniquement accéder aux membres de données statiques de la classe (et n'a pas `this` de pointeur). Pour plus d'informations, consultez Membres de données statiques.

Fonctions membres spéciales

Les fonctions membres spéciales sont des fonctions que le compilateur fournit automatiquement si vous ne les spécifiez pas dans votre code source.

- Constructeur par défaut
- Constructeur de copie
- **(C++11)** Déplacer un constructeur
- Opérateur d'assignation de copie
- **(C++11)** Opérateur d'affectation de déplacement
- Destructeur

Pour plus d'informations, consultez Fonctions membres spéciales.

Initialisation au niveau des membres

En C++11 et versions ultérieures, les déclarateurs de membres non statiques peuvent contenir des initialiseurs.

C++

```
class CanInit
{
public:
    long num {7};       // OK in C++11
    int k = 9;          // OK in C++11
    static int i = 9; // Error: must be defined and initialized
```

```
                        // outside of class declaration.

    // initializes num to 7 and k to 9
    CanInit(){}

    // overwrites original initialized value of num:
    CanInit(int val) : num(val) {}
};
int main()
{
}
```

Si une valeur est affectée à un membre dans un constructeur, cette valeur remplace la valeur affectée lors de la déclaration.

Il n'existe qu'une seule copie partagée des membres de données statiques pour tous les objets d'un type de classe donné. Les membres de données statiques doivent être définies et peuvent être initialisées au niveau de la portée du fichier. Pour plus d'informations sur les membres de données statiques, consultez Membres de données statiques. L'exemple suivant montre comment initialiser des membres de données statiques :

C++

```
// class_members2.cpp
class CanInit2
{
public:
    CanInit2() {} // Initializes num to 7 when new objects of type
                  //  CanInit are created.
    long     num {7};
    static int i;
    static int j;
};

// At file scope:

// i is defined at file scope and initialized to 15.
// The initializer is evaluated in the scope of CanInit.
int CanInit2::i = 15;

// The right side of the initializer is in the scope
// of the object being initialized
int CanInit2::j = i;
```

ⓘ Notes

Le nom de classe, `CanInit2`, doit précéder `i` pour spécifier que le `i` défini est un membre de classe `CanInit2`.

Voir aussi

Classes et structs

Contrôle d'accès aux membres (C++)

Article • 06/04/2023

Les contrôles d'accès vous permettent de séparer l'interface public d'une classe des détails de l'implémentation private et des protected membres destinés uniquement aux classes dérivées. Le spécificateur d'accès s'applique à tous les membres déclarés après lui jusqu'à ce que le spécificateur d'accès suivant soit rencontré.

C++

```
class Point
{
public:
    Point( int, int ) // Declare public constructor.;
    Point();// Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:                // Declare private state variables.
    int _x;
    int _y;

protected:      // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

L'accès par défaut est `private` dans une classe et `public` dans un struct ou une union. Les spécificateurs d'accès dans une classe peuvent être utilisés autant de fois que nécessaire dans n'importe quel ordre. L'allocation du stockage pour les objets de types de classes dépend de l'implémentation. Toutefois, les compilateurs doivent garantir l'attribution des membres à des adresses mémoire successivement plus élevées entre les spécificateurs d'accès.

Contrôle d'accès aux membres

Type d'accès	Signification
private	Les membres de classe déclarés comme ne peuvent être utilisés que `private` par les fonctions membres et les amis (classes ou fonctions) de la classe.
protected	Les membres de classe déclarés comme `protected` peuvent être utilisés par les fonctions membres et les amis (classes ou fonctions) de la classe. En outre, ils peuvent être utilisés par des classes dérivées de la classe.

Type d'accès	Signification
public	Les membres de classe déclarés comme `public` peuvent être utilisés par n'importe quelle fonction.

Le contrôle d'accès vous empêche d'utiliser des objets d'une manière qu'ils n'étaient pas destinées à être utilisées. Cette protection est perdue lorsque vous effectuez des conversions de type explicites (casts).

> ⓘ Notes
>
> Le contrôle d'accès s'applique également à tous les noms : fonctions membres, données membres, classes imbriquées et énumérateurs.

Contrôle d'accès dans les classes dérivées

Deux facteurs contrôlent les membres d'une classe de base qui sont accessibles dans une classe dérivée. Ces mêmes facteurs contrôlent l'accès aux membres hérités dans la classe dérivée :

- Indique si la classe dérivée déclare la classe de base à l'aide du spécificateur d'accès `public` .
- Quel est l'accès au membre de la classe de base.

Le tableau suivant montre l'interaction entre ces facteurs et la procédure de détermination de l'accès au membre de la classe de base.

Accès au membre de la classe de base

`private`	`protected`	`public`
Toujours inaccessible avec n'importe quel accès de dérivation	`private` dans la classe dérivée si vous utilisez `private` la dérivation	`private` dans la classe dérivée si vous utilisez `private` la dérivation
	`protected` dans la classe dérivée si vous utilisez `protected` la dérivation	`protected` dans la classe dérivée si vous utilisez `protected` la dérivation

`private`	`protected`	`public`
	`protected` dans la classe dérivée si vous utilisez `public` la dérivation	`public` dans la classe dérivée si vous utilisez `public` la dérivation

L'exemple suivant illustre la dérivation d'accès :

C++

```
// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
}
```

Dans `DerivedClass1`, la fonction `PublicFunc` membre est un **`public`** membre et `ProtectedFunc` est membre **`protected`** , car `BaseClass` est une **`public`** classe de base. `PrivateFunc` est **`private`** à `BaseClass`, et il est inaccessible à toutes les classes dérivées.

Dans `DerivedClass2`, les fonctions `PublicFunc` et `ProtectedFunc` sont considérés comme **`private`** membres, car est `BaseClass` une **`private`** classe de base. Là encore, `PrivateFunc` est **`private`** à `BaseClass`, et il est inaccessible à toutes les classes dérivées.

Vous pouvez déclarer une classe dérivée sans spécificateur d'accès de classe de base. Dans ce cas, la dérivation est prise en compte **`private`** si la déclaration de classe dérivée utilise le **`class`** mot clé. La dérivation est prise en compte **`public`** si la déclaration de classe dérivée utilise le **`struct`** mot clé. Par exemple, le code suivant :

```
C++

class Derived : Base
...
```

équivaut à :

```
C++

class Derived : private Base
...
```

De même, le code suivant :

```
C++

struct Derived : Base
...
```

équivaut à :

```
C++

struct Derived : public Base
...
```

Les membres déclarés comme ayant **`private`** accès ne sont pas accessibles aux fonctions ou classes dérivées, sauf si ces fonctions ou classes sont déclarées à l'aide de la **`friend`** déclaration dans la classe de base.

Un `union` type ne peut pas avoir de classe de base.

> ⓘ Notes
>
> Lors de la spécification d'une private classe de base, il est conseillé d'utiliser explicitement le `private` mot clé afin que les utilisateurs de la classe dérivée comprennent l'accès membre.

Contrôle d'accès et membres statiques

Lorsque vous spécifiez une classe de base comme `private`, elle affecte uniquement les membres non statiques. Les membres publics static sont toujours accessibles dans les classes dérivées. Toutefois, l'accès aux membres de la classe de base à l'aide de pointeurs, de références ou d'objets peut nécessiter une conversion, qui applique à nouveau le contrôle d'accès. Prenons l'exemple suivant :

C++

```
// access_control.cpp
class Base
{
public:
    int Print();             // Nonstatic member.
    static int CountOf();    // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};

// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();    // Nonstatic member.
};

// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
    // Call static member function CountOf explicitly.
    int cCount = ::Base::CountOf();     // OK.

    // Call static member function CountOf using pointer.
    cCount = this->CountOf();  // C2247: 'Base::CountOf'
                               // not accessible because
                               // 'Derived1' uses 'private'
                               // to inherit from 'Base'
```

```
    return cCount;
}
```

Dans le code précédent, le contrôle d'accès interdit la conversion d'un pointeur vers `Derived2` en un pointeur vers `Base`. Le **`this`** pointeur est implicitement de type `Derived2 *`. Pour sélectionner la `CountOf` fonction, **`this`** doit être converti en type `Base *`. Une telle conversion n'est pas autorisée, car `Base` est une private classe de base indirecte en `Derived2`. La conversion en un private type de classe de base est acceptable uniquement pour les pointeurs vers des classes dérivées immédiates. C'est pourquoi les pointeurs de type `Derived1 *` peuvent être convertis en type `Base *`.

Un appel explicite à la `CountOf` fonction, sans utiliser de pointeur, de référence ou d'objet pour la sélectionner, n'implique aucune conversion. C'est pourquoi l'appel est autorisé.

Les membres et amis d'une classe dérivée, `T`, peuvent convertir un pointeur vers `T` un pointeur vers une private classe de base directe de `T`.

Accès aux fonctions virtuelles

Le contrôle d'accès appliqué aux virtual fonctions est déterminé par le type utilisé pour effectuer l'appel de fonction. La substitution de déclarations de la fonction n'affecte pas le contrôle d'accès pour un type donné. Par exemple :

C++

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
   VFuncDerived vfd;             // Object of derived type.
   VFuncBase *pvfb = &vfd;       // Pointer to base type.
   VFuncDerived *pvfd = &vfd;    // Pointer to derived type.
```

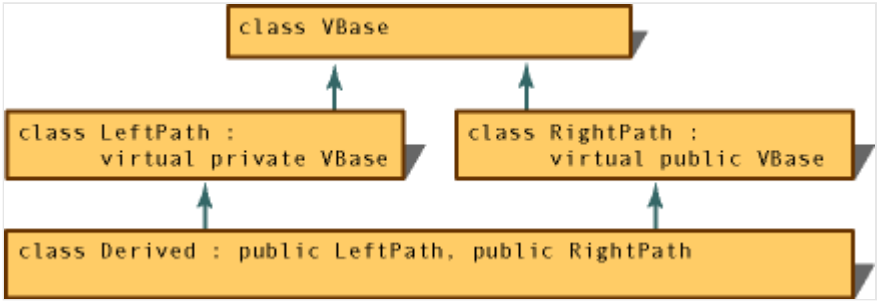
```
    int State;

    State = pvfb->GetState();     // GetState is public.
    State = pvfd->GetState();     // C2248 error expected; GetState is
private;
}
```

Dans l'exemple précédent, l'appel de la fonction `GetState` virtuelle à l'aide d'un pointeur vers le type `VFuncBase` appelle `VFuncDerived::GetState` et `GetState` est traité comme **`public`**. Toutefois, l'appel `GetState` à l'aide d'un pointeur vers le type `VFuncDerived` est une violation de contrôle d'accès, car `GetState` est déclaré **`private`** dans la classe `VFuncDerived`.

> ⊗ **Attention**
>
> La fonction virtuelle `GetState` peut être appelée en utilisant un pointeur vers la classe de base `VFuncBase`. Cela ne signifie pas que la fonction appelée est la version de classe de base de cette fonction.

Contrôle d'accès avec héritage multiple

Dans les treillis à héritage multiple impliquant des classes de base virtuelles, un nom donné est accessible via plusieurs chemins. Comme un contrôle d'accès différent peut être appliqué le long de ces différents chemins, le compilateur choisit le chemin qui fournit le plus souvent l'accès. Consultez la figure suivante :



Chemins d'accès d'un graphique d'héritage

Dans cette figure, un nom déclaré dans la classe `VBase` est toujours accessible via la classe `RightPath`. Le chemin d'accès droit est plus accessible, car `RightPath` déclare en tant **`public`** que classe de base, tandis que `LeftPath` déclare `VBase` en tant que **`private`** `VBase` .

Voir aussi

Informations de référence sur le langage C++

`friend` (C++)

Article • 06/04/2023

Dans certains cas, il est utile pour une classe d'accorder un accès au niveau membre à des fonctions qui ne sont pas membres de la classe, ou à tous les membres d'une classe distincte. Ces fonctions et classes gratuites sont appelées *amis*, marquées par le `friend` mot clé. Seul l'implémenteur de classe peut déclarer qui sont ses fonctions friend. Une fonction ou une classe ne peut pas se déclarer comme un ami d'une classe. Dans une définition de classe, utilisez le `friend` mot clé et le nom d'une fonction non membre ou d'une autre classe pour lui accorder l'accès aux membres privés et protégés de votre classe. Dans une définition de modèle, un paramètre de type peut être déclaré en tant que `friend`.

Syntaxe

`friend-declaration`:

- **`friend`** *`function-declaration`*
- **`friend`** *`function-definition`*
- **`friend`** *`elaborated-type-specifier`* **`;`**;
- **`friend`** *`simple-type-specifier`* **`;`**
- **`friend`** *`typename-specifier`* **`;`**

`friend` Déclarations

Si vous déclarez une `friend` fonction qui n'a pas été déclarée précédemment, cette fonction est exportée vers l'étendue non-classe englobante.

Les fonctions déclarées dans une `friend` déclaration sont traitées comme si elles avaient été déclarées à l'aide de la `extern` mot clé. Pour plus d'informations, consultez extern.

Bien que les fonctions avec étendue globale puissent être déclarées en tant que `friend` fonctions antérieures à leurs prototypes, les fonctions membres ne peuvent pas être déclarées en tant que `friend` fonctions avant l'apparition de leur déclaration de classe complète. Le code suivant montre comment une telle déclaration échoue :

C++

```
class ForwardDeclared;   // Class name is known.
class HasFriends
{
```

```
    friend int ForwardDeclared::IsAFriend();   // C2039 error expected
};
```

L'exemple précédent entre le nom `ForwardDeclared` de la classe dans l'étendue, mais la déclaration complète (plus précisément, la partie qui déclare la fonction `IsAFriend`) n'est pas connue. la déclaration dans la **`friend`** classe `HasFriends` génère une erreur.

Dans C++11, il existe deux formes de déclarations friend pour une classe :

```
C++

friend class F;
friend F;
```

Le premier formulaire introduit une nouvelle classe F si aucune classe existante de ce nom n'a été trouvée dans l'espace de noms le plus interne. **C++11** : le deuxième formulaire n'introduit pas de nouvelle classe ; il peut être utilisé lorsque la classe a déjà été déclarée, et il doit être utilisé lors de la déclaration d'un paramètre de type de modèle ou d'un `typedef` `friend`en tant que .

Utilisez `friend class F` lorsque le type référencé n'a pas encore été déclaré :

```
C++

namespace NS
{
    class M
    {
        friend class F;  // Introduces F but doesn't define it
    };
}
```

Une erreur se produit si vous utilisez `friend` avec un type de classe qui n'a pas été déclaré :

```
C++

namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data
declarations
    };
}
```

Dans l'exemple suivant, `friend F` fait référence à la `F` classe déclarée en dehors de l'étendue de NS.

C++

```
class F {};
namespace NS
{
    class M
    {
        friend F;  // OK
    };
}
```

Utilisez `friend F` pour déclarer un paramètre de modèle en tant qu'ami :

C++

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Utilisez `friend F` pour déclarer un typedef en tant qu'ami :

C++

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

Pour déclarer deux classes qui sont friends l'une de l'autre, la deuxième classe entière doit être spécifiée comme friend de la première classe. La raison de cette restriction est que le compilateur a suffisamment d'informations pour déclarer les fonctions friend uniquement au point où la deuxième classe est déclarée.

ⓘ Notes

Bien que la deuxième classe entière doive être friend de la première classe, vous pouvez sélectionner les fonctions de la première classe qui sont friends de la deuxième classe.

friend (fonctions)

Une `friend` fonction est une fonction qui n'est pas membre d'une classe, mais qui a accès aux membres privés et protégés de la classe. Les fonctions Friend ne sont pas considérées comme des membres de classe ; il s'agit de fonctions externes normales qui bénéficient de privilèges d'accès spéciaux. Les amis ne sont pas dans l'étendue de la classe et ils ne sont pas appelés à l'aide des opérateurs de sélection de membres (. et ->), sauf s'ils sont membres d'une autre classe. Une `friend` fonction est déclarée par la classe qui accorde l'accès. La `friend` déclaration peut être placée n'importe où dans la déclaration de classe. Il n'est pas affecté par les mots clés de contrôle d'accès.

L'exemple suivant présente une classe `Point` et une fonction friend, `ChangePrivate`. La `friend` fonction a accès au membre de données privé de l'objet `Point` qu'elle reçoit en tant que paramètre.

C++

```
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
// Output: 0
```

```
            1
}
```

Membres de classes en tant que friends

Les fonctions membres de classe peuvent être déclarées en tant que Friends dans d'autres classes. Prenons l'exemple suivant :

C++

```
// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
   int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Dans l'exemple précédent, seule la fonction `A::Func1( B& )` est autorisée `friend` à accéder à la classe `B`. Par conséquent, l'accès au membre `_b` privé est correct dans `Func1` de la classe `A` , mais pas dans `Func2`.

Une **`friend`** classe est une classe dont toutes les fonctions membres sont `friend` des fonctions d'une classe, c'est-à-dire dont les fonctions membres ont accès aux membres privés et protégés de l'autre classe. Supposez que la déclaration **`friend`** dans la classe `B` ait été :

C++

```
friend class A;
```

Dans ce cas, toutes les fonctions membres de la classe `A` auraient reçu `friend` l'accès à la classe `B`. Le code suivant est un exemple de `friend` classe :

C++

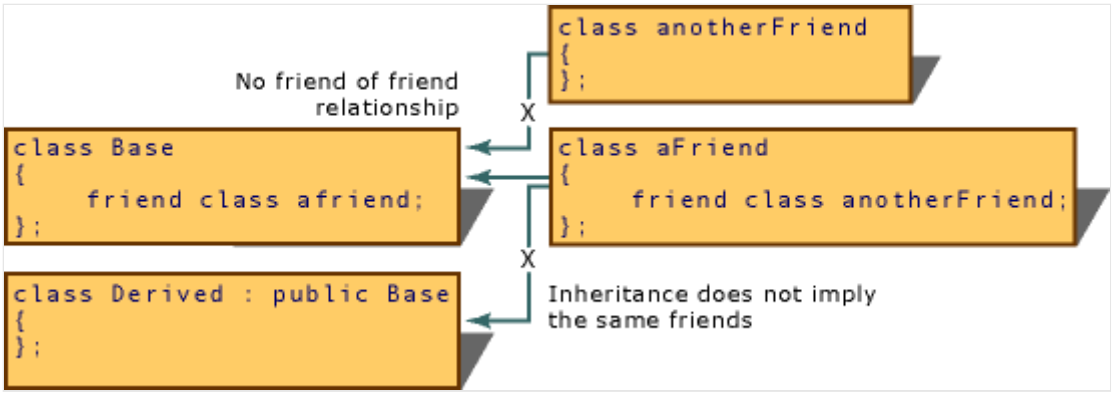
```
// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass;  // Declare a friend class
public:
   YourClass() : topSecret(0){}
   void printMember() { cout << topSecret << endl; }
private:
   int topSecret;
};

class YourOtherClass {
public:
   void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
   YourClass yc1;
   YourOtherClass yoc1;
   yc1.printMember();
   yoc1.change( yc1, 5 );
   yc1.printMember();
}
```

L'amitié n'est pas mutuelle, sauf indication explicite en tant que telle. Dans l'exemple ci-dessus, les fonctions membres de `YourClass` ne peuvent pas accéder aux membres privés de `YourOtherClass`.

Un type managé (en C++/CLI) ne peut pas avoir de `friend` fonctions, `friend` de classes ou `friend` d'interfaces.

L'amitié n'est pas héritée, ce qui signifie que les classes dérivées de `YourOtherClass` ne peuvent pas accéder aux `YourClass`membres privés de. L'amitié n'est pas transitive, de sorte que les classes qui sont amis de `YourOtherClass` ne peuvent pas accéder aux `YourClass`membres privés de.

L'illustration suivante montre quatre déclarations de classe : `Base`, `Derived`, `aFriend` et `anotherFriend`. Seule la classe `aFriend` a un accès direct aux membres privés de `Base` (et à tous les membres dont `Base` peut avoir hérité).



Définitions inline `friend`

Les fonctions Friend peuvent être définies (en fonction d'un corps de fonction) à l'intérieur des déclarations de classe. Ces fonctions sont des fonctions inline. Comme les fonctions inline membres, elles se comportent comme si elles avaient été définies immédiatement après que tous les membres de classe ont été vus, mais avant la fermeture de l'étendue de classe (à la fin de la déclaration de classe). Les fonctions Friend définies à l'intérieur des déclarations de classe se trouvent dans l'étendue de la classe englobante.

Voir aussi

Mots clés

private (C++)

Article • 16/06/2023

Syntaxe

```
private:
   [member-list]
private base-class
```

Notes

Lorsque vous précèdez une liste de membres de la classe, le **`private`** mot clé spécifie que ces membres sont accessibles uniquement à partir des fonctions membres et des amis de la classe. Cela s'applique à tous les membres déclarés jusqu'au spécificateur d'accès suivant de la classe.

Lorsque vous précèdez le nom d'une classe de base, le **`private`** mot clé spécifie que les membres publics et protégés de la classe de base sont des membres privés de la classe dérivée.

L'accès par défaut des membres d'une classe est privé. L'accès par défaut des membres d'une structure ou d'une union est public.

L'accès par défaut d'une classe de base est privé pour les classes et public pour les structures. Les unions ne peuvent pas avoir de classes de base.

Pour plus d'informations, consultez friend, public, protected et la table d'accès aux membres dans Contrôle de l'accès aux membres de la classe.

Spécifique /clr

Dans les types CLR, les mots clés du spécificateur d'accès C++ (**`public`**, **`private`**, et **`protected`**) peuvent affecter la visibilité des types et des méthodes en ce qui concerne les assemblys. Pour plus d'informations, consultez Access Control membre.

> ⓘ **Notes**

Les fichiers compilés avec **/LN** ne sont pas affectés par ce comportement. Dans ce cas, toutes les classes managées (public ou privées) seront visibles.

Spécifique END /clr

Exemple

C++

```
// keyword_private.cpp
class BaseClass {
public:
   // privMem accessible from member function
   int pubFunc() { return privMem; }
private:
   void privMem;
};

class DerivedClass : public BaseClass {
public:
   void usePrivate( int i )
      { privMem = i; }   // C2248: privMem not accessible
                         // from derived class
};

class DerivedClass2 : private BaseClass {
public:
   // pubFunc() accessible from derived class
   int usePublic() { return pubFunc(); }
};

int main() {
   BaseClass aBase;
   DerivedClass aDerived;
   DerivedClass2 aDerived2;
   aBase.privMem = 1;     // C2248: privMem not accessible
   aDerived.privMem = 1;  // C2248: privMem not accessible
                          //    in derived class
   aDerived2.pubFunc();   // C2247: pubFunc() is private in
                          //    derived class
}
```

Voir aussi

Contrôle de l'accès aux membres de classe

Mots clés

protected (C++)

Article • 03/04/2023

Syntaxe

```
protected:
   [member-list]
protected base-class
```

Notes

Le `protected` mot clé spécifie l'accès aux membres de la classe dans la *liste des membres* jusqu'au spécificateur d'accès suivant (`public` ou `private`) ou à la fin de la définition de classe. Les membres de classe déclarés comme `protected` peuvent être utilisés uniquement par les éléments suivants :

- Les fonctions membres de la classe qui a initialement déclaré ces membres.
- Les friends de la classe qui a initialement déclaré ces membres.
- Les classes dérivées ayant un accès public ou protégé à partir de la classe qui a initialement déclaré ces membres.
- Les classes directes dérivées de manière privée qui ont également un accès privé aux membres protégés.

Lorsque vous précèdez le nom d'une classe de base, le `protected` mot clé spécifie que les membres publics et protégés de la classe de base sont des membres protégés de ses classes dérivées.

Les membres protégés ne sont pas aussi privés que `private` les membres, qui sont accessibles uniquement aux membres de la classe dans laquelle ils sont déclarés, mais ils ne sont pas aussi publics que `public` les membres, qui sont accessibles dans n'importe quelle fonction.

Les membres protégés qui sont également déclarés comme `static` sont accessibles à toute fonction friend ou membre d'une classe dérivée. Les membres protégés qui ne sont pas déclarés comme `static` sont accessibles aux fonctions amis et membres d'une

classe dérivée uniquement par le biais d'un pointeur vers, d'une référence à ou d'un objet de la classe dérivée.

Pour plus d'informations, consultez friend, public, private et the member-access table in Controling Access to Class Members.

Spécifique /clr

Dans les types CLR, les mots clés du spécificateur d'accès C++ (`public`, `private`, et `protected`) peuvent affecter la visibilité des types et des méthodes en ce qui concerne les assemblys. Pour plus d'informations, consultez Access Control membres.

> ⓘ **Notes**
>
> Les fichiers compilés avec **/LN** ne sont pas affectés par ce comportement. Dans ce cas, toutes les classes managées (public ou privées) seront visibles.

Spécifique END /clr

Exemple

C++

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
   void setProtMemb( int i ) { m_protMemb = i; }
   void Display() { cout << m_protMemb << endl; }
protected:
   int  m_protMemb;
   void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
   void useProtfunc() { Protfunc(); }
} y;

int main() {
   // x.m_protMemb;         error, m_protMemb is protected
```

```
    x.setProtMemb( 0 );   // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );   // OK, uses public access function
    y.Display();
    // x.Protfunc();         error, Protfunc() is protected
    y.useProtfunc();      // OK, uses public access function
                          // in derived class
}
```

Voir aussi

Contrôle de l'accès aux membres de classe

Mots clés

public (C++)

Article • 03/04/2023

Syntaxe

```
public:
   [member-list]
public base-class
```

Notes

Lorsque vous précèdez une liste de membres de classe, le **`public`** mot clé spécifie que ces membres sont accessibles à partir de n'importe quelle fonction. Cela s'applique à tous les membres déclarés jusqu'au spécificateur d'accès suivant de la classe.

Lorsque vous précèdez le nom d'une classe de base, la **`public`** mot clé spécifie que les membres publics et protégés de la classe de base sont des membres publics et protégés, respectivement, de la classe dérivée.

L'accès par défaut des membres d'une classe est privé. L'accès par défaut des membres d'une structure ou d'une union est public.

L'accès par défaut d'une classe de base est privé pour les classes et public pour les structures. Les unions ne peuvent pas avoir de classes de base.

Pour plus d'informations, consultez private, protected, friend et the member-access table dans Contrôle de l'accès aux membres de la classe.

Spécifique /clr

Dans les types CLR, les mots clés du spécificateur d'accès C++ (**`public`**, **`private`**, et **`protected`**) peuvent affecter la visibilité des types et des méthodes en ce qui concerne les assemblys. Pour plus d'informations, consultez Access Control membres.

> **Notes**
>
> Les fichiers compilés avec **/LN** ne sont pas affectés par ce comportement. Dans ce cas, toutes les classes managées (public ou privées) seront visibles.

Spécifique END /clr

Exemple

C++

```
// keyword_public.cpp
class BaseClass {
public:
   int pubFunc() { return 0; }
};

class DerivedClass : public BaseClass {};

int main() {
   BaseClass aBase;
   DerivedClass aDerived;
   aBase.pubFunc();       // pubFunc() is accessible
                          //    from any function
   aDerived.pubFunc();    // pubFunc() is still public in
                          //    derived class
}
```

Voir aussi

Contrôle de l'accès aux membres de classe

Mots clés

Initialisation de l'accolade

Article • 16/06/2023

Il n'est pas toujours nécessaire de définir un constructeur pour un `class`, en particulier ceux qui sont relativement simples. Les utilisateurs peuvent initialiser des objets d'un ou à `struct` l'aide d'une `class` initialisation uniforme, comme illustré dans l'exemple suivant :

C++

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t)
:
       stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum},
minTemp{minimum} {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // When there's no constructor, an empty brace initializer does
    // value initialization = {0,0,0,0,0}
    TempData td_emptyInit{};

    // Uninitialized = if used, emits warning C4700 uninitialized local
```

```
variable
    TempData td_noInit;

    // Member declaration (in order of ctor parameters)
    TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

    return 0;
}
```

Lorsqu'un `class` ou `struct` n'a aucun constructeur, vous fournissez les éléments de liste dans l'ordre dans lequel les membres sont déclarés dans le `class`. Si a `class` un constructeur, fournissez les éléments dans l'ordre des paramètres. Si un type a un constructeur par défaut, déclaré implicitement ou explicitement, vous pouvez utiliser l'initialisation d'accolades avec des accolades vides pour l'appeler. Par exemple, les éléments suivants `class` peuvent être initialisés à l'aide de l'initialisation d'accolades vides et non vides :

C++

```
#include <string>
using namespace std;

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : m_string{ str }, m_double{ dbl } {}
double m_double;
string m_string;
};

int main()
{
    class_a c1{};
    class_a c1_1;

    class_a c2{ "ww" };
    class_a c2_1("xx");

    // order of parameters is the same as the constructor
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}
```

Si une classe a des constructeurs autres que les constructeurs par défaut, l'ordre dans lequel les membres de classe apparaissent dans l'initialiseur d'accolade est l'ordre dans lequel les paramètres correspondants apparaissent dans le constructeur, et non l'ordre dans lequel les membres sont déclarés (comme dans `class_a` l'exemple précédent).

Sinon, si le type n'a pas de constructeur déclaré, les initialiseurs membres doivent apparaître dans l'initialiseur d'accolades dans le même ordre qu'ils sont déclarés. Dans ce cas, vous pouvez initialiser autant de membres publics que vous le souhaitez, mais vous ne pouvez ignorer aucun membre. L'exemple suivant montre l'ordre utilisé dans l'initialisation d'accolade lorsqu'il n'y a pas de constructeur déclaré :

C++

```
class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // compiler error
    class_d d5{ "string", 'c', 2.0 }; // compiler error
}
```

Si le constructeur par défaut est explicitement déclaré mais marqué comme supprimé, l'initialisation d'accolades vides ne peut pas être utilisée :

C++

```
class class_f {
public:
    class_f() = delete;
    class_f(string x): m_string { x } {}
    string m_string;
};
int main()
{
    class_f cf{ "hello" };
    class_f cf1{}; // compiler error C2280: attempting to reference a
deleted function
}
```

Vous pouvez utiliser l'initialisation d'accolade partout où vous effectuez généralement une initialisation, par exemple, en tant que paramètre de fonction ou valeur de retour, ou avec l'mot clé `new` :

```
```

```
C++

class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };
```

En `/std:c++17` mode et versions ultérieures, les règles d'initialisation des accolades vides sont légèrement plus restrictives. Consultez Constructeurs dérivés et initialisation d'agrégation étendue.

constructeurs initializer_list

La classe initializer_list représente une liste d'objets d'un type spécifié qui peuvent être utilisés dans un constructeur et dans d'autres contextes. Vous pouvez construire un initializer_list à l'aide de l'initialisation d'accolade :

```
C++

initializer_list<int> int_list{5, 6, 7};
```

> ⓘ **Important**
>
> Pour utiliser cette classe, vous devez inclure **l'en-tête< initializer_list>**.

Un `initializer_list` peut être copié. Dans ce cas, les membres de la nouvelle liste sont des références aux membres de la liste d'origine :

```
C++

initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"
```

Les classes de conteneur de bibliothèque standard, ainsi que `string`, `wstring` et `regex`, ont `initializer_list` des constructeurs. Les exemples suivants montrent comment effectuer une initialisation avec ces constructeurs :

```
C++

vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
```

```
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };
```

Voir aussi

Classes et structs

Constructeurs

Gestion de la durée de vie et des ressources d'objet

Article • 16/06/2023

Contrairement aux langages managés, C++ n'a pas de *garbage collection* automatique, un processus interne qui libère de la mémoire du tas et d'autres ressources pendant l'exécution d'un programme. Un programme C++ est responsable du retour de toutes les ressources acquises au système d'exploitation. L'échec de la libération d'une ressource inutilisée est appelé *une fuite*. Les ressources divulguées ne sont pas disponibles pour d'autres programmes jusqu'à ce que le processus se termine. Les fuites de mémoire en particulier sont une cause courante de bogues dans la programmation de style C.

Le C++ moderne évite autant que possible d'utiliser la mémoire tas en déclarant des objets sur la pile. Lorsqu'une ressource est trop grande pour la pile, elle doit *appartenir à* un objet. À mesure que l'objet est initialisé, il acquiert la ressource qu'il possède. L'objet est alors responsable de la libération de la ressource dans son destructeur. L'objet propriétaire lui-même est déclaré sur la pile. Le principe selon lequel les *objets possèdent des ressources* est également appelé « l'acquisition de ressources est l'initialisation », ou RAII.

Lorsqu'un objet stack propriétaire de ressources sort de l'étendue, son destructeur est automatiquement appelé. De cette façon, le garbage collection en C++ est étroitement lié à la durée de vie des objets et est déterministe. Une ressource est toujours publiée à un point connu du programme, que vous pouvez contrôler. Seuls les destructeurs déterministes comme ceux en C++ peuvent gérer de manière identique les ressources mémoire et non-mémoire.

L'exemple suivant montre un objet `w`simple . Il est déclaré sur la pile au niveau de l'étendue de la fonction et est détruit à la fin du bloc de fonction. L'objet `w` ne possède aucune *ressource* (par exemple, la mémoire allouée au tas). Son seul membre `g` est lui-même déclaré sur la pile et sort simplement de l'étendue avec `w`. Aucun code spécial n'est nécessaire dans le `widget` destructeur.

C++

```
class widget {
private:
    gadget g;   // lifetime automatically tied to enclosing object
public:
    void draw();
};
```

```
void functionUsingWidget () {
    widget w;   // lifetime automatically tied to enclosing scope
                // constructs w, including the w.g gadget member
    // ...
    w.draw();
    // ...
} // automatic destruction and deallocation for w and w.g
  // automatic exception safety,
  // as if "finally { w.dispose(); w.g.dispose(); }"
```

Dans l'exemple suivant, `w` possède une ressource de mémoire et doit donc avoir du code dans son destructeur pour supprimer la mémoire.

C++

```
class widget
{
private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data member
    w.do_something();

} // automatic destruction and deallocation for w and w.data
```

Depuis C++11, il existe une meilleure façon d'écrire l'exemple précédent : en utilisant un pointeur intelligent de la bibliothèque standard. Le pointeur intelligent gère l'allocation et la suppression de la mémoire qu'il possède. L'utilisation d'un pointeur intelligent élimine le besoin d'un destructeur explicite dans la `widget` classe.

C++

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};
```

```
void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

En utilisant des pointeurs intelligents pour l'allocation de mémoire, vous pouvez éliminer le risque de fuites de mémoire. Ce modèle fonctionne pour d'autres ressources, telles que les handles de fichiers ou les sockets. Vous pouvez gérer vos propres ressources de la même manière dans vos classes. Pour plus d'informations, consultez Pointeurs intelligents.

La conception de C++ garantit que les objets sont détruits lorsqu'ils sortent de l'étendue. C'est-à-dire qu'ils sont détruits à mesure que les blocs sont sortis, dans l'ordre inverse de construction. Lorsqu'un objet est détruit, ses bases et membres sont détruits dans un ordre précis. Les objets déclarés en dehors de n'importe quel bloc, dans une étendue globale, peuvent entraîner des problèmes. Il peut être difficile de déboguer si le constructeur d'un objet global lève une exception.

Voir aussi

Bienvenue dans C++
Informations de référence sur le langage C++
Bibliothèque C++ standard

Pimpl pour l'encapsulation au moment de la compilation (Modern C++)

Article • 16/06/2023

L'idiome simple est une technique C++ moderne permettant de masquer l'implémentation, de réduire le couplage et de séparer les interfaces. Pimpl est l'abréviation de « pointeur vers l'implémentation ». Vous êtes peut-être déjà familiarisé avec le concept, mais connaissez-le par d'autres noms comme Cheshire Cat ou Compiler Firewall idiom.

Pourquoi utiliser pimpl ?

Voici comment l'idiome simple peut améliorer le cycle de vie du développement de logiciels :

- Réduction des dépendances de compilation.
- Séparation de l'interface et de l'implémentation.
- Portabilité :

En-tête Pimpl

C++

```
// my_class.h
class my_class {
   //  ... all public and protected stuff goes here ...
private:
   class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

L'idiome simple évite les cascades de reconstruction et les dispositions d'objets cassants. Il est bien adapté aux types (transitivement) populaires.

Implémentation de Pimpl

Définissez la `impl` classe dans le fichier .cpp.

C++

```
// my_class.cpp
class my_class::impl {  // defined privately here
  // ... all private data and functions: all of these
  //     can now change without recompiling callers ...
};
my_class::my_class(): pimpl( new impl )
{
  // ... set impl values ...
}
```

Bonnes pratiques

Déterminez s'il faut ajouter la prise en charge de la spécialisation d'échange sans levée.

Voir aussi

Bienvenue dans C++

Informations de référence sur le langage C++

Bibliothèque C++ standard

Portabilité aux limites ABI

Article • 16/06/2023

Utilisez des types et des conventions suffisamment portables aux limites de l'interface binaire. Un « type portable » est un type intégré C ou un struct qui contient uniquement des types intégrés C. Les types de classes ne peuvent être utilisés que lorsque l'appelant et l'appelant sont d'accord sur la disposition, la convention d'appel, etc. Cela n'est possible que lorsque les deux sont compilés avec les mêmes paramètres de compilateur et de compilateur.

Comment aplatir une classe pour la portabilité C

Lorsque les appelants peuvent être compilés avec un autre compilateur/langage, « aplatissez » en une API « **C** » **extern** avec une convention d'appel spécifique :

C++

```
// class widget {
//   widget();
//   ~widget();
//   double method( int, gadget& );
// };
extern "C" {        // functions using explicit "this"
   struct widget;   // opaque type (forward declaration only)
   widget* STDCALL widget_create();      // constructor creates new "this"
   void STDCALL widget_destroy(widget*); // destructor consumes "this"
   double STDCALL widget_method(widget*, int, gadget*); // method uses
"this"
}
```

Voir aussi

Bienvenue dans C++
Informations de référence sur le langage C++
Bibliothèque C++ standard

Constructeurs (C++)

Article • 03/04/2023

Pour personnaliser la façon dont une classe initialise ses membres, ou pour appeler des fonctions lors de la création d'un objet de votre classe, définissez un *constructeur*. Un constructeur porte le même nom que la classe et n'a aucune valeur de retour. Vous pouvez définir autant de constructeurs surchargés que nécessaire pour personnaliser l'initialisation de différentes manières. En règle générale, les constructeurs disposent d'une accessibilité publique afin que le code en dehors de la définition de classe ou de la hiérarchie d'héritage puisse créer des objets de la classe. Mais vous pouvez également déclarer un constructeur en tant que **`protected`** ou **`private`**.

Les constructeurs peuvent éventuellement prendre une liste d'initialiseurs membres. Il s'agit d'un moyen plus efficace d'initialiser les membres de la classe que d'affecter des valeurs dans le corps du constructeur. L'exemple suivant montre une classe `Box` avec trois constructeurs surchargés. Les deux dernières listes d'init membres utilisent :

C++

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    explicit Box(int i) : m_width(i), m_length(i), m_height(i) // member
init list
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}

    int Volume() { return m_width * m_length * m_height; }

private:
    // Will have value of 0 when default constructor is called.
    // If we didn't zero-init here, default constructor would
    // leave them uninitialized with garbage values.
    int m_width{ 0 };
    int m_length{ 0 };
    int m_height{ 0 };
};
```

Lorsque vous déclarez un instance d'une classe, le compilateur choisit le constructeur à appeler en fonction des règles de résolution de surcharge :

C++

```
int main()
{
    Box b; // Calls Box()

    // Using uniform initialization (preferred):
    Box b2 {5}; // Calls Box(int)
    Box b3 {5, 8, 12}; // Calls Box(int, int, int)

    // Using function-style notation:
    Box b4(2, 4, 6); // Calls Box(int, int, int)
}
```

- Les constructeurs peuvent être déclarés en tant que `inline`, explicit, friendou constexpr.
- Un constructeur peut initialiser un objet qui a été déclaré en tant que `const`, `volatile` ou `const volatile`. L'objet devient `const` une fois le constructeur terminé.
- Pour définir un constructeur dans un fichier d'implémentation, attribuez-lui un nom qualifié comme n'importe quelle autre fonction membre : `Box::Box(){...}`.

Listes d'initialiseurs de membres

Un constructeur peut éventuellement avoir une *liste d'initialiseurs membres*, qui initialise les membres de la classe avant l'exécution du corps du constructeur. (Une liste d'initialiseurs de membres n'est pas la même chose qu'une *liste d'initialiseurs* de type std::initializer_list<T>.)

Préférer les listes d'initialiseur de membres à l'affectation de valeurs dans le corps du constructeur. Une liste d'initialiseurs de membres initialise directement les membres. L'exemple suivant montre la liste d'initialiseurs de membres, qui se compose de toutes les `identifier(argument)` expressions après les deux-points :

C++

```
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
```

L'identificateur doit faire référence à un membre de classe ; il est initialisé avec la valeur de l'argument . L'argument peut être l'un des paramètres du constructeur, un appel de fonction ou un std::initializer_list<T>.

`const` les membres et les membres de type référence doivent être initialisés dans la liste d'initialiseurs de membres.

Pour vous assurer que les classes de base sont entièrement initialisées avant l'exécution du constructeur dérivé, appelez tous les constructeurs de classes de base paramétrables dans la liste d'initialiseurs.

Constructeurs par défaut

Les constructeurs par défaut n'ont généralement aucun paramètre, mais ils peuvent avoir des paramètres avec des valeurs par défaut.

C++

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/}

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h),
m_length(l){}
...
}
```

Les constructeurs par défaut sont l'une des fonctions membres spéciales. Si aucun constructeur n'est déclaré dans une classe, le compilateur fournit un constructeur par défaut implicite `inline` .

C++

```
#include <iostream>
using namespace std;

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
```

```
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}
```

Si vous vous appuyez sur un constructeur implicite par défaut, veillez à initialiser les membres dans la définition de classe, comme indiqué dans l'exemple précédent. Sans ces initialiseurs, les membres seraient non initialisés et l'appel Volume() produirait une valeur garbage. En général, il est recommandé d'initialiser les membres de cette façon, même en ne s'appuyant pas sur un constructeur par défaut implicite.

Vous pouvez empêcher le compilateur de générer un constructeur implicite par défaut en le définissant comme supprimé :

C++

```
    // Default constructor
    Box() = delete;
```

Un constructeur par défaut généré par le compilateur sera défini comme supprimé si les membres de classe ne sont pas constructibles par défaut. Par exemple, tous les membres du type de classe et leurs membres de type de classe doivent avoir un constructeur et des destructeurs par défaut accessibles. Tous les membres de données de type référence et tous les `const` membres doivent avoir un initialiseur de membre par défaut.

Lorsque vous appelez un constructeur par défaut généré par le compilateur et que vous essayez d'utiliser des parenthèses, un avertissement est émis :

C++

```
class myclass{};
int main(){
myclass mc();     // warning C4930: prototyped function not called (was a
variable definition intended?)
}
```

Cette instruction est un exemple du problème « Most Vexing Parse ». Vous pouvez interpréter `myclass md();` comme une déclaration de fonction ou comme l'appel d'un constructeur par défaut. Étant donné que les analyseurs C++ favorisent les déclarations par rapport à d'autres éléments, l'expression est traitée comme une déclaration de fonction. Pour plus d'informations, consultez Analyse la plus vexante.

Si des constructeurs autres que les constructeurs par défaut sont déclarés, le compilateur ne fournit pas de constructeur par défaut :

C++

```
class Box {
public:
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height){}
private:
    int m_width;
    int m_length;
    int m_height;

};

int main(){

    Box box1(1, 2, 3);
    Box box2{ 2, 3, 4 };
    Box box3; // C2512: no appropriate default constructor available
}
```

Si une classe n'a pas de constructeur par défaut, un tableau d'objets de cette classe ne peut pas être construit à l'aide de la seule syntaxe entre crochets. Par exemple, étant donné le bloc de code précédent, un tableau de cases ne peut pas être déclaré comme suit :

C++

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

Toutefois, vous pouvez utiliser un ensemble de listes d'initialiseurs pour initialiser un tableau d'objets Box :

C++

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Pour plus d'informations, consultez Initialiseurs.

Constructeurs de copie

Un *constructeur de copie* initialise un objet en copiant les valeurs membres d'un objet du même type. Si les membres de votre classe sont tous des types simples tels que des valeurs scalaires, le constructeur de copie généré par le compilateur est suffisant et vous n'avez pas besoin de définir le vôtre. Si votre classe nécessite une initialisation plus complexe, vous devez implémenter un constructeur de copie personnalisé. Par exemple,

si un membre de classe est un pointeur, vous devez définir un constructeur de copie pour allouer une nouvelle mémoire et copier les valeurs à partir de l'objet pointé de l'autre. Le constructeur de copie généré par le compilateur copie simplement le pointeur, afin que le nouveau pointeur pointe toujours vers l'emplacement de mémoire de l'autre.

Un constructeur de copie peut avoir l'une des signatures suivantes :

```
C++

    Box(Box& other); // Avoid if possible--allows modification of other.
    Box(const Box& other);
    Box(volatile Box& other);
    Box(volatile const Box& other);

    // Additional parameters OK if they have default values
    Box(Box& other, int i = 42, string label = "Box");
```

Lorsque vous définissez un constructeur de copie, vous devez également définir un opérateur d'affectation de copie (=). Pour plus d'informations, consultez Attribution et copie des constructeurs et opérateurs d'affectation de copie.

Vous pouvez empêcher la copie de votre objet en définissant le constructeur de copie comme supprimé :

```
C++

    Box (const Box& other) = delete;
```

La tentative de copie de l'objet génère l'erreur *C2280 : tentative de référence d'une fonction supprimée*.

Déplacer les constructeurs

Un *constructeur de déplacement* est une fonction membre spéciale qui déplace la propriété des données d'un objet existant vers une nouvelle variable sans copier les données d'origine. Il prend une référence rvalue comme premier paramètre, et tous les paramètres ultérieurs doivent avoir des valeurs par défaut. Les constructeurs de déplacement peuvent augmenter considérablement l'efficacité de votre programme lors du passage d'objets volumineux.

```
C++

Box(Box&& other);
```

Le compilateur choisit un constructeur de déplacement lorsque l'objet est initialisé par un autre objet du même type, si l'autre objet est sur le point d'être détruit et n'a plus besoin de ses ressources. L'exemple suivant montre un cas où un constructeur de déplacement est sélectionné par résolution de surcharge. Dans le constructeur qui appelle `get_Box()`, la valeur retournée est une *valeur x (* valeur eXpiring). Il n'est affecté à aucune variable et est donc sur le point de sortir de l'étendue. Pour fournir de la motivation pour cet exemple, donnons à Box un grand vecteur de chaînes qui représentent son contenu. Plutôt que de copier le vecteur et ses chaînes, le constructeur de déplacement le « vole » à partir de la valeur d'expiration « box » afin que le vecteur appartienne désormais au nouvel objet. L'appel à `std::move` est tout ce qui est nécessaire, car les classes `vector` et `string` implémentent leurs propres constructeurs de déplacement.

C++

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Box {
public:
    Box() { std::cout << "default" << std::endl; }
    Box(int width, int height, int length)
       : m_width(width), m_height(height), m_length(length)
    {
        std::cout << "int,int,int" << std::endl;
    }
    Box(Box& other)
       : m_width(other.m_width), m_height(other.m_height),
m_length(other.m_length)
    {
        std::cout << "copy" << std::endl;
    }
    Box(Box&& other) : m_width(other.m_width), m_height(other.m_height),
m_length(other.m_length)
    {
        m_contents = std::move(other.m_contents);
        std::cout << "move" << std::endl;
    }
    int Volume() { return m_width * m_height * m_length; }
    void Add_Item(string item) { m_contents.push_back(item); }
    void Print_Contents()
    {
        for (const auto& item : m_contents)
        {
            cout << item << " ";
        }
```

```
    }
private:
    int m_width{ 0 };
    int m_height{ 0 };
    int m_length{ 0 };
    vector<string> m_contents;
};

Box get_Box()
{
    Box b(5, 10, 18); // "int,int,int"
    b.Add_Item("Toupee");
    b.Add_Item("Megaphone");
    b.Add_Item("Suit");

    return b;
}

int main()
{
    Box b; // "default"
    Box b1(b); // "copy"
    Box b2(get_Box()); // "move"
    cout << "b2 contents: ";
    b2.Print_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open
    return 0;
}
```

Si une classe ne définit pas de constructeur de déplacement, le compilateur en génère un implicite s'il n'existe pas de constructeur de copie déclaré par l'utilisateur, d'opérateur d'affectation de copie, d'opérateur d'affectation de déplacement ou de destructeur. Si aucun constructeur de déplacement explicite ou implicite n'est défini, les opérations qui utiliseraient sinon un constructeur de déplacement utilisent le constructeur de copie à la place. Si une classe déclare un constructeur de déplacement ou un opérateur d'affectation de déplacement, le constructeur de copie déclaré implicitement est défini comme supprimé.

Un constructeur de déplacement déclaré implicitement est défini comme supprimé si des membres qui sont des types de classes ne disposent pas d'un destructeur ou si le compilateur ne peut pas déterminer le constructeur à utiliser pour l'opération de déplacement.

Pour plus d'informations sur l'écriture d'un constructeur de déplacement non trivial, consultez Déplacer les constructeurs et les opérateurs d'affectation de déplacement (C++).

Constructeurs explicitement par défaut et supprimés

Vous pouvez explicitement les constructeurs *de copie par défaut* , les constructeurs par défaut, les constructeurs de déplacement, les opérateurs d'affectation de copie, les opérateurs d'affectation de déplacement et les destructeurs. Vous pouvez *supprimer* explicitement toutes les fonctions membres spéciales.

C++

```
class Box2
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

Pour plus d'informations, consultez Fonctions explicitement par défaut et supprimées.

constructeurs constexpr

Un constructeur peut être déclaré en tant que constexpr si

- il est soit déclaré comme étant par défaut, soit il remplit toutes les conditions pour les fonctions constexpr en général ;
- la classe n'a pas de classes de base virtuelles ;
- chacun des paramètres est un type littéral ;
- le corps n'est pas un try-block de fonction ;
- tous les membres de données non statiques et les sous-objets de classe de base sont initialisés ;
- si la classe est (a) une union ayant des membres variants, ou (b) a des unions anonymes, un seul des membres de l'union est initialisé ;
- chaque membre de données non statique de type de classe, et tous les sous-objets de classe de base ont un constructeur constexpr

Constructeurs de liste d'initialiseurs

Si un constructeur prend un std::initializer_list<T> comme paramètre et que tous les autres paramètres ont des arguments par défaut, ce constructeur est sélectionné en résolution de surcharge lorsque la classe est instanciée via une initialisation directe. Vous pouvez utiliser le initializer_list pour initialiser n'importe quel membre qui peut l'accepter. Par exemple, supposons que la classe Box (illustrée précédemment) a un `std::vector<string>` membre `m_contents`. Vous pouvez fournir un constructeur comme suit :

```
C++

    Box(initializer_list<string> list, int w = 0, int h = 0, int l = 0)
        : m_contents(list), m_width(w), m_height(h), m_length(l)
{}
```

Puis créez des objets Box comme suit :

```
C++

    Box b{ "apples", "oranges", "pears" }; // or ...
    Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

Constructeurs explicites

Si une classe a un constructeur avec un paramètre unique, ou si tous les paramètres sauf un ont une valeur par défaut, le type de paramètre peut être implicitement converti en type de classe. Par exemple, si la classe `Box` a un constructeur semblable au suivant :

```
C++

Box(int size): m_width(size), m_length(size), m_height(size){}
```

Il est possible d'initialiser une box comme suit :

```
C++

Box b = 42;
```

Ou de passer une valeur int à une fonction qui accepte un objet Box :

```
C++

class ShippingOrder
{
```

```
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage){}

private:
    Box m_box;
    double m_postage;
}
//elsewhere...
    ShippingOrder so(42, 10.8);
```

Ces conversions peuvent être utiles dans certains cas mais, le plus souvent, elles peuvent entraîner des erreurs subtiles mais graves dans votre code. En règle générale, vous devez utiliser le `explicit` mot clé sur un constructeur (et des opérateurs définis par l'utilisateur) pour empêcher ce type de conversion de type implicite :

```
C++

explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

Si le constructeur est explicite, cette ligne entraîne une erreur du compilateur : `ShippingOrder so(42, 10.8);`. Pour plus d'informations, consultez Conversions de types définis par l'utilisateur.

Ordre de construction

Un constructeur effectue son travail dans l'ordre suivant :

1. Il appelle les constructeurs membres et de classe de base dans l'ordre de déclaration.
2. Si la classe est dérivée de classes de base virtuelles, il initialise les pointeurs de base virtuels de l'objet.
3. Si la classe possède ou hérite des fonctions virtuelles, il initialise les pointeurs de fonction virtuelle de l'objet. Les pointeurs de fonction virtuelle pointent sur la table de fonctions virtuelles de la classe pour permettre la liaison correcte des appels de fonction virtuelle au code.
4. Il exécute le code dans son corps de fonction.

L'exemple suivant montre l'ordre dans lequel les constructeurs de classe de base et membres sont appelés dans le constructeur pour une classe dérivée. Tout d'abord, le constructeur de base est appelé. Ensuite, les membres de la classe de base sont

initialisés dans l'ordre dans lequel ils apparaissent dans la déclaration de classe. Enfin, le constructeur dérivé est appelé.

C++

```
#include <iostream>

using namespace std;

class Contained1 {
public:
    Contained1() { cout << "Contained1 ctor\n"; }
};

class Contained2 {
public:
    Contained2() { cout << "Contained2 ctor\n"; }
};

class Contained3 {
public:
    Contained3() { cout << "Contained3 ctor\n"; }
};

class BaseContainer {
public:
    BaseContainer() { cout << "BaseContainer ctor\n"; }
private:
    Contained1 c1;
    Contained2 c2;
};

class DerivedContainer : public BaseContainer {
public:
    DerivedContainer() : BaseContainer() { cout << "DerivedContainer
ctor\n"; }
private:
    Contained3 c3;
};

int main() {
    DerivedContainer dc;
}
```

Voici la sortie :

Output

```
Contained1 ctor
Contained2 ctor
BaseContainer ctor
```

```
Contained3 ctor
DerivedContainer ctor
```

Un constructeur de classe dérivée appelle toujours un constructeur de classe de base, afin de pouvoir s'appuyer sur des classes de base complètement construites avant d'effectuer tout travail supplémentaire. Les constructeurs de classe de base sont appelés dans l'ordre de dérivation, par exemple, si `ClassA` est dérivé de `ClassB`, qui est dérivé de `ClassC`, le `ClassC` constructeur est appelé en premier, puis le `ClassB` constructeur, puis le `ClassA` constructeur.

Si une classe de base n'a pas de constructeur par défaut, vous devez fournir les paramètres du constructeur de classe de base dans le constructeur de classe dérivé :

C++

```
class Box {
public:
    Box(int width, int length, int height){
       m_width = width;
       m_length = length;
       m_height = height;
    }

private:
    int m_width;
    int m_length;
    int m_height;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, const string label&) :
Box(width, length, height){
        m_label = label;
    }
private:
    string m_label;
};

int main(){

    const string aLabel = "aLabel";
    StorageBox sb(1, 2, 3, aLabel);
}
```

Si un constructeur lève une exception, l'ordre de destruction est l'inverse de l'ordre de construction :

1. Le code dans le corps de la fonction constructeur est déroulé.

2. Les objets de classe de base et membres sont détruits dans l'ordre inverse de déclaration.

3. Si le constructeur n'est pas délégué, tous les objets et membres de classe de base entièrement construits sont détruits. Toutefois, étant donné que l'objet lui-même n'est pas entièrement construit, le destructeur n'est pas exécuté.

Constructeurs dérivés et initialisation d'agrégation étendue

Si le constructeur d'une classe de base n'est pas public, mais accessible à une classe dérivée, vous ne pouvez pas utiliser d'accolades vides pour initialiser un objet du type dérivé en `/std:c++17` mode et ultérieurement dans Visual Studio 2017 et versions ultérieures.

L'exemple suivant montre le comportement conforme à C++14 :

C++

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
               // which can call Base ctor.
```

Dans C++17, `Derived` est désormais considéré comme un type d'agrégat. Cela signifie que l'initialisation de `Base` par le biais du constructeur par défaut privé se produit donc directement dans le cadre de la règle d'initialisation d'agrégats étendue. Auparavant, le `Base` constructeur privé était appelé via le `Derived` constructeur, et il a réussi en raison de la `friend` déclaration .

L'exemple suivant montre le comportement de C++17 dans Visual Studio 2017 et versions ultérieures en `/std:c++17` mode :

C++

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': can't access
               // private member declared in class 'Base'
```

Constructeurs pour les classes qui ont plusieurs héritages

Si une classe est dérivée de plusieurs classes de base, les constructeurs de classe de base sont appelés dans l'ordre dans lequel ils sont répertoriés dans la déclaration de la classe dérivée :

C++

```
#include <iostream>
using namespace std;

class BaseClass1 {
public:
    BaseClass1() { cout << "BaseClass1 ctor\n"; }
};
class BaseClass2 {
public:
    BaseClass2() { cout << "BaseClass2 ctor\n"; }
};
class BaseClass3 {
public:
    BaseClass3() { cout << "BaseClass3 ctor\n"; }
};
class DerivedClass : public BaseClass1,
                     public BaseClass2,
                     public BaseClass3
                     {
public:
    DerivedClass() { cout << "DerivedClass ctor\n"; }
};

int main() {
```

```
    DerivedClass dc;
}
```

Vous devez obtenir la sortie suivante :

Output

```
BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor
```

Constructeurs effectuant une délégation

Un *constructeur de délégation* appelle un constructeur différent dans la même classe pour effectuer une partie du travail d'initialisation. Cette fonctionnalité est utile lorsque vous avez plusieurs constructeurs qui doivent tous effectuer un travail similaire. Vous pouvez écrire la logique main dans un constructeur et l'appeler à partir d'autres. Dans l'exemple trivial suivant, Box(int) délègue son travail à Box(int,int,int) :

C++

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    Box(int i) :  Box(i, i, i)  // delegating constructor
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
    //... rest of class as before
};
```

L'objet créé par les constructeurs est entièrement initialisé dès qu'un constructeur est terminé. Pour plus d'informations, consultez Délégation de constructeurs.

Constructeurs d'héritage (C++11)

Une classe dérivée peut hériter des constructeurs d'une classe de base directe à l'aide d'une `using` déclaration, comme illustré dans l'exemple suivant :

C++

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base() { cout << "Base()" << endl; }
    Base(const Base& other) { cout << "Base(Base&)" << endl; }
    explicit Base(int i) : num(i) { cout << "Base(int)" << endl; }
    explicit Base(char c) : letter(c) { cout << "Base(char)" << endl; }

private:
    int num;
    char letter;
};

class Derived : Base
{
public:
    // Inherit all constructors from Base
    using Base::Base;

private:
    // Can't initialize newMember from Base constructors.
    int newMember{ 0 };
};

int main()
{
    cout << "Derived d1(5) calls: ";
    Derived d1(5);
    cout << "Derived d1('c') calls: ";
    Derived d2('c');
    cout << "Derived d3 = d2 calls: " ;
    Derived d3 = d2;
    cout << "Derived d4 calls: ";
    Derived d4;
}

/* Output:
Derived d1(5) calls: Base(int)
Derived d1('c') calls: Base(char)
Derived d3 = d2 calls: Base(Base&)
Derived d4 calls: Base()*/
```

Visual Studio 2017 et versions ultérieures : l'instruction `using` en `/std:c++17` mode et versions ultérieures place dans l'étendue tous les constructeurs de la classe de base, à l'exception de ceux qui ont une signature identique à celle des constructeurs de la classe dérivée. En général, il est préférable d'utiliser des constructeurs hérités lorsque la classe dérivée ne déclare aucun nouveau membre de données ou constructeur.

Un modèle de classe peut hériter de tous les constructeurs d'un argument de type si ce type spécifie une classe de base :

C++

```
template< typename T >
class Derived : T {
    using T::T;   // declare the constructors from T
    // ...
};
```

Une classe dérivée ne peut pas hériter de plusieurs classes de base si ces classes de base ont des constructeurs qui ont une signature identique.

Constructeurs et classes composites

Les classes qui contiennent des membres de type classe sont appelées *classes composites*. Lorsqu'un membre de type classe d'une classe composite est créé, le constructeur est appelé avant le propre constructeur de la classe. Lorsqu'une classe contenue n'a pas de constructeur par défaut, vous devez utiliser une liste d'initialisation dans le constructeur de la classe composite. Dans l'exemple `StorageBox` précédent, si vous remplacez le type de la variable membre `m_label` par une nouvelle classe `Label`, vous devez appeler le constructeur de classe de base et initialiser la variable `m_label` dans le constructeur `StorageBox` :

C++

```
class Label {
public:
    Label(const string& name, const string& address) { m_name = name;
m_address = address; }
    string m_name;
    string m_address;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, Label label)
        : Box(width, length, height), m_label(label){}
private:
    Label m_label;
};

int main(){
// passing a named Label
    Label label1{ "some_name", "some_address" };
    StorageBox sb1(1, 2, 3, label1);
```

```
    // passing a temporary label
    StorageBox sb2(3, 4, 5, Label{ "another name", "another address" });

    // passing a temporary label as an initializer list
    StorageBox sb3(1, 2, 3, {"myname", "myaddress"});
}
```

Contenu de cette section

- Constructeurs de copie et opérateurs d'affectation de copie
- Constructeurs de déplacement et opérateurs d'affectation de déplacement
- Constructeurs effectuant une délégation

Voir aussi

Classes et structs

Constructeurs de copie et opérateurs d'assignation de copie (C++)

Article • 03/04/2023

> ⓘ **Notes**
>
> À compter de C++11, deux types d'affectation sont pris en charge dans la langue : *copier l'affectation* et *déplacer l'affectation*. Dans cet article « assignation » signifie assignation de copie, sauf spécification contraire. Pour plus d'informations sur l'affectation de déplacement, consultez **Déplacer les constructeurs et les opérateurs d'assignation de déplacement (C++).**
>
> Les opérations d'assignation et d'initialisation génèrent une copie des objets.

- **Affectation** : lorsque la valeur d'un objet est affectée à un autre objet, le premier objet est copié dans le deuxième objet. Ainsi, ce code copie la valeur de `b` dans `a`:

  ```
  C++

  Point a, b;
  ...
  a = b;
  ```

- **Initialisation** : l'initialisation se produit lorsque vous déclarez un nouvel objet, lorsque vous passez des arguments de fonction par valeur ou lorsque vous retournez par valeur à partir d'une fonction.

Vous pouvez définir la sémantique de « copie » pour les objets de type classe. Considérez par exemple le code suivant :

```
C++

TextFile a, b;
a.Open( "FILE1.DAT" );
b.Open( "FILE2.DAT" );
b = a;
```

Le code précédent peut signifier « copier le contenu de FILE1 . DAT à FILE2. DAT » ou « ignorer FILE2 ». DAT et créez `b` un deuxième handle pour FILE1.DAT. » Vous devez joindre la sémantique de copie appropriée à chaque classe, comme suit :

- Utilisez un opérateur `operator=` d'affectation qui retourne une référence au type de classe et prend un paramètre passé par `const` référence, par exemple `ClassName& operator=(const ClassName& x);`.

- Utilisez le constructeur de copie.

Si vous ne déclarez pas de constructeur de copie, le compilateur génère un constructeur de copie au niveau des membres pour vous. De même, si vous ne déclarez pas d'opérateur d'affectation de copie, le compilateur génère pour vous un opérateur d'affectation de copie au niveau des membres. La déclaration d'un constructeur de copie ne supprime pas l'opérateur d'assignation de copie généré par le compilateur, et vice versa. Si vous implémentez l'une ou l'autre, nous vous recommandons également d'implémenter l'autre. Lorsque vous implémentez les deux, la signification du code est claire.

Le constructeur de copie prend un argument de type `ClassName&`, où *`ClassName`* est le nom de la classe . Par exemple :

C++

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& );            // Declare copy constructor.
    Window& operator=(const Window& x); // Declare copy assignment.
    // ...
};

int main()
{
}
```

> ⓘ Notes
>
> Définissez le type de l'argument `const ClassName&` du constructeur de copie dans la mesure du possible. Cela empêche le constructeur de copie de modifier accidentellement l'objet copié. Il vous permet également de copier à partir d'objets `const` .

Constructeurs de copie générés par le compilateur

Les constructeurs de copie générés par le compilateur, comme les constructeurs de copie définis par l'utilisateur, ont un seul argument de type « référence à *class-name* ». Une exception est quand toutes les classes de base et les classes membres ont des constructeurs de copie déclarés comme prenant un seul argument de type **`const`***class-name*&. Dans ce cas, l'argument du constructeur de copie généré par le compilateur est également **`const`**.

Lorsque le type d'argument du constructeur de copie n'est pas **`const`**, l'initialisation en copiant un **`const`** objet génère une erreur. L'inverse n'est pas vrai : si l'argument est **`const`**, vous pouvez initialiser en copiant un objet qui n'est pas **`const`**.

Les opérateurs d'affectation générés par le compilateur suivent le même modèle pour **`const`**. Ils prennent un seul argument de type `ClassName&` , sauf si les opérateurs d'affectation dans toutes les classes de base et membres prennent des arguments de type `const ClassName&`. Dans ce cas, l'opérateur d'affectation généré pour la classe prend un **`const`** argument.

> ⓘ Notes
>
> Lorsque des classes de base virtuelles sont initialisées par des constructeurs de copie, qu'elles soient générées par le compilateur ou définies par l'utilisateur, elles ne sont initialisées qu'une seule fois : au moment où elles sont construites.

Les implications sont similaires au constructeur de copie. Lorsque le type d'argument n'est pas **`const`**, l'affectation à partir d'un **`const`** objet génère une erreur. L'inverse n'est pas vrai : si une **`const`** valeur est affectée à une valeur qui n'est pas **`const`**, l'affectation réussit.

Pour plus d'informations sur les opérateurs d'affectation surchargés, consultez Affectation.

Constructeurs de déplacement et opérateurs d'assignation de déplacement (C++)

Article • 03/04/2023

Cette rubrique explique comment écrire un *constructeur de déplacement* et un opérateur d'affectation de déplacement pour une classe C++. Un constructeur de déplacement permet aux ressources appartenant à un objet rvalue d'être déplacées dans une valeur lvalue sans copier. Pour plus d'informations sur la sémantique de déplacement, consultez Rvalue Reference Declarator : &&.

Cette rubrique repose sur la classe C++ suivante, `MemoryBlock`, qui gère une mémoire tampon.

C++

```
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

   // Simple constructor that initializes the resource.
   explicit MemoryBlock(size_t length)
      : _length(length)
      , _data(new int[length])
   {
      std::cout << "In MemoryBlock(size_t). length = "
                << _length << "." << std::endl;
   }

   // Destructor.
   ~MemoryBlock()
   {
      std::cout << "In ~MemoryBlock(). length = "
                << _length << ".";

      if (_data != nullptr)
      {
         std::cout << " Deleting resource.";
         // Delete the resource.
         delete[] _data;
      }
```

```
        std::cout << std::endl;
    }

    // Copy constructor.
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length)
        , _data(new int[other._length])
    {
        std::cout << "In MemoryBlock(const MemoryBlock&). length = "
                  << other._length << ". Copying resource." << std::endl;

        std::copy(other._data, other._data + _length, _data);
    }

    // Copy assignment operator.
    MemoryBlock& operator=(const MemoryBlock& other)
    {
        std::cout << "In operator=(const MemoryBlock&). length = "
                  << other._length << ". Copying resource." << std::endl;

        if (this != &other)
        {
            // Free the existing resource.
            delete[] _data;

            _length = other._length;
            _data = new int[_length];
            std::copy(other._data, other._data + _length, _data);
        }
        return *this;
    }

    // Retrieves the length of the data resource.
    size_t Length() const
    {
        return _length;
    }

private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};
```

Les procédures suivantes décrivent comment écrire un constructeur de déplacement ou un opérateur d'assignation de déplacement pour l'exemple de classe C++.

Pour créer un constructeur de déplacement pour une classe C++

1. Définissez une méthode de constructeur vide qui accepte une référence rvalue au type de classe comme paramètre, comme illustré dans l'exemple suivant :

```
C++

MemoryBlock(MemoryBlock&& other)
   : _data(nullptr)
   , _length(0)
{
}
```

2. Dans le constructeur de déplacement, assignez les données membres de classe de l'objet source à l'objet en cours de construction :

```
C++

_data = other._data;
_length = other._length;
```

3. Assignez les données membres de l'objet source aux valeurs par défaut. Cela empêche le destructeur de libérer des ressources (telles que de la mémoire) à plusieurs reprises :

```
C++

other._data = nullptr;
other._length = 0;
```

Pour créer un opérateur d'assignation de déplacement pour une classe C++

1. Définissez un opérateur d'assignation vide qui accepte une référence rvalue au type de classe comme paramètre et retourne une référence au type de classe, comme illustré dans l'exemple suivant :

```
C++

MemoryBlock& operator=(MemoryBlock&& other)
{
}
```

2. Dans l'opérateur d'assignation de déplacement, ajoutez une instruction conditionnelle qui n'effectue aucune opération si vous essayez d'assigner l'objet à lui-même.

```
C++
```

```
if (this != &other)
{
}
```

3. Dans l'instruction conditionnelle, libérez des ressources (telles que de la mémoire) à partir de l'objet auquel elles sont assignées.

   L'exemple suivant libère le membre `_data` de l'objet auquel il est assigné :

   C++

   ```
   // Free the existing resource.
   delete[] _data;
   ```

   Suivez les étapes 2 et 3 de la première procédure pour transférer les données membres de l'objet source vers l'objet en cours de construction :

   C++

   ```
   // Copy the data pointer and its length from the
   // source object.
   _data = other._data;
   _length = other._length;

   // Release the data pointer from the source object so that
   // the destructor does not free the memory multiple times.
   other._data = nullptr;
   other._length = 0;
   ```

4. Retournez une référence à l'objet actif, comme indiqué dans l'exemple suivant :

   C++

   ```
   return *this;
   ```

Exemple : Terminer le constructeur de déplacement et l'opérateur d'affectation

L'exemple suivant montre le constructeur de déplacement et l'opérateur d'assignation de déplacement complets pour la classe `MemoryBlock` :

C++

```
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
   : _data(nullptr)
   , _length(0)
{
   std::cout << "In MemoryBlock(MemoryBlock&&). length = "
             << other._length << ". Moving resource." << std::endl;

   // Copy the data pointer and its length from the
   // source object.
   _data = other._data;
   _length = other._length;

   // Release the data pointer from the source object so that
   // the destructor does not free the memory multiple times.
   other._data = nullptr;
   other._length = 0;
}

// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other) noexcept
{
   std::cout << "In operator=(MemoryBlock&&). length = "
             << other._length << "." << std::endl;

   if (this != &other)
   {
      // Free the existing resource.
      delete[] _data;

      // Copy the data pointer and its length from the
      // source object.
      _data = other._data;
      _length = other._length;

      // Release the data pointer from the source object so that
      // the destructor does not free the memory multiple times.
      other._data = nullptr;
      other._length = 0;
   }
   return *this;
}
```

Exemple Utiliser la sémantique de déplacement pour améliorer les performances

L'exemple suivant montre comment la sémantique de déplacement peut améliorer les performances de vos applications. L'exemple ajoute deux éléments à un objet vectoriel, puis insère un nouvel élément entre les deux éléments existants. La `vector` classe utilise

la sémantique de déplacement pour effectuer efficacement l'opération d'insertion en déplaçant les éléments du vecteur au lieu de les copier.

C++

```
// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
   // Create a vector object and add a few elements to it.
   vector<MemoryBlock> v;
   v.push_back(MemoryBlock(25));
   v.push_back(MemoryBlock(75));

   // Insert a new element into the second position of the vector.
   v.insert(v.begin() + 1, MemoryBlock(50));
}
```

Cet exemple produit la sortie suivante :

Output

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.
```

Avant Visual Studio 2010, cet exemple a produit la sortie suivante :

Output

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.
In ~MemoryBlock(). length = 75. Deleting resource.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In operator=(const MemoryBlock&). length = 75. Copying resource.
In operator=(const MemoryBlock&). length = 50. Copying resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.
```

La version de cet exemple qui utilise la sémantique de déplacement est plus efficace que celle qui ne l'utilise pas, car elle exécute moins d'opérations de copie, d'allocation de mémoire et de libération de mémoire.

Programmation fiable

Pour éviter les fuites de ressources, libérez toujours des ressources (telles que la mémoire, les handles de fichiers et les sockets) dans l'opérateur d'assignation de déplacement.

Pour empêcher la destruction irrécupérable des ressources, gérez correctement l'auto-assignation dans l'opérateur d'assignation de déplacement.

Si vous fournissez un constructeur de déplacement et un opérateur d'assignation de déplacement pour votre classe, vous pouvez supprimer le code redondant en écrivant le constructeur de déplacement de sorte qu'il appelle l'opérateur d'assignation de déplacement. L'exemple suivant présente une version modifiée du constructeur de déplacement qui appelle l'opérateur d'assignation de déplacement :

C++

```
// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
   : _data(nullptr)
   , _length(0)
{
   *this = std::move(other);
}
```

La fonction std::move convertit la valeur lvalue `other` en rvalue.

Voir aussi

Déclarateur de référence Rvalue : &&
std::move

Constructeurs effectuant une délégation

Article • 16/06/2023

De nombreuses classes ont plusieurs constructeurs qui effectuent des opérations similaires, par exemple, valident les paramètres :

C++

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Vous pouvez réduire le code répétitif en ajoutant une fonction qui effectue toute la validation, mais le code pour `class_c` serait plus facile à comprendre et à gérer si un constructeur pouvait déléguer une partie du travail à un autre. Pour ajouter des constructeurs de délégation, utilisez la `constructor (. . .) : constructor (. . .)` syntaxe :

C++

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
```

```
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max,
my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
}
};
int main() {

    class_c c1{ 1, 3, 2 };
}
```

Lorsque vous parcourez l'exemple précédent, notez que le constructeur `class_c(int, int, int)` appelle d'abord le constructeur `class_c(int, int)`, qui appelle `class_c(int)` à son tour . Chacun des constructeurs effectue uniquement le travail qui n'est pas effectué par les autres constructeurs.

Le premier constructeur appelé initialise l'objet afin que tous ses membres soient initialisés à ce stade. Vous ne pouvez pas effectuer l'initialisation de membre dans un constructeur qui délègue à un autre constructeur, comme illustré ici :

C++

```
class class_a {
public:
    class_a() {}
    // member initialization here, no delegate
    class_a(string str) : m_string{ str } {}

    //can't do member initialization here
    // error C3511: a call to a delegating constructor shall be the only
member-initializer
    class_a(string str, double dbl) : class_a(str) , m_double{ dbl } {}

    // only member assignment
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string;
};
```

L'exemple suivant montre l'utilisation d'initialiseurs de membres de données non statiques. Notez que si un constructeur initialise également un membre de données donné, l'initialiseur de membre est remplacé :

C++

```
class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
```

```
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }
    double m_double{ 1.0 };
    string m_string{ m_double < 10.0 ? "alpha" : "beta" };
};

int main() {
    class_a a{ "hello", 2.0 };  //expect a.m_double == 2.0, a.m_string ==
"hello"
    int y = 4;
}
```

La syntaxe de délégation du constructeur n'empêche pas la création accidentelle d'une récursion du constructeur (Constructeur1 appelle Constructeur2 qui appelle Constructeur1) et aucune erreur n'est levée tant qu'il n'y a pas de dépassement de pile. Il est de votre responsabilité d'éviter les cycles.

C++

```
class class_f{
public:
    int max;
    int min;

    // don't do this
    class_f() : class_f(6, 3){ }
    class_f(int my_max, int my_min) : class_f() { }
};
```

Destructeurs (C++)

Article • 03/04/2023

Un destructeur est une fonction membre qui est appelée automatiquement lorsque l'objet sort de l'étendue ou est explicitement détruit par un appel à `delete`. Un destructeur porte le même nom que la classe , précédé d'un tilde (`~`). Par exemple, le destructeur de la classe `String` est déclaré : `~String()`.

Si vous ne définissez pas de destructeur, le compilateur en fournit un par défaut ; pour de nombreuses classes, cela suffit. Vous devez définir un destructeur personnalisé uniquement lorsque la classe stocke des handles vers des ressources système qui doivent être libérées ou des pointeurs qui possèdent la mémoire vers laquelle ils pointent.

Prenons la déclaration suivante d'une classe `String` :

C++

```
// spec1_destructors.cpp
#include <string>

class String {
public:
   String( char *ch );  // Declare constructor
   ~String();           //  and destructor.
private:
   char    *_text;
   size_t  sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
   sizeOfText = strlen( ch ) + 1;

   // Dynamically allocate the correct amount of memory.
   _text = new char[ sizeOfText ];

   // If the allocation succeeds, copy the initialization string.
   if( _text )
      strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
   // Deallocate the memory that was previously reserved
   //  for this string.
   delete[] _text;
}
```

```
int main() {
   String str("The piper in the glen...");
}
```

Dans l'exemple précédent, le destructeur `String::~String` utilise l'opérateur **`delete`** pour libérer l'espace alloué dynamiquement pour le stockage de texte.

Déclaration de destructeurs

Les destructeurs sont des fonctions ayant le même nom que la classe, mais précédé d'un tilde (`~`).

Plusieurs règles régissent la déclaration des destructeurs. Les destructeurs :

- N'acceptez pas les arguments.
- Ne retournez pas de valeur (ou **`void`**).
- Ne peut pas être déclaré en tant que **`const`**, **`volatile`** ou **`static`**. Toutefois, ils peuvent être appelés pour la destruction d'objets déclarés comme **`const`**, **`volatile`** ou **`static`**.
- Peut être déclaré en tant que **`virtual`**. À l'aide de destructeurs virtuels, vous pouvez détruire des objets sans connaître leur type: le destructeur correct pour l'objet est appelé à l'aide du mécanisme de fonction virtuelle. Les destructeurs peuvent également être déclarés en tant que fonctions virtuelles pures pour les classes abstraites.

Utilisation de destructeurs

Les destructeurs sont appelés lorsque l'un des événements suivants se produit :

- Un objet (automatique) local avec portée de bloc passe hors de portée.
- Un objet alloué à l'aide de l'opérateur **`new`** est explicitement libéré à l'aide de **`delete`**.
- La durée de vie d'un objet temporaire se termine.
- Un programme se termine et des objets globaux ou statiques existent.
- Le destructeur est appelé explicitement à l'aide du nom complet de la fonction destructeur.

Les destructeurs peuvent librement appeler des fonctions membres de classe et accéder aux données de membres de classe.

Il existe deux restrictions sur l'utilisation des destructeurs :

- Vous ne pouvez pas prendre son adresse.
- Les classes dérivées n'héritent pas du destructeur de leur classe de base.

Ordre de destruction

Lorsqu'un objet bascule hors de portée ou est supprimé, la séquence d'événements de sa suppression complète est la suivante :

1. Le destructeur de la classe est appelé et le corps de la fonction destructeur est exécuté.
2. Les destructeurs des objets membres non statiques sont appelés dans l'ordre inverse dans lequel ils apparaissent dans la déclaration de classe. La liste d'initialisation des membres facultative utilisée dans la construction de ces membres n'affecte pas l'ordre de construction ou de destruction.
3. Les destructeurs pour les classes de base non virtuelles sont appelés dans l'ordre inverse de déclaration.
4. Les destructeurs pour les classes de base virtuelles sont appelés dans l'ordre inverse de leur déclaration.

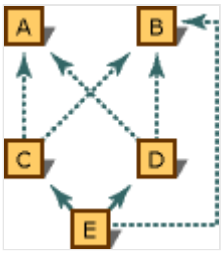
C++

```
// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
   A1 * a = new A3;
   delete a;
   printf("\n");

   B1 * b = new B3;
   delete b;
```

```
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}

Output: A3 dtor
A2 dtor
A1 dtor

B1 dtor

B3 dtor
B2 dtor
B1 dtor
```

Classes de base virtuelles

Les destructeurs pour les classes de base virtuelles sont appelés dans l'ordre inverse d'apparition dans un graphique acyclique dirigé (balayage à profondeur prioritaire, de gauche à droite, post-ordre). L'illustration suivante représente un graphique d'héritage.



L'exemple suivant répertorie les titres des classes représentées dans l'illustration.

C++

```
class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B
```

Pour déterminer l'ordre de destruction des classes de base virtuelles d'un objet de type `E`, le compilateur génère une liste en appliquant l'algorithme suivant :

1. Balayez le graphique vers la gauche, en démarrant au point le plus élevé dans le graphique (dans ce cas, `E`).

2. Exécutez des balayages vers la gauche jusqu'à ce que tous les nœuds aient été consultés. Notez le nom du nœud actuel.

3. Reprenez le nœud précédent (en bas et à droite) pour déterminer si le nœud mémorisé est une classe de base virtuelle.
4. Si le nœud mémorisé est une classe de base virtuelle, analysez la liste pour voir si elle a déjà été entrée. S'il ne s'agit pas d'une classe de base virtuelle, ignorez-la.
5. Si le nœud mémorisé ne figure pas encore dans la liste, ajoutez-le en bas de la liste.
6. Balayez le graphique vers le haut et le long du tracé suivant vers la droite.
7. Passez à l'étape 2.
8. Lorsque le dernier tracé vers le haut est écoulé, notez le nom du nœud actuel.
9. Passez à l'étape 3.
10. Continuez ainsi jusqu'à ce que le nœud inférieur soit de nouveau le nœud actuel.

Par conséquent, pour la classe `E`, l'ordre de destruction est le suivant :

1. Classe `E` de base non virtuelle .
2. Classe `D` de base non virtuelle .
3. Classe `C` de base non virtuelle .
4. Classe de base virtuelle `B`.
5. Classe de base virtuelle `A`.

Ce processus crée une liste triée d'entrées uniques. Aucun nom de classe n'apparaît deux fois. Une fois la liste construite, elle est parcourue dans l'ordre inverse, et le destructeur pour chacune des classes de la liste de la dernière à la première est appelé.

L'ordre de construction ou de destruction est principalement important lorsque les constructeurs ou les destructeurs d'une classe s'appuient sur l'autre composant créé en premier ou persistant plus longtemps, par exemple, si le destructeur pour `A` (dans la figure ci-dessus) s'appuyait sur le `B` fait que son code était toujours présent lors de l'exécution de son code, ou vice versa.

Ces interdépendances entre les classes dans un graphique d'héritage sont fondamentalement dangereuses car les classes dérivées ultérieurement peuvent modifier le tracé à l'extrême gauche, modifiant ainsi l'ordre de construction et de destruction.

Classes de base non virtuelles

Les destructeurs pour les classes de base non virtuelles sont appelés dans l'ordre inverse dans lequel les noms de classe de base sont déclarés. Prenons la déclaration de classe suivante :

```
C++

class MultInherit : public Base1, public Base2
...
```

Dans l'exemple précédent, le destructeur de `Base2` est appelé avant le destructeur de `Base1`.

Appels de destructeur explicites

Appeler un destructeur explicitement est rarement nécessaire. Toutefois, il peut être utile d'effectuer un nettoyage des objets placés à des adresses absolues. Ces objets sont généralement alloués à l'aide d'un opérateur défini par `new` l'utilisateur qui accepte un argument de placement. L'opérateur `delete` ne peut pas libérer cette mémoire, car elle n'est pas allouée à partir du magasin gratuit (pour plus d'informations, voir Les opérateurs nouveaux et supprimés). Un appel au destructeur, toutefois, permet d'effectuer un nettoyage approprié. Pour appeler explicitement le destructeur pour un objet, `s`, de classe `String`, utilisez l'une des instructions suivantes :

```
C++

s.String::~String();     // non-virtual call
ps->String::~String();   // non-virtual call

s.~String();       // Virtual call
ps->~String();     // Virtual call
```

La notation pour les appels explicites aux destructeurs, illustrée dans l'exemple précédent, peut être utilisée que le type définisse ou non un destructeur. Vous pouvez ainsi effectuer ce type d'appels explicites sans savoir si un destructeur est défini pour le type. Un appel explicite à un destructeur n'a aucun effet lorsqu'aucun destructeur n'est défini.

Programmation fiable

Une classe a besoin d'un destructeur si elle acquiert une ressource et, pour gérer la ressource en toute sécurité, elle doit probablement implémenter un constructeur de copie et une affectation de copie.

Si ces fonctions spéciales ne sont pas définies par l'utilisateur, elles sont implicitement définies par le compilateur. Les constructeurs et opérateurs d'affectation générés implicitement effectuent une copie superficielle au niveau des membres, ce qui est presque certainement incorrect si un objet gère une ressource.

Dans l'exemple suivant, le constructeur de copie généré implicitement crée les `str1.text` pointeurs et `str2.text` fait référence à la même mémoire, et lorsque nous revenons de `copy_strings()`, cette mémoire est supprimée deux fois, ce qui est un comportement non défini :

C++

```
void copy_strings()
{
   String str1("I have a sense of impending disaster...");
   String str2 = str1; // str1.text and str2.text now refer to the same
object
} // delete[] _text; deallocates the same memory twice
  // undefined behavior
```

La définition explicite d'un destructeur, d'un constructeur de copie ou d'un opérateur d'assignation de copie empêche la définition implicite du constructeur de déplacement et de l'opérateur d'affectation de déplacement. Dans ce cas, l'échec de fournir des opérations de déplacement est généralement, si la copie est coûteuse, une opportunité d'optimisation manquée.

Voir aussi

Constructeurs de copie et opérateurs
d'assignation de copie Déplacer des constructeurs et des opérateurs d'affectation de déplacement

Vue d'ensemble des fonctions membres

Article • 03/04/2023

Les fonctions membres sont statique ou non statique. Le comportement des fonctions membres statiques diffère des autres fonctions membres, car les fonctions membres statiques n'ont pas d'argument implicite `this` . Les fonctions membres non statiques ont un `this` pointeur. Les fonctions membres (statique ou non statique) peuvent être définies à l'intérieur ou à l'extérieur de la déclaration de classe.

Si une fonction membre est définie dans une déclaration de classe, elle est traitée comme une fonction inline, et il est inutile de qualifier le nom de fonction avec son nom de classe. Bien que les fonctions définies à l'intérieur des déclarations de classe soient déjà traitées comme des fonctions inline, vous pouvez utiliser la mot clé pour documenter le `inline` code.

Un exemple de déclaration d'une fonction dans une déclaration de classe suit :

C++

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    //  of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{
}
```

Si la définition d'une fonction membre est en dehors de la déclaration de classe, elle est traitée comme une fonction inline uniquement si elle est explicitement déclarée en tant que `inline`. En outre, le nom de fonction dans la définition doit être qualifié avec son nom de classe par l'opérateur de résolution de portée (`::`).

L'exemple suivant est identique à la déclaration précédente de la classe `Account`, sauf que la fonction `Deposit` est définie à l'extérieur de la déclaration de classe :

C++

```cpp
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

ⓘ Notes

Bien que les fonctions membres puissent être définies dans une déclaration de classe ou séparément, aucune fonction membre ne peut être ajoutée à une classe après la classe définie.

Les classes contenant des fonctions membres peuvent avoir plusieurs déclarations, mais les fonctions membres elles-mêmes doivent avoir une seule définition dans un programme. Plusieurs définitions génèrent un message d'erreur au moment de la liaison. Si une classe contient des définitions de fonction inline, les définitions de fonction doivent être identiques pour observer cette règle « d'une définition ».

Spécificateur virtuel

Article • 03/04/2023

Le mot clé virtuel ne peut être appliqué qu’aux fonctions membres de classe non statiques. Cela signifie que la liaison des appels à la fonction est différée jusqu'au moment de l'exécution. Pour plus d’informations, consultez Fonctions virtuelles.

Spécificateur de substitution

Article • 16/06/2023

Vous pouvez utiliser le mot clé **de remplacement** pour désigner des fonctions membres qui remplacent une fonction virtuelle dans une classe de base.

Syntaxe

```
function-declaration override;
```

Notes

la substitution respecte le contexte et a une signification particulière uniquement lorsqu'elle est utilisée après une déclaration de fonction membre ; sinon, ce n'est pas une mot clé réservée.

Exemple

Utilisez **la substitution** pour empêcher le comportement d'héritage par inadvertance dans votre code. L'exemple suivant montre où, sans utiliser **la substitution**, le comportement de la fonction membre de la classe dérivée n'a peut-être pas été prévu. Le compilateur n'émet pas d'erreurs pour ce code.

C++

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const, so it does
not
                          // override BaseClass::funcB() const and it is a
new member function
```

```
    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a
different
                                      // parameter type than
BaseClass::funcC(int), so
                                      // DerivedClass::funcC(double) is a
new member function
};
```

Lorsque vous utilisez **la substitution**, le compilateur génère des erreurs au lieu de créer silencieusement de nouvelles fonctions membres.

C++

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB()
does not
                                   // override BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override; // compiler error:
                                                 //
DerivedClass::funcC(double) does not
                                                 // override
BaseClass::funcC(int)

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                           // override the non-virtual BaseClass::funcD()
};
```

Pour spécifier que les fonctions ne peuvent pas être remplacées et que les classes ne peuvent pas être héritées, utilisez la mot clé finale.

Voir aussi

spécificateur final

Mots clés

Spécificateur final

Article • 16/06/2023

Vous pouvez utiliser la mot clé **finale** pour désigner des fonctions virtuelles qui ne peuvent pas être substituées dans une classe dérivée. Vous pouvez également l'utiliser pour désigner les classes qui ne peuvent pas être héritées.

Syntaxe

```
function-declaration final;
class class-name final base-classes
```

Notes

final respecte le contexte et a une signification particulière uniquement lorsqu'il est utilisé après une déclaration de fonction ou un nom de classe ; sinon, ce n'est pas un mot clé réservé.

Lorsque **final** est utilisé dans les déclarations de classe, `base-classes` est une partie facultative de la déclaration.

Exemple

L'exemple suivant utilise la mot clé **finale** pour spécifier qu'une fonction virtuelle ne peut pas être remplacée.

C++

```
class BaseClass
{
    virtual void func() final;
};

class DerivedClass: public BaseClass
{
    virtual void func(); // compiler error: attempting to
                         // override a final function
};
```

Pour plus d'informations sur la façon de spécifier que les fonctions membres peuvent être remplacées, consultez Spécificateur de remplacement.

L'exemple suivant utilise la mot clé **finale** pour spécifier qu'une classe ne peut pas être héritée.

C++

```
class BaseClass final
{
};

class DerivedClass: public BaseClass // compiler error: BaseClass is
                                     // marked as non-inheritable
{
};
```

Voir aussi

Mots clés
spécificateur de remplacement

Héritage (C++)

Article • 03/04/2023

Cette section explique comment utiliser les classes dérivées pour générer des programmes extensibles.

Vue d'ensemble

De nouvelles classes peuvent être dérivées de classes existantes à l'aide d'un mécanisme appelé « héritage » (consultez les informations commençant dans Héritage unique). Les classes utilisées pour la dérivation sont appelées classes de base d'une classe dérivée particulière. Une classe dérivée est déclarée en utilisant la syntaxe suivante :

C++

```
class Derived : [virtual] [access-specifier] Base
{
   // member list
};
class Derived : [virtual] [access-specifier] Base1,
   [virtual] [access-specifier] Base2, . . .
{
   // member list
};
```

Après l'étiquette (nom) de la classe, un signe deux-points apparaît, suivi d'une liste de spécifications de base. Les classes de base ainsi nommées doivent avoir été déclarées précédemment. Les spécifications de base peuvent contenir un spécificateur d'accès, qui est l'un des mots clés `public`, `protected` ou `private`. Ces spécificateurs d'accès apparaissent avant le nom de classe de base et s'appliquent uniquement à cette classe de base. Ces spécificateurs contrôlent l'autorisation de la classe dérivée à utiliser pour les membres de la classe de base. Pour plus d'informations sur l'accès aux membres de la classe de base, consultez Access Control membre. Si le spécificateur d'accès est omis, l'accès à cette base est considéré comme `private`. Les spécifications de base peuvent contenir les mot clé `virtual` pour indiquer l'héritage virtuel. Ce mot clé peut apparaître avant ou après le spécificateur d'accès, le cas échéant. Si l'héritage virtuel est utilisé, la classe de base est appelée classe de base virtuelle.

Plusieurs classes de base peuvent être spécifiées en les séparant par des virgules. Si une seule classe de base est spécifiée, le modèle d'héritage est Héritage unique. Si plusieurs classes de base sont spécifiées, le modèle d'héritage est appelé Héritage multiple.

Cet article contient les rubriques suivantes :

- Héritage simple
- Plusieurs classes de base
- Fonctions virtuelles
- Substitutions explicites
- Classes abstraites
- Résumé des règles d'étendue

Les mots clés __super et __interface sont documentés dans cette section.

Voir aussi

Informations de référence sur le langage C++

Fonctions virtuelles

Article • 03/04/2023

Une fonction virtuelle est une fonction membre dont vous souhaitez la redéfinition dans des classes dérivées. Lorsque vous faites référence à un objet de classe dérivée à l'aide d'un pointeur ou d'une référence à la classe de base, vous pouvez appeler une fonction virtuelle pour cet objet et exécuter la version de la classe dérivée de la fonction.

Les fonctions virtuelles garantissent que la fonction correcte est appelée pour un objet, quelle que soit l'expression utilisée pour créer l'appel de fonction.

Supposons qu'une classe de base contienne une fonction déclarée comme virtuelle et qu'une classe dérivée définit la même fonction. La fonction de la classe dérivée est appelée pour les objets de la classe dérivée, même si elle est appelée à l'aide d'un pointeur ou d'une référence à la classe de base. L'exemple suivant montre une classe de base qui fournit une implémentation de la fonction `PrintBalance` et de deux classes dérivées.

C++

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
   Account( double d ) { _balance = d; }
   virtual ~Account() {}
   virtual double GetBalance() { return _balance; }
   virtual void PrintBalance() { cerr << "Error. Balance not available for
base type." << endl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
   CheckingAccount(double d) : Account(d) {}
   void PrintBalance() { cout << "Checking account balance: " <<
GetBalance() << endl; }
};

class SavingsAccount : public Account {
public:
   SavingsAccount(double d) : Account(d) {}
   void PrintBalance() { cout << "Savings account balance: " <<
GetBalance(); }
```

```
};

int main() {
   // Create objects of type CheckingAccount and SavingsAccount.
   CheckingAccount checking( 100.00 );
   SavingsAccount  savings( 1000.00 );

   // Call PrintBalance using a pointer to Account.
   Account *pAccount = &checking;
   pAccount->PrintBalance();

   // Call PrintBalance using a pointer to Account.
   pAccount = &savings;
   pAccount->PrintBalance();
}
```

Dans le code précédent, les appels à `PrintBalance` sont identiques, sauf pour l'objet vers lequel `pAccount` pointe. Étant donné que `PrintBalance` est virtuelle, c'est la version de la fonction définie pour chaque objet qui est appelée. La fonction `PrintBalance` dans les classes dérivées `CheckingAccount` et `SavingsAccount` « remplacent » la fonction dans la classe de base `Account`.

Si une classe qui ne fournit pas d'implémentation de substitution de la fonction `PrintBalance` est déclarée, c'est l'implémentation par défaut de la classe de base `Account` qui est utilisée.

Les fonctions dans les classes dérivées remplacent les fonctions virtuelles dans les classes de base uniquement si leur type est identique. Une fonction dans une classe dérivée ne peut pas différer d'une fonction virtuelle dans une classe de base uniquement par son type de retour ; la liste d'arguments doit différer également.

Lors de l'appel d'une fonction via des pointeurs ou des références, les règles suivantes s'appliquent :

- Un appel à une fonction virtuelle est résolu en fonction du type sous-jacent d'objet pour lequel cette fonction est appelée.
- Un appel à une fonction non virtuelle est résolu en fonction du type du pointeur ou de la référence.

L'exemple suivant montre comment les fonctions virtuelles et non virtuelles se comportent lorsqu'elles sont appelées via des pointeurs :

C++

```
// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
```

```
#include <iostream>
using namespace std;

class Base {
public:
   virtual void NameOf();   // Virtual function.
   void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
   cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
   cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
   void NameOf();   // Virtual function.
   void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
   cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
   cout << "Invoked by Derived\n";
}

int main() {
   // Declare an object of type Derived.
   Derived aDerived;

   // Declare two pointers, one of type Derived * and the other
   //  of type Base *, and initialize them to point to aDerived.
   Derived *pDerived = &aDerived;
   Base    *pBase    = &aDerived;

   // Call the functions.
   pBase->NameOf();           // Call virtual function.
   pBase->InvokingClass();    // Call nonvirtual function.
   pDerived->NameOf();        // Call virtual function.
   pDerived->InvokingClass(); // Call nonvirtual function.
}
```

Output

```
Derived::NameOf
Invoked by Base
```

```
Derived::NameOf
Invoked by Derived
```

Comme vous pouvez le constater, que la fonction `NameOf` soit appelée via un pointeur vers `Base` ou un pointeur vers `Derived`, c'est la fonction pour `Derived` qui est appelée. La fonction pour `Derived` est appelée car `NameOf` est une fonction virtuelle, et `pBase` et `pDerived` pointent vers un objet de type `Derived`.

Étant donné que les fonctions virtuelles sont appelées uniquement pour les objets de types de classe, vous ne pouvez pas déclarer de fonctions globales ou statiques en tant que **`virtual`**.

Le **`virtual`** mot clé peut être utilisé lors de la déclaration de fonctions de substitution dans une classe dérivée, mais il n'est pas nécessaire ; les remplacements de fonctions virtuelles sont toujours virtuels.

Les fonctions virtuelles d'une classe de base doivent être définies, sauf si elles sont déclarées à l'aide du *spécificateur pur*. (Pour plus d'informations sur les fonctions virtuelles pures, consultez Classes abstraites.)

Le mécanisme d'appel de fonction virtuelle peut être supprimé en qualifiant explicitement le nom de fonction à l'aide de l'opérateur de résolution de portée (`::`). Prenons l'exemple précédent qui implique la classe `Account`. Pour appeler `PrintBalance` dans la classe de base, utilisez le code suivant :

C++

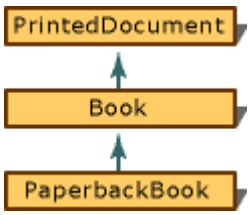
```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );

pChecking->Account::PrintBalance();  //  Explicit qualification.

Account *pAccount = pChecking;  // Call Account::PrintBalance

pAccount->Account::PrintBalance();   //  Explicit qualification.
```

Les deux appels à `PrintBalance` dans l'exemple précédent suppriment le mécanisme d'appel de fonction virtuelle.

Héritage simple

Article • 03/04/2023

Dans « l'héritage unique », une forme courante d'héritage, les classes possèdent une seule classe de base. Considérons la relation illustrée à la figure ci-dessous.



Graphique simple illustrant un héritage unique

Notez la progression d'un niveau général à un niveau spécifique dans la figure. Un autre attribut courant figurant dans la conception de la plupart des hiérarchies de classes est que la classe dérivée possède une « sorte » de relation avec la classe de base. Dans la figure ci-dessus, un objet `Book` est une sorte d'objet `PrintedDocument` et un objet `PaperbackBook` est une sorte d'objet `book`.

Autre élément à retenir dans la figure : `Book` est à la fois une classe dérivée (de `PrintedDocument`) et une classe de base (`PaperbackBook` est dérivé de `Book`). Le squelette de la déclaration d'une telle hiérarchie de classes est illustré dans l'exemple suivant :

C++

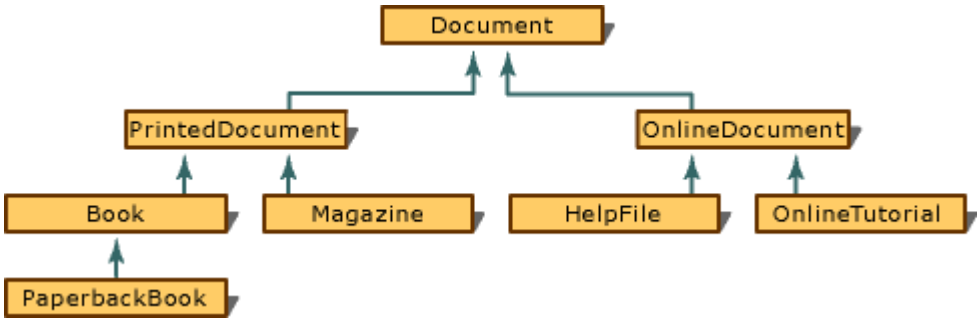
```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

`PrintedDocument` est considéré comme une classe « de base directe » pour `Book` ; c'est une classe « de base indirecte » pour `PaperbackBook`. La différence tient au fait qu'une classe de base directe figure dans la liste de base d'une déclaration de classe, alors qu'une classe de base indirecte n'y figure pas.

La classe de base dont chaque classe est dérivée est déclarée avant la déclaration de la classe dérivée. Il ne suffit pas de fournir une déclaration de référence avant pour une classe de base ; il faut fournir une déclaration complète.

Dans l'exemple précédent, le spécificateur `public` d'accès est utilisé. La signification de l'héritage public, protégé et privé est décrite dans Access Control membres.

Une classe peut servir de classe de base pour de nombreuses classes spécifiques, comme illustré à la figure suivante.



Exemple de graphique acyclique dirigé

Dans le diagramme ci-dessus, appelé « graphique acyclique dirigé », certaines classes sont des classes de base pour plusieurs classes dérivées. Toutefois, l'inverse n'est pas vrai : il existe une seule classe de base directe pour chaque classe dérivée. Le graphique de la figure représente une structure « d'héritage unique ».

> ⓘ **Notes**
>
> Les graphiques acycliques dirigés ne sont pas propres à l'héritage unique. Ils sont également utilisés pour représenter des cas d'héritage multiple.

Dans un héritage, la classe dérivée contient les membres de la classe de base, ainsi que tous les nouveaux membres que vous ajoutez. Par conséquent, une classe dérivée peut faire référence à des membres de la classe de base (à moins que ces membres soient redéfinis dans la classe dérivée). L'opérateur de résolution de portée (`::`) peut être utilisé pour faire référence à des membres de classes de base directes ou indirectes, lorsque ces membres ont été redéfinis dans la classe dérivée. Examinez cet exemple :

C++

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
   char *Name;   // Document name.
   void PrintNameOf();   // Print name.
};

// Implementation of PrintNameOf function from class Document.
```

```
void Document::PrintNameOf() {
   cout << Name << endl;
}

class Book : public Document {
public:
   Book( char *name, long pagecount );
private:
   long  PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
   Name = new char[ strlen( name ) + 1 ];
   strcpy_s( Name, strlen(Name), name );
   PageCount = pagecount;
};
```

Notez que le constructeur de `Book`, (`Book::Book`), a accès aux données membres, `Name`. Dans un programme, un objet de type `Book` peut être créé et utilisé comme suit :

C++

```
//  Create a new object of type Book. This invokes the
//   constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...

//  Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

Comme le montre l'exemple précédent, le membre de classe et les données et fonctions héritées sont utilisés de façon identique. Si l'implémentation de la classe `Book` demande une réimplémentation de la fonction `PrintNameOf`, la fonction qui appartient à la classe `Document` peut être appelée uniquement à l'aide de l'opérateur de résolution de portée (`::`) :

C++

```
// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
   char *Name;          // Document name.
   void  PrintNameOf() {}  // Print name.
```

```
};

class Book : public Document {
   Book( char *name, long pagecount );
   void PrintNameOf();
   long  PageCount;
};

void Book::PrintNameOf() {
   cout << "Name of book: ";
   Document::PrintNameOf();
}
```

Les pointeurs et les références des classes dérivées peuvent être convertis implicitement en pointeurs et références de leurs classes de base, s'il existe une classe de base accessible et non équivoque. Le code suivant illustre ce concept à l'aide de pointeurs (le même principe s'applique aux références) :

C++

```
// deriv_SingleInheritance4.cpp
// compile with: /W3
struct Document {
   char *Name;
   void PrintNameOf() {}
};

class PaperbackBook : public Document {};

int main() {
   Document * DocLib[10];   // Library of ten documents.
   for (int i = 0 ; i < 5 ; i++)
      DocLib[i] = new Document;
   for (int i = 5 ; i < 10 ; i++)
      DocLib[i] = new PaperbackBook;
}
```

Dans l'exemple précédent, des types différents sont créés. Toutefois, comme ces types sont tous dérivés de la classe `Document`, il existe une conversion implicite vers `Document *`. Par conséquent, `DocLib` est une « liste hétérogène » (une liste dans laquelle tous les objets ne sont pas du même type) contenant différents types d'objets.

Comme la classe `Document` possède une fonction `PrintNameOf`, elle peut imprimer le nom de chaque ouvrage de la bibliothèque, mais elle peut omettre certaines informations spécifiques au type de document (nombre de pages pour `Book`, nombre d'octets pour `HelpFile`, etc.).

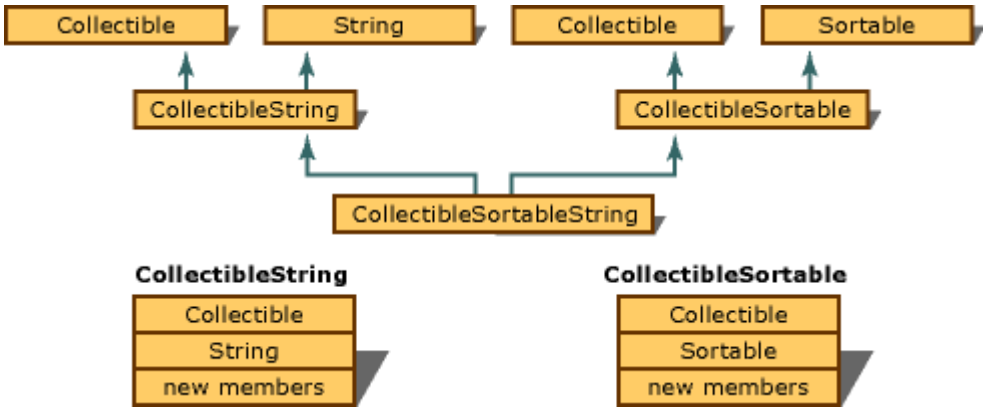
ⓘ Notes

Obliger la classe de base à implémenter une fonction telle que `PrintNameOf` est rarement le meilleur choix de conception. **Virtual Functions** offre d'autres alternatives de conception.

Classes de base

Article • 16/06/2023

Le processus d'héritage crée une classe dérivée qui se compose des membres des classes de base et tous les nouveaux membres ajoutés par la classe dérivée. Dans un héritage multiple, il est possible de construire un graphique d'héritage, où la même classe de base fait partie de plusieurs classes dérivées. L'illustration suivante montre ce graphique.



Plusieurs instances d’une seule classe de base

Dans l'illustration, les représentations illustrées des composants de `CollectibleString` et de `CollectibleSortable` sont affichés. Toutefois, la classe de base, `Collectible`, est dans `CollectibleSortableString` via le chemin d’accès `CollectibleString` et le chemin d’accès `CollectibleSortable`. Pour éliminer cette redondance, ces classes peuvent être déclarées comme classes de base virtuelles lorsqu'elles sont héritées.

Plusieurs classes de base

Article • 11/04/2023

Une classe peut être dérivée de plusieurs classes de base. Dans un modèle à héritage multiple (où les classes sont dérivées de plusieurs classes de base), les classes de base sont spécifiées à l'aide de l'élément de grammaire *de liste de base* . Par exemple, la déclaration de classe pour `CollectionOfBook`, dérivé de `Collection` et `Book`, peut être spécifiée :

C++

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

L'ordre dans lequel les classes de base sont spécifiées n'est pas significatif, sauf dans certains cas où les constructeurs et les destructeurs sont appelés. Dans ces cas, l'ordre dans lequel les classes de base sont spécifiées a une incidence sur ce qui suit :

- Ordre dans lequel les constructeurs sont appelés. Si votre code repose sur la partie `Book` de `CollectionOfBook` à initialiser avant la partie `Collection`, l'ordre des spécifications est important. L'initialisation a lieu dans l'ordre dans lequel les classes sont spécifiées dans la *liste de base*.

- L'ordre dans lequel les destructeurs sont appelés pour effectuer le nettoyage. Là encore, si un « élément » particulier de la classe doit être présent lorsque l'autre élément est détruit, l'ordre a une importance. Les destructeurs sont appelés dans l'ordre inverse des classes spécifiées dans la *liste de base*.

  > ⓘ **Notes**
  >
  > L'ordre de spécification des classes de base peut avoir une incidence sur la disposition de mémoire de la classe. Ne prenez aucune décision de programmation concernant l'ordre des membres de base en mémoire.

Lorsque vous spécifiez la *liste de base*, vous ne pouvez pas spécifier le même nom de classe plusieurs fois. Toutefois, il est possible qu'une classe soit une base indirecte d'une

classe dérivée plusieurs fois.

Classes de base virtuelles

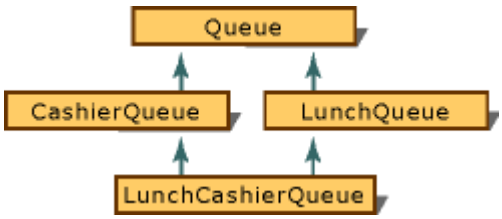
Comme une classe peut être une classe de base indirecte à une classe dérivée plusieurs fois, C++ offre un moyen d'optimiser la façon dont fonctionnent de telles classes de base. Les classes de base virtuelles permettent d'économiser de l'espace et d'éviter les ambiguïtés dans les hiérarchies de classes qui utilisent un héritage multiple.

Chaque objet non virtuel contient une copie des données membres définies dans la classe de base. Cette duplication gaspille de l'espace et vous oblige à spécifier quelle copie des membres de classe de base vous souhaitez chaque fois que vous accédez à eux.

Lorsqu'une classe de base est spécifiée comme base virtuelle, elle peut agir comme base indirecte plusieurs fois sans duplication de ses données membres. Une copie unique de ses données membres est partagée par toutes les classes de base qui l'utilisent comme base virtuelle.

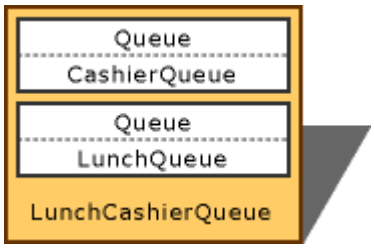
Lors de la déclaration d'une classe de base virtuelle, le `virtual` mot clé apparaît dans les listes de base des classes dérivées.

Considérez la hiérarchie de classes dans la figure suivante, qui illustre une ligne de déjeuner simulée :



Graphique de ligne de déjeuner simulé

Dans la figure, `Queue` représente la classe de base de `CashierQueue` et `LunchQueue`. Toutefois, lorsque les deux classes sont combinées pour former `LunchCashierQueue`, le problème suivant survient : la nouvelle classe contient deux sous-objets de type `Queue`, l'un provenant de `CashierQueue` et l'autre de `LunchQueue`. La figure suivante illustre la disposition de la mémoire conceptuelle (la disposition de mémoire réelle peut être optimisée) :



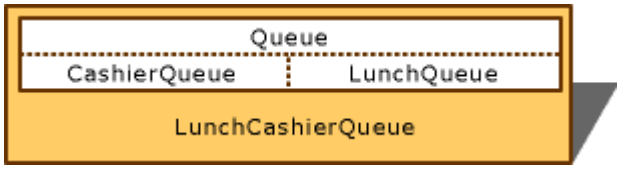
Objet Lunch-Line simulé

Il existe deux `Queue` sous-objets dans l'objet `LunchCashierQueue` . Le code suivant déclare `Queue` en tant que classe de base virtuelle :

C++

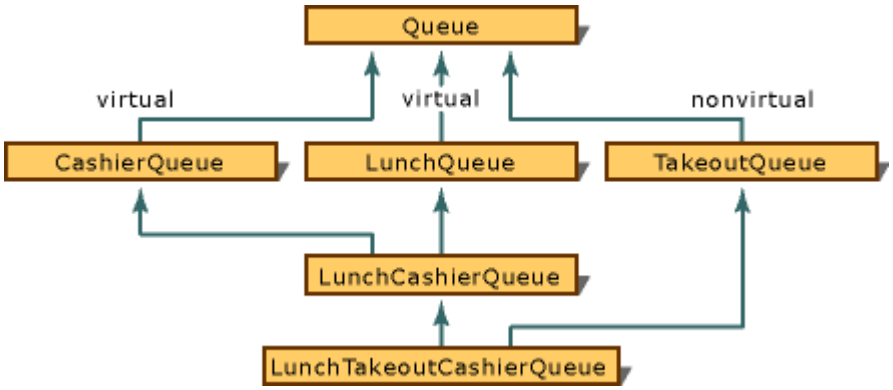
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

Le `virtual` mot clé garantit qu'une seule copie du sous-objet `Queue` est incluse (voir la figure suivante).



Objet lunch-line simulé avec classes de base virtuelles

Une classe peut avoir à la fois un composant virtuel et un composant non virtuel d'un type donné. Cela se produit dans les conditions illustrées dans la figure suivante :



Composants virtuels et non virtuels de la même classe

Dans cette figure, `CashierQueue` et `LunchQueue` utilisent `Queue` comme classe de base virtuelle. Toutefois, `TakeoutQueue` spécifie `Queue` en tant que classe de base, et non pas

comme classe de base virtuelle. Par conséquent, `LunchTakeoutCashierQueue` a deux sous-objets de type `Queue` : l'un provenant du chemin d'héritage qui inclut `LunchCashierQueue` et l'autre provenant du chemin qui inclut `TakeoutQueue`. La figure ci-dessous illustre cela.

Disposition d'objet avec héritage virtuel et non virtuel

> ⓘ **Notes**
>
> L'héritage virtuel fournit des avantages de taille significatifs par rapport à l'héritage non virtuel. Toutefois, il peut introduire une certaine surcharge de traitement.

Si une classe dérivée remplace une fonction virtuelle qu'elle hérite d'une classe de base virtuelle, et si un constructeur ou un destructeur pour la classe de base dérivée appelle cette fonction à l'aide d'un pointeur vers la classe de base virtuelle, le compilateur peut introduire d'autres champs « vtordisp » masqués dans les classes avec des bases virtuelles. L'option `/vd0` du compilateur supprime l'ajout du membre de déplacement du constructeur/destructeur vtordisp masqué. L'option `/vd1` du compilateur, par défaut, les active là où elles sont nécessaires. Désactivez vtordisps uniquement si vous êtes sûr que tous les constructeurs et destructeurs de classe appellent virtuellement des fonctions virtuelles.

L'option `/vd` du compilateur affecte un module de compilation entier. Utilisez le `vtordisp` pragma pour supprimer, puis réactiver `vtordisp` les champs classe par classe :

C++

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
\#pragma vtordisp( on )
```

Ambiguïtés au niveau du nom

L'héritage multiple présente le risque que les noms soient hérités le long de plusieurs chemins. Les noms des membres de la classe le long de ces chemins ne sont pas nécessairement uniques. Ces conflits de nom sont appelés « ambiguïtés. »

Toute expression qui fait référence à un membre de classe doit faire une référence qui ne soit pas ambigu. L'exemple suivant montre comment développer les ambiguïtés :

C++

```
// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a();  // class A also has a member "a"
    int b();       //  and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};
```

Compte tenu des déclarations de classe précédentes, le code tel que le suivant est ambigu, car il n'est pas clair si `b` fait référence au `b` dans `A` ou dans `B`:

C++

```
C *pc = new C;

pc->b();
```

Prenons l'exemple précédent. Étant donné que le nom `a` est membre de la classe `A` et de la classe `B`, le compilateur ne peut pas discerner qui `a` désigne la fonction à appeler. L'accès à un membre est ambigu s'il peut faire référence à plusieurs fonctions, objets, types ou énumérateurs.

Le compilateur détecte les ambiguïtés en exécutant des tests dans cet ordre :

1. Si l'accès au nom est ambigu (comme il vient d'être décrit), un message d'erreur est généré.

2. Si les fonctions surchargées sont non ambiguës, elles sont résolues.

3. Si l'accès au nom ne respecte pas l'autorisation accès-membre, un message d'erreur est généré. (Pour plus d'informations, consultez Member-Access Control.)

Lorsqu'une expression génère une ambiguïté par héritage, vous pouvez manuellement la résoudre en qualifiant le nom en question avec son nom de classe. Pour que l'exemple précédent se compile correctement sans ambiguïtés, utilisez par exemple ce code :

C++

```
C *pc = new C;

pc->B::a();
```

> ⓘ **Notes**
>
> Lorsque `C` est déclaré, il a la possibilité de provoquer des erreurs lorsque `B` est référencé dans la portée de `C`. Aucune erreur n'est publiée, toutefois, jusqu'à ce qu'une référence non qualifiée à `B` soit effectuée en réalité dans la portée de `C`.

Dominance

il est possible d'atteindre plusieurs noms (fonction, objet ou énumérateur) via un graphe d'héritage. Ces cas sont jugés équivoques par rapport aux classes de base non virtuelles. Elles sont également ambiguës avec les classes de base virtuelles, sauf si l'un des noms « domine » les autres.

Un nom domine un autre nom s'il est défini dans les deux classes et si une classe est dérivée de l'autre. Le nom dominant est le nom figurant dans la classe dérivée. Ce nom est utilisé lorsqu'une ambiguïté serait autrement apparue, comme indiqué dans l'exemple suivant :

C++

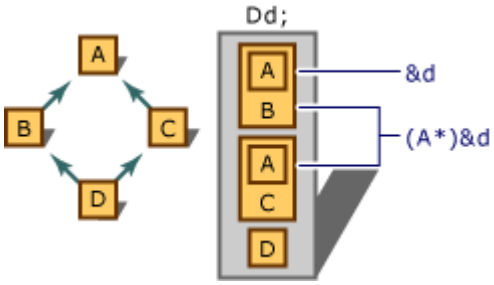
```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

class C : public virtual A {};
```

```
class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

Conversions ambiguës

Les conversions explicites et implicites à partir de pointeurs ou de références vers des types classe peuvent provoquer des ambiguïtés. L'illustration suivante, Conversion ambiguë des pointeurs vers des classes de base, montre ce qui suit :

- La déclaration d'un objet de type `D`.
- Effet de l'application de l'opérateur d'adresse (**&**) à cet objet. L'opérateur d'adresse de fournit toujours l'adresse de base de l'objet.
- L'effet de la conversion explicite du pointeur obtenu à l'aide de l'opérateur d'adresse vers le type de classe de base `A`. Le fait de contraindre l'adresse de l'objet à taper `A*` ne fournit pas toujours au compilateur suffisamment d'informations sur le sous-objet de type `A` à sélectionner ; dans ce cas, il existe deux sous-objets.



Conversion ambiguë de pointeurs vers les classes de base

La conversion en type `A*` (pointeur vers `A`) est ambiguë, car il n'existe aucun moyen de déterminer quel sous-objet de type `A` est le bon. Vous pouvez éviter l'ambiguïté en spécifiant explicitement le sous-objet que vous voulez utiliser, comme suit :

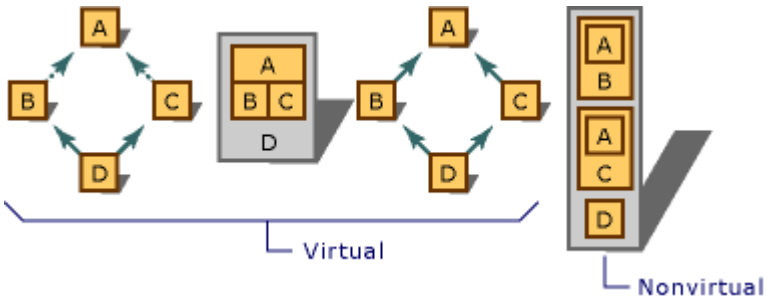
C++

```
(A *)(B *)&d       // Use B subobject.
(A *)(C *)&d       // Use C subobject.
```

Ambiguïtés et classes de base virtuelles

Si des classes de base virtuelles sont utilisées, les fonctions, les objets, les types et les énumérateurs sont accessibles via plusieurs chemins d'héritage. Étant donné qu'il n'existe qu'une seule instance de la classe de base, il n'y a aucune ambiguïté lors de l'accès à ces noms.

L'illustration suivante montre comment les objets sont composés à l'aide de l'héritage virtuel et non virtuel.



Dérivation virtuelle et non virtuelle

Dans l'illustration, accéder à un membre de classe `A` via des classes de base non virtuelles provoque une ambiguïté ; le compilateur ne propose aucune information indiquant s'il convient d'utiliser le sous-objet associé à `B` ou le sous-objet associé à `C`. Toutefois, lorsque `A` est spécifié en tant que classe de base virtuelle, il n'est pas question de déterminer quel sous-objet est accessible.

Voir aussi

Héritage

Substitutions explicites (C++)

Article • 16/06/2023

Section spécifique à Microsoft

Si la même fonction virtuelle est déclarée dans plusieurs interfaces et si une classe est dérivée de ces interfaces, vous pouvez remplacer explicitement chaque fonction virtuelle.

Pour plus d'informations sur les remplacements explicites dans le code managé à l'aide de C++/CLI, consultez Remplacements explicites.

FIN de la section spécifique à Microsoft

Exemple

L'exemple de code suivant montre comment utiliser les substitutions explicites :

C++

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
   void mf1();
   void mf1(int);
   void mf2();
   void mf2(int);
};

__interface IMyInt2 {
   void mf1();
   void mf1(int);
   void mf2();
   void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
   void IMyInt1::mf1() {
      printf_s("In CMyClass::IMyInt1::mf1()\n");
   }

   void IMyInt1::mf1(int) {
      printf_s("In CMyClass::IMyInt1::mf1(int)\n");
   }
```

```
   void IMyInt1::mf2();
   void IMyInt1::mf2(int);

   void IMyInt2::mf1() {
      printf_s("In CMyClass::IMyInt2::mf1()\n");
   }

   void IMyInt2::mf1(int) {
         printf_s("In CMyClass::IMyInt2::mf1(int)\n");
   }

   void IMyInt2::mf2();
   void IMyInt2::mf2(int);
};

void CMyClass::IMyInt1::mf2() {
   printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
   printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
   printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
   printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
   IMyInt1 *pIMyInt1 = new CMyClass();
   IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

   pIMyInt1->mf1();
   pIMyInt1->mf1(1);
   pIMyInt1->mf2();
   pIMyInt1->mf2(2);
   pIMyInt2->mf1();
   pIMyInt2->mf1(3);
   pIMyInt2->mf2();
   pIMyInt2->mf2(4);

   // Cast to a CMyClass pointer so that the destructor gets called
      CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
      delete p;
}
```

Output

```
In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
```

```
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)
```

Voir aussi

Héritage

Classes abstraites (C++)

Article • 03/04/2023

Les classes abstraites agissent comme des expressions de concepts généraux dont des classes plus spécifiques peuvent être dérivées. Vous ne pouvez pas créer un objet d'un type de classe abstraite. Toutefois, vous pouvez utiliser des pointeurs et des références vers des types de classes abstraites.

Vous créez une classe abstraite en déclarant au moins une fonction membre virtuelle pure. Il s'agit d'une fonction virtuelle déclarée à l'aide de la syntaxe du spécificateur *pur* (`= 0`). Les classes dérivées de la classe abstraite doivent implémenter la fonction virtuelle pure, sinon elles aussi sont des classes abstraites.

Prenons l'exemple présenté dans Fonctions virtuelles. L'objectif de la classe `Account` est de fournir une fonctionnalité générale, mais les objets de type `Account` sont trop généraux pour être utiles. Cela signifie qu'est `Account` un bon candidat pour une classe abstraite :

C++

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
   Account( double d );   // Constructor.
   virtual double GetBalance();   // Obtain balance.
   virtual void PrintBalance() = 0;   // Pure virtual function.
private:
    double _balance;
};
```

La seule différence entre cette déclaration et la précédente vient du fait que `PrintBalance` est déclaré avec le spécificateur pure (`= 0`).

Restrictions sur les classes abstraites

Les classes abstraites ne peuvent pas être utilisées pour :

- Variables ou données membres
- Types d'arguments
- Types de retour de fonction

- Types de conversions explicites

Si le constructeur d'une classe abstraite appelle une fonction virtuelle pure, directement ou indirectement, le résultat n'est pas défini. Toutefois, les constructeurs et les destructeurs des classes abstraites peuvent appeler d'autres fonctions membres.

Fonctions virtuelles pures définies

Les fonctions virtuelles pures dans des classes abstraites peuvent être *définies* ou avoir une implémentation. Vous pouvez uniquement appeler de telles fonctions à l'aide de la syntaxe complète :

abstract-class-name::*function-name*()

Les fonctions virtuelles pures définies sont utiles lorsque vous concevez des hiérarchies de classes dont les classes de base incluent des destructeurs virtuels purs. Cela est dû au fait que les destructeurs de classe de base sont toujours appelés pendant la destruction d'objets. Prenons l'exemple suivant :

C++

```
// deriv_RestrictionsOnUsingAbstractClasses.cpp
// Declare an abstract base class with a pure virtual destructor.
// It's the simplest possible abstract class.
class base
{
public:
    base() {}
    // To define the virtual destructor outside the class:
    virtual ~base() = 0;
    // Microsoft-specific extension to define it inline:
//  virtual ~base() = 0 {};
};

base::~base() {} // required if not using Microsoft extension

class derived : public base
{
public:
    derived() {}
    ~derived() {}
};

int main()
{
    derived aDerived; // destructor called when it goes out of scope
}
```

L'exemple montre comment une extension de compilateur Microsoft vous permet d'ajouter une définition inline à des données virtuelles `~base()` pures. Vous pouvez également le définir en dehors de la classe à l'aide de `base::~base() {}`.

Lorsque l'objet `aDerived` sort de l'étendue, le destructeur de la classe `derived` est appelé. Le compilateur génère du code pour appeler implicitement le destructeur pour la classe `base` après le `derived` destructeur. L'implémentation vide de la fonction `~base` virtuelle pure garantit qu'au moins une implémentation existe pour la fonction. Sans elle, l'éditeur de liens génère une erreur de symbole externe non résolue pour l'appel implicite.

> ⓘ **Notes**
>
> Dans l'exemple précédent, la fonction virtuelle pure `base::~base` est appelée implicitement depuis `derived::~derived`. Il est également possible d'appeler explicitement des fonctions virtuelles pures à l'aide d'un nom de fonction membre complet. Ces fonctions doivent avoir une implémentation, sinon l'appel génère une erreur au moment du lien.

Voir aussi

Héritage

Résumé des règles de portée

Article • 03/04/2023

L'utilisation d'un nom doit être non équivoque dans sa portée (jusqu'au point où la surcharge est déterminée). Si le nom indique une fonction, celle-ci doit être non équivoque en ce qui concerne le nombre et le type de paramètres. Si le nom reste sans ambiguïté, des règles d'accès aux membres sont appliquées.

Initialiseurs de constructeur

Les initialiseurs de constructeur sont évalués dans l'étendue du bloc le plus externe du constructeur pour lequel ils sont spécifiés. Par conséquent, ils peuvent utiliser les noms de paramètres du constructeur.

Noms globaux

Un nom d'objet, de fonction ou d'énumérateur est global s'il est placé en dehors de toute fonction ou classe ou préfixé par l'opérateur de portée unaire globale (`::`), et s'il n'est pas utilisé conjointement avec l'un des opérateurs binaires suivants :

- Résolution de portée (`::`)
- Sélection de membres pour les objets et références (.)
- Sélection de membres pour les pointeurs (->)

Noms qualifiés

Les noms utilisés avec l'opérateur de résolution d'étendue binaire (`::`) sont appelés « noms qualifiés ». Le nom spécifié après l'opérateur de résolution d'étendue binaire doit être membre de la classe spécifiée à gauche de l'opérateur ou membre de sa ou de ses classes de base.

Les noms spécifiés après l'opérateur de sélection membre (. ou ->) doivent être membres du type de classe de l'objet spécifié à gauche de l'opérateur ou des membres de sa ou de ses classes de base. Les noms spécifiés à droite de l'opérateur de sélection membre (->) peuvent également être des objets d'un autre type de classe, à condition que le côté gauche de -> soit un objet de classe et que la classe définit un opérateur de sélection de membres surchargé (->) qui s'évalue vers un autre type de classe. (Cette disposition est abordée plus en détail dans l'article Accès aux membres de classe.)

Le compilateur recherche les noms dans l'ordre suivant et s'arrête après avoir trouvé le nom :

1. Portée de bloc active si le nom est utilisé dans une fonction. Sinon, portée globale.
2. Vers l'extérieur dans chaque portée de bloc englobante, notamment la portée de fonction externe (qui inclut des paramètres de fonction).
3. Si le nom est utilisé dans une fonction membre, il est recherché dans la portée de la classe.
4. Le nom est recherché dans les classes de base de la classe.
5. La recherche est effectuée dans la portée de classe imbriquée englobante (le cas échéant) et ses bases. La recherche se poursuit jusqu'à atteindre la portée de classe englobante externe.
6. La recherche est effectuée dans la portée globale.

Toutefois, vous pouvez apporter des modifications à cet ordre de recherche comme suit :

1. Les noms précédés de `::` forcent la recherche à commencer au niveau de la portée globale.
2. Les noms précédés des `class` mots clés , `struct` et `union` forcent le compilateur à rechercher uniquement les `class` noms , `struct` ou `union` .
3. Les noms sur le côté gauche de l'opérateur de résolution d'étendue (`::`) ne peuvent être que `class` , `struct` , `namespace` ou `union` des noms.

Si le nom fait référence à un membre non statique mais est utilisé dans une fonction membre statique, un message d'erreur est généré. De même, si le nom fait référence à un membre non statique dans une classe englobante, un message d'erreur est généré, car les classes fermées n'ont pas de pointeurs de classe `this` englobante.

Noms de paramètres de fonction

Les noms de paramètres de fonction dans les définitions de fonction sont considérés comme étant dans la portée du bloc le plus à l'extérieur de la fonction. Par conséquent, il s'agit de noms régionaux. Ils se trouvent hors de portée lors de la sortie de la fonction.

Les noms de paramètres de fonction dans les déclarations de fonctions (prototypes) sont dans la portée locale de la déclaration et sortent de la portée à la fin de la

déclaration.

Les paramètres par défaut sont dans la portée du paramètre pour lequel ils sont la valeur par défaut, comme décrit dans les deux paragraphes précédents. Toutefois, ils ne peuvent pas accéder à des variables locales ou à des membres de classes non statiques. Les paramètres par défaut sont évalués au point de l'appel de fonction, mais ils sont évalués dans la portée d'origine de la déclaration de fonction. Par conséquent, les paramètres par défaut pour les fonctions membres sont toujours évalués dans la portée de classe.

Voir aussi

Héritage

Mots clé de l'héritage

Article • 16/06/2023

Section spécifique à Microsoft

> **`class`** *`class-name`*
>
> **`class __single_inheritance`** *`class-name`*
>
> **`class __multiple_inheritance`** *`class-name`*
>
> **`class __virtual_inheritance`** *`class-name`*

où :

`class-name`
est le nom de la variable en cours de déclaration.

C++ vous permet de déclarer un pointeur vers un membre de classe avant la définition de la classe. Par exemple :

```
C++

class S;
int S::*p;
```

Dans le code ci-dessus, `p` est déclaré comme un pointeur vers le membre entier de la classe S. Toutefois, `class S` n'a pas encore été défini dans ce code ; il n'a été déclaré que. Lorsque le compilateur détecte ce type de pointeur, il doit créer une représentation généralisée du pointeur. La taille de la représentation dépend du modèle d'héritage spécifié. Il existe trois façons de spécifier un modèle d'héritage pour le compilateur :

- Sur la ligne de commande à l'aide du /vmg commutateur
- Utilisation du pointers_to_members pragma
- À l'aide des mots clés d'héritage **`__single_inheritance`**, **`__multiple_inheritance`**, et **`__virtual_inheritance`**. Cette technique contrôle le modèle d'héritage pour chaque classe.

> **Notes**
>
> Si vous déclarez toujours un pointeur vers un membre d'une classe après la définition de la classe, vous n'avez pas besoin d'utiliser l'une de ces options.

Si vous déclarez un pointeur vers un membre de classe avant que la classe ne soit définie, cela peut affecter négativement la taille et la vitesse du fichier exécutable résultant. Plus l'héritage utilisé par une classe est complexe, plus le nombre d'octets requis pour représenter un pointeur vers un membre de la classe est élevé. Et plus le code requis pour interpréter le pointeur est important. L'héritage unique (ou aucun) est le moins complexe, et l'héritage virtuel est le plus complexe. Les pointeurs vers les membres que vous déclarez avant la définition de la classe utilisent toujours la représentation la plus grande et la plus complexe.

Si l'exemple ci-dessus est modifié comme suit :

```
C++

class __single_inheritance S;
int S::*p;
```

peu importe les options de ligne de commande ou les pragmas que vous spécifiez, les pointeurs vers les membres de `class S` utiliseront la plus petite représentation possible.

> ⓘ **Notes**
>
> La même déclaration anticipée d'une représentation de pointeur de classe vers membre devrait se produire dans chaque unité de traduction qui déclare des pointeurs vers des membres de cette classe, et la déclaration devrait se produire avant que les pointeurs vers des membres soient déclarées.

Pour la compatibilité avec les versions précédentes, , et sont synonymes de `__single_inheritance`, `__multiple_inheritance` et `__virtual_inheritance`, sauf si l'option /Za du compilateur (Désactiver les extensions de langage) est spécifiée. `_virtual_inheritance` `_multiple_inheritance` `_single_inheritance`

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés

virtual (C++)

Article • 03/04/2023

Le `virtual` mot clé déclare une fonction virtuelle ou une classe de base virtuelle.

Syntaxe

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Paramètres

spécificateurs de type
Spécifie le type de retour de la fonction membre virtuelle.

member-function-declarator
Déclare une fonction membre.

spécificateur d'accès
Définit le niveau d'accès à la classe de base , `public` `protected` ou `private`. Peut apparaître avant ou après le `virtual` mot clé.

base-class-name
Identifie un type de classe déclaré précédemment.

Notes

Pour plus d'informations, consultez Virtual Functions .

Consultez également les mots clés suivants : classe, privé, public et protégé.

Voir aussi

Mots clés

`__super`

Article • 16/06/2023

Section spécifique à Microsoft

Permet de déclarer explicitement que vous appelez une implémentation de classe de base pour une fonction que vous substituez.

Syntaxe

```
__super::member_function();
```

Notes

Toutes les méthodes de classe de base accessibles sont considérées pendant la phase de résolution de surcharge, et la fonction qui fournit la meilleure correspondance est celle qui est appelée.

`__super` peut apparaître uniquement dans le corps d'une fonction membre.

`__super` ne peut pas être utilisé avec une déclaration using. Pour plus d'informations, consultez Utilisation de la déclaration .

Avec l'introduction d'attributs qui injectent du code, votre code peut contenir une ou plusieurs classes de base dont vous ne connaissez peut-être pas le nom, mais qui contiennent des méthodes que vous souhaitez appeler.

Exemple

C++

```
// deriv_super.cpp
// compile with: /c
struct B1 {
   void mf(int) {}
};

struct B2 {
   void mf(short) {}
```

```
   void mf(char) {}
};

struct D : B1, B2 {
   void mf(short) {
      __super::mf(1);   // Calls B1::mf(int)
      __super::mf('s');   // Calls B2::mf(char)
   }
};
```

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés

__interface

Article • 19/06/2023

Section spécifique à Microsoft

Une interface Microsoft C++ peut être définie comme suit :

- Peut hériter de zéro ou plusieurs interfaces de base.
- Ne peut pas hériter d'une classe de base.
- Ne peut contenir que des méthodes publiques, virtuelles pures.
- Ne peut pas contenir de constructeurs, de destructeurs ou d'opérateurs.
- Ne peut pas contenir de méthodes statiques.
- Ne peut pas contenir de données membres ; les propriétés sont autorisées.

Syntaxe

```
modifier __interface interface-name {interface-definition};
```

Notes

Une classe ou un struct C++ peut être implémenté avec ces règles, mais `__interface` les applique.

Par exemple, voici un exemple de définition d'interface :

C++

```
__interface IMyInterface {
   HRESULT CommitX();
   HRESULT get_X(BSTR* pbstrName);
};
```

Pour plus d'informations sur les interfaces managées, consultez classe d'interface.

Notez que vous n'êtes pas obligé d'indiquer explicitement que `CommitX` et `get_X` sont des fonctions virtuelles pures. Une déclaration équivalente pour la première fonction se

présenterait comme suit :

```
C++

virtual HRESULT CommitX() = 0;
```

`__interface` implique le modificateur novable`__declspec` .

Exemple

L'exemple suivant montre comment utiliser les propriétés déclarées dans une interface.

```
C++

// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
   [ id(0) ] int int_data;
   [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
```

```
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}
```

Output

```
p->int_data = 100
bstr_data = Testing
```

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Attributs d'interface

Fonctions membres spéciales

Article • 16/06/2023

Les *fonctions membres spéciales* sont des fonctions membres de classe (ou struct) que, dans certains cas, le compilateur génère automatiquement pour vous. Ces fonctions sont le constructeur par défaut, le destructeur, le constructeur de copie et l'opérateur d'assignation de copie, ainsi que le constructeur de déplacement et l'opérateur d'affectation de déplacement. Si votre classe ne définit pas une ou plusieurs fonctions membres spéciales, le compilateur peut déclarer et définir implicitement les fonctions utilisées. Les implémentations générées par le compilateur sont appelées fonctions membres spéciales *par défaut* . Le compilateur ne génère pas de fonctions si elles ne sont pas nécessaires.

Vous pouvez déclarer explicitement une fonction membre spéciale par défaut à l'aide de la **mot clé par défaut**. Cela oblige le compilateur à définir la fonction uniquement si nécessaire, de la même façon que si la fonction n'était pas déclarée du tout.

Dans certains cas, le compilateur peut générer des fonctions membres spéciales *supprimées* , qui ne sont pas définies et ne peuvent donc pas être appelées. Cela peut se produire dans les cas où un appel à une fonction membre spéciale particulière sur une classe n'a pas de sens, compte tenu d'autres propriétés de la classe. Pour empêcher explicitement la génération automatique d'une fonction membre spéciale, vous pouvez la déclarer comme supprimée à l'aide de la mot clé = **delete**.

Le compilateur génère un *constructeur par défaut*, un constructeur qui ne prend aucun argument, uniquement lorsque vous n'avez pas déclaré d'autre constructeur. Si vous avez déclaré uniquement un constructeur qui accepte des paramètres, le code qui tente d'appeler un constructeur par défaut provoque la production d'un message d'erreur par le compilateur. Le constructeur par défaut généré par le compilateur effectue une initialisation par défaut simple au niveau des membres de l'objet. L'initialisation par défaut laisse toutes les variables membres dans un état indéterminé.

Le destructeur par défaut effectue la destruction de l'objet au niveau des membres. Elle est virtuelle uniquement si un destructeur de classe de base est virtuel.

Les opérations de construction et d'affectation de copie et de déplacement par défaut effectuent des copies de modèle de bits au niveau des membres ou des déplacements de membres de données non statiques. Les opérations de déplacement sont générées uniquement lorsqu'aucune opération de destructeur ou de déplacement ou de copie n'est déclarée. Un constructeur de copie par défaut n'est généré que lorsqu'aucun constructeur de copie n'est déclaré. Elle est implicitement supprimée si une opération

de déplacement est déclarée. Un opérateur d'affectation de copie par défaut est généré uniquement lorsqu'aucun opérateur d'affectation de copie n'est explicitement déclaré. Elle est implicitement supprimée si une opération de déplacement est déclarée.

Voir aussi

Informations de référence sur le langage C++

Membres statiques (C++)

Article • 16/06/2023

Les classes peuvent contenir des données membres et des fonctions membres statiques. Lorsqu'un membre de données est déclaré en tant que `static`, une seule copie des données est conservée pour tous les objets de la classe.

Les données membres statiques ne font pas partie des objets d'un type donné de classe. Par conséquent, la déclaration de données membres statiques n'est pas considérée comme une définition. Les données membres sont déclarées dans la portée de classe, mais la définition est exécutée au niveau de la portée de fichier. Ces membres statiques ont une liaison externe. L'exemple suivant illustre ce mécanisme :

C++

```
// static_data_members.cpp
class BufferedOutput
{
public:
   // Return number of bytes written by any object of this class.
   short BytesWritten()
   {
      return bytecount;
   }

   // Reset the counter.
   static void ResetCount()
   {
      bytecount = 0;
   }

   // Static member declaration.
   static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

Dans le code précédent, le membre `bytecount` est déclaré dans la classe `BufferedOutput`, mais doit être défini en dehors de la déclaration de classe.

Les données membres statiques peuvent être mentionnées sans faire référence à un objet de type classe. Le nombre d'octets écrits avec des objets `BufferedOutput` peut être

obtenu comme suit :

```
C++

long nBytes = BufferedOutput::bytecount;
```

Pour que le membre statique existe, il n'est pas nécessaire que des objets du type classe existent. Les membres statiques sont également accessibles à l'aide des opérateurs de sélection de membres (. et ->). Par exemple :

```
C++

BufferedOutput Console;

long nBytes = Console.bytecount;
```

Dans le cas précédent, la référence à l'objet (`Console`) n'est pas évaluée ; la valeur retournée est celle de l'objet statique `bytecount`.

Les données membres statiques étant soumises à des règles d'accès de membre de classe, l'accès privé à ces données est autorisé uniquement pour les fonctions membres de classe et friend. Ces règles sont décrites dans Member-Access Control. L'exception est que les données membres statiques doivent être définies dans la portée de fichier quelles que soient les restrictions d'accès. Si les données membres doivent être explicitement initialisées, un initialiseur doit être fourni avec la définition.

Le type d'un membre statique n'est pas qualifié par son nom de classe. Par conséquent, le type de `BufferedOutput::bytecount` est **`long`**.

Voir aussi

Classes et structs

Conversions de type définies par l'utilisateur (C++)

Article • 16/06/2023

Une *conversion* produit une nouvelle valeur d'un type à partir d'une valeur d'un autre type. *Les conversions standard* sont intégrées au langage C++ et prennent en charge ses types intégrés, et vous pouvez créer *des conversions définies par l'utilisateur* pour effectuer des conversions vers, depuis ou entre des types définis par l'utilisateur.

Les conversions standard effectuent des conversions entre des types intégrés, entre des pointeurs ou références vers des types apparentés par héritage, vers et à partir de pointeurs void, et vers le pointeur null. Pour plus d'informations, consultez Conversions standard. Les conversions définies par l'utilisateur effectuent des conversions entre des types définis par l'utilisateur ou entre des types définis par l'utilisateur et des types intégrés. Vous pouvez les implémenter en tant que constructeurs de conversion ou en tant que fonctions de conversion.

Les conversions peuvent être explicites (quand le programmeur appelle un type pour qu'il soit converti en un autre type, comme dans une diffusion ou une initialisation directe) ou implicites (quand le langage ou le programme appelle un autre type que celui qui a été donné par le programmeur).

Les conversions implicites sont tentées dans les conditions suivantes :

- un argument fourni à une fonction n'est pas du même type que le paramètre correspondant ;
- la valeur retournée d'une fonction n'est pas du même type que le type de retour de la fonction ;
- une expression d'initialiseur n'est pas du même type que l'objet qu'elle initialise ;
- une expression qui contrôle une instruction conditionnelle, une construction de boucles ou un commutateur n'est pas du type requis pour exercer ce contrôle.
- un opérande fourni à un opérateur n'est pas du même type que le paramètre d'opérande correspondant. Pour les opérateurs intégrés, les deux opérandes doivent être du même type et sont convertis en un type commun qui représente les deux. Pour plus d'informations, consultez Conversions standard. Pour les opérateurs définis par l'utilisateur, chaque opérande doit être du même type que le paramètre d'opérande correspondant.

Quand une conversion standard ne peut pas terminer une conversion implicite, le compilateur peut utiliser une conversion définie par l'utilisateur, suivie éventuellement d'une autre conversion standard, pour la terminer.

Quand plusieurs conversions définies par l'utilisateur effectuant la même conversion sont disponibles sur un site de conversion, la conversion est dite ambiguë. Ces ambiguïtés sont une erreur, car le compilateur ne peut pas déterminer quelle conversion choisir parmi les conversions disponibles. Toutefois, ce n'est pas une erreur de définir plusieurs façons d'effectuer la même conversion, car l'ensemble des conversions disponibles peut être différent selon l'emplacement où elles se trouvent dans le code source (par exemple, en fonction des fichiers d'en-tête inclus dans un fichier source). Si une seule conversion est disponible sur le site de conversion, il n'y a pas d'ambiguïté. Il existe plusieurs raisons aux conversions ambiguës, mais les plus courantes sont les suivantes :

- Héritage multiple. La conversion est définie dans plusieurs classes de base.
- Appel de fonction ambigu. La conversion est définie en tant que constructeur de conversion du type cible et en tant que fonction de conversion du type source. Pour plus d'informations, consultez Fonctions de conversion.

Vous pouvez généralement résoudre une ambiguïté en qualifiant le nom du type impliqué de manière plus complète ou en effectuant une diffusion explicite pour clarifier votre intention.

Les constructeurs de conversion et les fonctions de conversion sont régis par des règles de contrôle d'accès par membre, mais l'accessibilité des conversions n'est possible que si et quand une conversion n'est pas considérée comme ambiguë. Cela signifie qu'une conversion peut être ambiguë même si le niveau d'accès d'une conversion concurrente l'empêche d'être utilisée. Pour plus d'informations sur l'accessibilité des membres, consultez Access Control membres.

Le mots clé explicite et les problèmes liés à la conversion implicite

Par défaut, quand vous créez une conversion définie par l'utilisateur, le compilateur peut l'utiliser pour effectuer des conversions implicites. Parfois, c'est ce que vous voulez, mais parfois, les règles simples qui guident le compilateur dans la création de conversions implicites peut l'amener à accepter du code qu'il ne devrait pas.

Un exemple bien connu d'une conversion implicite qui peut provoquer des problèmes est la conversion en `bool`. Il existe de nombreuses raisons pour lesquelles vous pouvez

créer un type de classe qui peut être utilisé dans un contexte booléen(par exemple, afin qu'il puisse être utilisé pour contrôler une instruction ou une `if` boucle), mais lorsque le compilateur effectue une conversion définie par l'utilisateur vers un type intégré, le compilateur est autorisé à appliquer une conversion standard supplémentaire par la suite. L'objectif de cette conversion standard supplémentaire est de permettre la promotion de `short` vers `int`, mais elle ouvre également la porte aux conversions moins évidentes, par exemple, de `bool` à `int`, qui permet à votre type de classe d'être utilisé dans des contextes d'entiers que vous n'avez jamais prévus. Ce problème particulier est connu sous le nom *de problème de bool sécurisé*. C'est dans ce genre de problème que le `explicit` mot clé peut vous aider.

Le `explicit` mot clé indique au compilateur que la conversion spécifiée ne peut pas être utilisée pour effectuer des conversions implicites. Si vous vouliez la commodité syntaxique des conversions implicites avant l'introduction de la `explicit` mot clé, vous deviez accepter les conséquences inattendues que la conversion implicite a parfois créées ou utiliser des fonctions de conversion nommées moins pratiques comme solution de contournement. À présent, à l'aide de la `explicit` mot clé, vous pouvez créer des conversions pratiques qui ne peuvent être utilisées que pour effectuer des casts explicites ou une initialisation directe, et qui n'entraînent pas le type de problèmes illustrés par le problème de bool sécurisé.

La `explicit` mot clé peut être appliquée aux constructeurs de conversion depuis C++98 et aux fonctions de conversion depuis C++11. Les sections suivantes contiennent plus d'informations sur l'utilisation des `explicit` mot clé.

Constructeurs de conversion

Les constructeurs de conversion définissent les conversions de types définis par l'utilisateur ou intégrés en type défini par l'utilisateur. L'exemple suivant illustre un constructeur de conversion qui convertit le type `double` intégré en type `Money` défini par l'utilisateur .

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
```

```
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}
```

Notez que le premier appel de la fonction `display_balance`, qui prend un argument de type `Money`, ne requiert aucune conversion, car le type de son argument est correct. Toutefois, lors du deuxième appel à `display_balance`, une conversion est nécessaire, car le type de l'argument, un **`double`** avec la valeur , `49.95` n'est pas ce que la fonction attend. La fonction ne peut pas utiliser cette valeur directement, mais étant donné qu'il existe une conversion du type de l'argument en **`double`** type du paramètre `Money` correspondant, une valeur temporaire de type `Money` est construite à partir de l'argument et utilisée pour terminer l'appel de fonction. Dans le troisième appel à `display_balance`, notez que l'argument n'est pas un **`double`**, mais qu'il est plutôt un **`float`** avec une valeur de , et pourtant l'appel de `9.99` fonction peut toujours être effectué, car le compilateur peut effectuer une conversion standard (dans ce cas, de **`float`** à **`double`**) puis effectuer la conversion définie par l'utilisateur de **`double`** à `Money` pour terminer la conversion nécessaire.

Déclaration des constructeurs de conversion

Les règles suivantes s'appliquent à la déclaration d'un constructeur de conversion :

- Le type cible de la conversion est le type défini par l'utilisateur en cours de construction.
- Les constructeurs de conversion prennent généralement un seul argument, qui est de type source. Toutefois, un constructeur de conversion peut spécifier d'autres paramètres si chaque paramètre supplémentaire a une valeur par défaut. Le type source reste le type du premier paramètre.

- Les constructeurs de conversion, comme tous les constructeurs, ne spécifient pas de type de retour. La spécification d'un type de retour dans la déclaration constitue une erreur.

- Les constructeurs de conversion peuvent être explicites.

Constructeurs de conversion explicites

En déclarant un constructeur de conversion comme `explicit`, il ne peut être utilisé que pour effectuer une initialisation directe d'un objet ou pour effectuer un cast explicite. Cela empêche les fonctions acceptant un argument du type de la classe d'accepter également de manière implicite des arguments du type source du constructeur de conversion, et empêche le type de la classe d'être initialisé par copie d'une valeur du type source. L'exemple suivant décrit comment définir un constructeur de conversion explicite et son impact sur le code formé correctement.

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    explicit Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };        // Legal: direct initialization is
explicit.

    display_balance(payable);      // Legal: no conversion required
    display_balance(49.95);        // Error: no suitable conversion exists
to convert from double to Money.
    display_balance((Money)9.99f); // Legal: explicit cast to Money

    return 0;
}
```

Dans cet exemple, notez que vous pouvez encore utiliser le constructeur de conversion explicite pour effectuer une initialisation directe de `payable`. Si au lieu de cela vous devez initialiser par copie `Money payable = 79.99;`, cela constituerait une erreur. Le premier appel de `display_balance` n'est pas affecté, car l'argument est de type correct. Le deuxième appel de `display_balance` constitue une erreur, car le constructeur de conversion ne peut pas être utilisé pour effectuer des conversions implicites. Le troisième appel à `display_balance` est légal en raison de la conversion explicite en `Money`, mais notez que le compilateur a quand même aidé à terminer la conversion en insérant un cast implicite de **`float`** à **`double`**.

Même si vous pouvez être tenté par la commodité de l'autorisation des conversions implicites, celle-ci peut introduire des bogues difficiles à trouver. De manière générale, il vaut mieux que tous les constructeurs de conversion soient explicites, sauf quand vous êtes sûr que vous voulez qu'une conversion spécifique se produise de manière implicite.

Fonctions de conversion

Les fonctions de conversion définissent des conversions d'un type défini par l'utilisateur en d'autres types. Ces fonctions sont parfois qualifiées d'« opérateurs de transtypage », car elles sont appelées, en même temps que les constructeurs de conversion, quand une valeur est transtypée vers un type différent. L'exemple suivant illustre une fonction de conversion qui convertit du type défini par l'utilisateur, `Money`, en un type intégré, **`double`**:

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}
```

Notez que la variable `amount` membre est rendue privée et qu'une fonction de conversion publique en type `double` est introduite simplement pour retourner la valeur de `amount`. Dans la fonction `display_balance`, une conversion implicite se produit quand la valeur de `balance` est diffusée sur la sortie standard à l'aide de l'opérateur d'insertion de flux `<<`. Étant donné qu'aucun opérateur d'insertion de flux n'est défini pour le type `Money` défini par l'utilisateur , mais qu'il en existe un pour le type `double` intégré , le compilateur peut utiliser la fonction de conversion de `Money` à `double` pour satisfaire l'opérateur d'insertion de flux.

Les fonctions de conversion sont héritées dans les classes dérivées. Les fonctions de conversion d'une classe dérivée ne peuvent remplacer une fonction de conversion héritée que lorsqu'elles convertissent vers le même type exactement. Par exemple, une fonction de conversion définie par l'utilisateur de l'opérateur de classe dérivée **int** ne remplace pas, ni même influence, une fonction de conversion définie par l'utilisateur de l'opérateur de classe **de base short**, même si les conversions standard définissent une relation de conversion entre `int` et .`short`

Déclaration des fonctions de conversion

Les règles suivantes s'appliquent à la déclaration d'une fonction de conversion :

- Le type cible de la conversion doit être déclaré avant la déclaration de la fonction de conversion. Les classes, structures, énumérations et typedefs ne peuvent pas être déclarés dans la déclaration de la fonction de conversion.

  ```
  C++
  ```

  ```
  operator struct String { char string_storage; }() // illegal
  ```

- Les fonctions de conversion ne prennent pas d'arguments. La spécification d'un paramètre dans la déclaration constitue une erreur.
- Les fonctions de conversion ont un type de retour spécifié par le nom de la fonction de conversion, qui est également le nom du type cible de la conversion. La spécification d'un type de retour dans la déclaration constitue une erreur.
- Les fonctions de conversion peuvent être virtuelles.
- Les fonctions de conversion peuvent être explicites.

Fonctions de conversion explicite

Quand une fonction de conversion est déclarée comme explicite, elle ne peut être utilisée que pour effectuer un transtypage explicite. Cela empêche les fonctions acceptant un argument du type cible de la fonction de conversion d'accepter également de manière implicite des arguments du type de la classe, et empêche les instances du type cible d'être initialisées par copie d'une valeur du type de la classe. L'exemple suivant décrit comment définir une fonction de conversion explicite et son impact sur le code formé correctement.

C++

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    explicit operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << (double)balance << std::endl;
}
```

Ici, **l'opérateur de** fonction de conversion double a été rendu explicite, et un cast explicite en type `double` a été introduit dans la fonction `display_balance` pour effectuer la conversion. Si ce transtypage est omis, le compilateur est incapable de rechercher un opérateur d'insertion de flux `<<` convenable pour le type `Money` et une erreur se produit.

Membres de données mutables (C++)

Article • 03/04/2023

Ce mot clé peut être appliqué uniquement aux données membres non statiques et non constantes d'une classe. Si un membre de données est déclaré `mutable`, il est légal d'attribuer une valeur à ce membre de données à partir d'une `const` fonction membre.

Syntaxe

```
mutable member-variable-declaration;
```

Notes

Par exemple, le code suivant est compilé sans erreur, car `m_accessCount` a été déclaré comme étant `mutable`, et peut donc être modifié par `GetFlag` même si `GetFlag` est une fonction membre const.

C++

```
// mutable.cpp
class X
{
public:
   bool GetFlag() const
   {
      m_accessCount++;
      return m_flag;
   }
private:
   bool m_flag;
   mutable int m_accessCount;
};

int main()
{
}
```

Voir aussi

Mots clés

Déclarations de classes imbriquées

Article • 03/04/2023

Une classe peut être déclarée dans la portée d'une autre classe. Une telle classe est appelée « classe imbriquée ». Les classes imbriquées sont considérées comme appartenant à l'étendue de la classe englobante et peuvent être utilisées dans cette étendue. Pour faire référence à une classe imbriquée à partir d'une portée différente de sa portée englobante immédiate, vous devez utiliser un nom qualifié complet.

L'exemple suivant montre comment déclarer des classes imbriquées :

C++

```
// nested_class_declarations.cpp
class BufferedIO
{
public:
   enum IOError { None, Access, General };

   // Declare nested class BufferedInput.
   class BufferedInput
   {
   public:
      int read();
      int good()
      {
         return _inputerror == None;
      }
   private:
       IOError _inputerror;
   };

   // Declare nested class BufferedOutput.
   class BufferedOutput
   {
      // Member list
   };
};

int main()
{
}
```

`BufferedIO::BufferedInput` et `BufferedIO::BufferedOutput` sont déclarés dans `BufferedIO`. Ces noms de classe ne sont pas visibles en dehors de la portée de la classe `BufferedIO`. Toutefois, un objet de type `BufferedIO` ne contient aucun objet de type `BufferedInput` ou `BufferedOutput`.

Les classes imbriquées ne peuvent utiliser directement les noms, les noms de types, les noms des membres statiques et les énumérateurs qu'à partir de la classe englobante. Pour utiliser des noms d'autres membres de classe, vous devez utiliser des pointeurs, les références ou des noms d'objet.

Dans l'exemple `BufferedIO` précédent, l'énumération `IOError` est accessible directement par les fonctions membres des classes imbriquées, `BufferedIO::BufferedInput` ou `BufferedIO::BufferedOutput`, comme indiqué dans la fonction `good`.

> ⓘ **Notes**
>
> Les classes imbriquées déclarent uniquement des types dans la portée de classe. Elles n'entraînent pas la création d'objets contenus de la classe imbriquée. L'exemple précédent déclare deux classes imbriquées mais ne déclare aucun objet de ces types de classe.

Il existe une exception à la visibilité de portée d'une déclaration de classe imbriquée : lorsqu'un nom de type est déclaré avec une déclaration anticipée. Dans ce cas, le nom de classe déclaré par la déclaration anticipée est visible en dehors de la classe englobante, sa portée étant définie de telle sorte qu'elle est la plus petite portée englobante sans classe. Par exemple :

C++

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

Privilège d'accès dans les classes imbriquées

Le fait d'imbriquer une classe dans une autre classe ne donne pas de privilèges d'accès spécifiques aux fonctions membres de la classe imbriquée. De même, les fonctions membres de la classe englobante ne disposent d'aucun accès spécial aux membres de la classe imbriquée.

Fonctions membres dans les classes imbriquées

Les fonctions membres déclarées dans des classes imbriquées peuvent être définies dans la portée du fichier. L'exemple précédent pourrait avoir été écrit comme suit :

C++

```
// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
    public:
        int read(); // Declare but do not define member
        int good(); //  functions read and good.
    private:
        IOError _inputerror;
    };

    class BufferedOutput
    {
        // Member list.
    };
};
// Define member functions read and good in
//  file scope.
int BufferedIO::BufferedInput::read()
{
   return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}
int main()
{
}
```

Dans l'exemple précédent, la syntaxe *qualified-type-name* est utilisée pour déclarer le nom de la fonction. La déclaration :

```
C++

BufferedIO::BufferedInput::read()
```

signifie « la `read` fonction qui est membre de la `BufferedInput` classe qui se trouve dans l'étendue de la `BufferedIO` classe ». Étant donné que cette déclaration utilise la syntaxe *qualified-type-name* , les constructions du formulaire suivant sont possibles :

```
C++

typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()
```

La déclaration précédente est équivalente à la déclaration précédente, mais elle utilise un `typedef` nom à la place des noms de classe.

Fonctions friend dans les classes imbriquées

Les fonctions friend déclarées dans une classe imbriquée sont considérées comme étant dans la portée de la classe imbriquée, pas dans la classe englobante. Par conséquent, les fonctions friend ne comprennent pas de privilèges d'accès spéciaux sur les membres ou les fonctions membres de la classe englobante. Si vous souhaitez utiliser un nom déclaré dans une classe imbriquée dans une fonction friend et que la fonction friend est définie dans la portée du fichier, utilisez les noms de types qualifiés comme suit :

```
C++

// friend_functions_and_nested_classes.cpp

#include <string.h>

enum
{
    sizeOfMessage = 255
};

char *rgszMessage[sizeOfMessage];

class BufferedIO
{
public:
    class BufferedInput
```

```
    {
    public:
        friend int GetExtendedErrorStatus();
        static char *message;
        static int  messageSize;
        int iMsgNo;
   };
};

char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[iMsgNo] );

    return iMsgNo;
}

int main()
{
}
```

Le code suivant illustre la fonction `GetExtendedErrorStatus` déclarée comme fonction friend. Dans la fonction, définie dans la portée du fichier, un message est copié d'un tableau statique dans un membre de classe. Notez qu'une meilleure implémentation de `GetExtendedErrorStatus` consiste à le déclarer comme suit :

C++

```
int GetExtendedErrorStatus( char *message )
```

Avec l'interface précédente, plusieurs classes peuvent utiliser les services de cette fonction en passant par un emplacement en mémoire où ils souhaitent copier le message d'erreur.

Voir aussi

Classes et structs

Types de classe anonymes

Article • 16/06/2023

Les classes peuvent être anonymes, c'est-à-dire qu'elles peuvent être *déclarées sans identificateur*. Cela est utile lorsque vous remplacez un nom de classe par un `typedef` nom, comme dans le cas suivant :

C++

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

> ⓘ **Notes**
>
> L'utilisation de classes anonymes indiquée dans l'exemple précédent est utile pour conserver la compatibilité avec un code C existant. Dans certains codes C, l'utilisation de conjointement avec des `typedef` structures anonymes est répandue.

Les classes anonymes sont également utiles lorsque vous souhaitez qu'une référence à un membre de classe apparaisse comme si elle n'était pas été contenue dans une classe distincte, comme dans l'exemple suivant :

C++

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PTValue ptv;
```

Dans le code précédent, `iValue` vous pouvez accéder à l'aide de l'opérateur de sélection des membres de l'objet (.) comme suit :

C++

```
int i = ptv.iValue;
```

Les classes anonymes sont soumises à certaines restrictions. (Pour plus d'informations sur les syndicats anonymes, consultez Unions.) Classes anonymes :

- Ne peut pas avoir de constructeur ou de destructeur.
- Impossible de passer en tant qu'arguments aux fonctions (sauf si la vérification de type est défaite à l'aide de points de suspension).
- Ne peut pas être retourné comme valeurs de retour à partir des fonctions.

Structs anonymes

Section spécifique à Microsoft

Une extension Microsoft C vous permet de déclarer une variable de structure dans une autre structure sans lui donner un nom. Ces structures imbriquées sont appelées des structures anonymes. C++ n'autorise pas les structures anonymes.

Vous pouvez accéder aux membres d'une structure anonyme comme s'ils étaient des membres de la structure conteneur.

C++

```
// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int  areacode;
    long number;
};

struct person
{
    char   name[30];
    char   gender;
    int    age;
    int    weight;
    struct phone;    // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
```

```
}
//Output: 1234567
```

FIN de la section spécifique à Microsoft

Pointeurs vers membres

Article • 03/04/2023

Les déclarations de pointeurs vers des membres sont des cas spéciaux de déclarations de pointeur. Ils sont déclarés à l'aide de la séquence suivante :

> *storage-class-specifiers*$_{opt}$*cv-qualifiers*$_{opt}$*type-specifierms-modifier*$_{opt}$*qualified-name* `::*` *cv-qualifiers*$_{opt}$*identifierpm-initializer*$_{opt}$ `;`

1. Spécificateur de déclaration :
    - Spécificateur de classe de stockage facultatif.
    - Facultatifs **`const`** et **`volatile`** spécificateurs.
    - Spécificateur de type : nom d'un type. Il s'agit du type du membre à pointer, et non de la classe.
2. Déclarateur :
    - Modificateur spécifique à Microsoft facultatif. Pour plus d'informations, consultez Modificateurs spécifiques à Microsoft.
    - Nom qualifié de la classe contenant les membres vers lesquels pointer.
    - opérateur `::`,
    - opérateur `*`,
    - Facultatifs **`const`** et **`volatile`** spécificateurs.
    - Identificateur nommant le pointeur vers le membre.
3. Initialiseur de pointeur à membre facultatif :
    - opérateur `=`,
    - opérateur `&`,
    - Nom qualifié de la classe.
    - opérateur `::`,
    - Nom d'un membre non statique de la classe du type approprié.

Comme toujours, les déclarateurs multiples (et tout initialiseur associé) sont autorisés dans une même déclaration. Un pointeur vers un membre ne peut pas pointer vers un membre statique de la classe, un membre de type référence ou `void`.

Un pointeur vers un membre d'une classe diffère d'un pointeur normal : il a à la fois des informations de type pour le type du membre et pour la classe à laquelle appartient le membre. Un pointeur normal identifie (a l'adresse de) un seul objet en mémoire. Un pointeur vers un membre d'une classe identifie ce membre dans toute instance de la classe. L'exemple suivant déclare une classe, `Window`, et certains pointeurs vers des données membres.

C++

```
// pointers_to_members1.cpp
class Window
{
public:
   Window();                                  // Default constructor.
   Window( int x1, int y1,                    // Constructor specifying
   int x2, int y2 );                          // Window size.
   bool SetCaption( const char *szTitle );    // Set window caption.
   const char *GetCaption();                  // Get window caption.
   char *szWinCaption;                        // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
}
```

Dans l'exemple précédent, `pwCaption` est un pointeur vers n'importe quel membre de la classe `Window` de type `char*`. Le type de `pwCaption` est `char * Window::*`. Le fragment de code suivant déclare des pointeurs vers les fonctions membres `SetCaption` et `GetCaption`.

C++

```
const char * (Window::* pfnwGC)() = &Window::GetCaption;
bool (Window::* pfnwSC)( const char * ) = &Window::SetCaption;
```

Les pointeurs `pfnwGC` et `pfnwSC` pointent respectivement vers `GetCaption` et `SetCaption` de la classe `Window`. Le code copie directement des informations vers le titre de la fenêtre à l'aide du pointeur vers le membre `pwCaption` :

C++

```
Window  wMainWindow;
Window *pwChildWindow = new Window;
char   *szUntitled    = "Untitled -  ";
int     cUntitledLen  = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1';     // same as
// wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; // same as
// pwChildWindow->szWinCaption[cUntitledLen - 1] = '2';
```

La différence entre les `.*` opérateurs et `->*` (opérateurs pointeur vers membre) est que l'opérateur `.*` sélectionne les membres en fonction d'une référence d'objet ou d'objet, tandis que l'opérateur `->*` sélectionne les membres via un pointeur. Pour plus d'informations sur ces opérateurs, consultez Expressions avec des opérateurs pointeur vers membre.

Le résultat des opérateurs pointeur à membre est le type du membre. Dans ce cas, il s'agit de `char *`.

Le fragment de code suivant appelle les fonctions membres `GetCaption` et `SetCaption` à l'aide de pointeurs vers des membres :

C++

```
// Allocate a buffer.
enum {
    sizeOfBuffer = 100
};
char szCaptionBase[sizeOfBuffer];

// Copy the main window caption into the buffer
//  and append " [View 1]".
strcpy_s( szCaptionBase, sizeOfBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeOfBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

Restrictions sur les pointeurs vers des membres

L'adresse d'un membre statique n'est pas un pointeur vers un membre. Il s'agit d'un pointeur normal vers le instance du membre statique. Un seul instance d'un membre statique existe pour tous les objets d'une classe donnée. Cela signifie que vous pouvez utiliser les opérateurs d'adresse ordinaire de (**&**) et de déréférencement (*).

Pointeurs vers des membres et des fonctions virtuelles

L'appel d'une fonction virtuelle via une fonction pointeur à membre fonctionne comme si la fonction avait été appelée directement. La fonction correcte est recherchée dans la table v-table et appelée.

La clé du bon fonctionnement des fonctions virtuelles est, comme toujours, de les appeler via un pointeur vers une classe de base. (Pour plus d'informations sur les fonctions virtuelles, consultez Fonctions virtuelles.)

Le code suivant montre comment appeler une fonction virtuelle via un pointeur de fonction membre :

C++

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void Print();
};
void (Base::* bfnPrint)() = &Base::Print;
void Base::Print()
{
    cout << "Print function for class Base" << endl;
}

class Derived : public Base
{
public:
    void Print();  // Print is still a virtual function.
};

void Derived::Print()
{
    cout << "Print function for class Derived" << endl;
}

int main()
{
    Base   *bPtr;
    Base    bObject;
    Derived dObject;
    bPtr = &bObject;    // Set pointer to address of bObject.
    (bPtr->*bfnPrint)();
```

```
    bPtr = &dObject;    // Set pointer to address of dObject.
    (bPtr->*bfnPrint)();
}

// Output:
// Print function for class Base
// Print function for class Derived
```

Pointeur `this`

Article • 16/06/2023

Le `this` pointeur est un pointeur accessible uniquement dans les fonctions membres non statiques d'un `class` type , `struct` ou `union` . Il pointe vers l'objet pour lequel la fonction membre est appelée. Les fonctions membres statiques n'ont pas de `this` pointeur.

Syntaxe

```
C++
```

```
this
this->member-identifier
```

Notes

Le pointeur d'un `this` objet ne fait pas partie de l'objet lui-même. Il n'est pas reflété dans le résultat d'une `sizeof` instruction sur l'objet. Lorsqu'une fonction membre non statique est appelée pour un objet, le compilateur transmet l'adresse de l'objet à la fonction en tant qu'argument masqué. Par exemple, l'appel de fonction suivant :

```
C++
```

```
myDate.setMonth( 3 );
```

peut être interprété comme suit :

```
C++
```

```
setMonth( &myDate, 3 );
```

L'adresse de l'objet est disponible à partir de la fonction membre en tant que `this` pointeur. La plupart des `this` utilisations de pointeurs sont implicites. Il est légal, bien qu'inutile, d'utiliser un explicite `this` lorsque vous faites référence à des membres du class. Par exemple :

```
C++
```

```
void Date::setMonth( int mn )
{
   month = mn;            // These three statements
   this->month = mn;      // are equivalent
   (*this).month = mn;
}
```

L'expression `*this` est utilisée couramment pour retourner l'objet actuel d'une fonction membre :

C++

```
return *this;
```

Le `this` pointeur est également utilisé pour se prémunir contre l'autoréférence :

C++

```
if (&Object != this) {
// do not execute in cases of self-reference
```

> ⓘ Notes
>
> Étant donné que le `this` pointeur n'est pas modifiable, les affectations au `this` pointeur ne sont pas autorisées. Les implémentations antérieures de C++ autorisant l'affectation à `this`.

Parfois, le `this` pointeur est utilisé directement, par exemple pour manipuler des données structautoréférentielles, où l'adresse de l'objet actif est requise.

Exemple

C++

```
// this_pointer.cpp
// compile with: /EHsc

#include <iostream>
#include <string.h>

using namespace std;

class Buf
```

```
{
public:
    Buf( char* szBuffer, size_t sizeOfBuffer );
    Buf& operator=( const Buf & );
    void Display() { cout << buffer << endl; }

private:
    char*   buffer;
    size_t  sizeOfBuffer;
};

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
    if( &otherbuf != this )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer =  strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}
```

Output

```
my buffer
your buffer
```

Type du `this` pointeur

Le **`this`** type du pointeur peut être modifié dans la déclaration de fonction par les **`const`** mots clés et **`volatile`** . Pour déclarer une fonction qui a l'un de ces attributs, ajoutez la ou les mot clé après la liste d'arguments de fonction.

Prenons un exemple :

C++

```
// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

Le code précédent déclare une fonction membre, `X`, dans laquelle le **`this`** pointeur est traité comme un **`const`** pointeur vers un **`const`** objet. Les combinaisons d'options *cv-mod-list* peuvent être utilisées, mais elles modifient toujours l'objet pointé vers le **`this`** pointeur, et non le pointeur lui-même. La déclaration suivante déclare la fonction `X`, où le **`this`** pointeur est un **`const`** pointeur vers un **`const`** objet :

C++

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

Le type de **`this`** dans une fonction membre est décrit par la syntaxe suivante. le *`cv-qualifier-list`* est déterminé à partir du déclarateur de la fonction membre. Il peut s'agir **`const`** de ou **`volatile`** (ou les deux). *`class-type`* est le nom de :class

[*`cv-qualifier-list`*] *`class-type`* **`* const this`**

En d'autres termes, le `this` pointeur est toujours un `const` pointeur. Il ne peut pas être réaffecté. Les `const` qualificateurs ou `volatile` utilisés dans la déclaration de fonction membre s'appliquent à la class instance le `this` pointeur pointe vers, dans l'étendue de cette fonction.

Le tableau ci-dessous fournit davantage d'explications sur le fonctionnement de ces modificateurs.

Sémantique des `this` modificateurs

Modificateur	Signification
`const`	Impossible de modifier les données de membre ; ne peut pas appeler les fonctions membres qui ne sont pas `const`.
`volatile`	Les données membres sont chargées à partir de la mémoire chaque fois qu'elles sont consultées ; désactive certaines optimisations.

Il s'agit d'une erreur de passer un `const` objet à une fonction membre qui n'est pas `const`.

De même, il s'agit également d'une erreur de passer un `volatile` objet à une fonction membre qui n'est pas `volatile`.

Les fonctions membres déclarées comme `const` ne peuvent pas modifier les données membres : dans ces fonctions, le `this` pointeur est un pointeur vers un `const` objet.

> ⓘ Notes
>
> Les constructors et destructors ne peuvent pas être déclarés en tant que `const` ou `volatile`. Toutefois, ils peuvent être appelés sur `const` des objets ou `volatile` .

Voir aussi

Mots clés

Champs de bits C++

Article • 17/06/2023

Les classes et les structures peuvent contenir des membres qui occupent moins d'espace de stockage qu'un type intégral. Ces membres sont spécifiés en tant que champs de bits. La syntaxe de la spécification *member-declarator de* champ binaire est la suivante :

Syntaxe

declarator:*constant-expression*

Notes

Le *déclarateur* (facultatif) est le nom par lequel le membre est accessible dans le programme. Il doit s'agir d'un type intégral (y compris les types énumérés). La *constante-expression* spécifie le nombre de bits occupés par le membre dans la structure. Les champs de bits anonymes, c'est-à-dire les membres de champ de bits sans identificateur, peuvent être utilisés pour le remplissage.

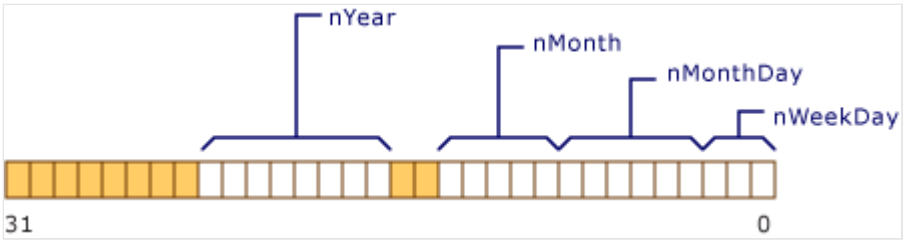
> ⓘ **Notes**
>
> Un champ de bits sans nom de largeur 0 force l'alignement du champ de bits suivant sur la limite de **type** suivant, où **type** est le type du membre.

L'exemple suivant déclare une structure qui contient des champs de bits :

C++

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
   unsigned short nWeekDay  : 3;    // 0..7   (3 bits)
   unsigned short nMonthDay : 6;    // 0..31  (6 bits)
   unsigned short nMonth    : 5;    // 0..12  (5 bits)
   unsigned short nYear     : 8;    // 0..100 (8 bits)
};
```

La disposition de la mémoire conceptuelle d'un objet de type `Date` est illustrée dans la figure suivante :



`nYear` a une longueur de 8 bits, ce qui dépasserait la limite du mot du type déclaré, **`unsigned short`**. Par conséquent, il commence au début d'un nouveau **`unsigned short`**. Il n'est pas nécessaire que tous les champs de bits tiennent dans un objet du type sous-jacent ; de nouvelles unités de stockage sont allouées, en fonction du nombre de bits demandés dans la déclaration.

Section spécifique à Microsoft

L'ordre des données déclarées en tant que champs de bits est de faible à bit élevé, comme illustré dans la figure précédente.

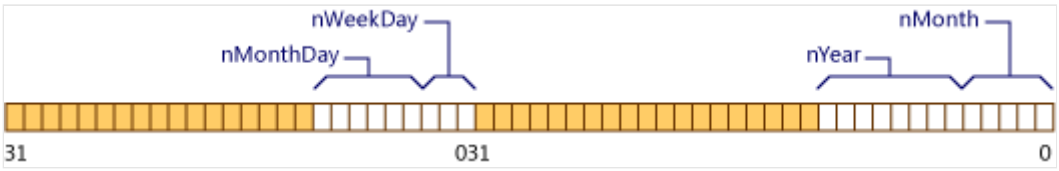
FIN de la section spécifique à Microsoft

Si la déclaration d'une structure inclut un champ sans nom de longueur 0, comme illustré dans l'exemple suivant :

C++

```
// bit_fields2.cpp
// compile with: /LD
struct Date {
   unsigned nWeekDay  : 3;    // 0..7   (3 bits)
   unsigned nMonthDay : 6;    // 0..31  (6 bits)
   unsigned           : 0;    // Force alignment to next boundary.
   unsigned nMonth    : 5;    // 0..12  (5 bits)
   unsigned nYear     : 8;    // 0..100 (8 bits)
};
```

Ensuite, la disposition de la mémoire est comme indiqué dans la figure suivante :



Le type sous-jacent d'un champ bit doit être un type intégral, comme décrit dans Types intégrés.

Si l'initialiseur d'une référence de type `const T&` est une valeur lvalue qui fait référence à un champ bit de type `T`, la référence n'est pas liée directement au champ bit. Au lieu de

cela, la référence est liée à un temporaire initialisé pour contenir la valeur du champ bit.

Restrictions sur les champs de bits

La liste suivante détaille les opérations erronées sur les champs de bits :

- Prise de l'adresse d'un champ de bits.
- Initialisation d’une non-référence `const` avec un champ bit.

Voir aussi

Classes et structs

Expressions lambda en C++

Article • 03/04/2023

En C++11 et versions ultérieures, une expression lambda(souvent appelée *lambda*) est un moyen pratique de définir un objet de fonction anonyme (une *fermeture*) à l'emplacement où elle est appelée ou passée en tant qu'argument à une fonction. En règle générale, les lambdas sont utilisés pour encapsuler quelques lignes de code qui sont passées à des algorithmes ou à des fonctions asynchrones. Cet article définit ce que sont les lambdas et les compare à d'autres techniques de programmation. Il décrit leurs avantages et fournit des exemples de base.

Articles connexes

- Expressions lambda et objets de fonction
- Utilisation d'expressions lambda
- Expressions lambda constexpr

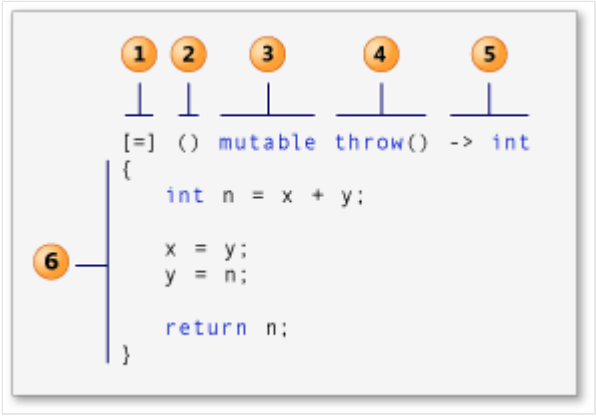
Parties d'une expression lambda

Voici un lambda simple qui est passé comme troisième argument à la `std::sort()` fonction :

C++

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

Cette illustration montre les parties de la syntaxe lambda :



1. *clause capture* (également appelée *lambda-introduction dans* la spécification C++)
2. *liste de paramètres* Optionnel. (Également appelé *déclarateur lambda*)
3. *spécification mutable* Optionnel.
4. *spécification d'exception* Optionnel.
5. *type de fin de retour* Optionnel.
6. *corps lambda*.

Clause capture

Un lambda peut introduire de nouvelles variables dans son corps (en **C++14**), et il peut également accéder ou *capturer* des variables à partir de l'étendue environnante. Un lambda commence par la clause de capture. Il spécifie quelles variables sont capturées et si la capture est par valeur ou par référence. Les variables qui ont le préfixe ampersand (`&`) sont accessibles par référence et les variables qui ne l'ont pas sont accessibles par valeur.

Une clause de capture vide, `[ ]`, indique que le corps de l'expression lambda n'accède à aucune variable dans la portée englobante.

Vous pouvez utiliser un mode de capture par défaut pour indiquer comment capturer toutes les variables externes référencées dans le corps lambda : `[&]` signifie que toutes les variables auxquelles vous faites référence sont capturées par référence et `[=]` qu'elles sont capturées par valeur. Vous pouvez utiliser un mode de capture par défaut, puis spécifier le mode opposé explicitement pour des variables spécifiques. Par exemple, si le corps d'une expression lambda accède à la variable externe `total` par référence et à la variable externe `factor` par valeur, les clauses de capture suivantes sont équivalentes :

```
C++
```

```
[&total, factor]
[factor, &total]
[&, factor]
[=, &total]
```

Seules les variables mentionnées dans le corps lambda sont capturées lorsqu'une capture par défaut est utilisée.

Si une clause de capture inclut une capture par défaut `&`, aucun identificateur dans une capture de cette clause de capture ne peut avoir la forme `&identifier`. De même, si la clause de capture inclut une capture par défaut `=`, aucune capture de cette clause de capture ne peut avoir la forme `=identifier`. Identificateur ou **`this`** ne peut pas apparaître plusieurs fois dans une clause de capture. L'extrait de code suivant illustre quelques exemples :

C++

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};      // OK
    [&, &i]{};     // ERROR: i preceded by & when & is the default
    [=, this]{};   // ERROR: this when = is the default
    [=, *this]{ }; // OK: captures this by value. See below.
    [i, i]{};      // ERROR: i repeated
}
```

Une capture suivie d'un point de suspension est une extension de pack, comme illustré dans cet exemple de modèle variadique :

C++

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

Pour utiliser des expressions lambda dans le corps d'une fonction membre de classe, passez le **`this`** pointeur vers la clause de capture pour fournir l'accès aux fonctions membres et aux membres de données de la classe englobante.

Visual Studio 2017 version 15.3 et ultérieure (disponible en mode et versions /std:c++17 ultérieures) : le **`this`** pointeur peut être capturé par valeur en spécifiant **`*this`** dans la clause de capture. La capture par valeur copie l'intégralité de la fermeture

à chaque site d'appel où l'lambda est appelé. (Une fermeture est l'objet de fonction anonyme qui encapsule l'expression lambda.) La capture par valeur est utile lorsque l'objet lambda s'exécute dans des opérations parallèles ou asynchrones. Il est particulièrement utile sur certaines architectures matérielles, telles que NUMA.

Pour obtenir un exemple qui montre comment utiliser des expressions lambda avec des fonctions membres de classe, consultez « Exemple : Utilisation d'une expression lambda dans une méthode » dans Exemples d'expressions lambda.

Lorsque vous utilisez la clause de capture, nous vous recommandons de garder ces points à l'esprit, en particulier lorsque vous utilisez des lambdas avec multithreading :

- Les captures de référence peuvent être utilisées pour modifier des variables à l'extérieur, mais pas les captures de valeurs. (`mutable` permet de modifier les copies, mais pas les originaux.)
- Les captures de référence reflètent les mises à jour des variables externes, mais pas les captures de valeurs.
- les captures de référence introduisent une dépendance liée à la durée de vie, contrairement aux captures de valeur. Il est particulièrement important lorsque le lambda s'exécute de manière asynchrone. Si vous capturez un local par référence dans un lambda asynchrone, ce local peut facilement disparaître au moment où l'lambda s'exécute. Votre code peut provoquer une violation d'accès au moment de l'exécution.

Capture généralisée (C++14)

En C++14, vous pouvez introduire et initialiser de nouvelles variables dans la clause de capture, sans que ces variables existent dans l'étendue englobante de la fonction lambda. L'initialisation peut être exprimée comme n'importe quelle expression arbitraire ; le type de la nouvelle variable est déduit du type produit par l'expression. Cette fonctionnalité vous permet de capturer des variables de déplacement uniquement (telles que `std::unique_ptr`) à partir de l'étendue environnante et de les utiliser dans un objet lambda.

C++

```
pNums = make_unique<vector<int>>(nums);
//...
      auto a = [ptr = move(pNums)]()
        {
           // use ptr
        };
```

Liste de paramètres

Les lambdas peuvent à la fois capturer des variables et accepter des paramètres d'entrée. Une liste de paramètres (*déclarateur lambda* dans la syntaxe Standard) est facultative et, dans la plupart des aspects, ressemble à la liste de paramètres d'une fonction.

C++

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

En **C++14**, si le type de paramètre est générique, vous pouvez utiliser le `auto` mot clé comme spécificateur de type. Ce mot clé indique au compilateur de créer l'opérateur d'appel de fonction en tant que modèle. Chaque instance de dans une liste de `auto` paramètres équivaut à un paramètre de type distinct.

C++

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Une expression lambda peut accepter une autre expression lambda comme argument. Pour plus d'informations, consultez « Expressions lambda d'ordre supérieur » dans l'article Exemples d'expressions lambda.

Étant donné qu'une liste de paramètres est facultative, vous pouvez omettre les parenthèses vides si vous ne passez pas d'arguments à l'expression lambda et que son lambda-declarateur ne contient pas *exception-specification*, *trailing-return-type* ou `mutable`.

Spécification mutable

En règle générale, l'opérateur d'appel de fonction d'un lambda est const-by-value, mais l'utilisation du `mutable` mot clé annule cette opération. Il ne produit pas de membres de données mutables. La `mutable` spécification permet au corps d'une expression lambda

de modifier des variables capturées par valeur. Certains des exemples plus loin dans cet article montrent comment utiliser `mutable`.

Spécification d'exception

Vous pouvez utiliser la spécification d'exception `noexcept` pour indiquer que l'expression lambda ne lève aucune exception. Comme pour les fonctions ordinaires, le compilateur Microsoft C++ génère l'avertissement C4297 si une expression lambda déclare la spécification d'exception `noexcept` et si le corps lambda lève une exception, comme illustré ici :

C++

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
   []() noexcept { throw 5; }();
}
```

Pour plus d'informations, consultez Spécifications d'exception (throw).

Type de retour

Le type de retour d'une expression lambda est déduit automatiquement. Vous n'avez pas besoin d'utiliser le auto mot clé, sauf si vous *spécifiez un type de retour de fin*. Le *type de fin de retour* ressemble à la partie de type retour d'une fonction ordinaire ou d'une fonction membre. Toutefois, le type de retour doit suivre la liste de paramètres. Vous devez donc ajouter le mot clé trailing-return-type `->` avant le type de retour.

Vous pouvez omettre la partie de type retour d'une expression lambda si le corps lambda ne contient qu'une seule instruction de retour. Ou, si l'expression ne retourne pas de valeur. Si le corps d'une expression lambda contient une seule instruction return, le compilateur déduit le type de retour du type de l'expression de retour. Sinon, le compilateur déduit le type de retour comme `void`. Considérez les exemples d'extraits de code suivants qui illustrent ce principe :

C++

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; };  // ERROR: return type is void, deducing
                                  // return type from braced-init-list isn't
valid
```

Une expression lambda peut générer une autre expression lambda comme valeur de retour. Pour plus d'informations, consultez « Expressions lambda d'ordre supérieur » dans Exemples d'expressions lambda.

Corps lambda

Le corps lambda d'une expression lambda est une instruction composée. Il peut contenir tout ce qui est autorisé dans le corps d'une fonction ordinaire ou d'une fonction membre. Le corps d'une fonction ordinaire et d'une expression lambda peut accéder aux types de variables suivants :

- Variables capturées dans la portée englobante, comme décrit précédemment.
- Paramètres.
- Variables déclarées localement.
- Les membres de données de classe, lorsqu'ils sont déclarés à l'intérieur d'une classe et `this` sont capturés.
- Toute variable dont la durée de stockage est statique, par exemple, les variables globales.

L'exemple suivant contient une expression lambda qui capture explicitement la variable `n` par valeur, et capture implicitement la variable `m` par référence :

C++

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
   int m = 0;
   int n = 0;
   [&, n] (int a) mutable { m = ++n + a; }(4);
   cout << m << endl << n << endl;
}
```

Output

```
5
0
```

La variable `n` étant capturée par valeur, sa valeur reste `0` après l'appel à l'expression lambda. La `mutable` spécification permet `n` d'être modifiée dans le lambda.

Une expression lambda peut uniquement capturer des variables dont la durée de stockage est automatique. Toutefois, vous pouvez utiliser des variables qui ont une durée de stockage statique dans le corps d'une expression lambda. L'exemple suivant utilise la fonction `generate` et une expression lambda pour assigner une valeur à chaque élément dans un objet `vector`. L'expression lambda modifie la variable statique pour générer la valeur de l'élément suivant.

C++

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}
```

Pour plus d'informations, consultez générer.

L'exemple de code suivant utilise la fonction de l'exemple précédent et ajoute un exemple d'expression lambda qui utilise l'algorithme `generate_n`de bibliothèque standard C++ . Cette expression lambda affecte un élément d'un objet `vector` à la somme des deux éléments précédents. Le `mutable` mot clé est utilisé afin que le corps de l'expression lambda puisse modifier ses copies des variables `x` externes et `y`, que l'expression lambda capture par valeur. Étant donné que l'expression lambda capture les variables `x` et `y` d'origine par valeur, leurs valeurs restent égales à `1` après l'exécution de l'expression.

C++

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;
```

```
    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
    int y = 1;

    // Sets each element in the vector to the sum of the
    // previous two elements.
    generate_n(v.begin() + 2,
        elementCount - 2,
        [=]() mutable throw() -> int { // lambda is the 3rd parameter
        // Generate current value.
        int n = x + y;
        // Update previous two values.
        x = y;
        y = n;
        return n;
    });
    print("vector v after call to generate_n() with lambda: ", v);

    // Print the local variables x and y.
    // The values of x and y hold their initial values because
    // they are captured by value.
    cout << "x: " << x << " y: " << y << endl;

    // Fill the vector with a sequence of numbers
    fillVector(v);
    print("vector v after 1st call to fillVector(): ", v);
    // Fill the vector with the next sequence of numbers
```

```
    fillVector(v);
    print("vector v after 2nd call to fillVector(): ", v);
}
```

Output

```
vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18
```

Pour plus d'informations, consultez generate_n.

`constexpr` expressions lambda

Visual Studio 2017 version 15.3 et ultérieures (disponible en /std:c++17 mode et versions ultérieures) : vous pouvez déclarer une expression lambda comme `constexpr` (ou l'utiliser dans une expression constante) lorsque l'initialisation de chaque membre de données capturé ou introduit est autorisée dans une expression constante.

C++

```
    int y = 32;
    auto answer = [y]() constexpr
    {
        int x = 10;
        return y + x;
    };

    constexpr int Increment(int n)
    {
        return [n] { return n + 1; }();
    }
```

Un lambda est implicitement `constexpr` si son résultat répond aux exigences d'une `constexpr` fonction :

C++

```
    auto answer = [](int n)
    {
        return 32 + n;
    };

    constexpr int response = answer(10);
```

Si un lambda est implicitement ou explicitement `constexpr`, la conversion en pointeur de fonction produit une `constexpr` fonction :

```
C++
```

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Spécifique à Microsoft

Les lambdas ne sont pas pris en charge dans les entités managées CLR (Common Language Runtime) suivantes : `ref class`, `ref struct`, `value class` ou `value struct`.

Si vous utilisez un modificateur spécifique à Microsoft, tel que __declspec, vous pouvez l'insérer dans une expression lambda immédiatement après . `parameter-declaration-clause` Par exemple :

```
C++
```

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;
};
```

Pour déterminer si un modificateur particulier est pris en charge par les lambdas, consultez l'article sur le modificateur dans la section Modificateurs spécifiques à Microsoft .

Visual Studio prend en charge la fonctionnalité lambda C++11 Standard et *les lambdas sans état*. Un lambda sans état est convertible en pointeur de fonction qui utilise une convention d'appel arbitraire.

Voir aussi

Informations de référence sur le langage C++
Objets de fonction dans la bibliothèque standard C++
Appel de fonction
for_each

Syntaxe d’expression lambda

Article • 16/06/2023

Cet article décrit la syntaxe et les éléments structuraux des expressions lambda. Pour obtenir une description des expressions lambda, consultez Expressions lambda.

Comparaison des objets de fonction et des expressions lambda

Lorsque vous écrivez du code, vous utilisez probablement des pointeurs de fonction et des objets de fonction pour résoudre des problèmes et effectuer des calculs, en particulier lorsque vous utilisez des algorithmes de bibliothèque standard C++. Les pointeurs de fonction et les objets de fonction présentent chacun des avantages et des inconvénients. Par exemple, les pointeurs de fonction possèdent une charge mémoire syntaxique minimale, mais leur état n'est pas conservé au sein d'une portée. Les objets de fonction, quant à eux, peuvent conserver leur état, mais ils nécessitent la charge mémoire syntaxique d'une définition de classe.

Une expression lambda combine les avantages des pointeurs et des objets de fonction, tout en évitant leurs inconvénients. Comme un objet de fonction, un lambda est flexible et peut conserver l’état, mais contrairement à un objet de fonction, sa syntaxe compacte ne nécessite pas de définition de classe explicite. Grâce aux expressions lambda, votre code est moins encombré et moins sujet aux erreurs que le code nécessaire pour un objet de fonction équivalent.

Les exemples suivants comparent l'utilisation d'une expression lambda et d'un objet de fonction. Le premier exemple utilise une expression lambda pour afficher sur la console si chaque élément d'un objet `vector` est pair ou impair. Le deuxième exemple utilise un objet de fonction pour accomplir la même tâche.

Exemple 1 : Utilisation d'une expression lambda

Cet exemple passe un lambda à la fonction **for_each** . L'expression lambda imprime un résultat qui indique si chaque élément dans un objet `vector` est pair ou impair.

Code

C++

```cpp
// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   // Create a vector object that contains 9 elements.
   vector<int> v;
   for (int i = 1; i < 10; ++i) {
      v.push_back(i);
   }

   // Count the number of even numbers in the vector by
   // using the for_each function and a lambda.
   int evenCount = 0;
   for_each(v.begin(), v.end(), [&evenCount] (int n) {
      cout << n;
      if (n % 2 == 0) {
         cout << " is even " << endl;
         ++evenCount;
      } else {
         cout << " is odd " << endl;
      }
   });

   // Print the count of even numbers to the console.
   cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

Output

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

Commentaires

Dans l'exemple, le troisième argument de la fonction **for_each** est un lambda. La partie `[&evenCount]` spécifie la clause de capture de l'expression, `(int n)` spécifie la liste des

paramètres, et la partie restante spécifie le corps de l'expression.

Exemple 2 : Utilisation d'un objet de fonction

Il arrive qu'une expression lambda soit trop complexe pour être utilisée autrement que dans l'exemple précédent. L'exemple suivant utilise un objet de fonction au lieu d'un lambda, avec la fonction **for_each** , pour produire les mêmes résultats que l'exemple 1. Les deux exemples indiquent le nombre de chiffres pairs dans un objet `vector`. Pour conserver l'état de l'opération, la classe `FunctorClass` enregistre la variable `m_evenCount` par référence comme variable membre. Pour effectuer l'opération, `FunctorClass` implémente l'opérateur d'appel de fonction, **operator()**. Le compilateur Microsoft C++ génère du code comparable en taille et en performances au code lambda de l'exemple 1. Pour un problème simple comme celui de cet article, la plus simple des expressions lambda convient probablement mieux qu'un objet de fonction. Toutefois, si vous pensez que les fonctionnalités peuvent nécessiter une expansion significative à l'avenir, il serait plus judicieux d'utiliser un objet de fonction afin que la maintenance du code soit facilitée.

Pour plus d'informations sur **l'opérateur(),** consultez Appel de fonction. Pour plus d'informations sur la fonction **for_each** , consultez for_each.

Code

C++

```
// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
```

```
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

Output

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

Voir aussi

Expressions lambda

Exemples d'expressions lambda

Générer

generate_n
for_each
Spécifications d’exception (throw)
Avertissement du compilateur (niveau 1) C4297
Modificateurs spécifiques à Microsoft

Exemples d'expressions lambda

Article • 03/04/2023

Cet article explique comment utiliser des expressions lambda dans vos programmes. Pour obtenir une vue d'ensemble des expressions lambda, consultez Expressions lambda. Pour plus d'informations sur la structure d'une expression lambda, consultez Syntaxe de l'expression lambda.

Déclaration d'expressions lambda

Exemple 1

Étant donné qu'une expression lambda est typée, vous pouvez l'affecter à une `auto` variable ou à un function objet, comme illustré ici :

C++

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{

    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto
variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

L'exemple génère cette sortie :

Output

```
5
7
```

Notes

Pour plus d'informations, consultez auto, function Classe et Appel de fonction.

Bien que les expressions lambda soient le plus souvent déclarées dans le corps d'une fonction, vous pouvez les déclarer partout où vous pouvez initialiser une variable.

Exemple 2

Le compilateur Microsoft C++ lie une expression lambda à ses variables capturées lorsque l'expression est déclarée plutôt que lorsque l'expression est appelée. L'exemple suivant montre une expression lambda qui capture la variable locale `i` par valeur et la variable locale `j` par référence. Puisque l'expression lambda capture `i` par valeur, la réaffectation de `i` ultérieurement dans le programme n'affecte pas le résultat de l'expression. Toutefois, puisque l'expression lambda enregistre `j` par référence, la réaffectation de `j` influence le résultat de l'expression.

C++

```
// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
   using namespace std;

   int i = 3;
   int j = 5;

   // The following lambda expression captures i by value and
   // j by reference.
   function<int (void)> f = [i, &j] { return i + j; };

   // Change the values of i and j.
   i = 22;
   j = 44;

   // Call f and print its result.
   cout << f() << endl;
}
```

L'exemple génère cette sortie :

Output

```
47
```

[Dans cet article]

Appel d'expressions lambda

Vous pouvez appeler une expression lambda immédiatement, comme indiqué dans l'extrait de code suivant. Le deuxième extrait de code montre comment passer un lambda en tant qu'argument à des algorithmes de bibliothèque standard C++ tels que `find_if`.

Exemple 1

Cet exemple déclare une expression lambda qui retourne la somme de deux entiers et appelle l'expression immédiatement avec les arguments `5` et `4`:

C++

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
   using namespace std;
   int n = [] (int x, int y) { return x + y; }(5, 4);
   cout << n << endl;
}
```

L'exemple génère cette sortie :

Output

```
9
```

Exemple 2

Cet exemple passe une expression lambda comme argument de la fonction `find_if`. L'expression lambda retourne **`true`** si son paramètre est un nombre pair.

C++

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
    numbers.push_back(46);
    numbers.push_back(99);

    // Use the find_if function and a lambda expression to find the
    // first even number in the list.
    const list<int>::const_iterator result =
        find_if(numbers.begin(), numbers.end(),[](int n) { return (n % 2) ==
0; });

    // Print the result.
    if (result != numbers.end()) {
        cout << "The first even number in the list is " << *result << "." <<
endl;
    } else {
        cout << "The list contains no even numbers." << endl;
    }
}
```

L'exemple génère cette sortie :

Output

```
The first even number in the list is 42.
```

Notes

Pour plus d'informations sur la `find_if` fonction, consultez find_if. Pour plus d'informations sur les fonctions de bibliothèque standard C++ qui exécutent des algorithmes courants, consultez <algorithm>.

[Dans cet article]

Imbrication d’expressions lambda

Exemple

Vous pouvez imbriquer une expression lambda dans une autre, comme le montre l'exemple suivant. L’expression lambda interne multiplie son argument par 2 et retourne le résultat. L’expression lambda externe appelle l’expression lambda interne avec son argument et ajoute 3 au résultat.

C++

```
// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.
    int timestwoplusthree = [](int x) { return [](int y) { return y * 2; }
(x) + 3; }(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}
```

L'exemple génère cette sortie :

Output

```
13
```

Notes

Dans cet exemple, `[](int y) { return y * 2; }` est l’expression lambda imbriquée.

[Dans cet article]

Fonctions lambda d'ordre supérieur

Exemple

De nombreux langages de programmation prennent en charge le concept d'une *fonction d'ordre supérieur*. Une fonction d'ordre supérieur est une expression lambda qui prend une autre expression lambda comme argument ou retourne une expression lambda. Vous pouvez utiliser la function classe pour permettre à une expression lambda C++ de se comporter comme une fonction d'ordre supérieur. L'exemple suivant présente une expression lambda qui retourne un objet `function` et une expression lambda qui prend un objet `function` comme argument.

C++

```
// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}
```

L'exemple génère cette sortie :

Output

```
30
```

[Dans cet article]

Utilisation d'une expression lambda dans une fonction

Exemple

Vous pouvez utiliser les expressions lambda dans le corps d'une fonction. L'expression lambda peut accéder à toute donnée membre ou fonction accessible à la fonction englobante. Vous pouvez capturer explicitement ou implicitement le pointeur pour fournir l'accès `this` aux fonctions et aux membres de données de la classe englobante. **Visual Studio 2017 version 15.3 et ultérieure** (disponible avec /std:c++17 et versions ultérieures) : Capture `this` par valeur (`[*this]`) lorsque l'lambda est utilisé dans des opérations asynchrones ou parallèles où le code peut s'exécuter une fois que l'objet d'origine sort de l'étendue.

Vous pouvez utiliser le `this` pointeur explicitement dans une fonction, comme illustré ici :

C++

```
// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [*this](int n) { cout << n * _scale << endl; });
}
```

Vous pouvez également capturer le `this` pointeur implicitement :

C++

```
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [=](int n) { cout << n * _scale << endl; });
}
```

L'exemple suivant montre la classe `Scale`, qui encapsule une valeur d'échelle.

C++

```
// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale <<
endl; });
    }

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}
```

L'exemple génère cette sortie :

Output

```
3
6
9
12
```

Notes

La fonction `ApplyScale` utilise une expression lambda pour imprimer le produit de la valeur d'échelle et de chaque élément d'un objet `vector`. L'expression lambda capture `this` implicitement afin qu'elle puisse accéder au `_scale` membre.

[Dans cet article]

Utilisation d'expressions lambda avec des modèles

Exemple

Les expressions lambda étant typées, vous pouvez les utiliser avec des modèles C++. L'exemple suivant illustre les fonctions `negate_all` et `print_all`. La `negate_all` fonction applique l'unaire `operator-` à chaque élément de l'objet `vector` . La fonction `print_all` affiche chaque élément de l'objet `vector` sur la console.

C++

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes signed data type.
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}

int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
```

```
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all():" << endl;
    print_all(v);
}
```

L'exemple génère cette sortie :

Output

```
34
-43
56
After negate_all():
-34
43
-56
```

Notes

Pour plus d'informations sur les modèles C++, consultez Modèles.

[Dans cet article]

Gestion des exceptions

Exemple

Le corps d'une expression lambda suit les règles de la gestion structurée des exceptions (SEH) et la gestion des exceptions C++. Vous pouvez gérer une exception levée dans le corps d'une expression lambda ou différer la gestion des exceptions dans la portée englobante. L'exemple suivant utilise la `for_each` fonction et une expression lambda pour remplir un `vector` objet avec les valeurs d'un autre. Il utilise un `try`/`catch` bloc pour gérer l'accès non valide au premier vecteur.

C++

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
```

```
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);

    // Create another vector that contains index values.
    vector<int> indices(3);
    indices[0] = 0;
    indices[1] = -1; // This is not a valid subscript. It will trigger an
exception.
    indices[2] = 2;

    // Use the values from the vector of index values to
    // fill the elements vector. This example uses a
    // try/catch block to handle invalid access to the
    // elements vector.
    try
    {
        for_each(indices.begin(), indices.end(), [&](int index) {
            elements.at(index) = index;
        });
    }
    catch (const out_of_range& e)
    {
        cerr << "Caught '" << e.what() << "'." << endl;
    };
}
```

L'exemple génère cette sortie :

Output

```
Caught 'invalid vector<T> subscript'.
```

Notes

Pour plus d'informations sur la gestion des exceptions, consultez Gestion des exceptions.

[Dans cet article]

Utilisation d'expressions lambda avec des types managés (C++/CLI)

Exemple

La clause de capture d'une expression lambda ne peut pas contenir une variable qui a un type managé. Toutefois, vous pouvez passer un argument qui a un type managé dans la liste de paramètres d'une expression lambda. L'exemple suivant contient une expression lambda qui capture la variable non managée locale `ch` par valeur et prend un objet System.String comme paramètre.

C++

```
// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    }("Hello");
}
```

L'exemple génère cette sortie :

Output

```
Hello!
```

Notes

Vous pouvez également utiliser vos expressions lambda avec la bibliothèque STL/CLR. Pour plus d'informations, consultez Informations de référence sur la bibliothèque STL/CLR.

> **ⓘ Important**
>
> Les lambdas ne sont pas pris en charge dans ces entités gérées par le Common Language Runtime (CLR) : **`ref class`**, **`ref struct`**, **`value class`** et **`value struct`**.

[Dans cet article]

Voir aussi

Expressions lambda
Syntaxe d’expression lambda
auto
function Classe
find_if
<algorithm>
Appel de fonction
Modèles
Gestion des exceptions
Informations de référence sur la bibliothèque STL/CLR

Expressions lambda constexpr en C++

Article • 16/06/2023

Visual Studio 2017 version 15.3 et ultérieures (disponible en /std:c++17 mode et versions ultérieures) : une expression lambda peut être déclarée `constexpr` ou utilisée dans une expression constante lorsque l'initialisation de chaque membre de données qu'elle capture ou introduit est autorisée dans une expression constante.

C++

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

Un lambda est implicitement `constexpr` si son résultat répond aux exigences d'une `constexpr` fonction :

C++

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Si un lambda est implicitement ou explicitement `constexpr` et que vous le convertissez en pointeur de fonction, la fonction résultante est également `constexpr`:

C++

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

Voir aussi

Informations de référence sur le langage C++
Objets de fonction dans la bibliothèque standard C++
Appel de fonction
for_each

Tableaux (C++)

Article • 03/04/2023

Un tableau est une séquence d'objets du même type qui occupent une zone contiguë de mémoire. Les tableaux de style C traditionnels sont la source de nombreux bogues, mais ils sont encore courants, en particulier dans les anciennes bases de code. En C++moderne, nous vous recommandons vivement d'utiliser std::vector ou std::array au lieu de tableaux de style C décrits dans cette section. Ces deux types de bibliothèque standard stockent leurs éléments sous la forme d'un bloc de mémoire contigu. Toutefois, ils offrent une plus grande sécurité de type et prennent en charge les itérateurs qui sont garantis pour pointer vers un emplacement valide dans la séquence. Pour plus d'informations, consultez Conteneurs.

Déclarations de pile

Dans une déclaration de tableau C++, la taille du tableau est spécifiée après le nom de la variable, et non après le nom de type comme dans d'autres langues. L'exemple suivant déclare un tableau de 1 000 doubles à allouer sur la pile. Le nombre d'éléments doit être fourni en tant que littéral entier ou en tant qu'expression constante. Cela est dû au fait que le compilateur doit savoir combien d'espace de pile allouer ; il ne peut pas utiliser une valeur calculée au moment de l'exécution. Chaque élément du tableau se voit attribuer une valeur par défaut de 0. Si vous n'affectez pas de valeur par défaut, chaque élément contient initialement les valeurs aléatoires qui se trouvent à cet emplacement de mémoire.

C++

```
    constexpr size_t size = 1000;

    // Declare an array of doubles to be allocated on the stack
    double numbers[size] {0};

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i-1] * 1.1;
    }

    // Access each element
    for (size_t i = 0; i < size; i++)
```

```
    {
        std::cout << numbers[i] << " ";
    }
```

Le premier élément du tableau est le zéro élément. Le dernier élément est l'élément (*n-1*), où *n* est le nombre d'éléments que le tableau peut contenir. Le nombre d'éléments dans la déclaration doit être d'un type intégral et doit être supérieur à 0. Il est de votre responsabilité de vous assurer que votre programme ne transmet jamais à l'opérateur d'indice une valeur supérieure `(size - 1)` à .

Un tableau de taille zéro n'est légal que lorsque le tableau est le dernier champ d'un **`struct`** ou **`union`** et lorsque les extensions Microsoft sont activées (**`/Za`** ou **`/permissive-`** n'est pas définie).

Les tableaux basés sur la pile sont plus rapides à allouer et à accéder que les tableaux basés sur un tas. Toutefois, l'espace de pile est limité. Le nombre d'éléments de tableau ne peut pas être si important qu'il utilise trop de mémoire de pile. Combien est trop dépend de votre programme. Vous pouvez utiliser des outils de profilage pour déterminer si un tableau est trop volumineux.

Déclarations de tas

Vous pouvez avoir besoin d'un tableau trop volumineux pour être alloué sur la pile ou dont la taille n'est pas connue au moment de la compilation. Il est possible d'allouer ce tableau sur le tas à l'aide d'une new[] expression. L'opérateur retourne un pointeur vers le premier élément. L'opérateur d'indice fonctionne sur la variable pointeur de la même façon qu'il le fait sur un tableau basé sur une pile. Vous pouvez également utiliser l'arithmétique du pointeur pour déplacer le pointeur vers tous les éléments arbitraires du tableau. Il est de votre responsabilité de vous assurer que :

- vous conservez toujours une copie de l'adresse du pointeur d'origine afin de pouvoir supprimer la mémoire lorsque vous n'avez plus besoin du tableau.
- vous n'incrémentez pas ou décrémentez l'adresse du pointeur au-delà des limites du tableau.

L'exemple suivant montre comment définir un tableau sur le tas au moment de l'exécution. Il montre comment accéder aux éléments de tableau à l'aide de l'opérateur subscript et à l'aide de pointeur arithmétique :

```
C++
```

```
void do_something(size_t size)
{
```

```
    // Declare an array of doubles to be allocated on the heap
    double* numbers = new double[size]{ 0 };

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i - 1] * 1.1;
    }

    // Access each element with subscript operator
    for (size_t i = 0; i < size; i++)
    {
        std::cout << numbers[i] << " ";
    }

    // Access each element with pointer arithmetic
    // Use a copy of the pointer for iterating
    double* p = numbers;

    for (size_t i = 0; i < size; i++)
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    // Alternate method:
    // Reset p to numbers[0]:
    p = numbers;

    // Use address of pointer to compute bounds.
    // The compiler computes size as the number
    // of elements * (bytes per element).
    while (p < (numbers + size))
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    delete[] numbers; // don't forget to do this!

}
int main()
{
    do_something(108);
}
```

Initialisation des tableaux

Vous pouvez initialiser un tableau dans une boucle, un élément à la fois ou dans une instruction unique. Le contenu des deux tableaux suivants est identique :

C++

```
    int a[10];
    for (int i = 0; i < 10; ++i)
    {
        a[i] = i + 1;
    }

    int b[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Passage de tableaux à des fonctions

Lorsqu'un tableau est passé à une fonction, il est passé en tant que pointeur vers le premier élément, qu'il s'agisse d'un tableau basé sur une pile ou sur un tas. Le pointeur ne contient aucune autre information sur la taille ou le type. Ce comportement est appelé *décroissance de pointeur*. Lorsque vous passez un tableau à une fonction, vous devez toujours spécifier le nombre d'éléments dans un paramètre distinct. Ce comportement implique également que les éléments du tableau ne sont pas copiés lorsque le tableau est passé à une fonction. Pour empêcher la fonction de modifier les éléments, spécifiez le paramètre comme pointeur vers des `const` éléments.

L'exemple suivant montre une fonction qui accepte un tableau et une longueur. Le pointeur pointe vers le tableau d'origine, pas une copie. Étant donné que le paramètre n'est pas `const`, la fonction peut modifier les éléments du tableau.

C++

```
void process(double *p, const size_t len)
{
    std::cout << "process:\n";
    for (size_t i = 0; i < len; ++i)
    {
        // do something with p[i]
    }
}
```

Déclarez et définissez le paramètre `p` de tableau comme `const` pour le rendre en lecture seule dans le bloc de fonction :

C++

```
void process(const double *p, const size_t len);
```

La même fonction peut également être déclarée de cette manière, sans changement de comportement. Le tableau est toujours passé en tant que pointeur vers le premier élément :

```
C++
```

```
// Unsized array
void process(const double p[], const size_t len);

// Fixed-size array. Length must still be specified explicitly.
void process(const double p[1000], const size_t len);
```

Tableaux multidimensionnels

Les tableaux construits à partir d'autres tableaux sont des tableaux multidimensionnels. Ces tableaux multidimensionnels sont spécifiés en plaçant dans l'ordre plusieurs expressions de constantes entre accolades. Par exemple, observez cette déclaration :

```
C++
```

```
int i2[5][7];
```

Il spécifie un tableau de type `int`, disposé de manière conceptuelle dans une matrice à deux dimensions de cinq lignes et sept colonnes, comme illustré dans la figure suivante :

L'image est une grille de 7 cellules de large et de 5 cellules haute. Chaque cellule contient l'index de la cellule. Le premier index de cellule est étiqueté 0,0. La cellule suivante de cette ligne est 0,1 et ainsi de suite à la dernière cellule de cette ligne qui est 0,6. La ligne suivante commence par l'index 1,0. La cellule qui suit a un index de 1,1. La dernière cellule de cette ligne est 1,6. Ce modèle se répète jusqu'à la dernière ligne, qui commence par l'index 4,0. La dernière cellule de la dernière ligne a un index de 4,6. :::image-end

Vous pouvez déclarer des tableaux multidimensionnés qui ont une liste d'initialiseurs (comme décrit dans Initialiseurs). Dans ces déclarations, l'expression constante qui spécifie les limites de la première dimension peut être omise. Par exemple :

C++

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][cMarkets] = {
   { 32.19, 47.29, 31.99, 19.11 },
   { 11.29, 22.49, 33.47, 17.29 },
   { 41.97, 22.09,  9.76, 22.55 }
};
```

La déclaration précédente définit un tableau composé de trois lignes et de quatre colonnes. Les lignes représentent les fabriques et les colonnes les marchés auxquels les fabriques livrent. Les valeurs sont les coûts de transport entre les fabriques et les marchés. La première dimension du tableau est omise, mais le compilateur la complète en examinant l'initialiseur.

L'utilisation de l'opérateur indirection (*) sur un type de tableau à n dimensions génère un tableau n-1 dimensionnel. Si n correspond à 1, une variable scalaire (ou un élément de tableau) est générée.

Les tableaux C++ sont stockés dans l'ordre row-major. L'ordre row-major signifie que le dernier indice varie le plus rapidement.

Exemple

Vous pouvez également omettre la spécification de limites pour la première dimension d'un tableau multidimensionnel dans les déclarations de fonction, comme illustré ici :

C++

```
// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits>   // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][cMkts] = {
   { 32.19, 47.29, 31.99, 19.11 },
   { 11.29, 22.49, 33.47, 17.29 },
   { 41.97, 22.09,  9.76, 22.55 }
};

// Calculate size of unspecified dimension
```

```
const int cFactories = sizeof TransportCosts /
                  sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int
mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
   double MinCost;

   if (argv[1] == 0) {
      cout << "You must specify the number of markets." << endl;
      exit(0);
   }
   MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
   cout << "The minimum cost to Market " << argv[1] << " is: "
       << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts)
{
   double MinCost = DBL_MAX;

   for( size_t i = 0; i < cFacts; ++i )
      MinCost = (MinCost < TransportCosts[i][Mkt]) ?
         MinCost : TransportCosts[i][Mkt];

   return MinCost;
}
```

Output

```
The minimum cost to Market 3 is: 17.29
```

La fonction `FindMinToMkt` est écrite de telle sorte que l'ajout de nouvelles fabriques ne nécessite aucune modification du code, mais simplement une recompilation.

Initialisation des tableaux

Les tableaux d'objets qui ont un constructeur de classe sont initialisés par le constructeur. Quand il y a moins d'éléments dans la liste d'initialiseurs que d'éléments dans le tableau, le constructeur par défaut est utilisé pour les éléments restants. Si aucun constructeur par défaut n'est défini pour la classe, la liste d'initialiseurs doit être *complète*, c'est-à-dire qu'il doit y avoir un initialiseur pour chaque élément du tableau.

Prenons la classe `Point` qui définit deux constructeurs :

```
```

C++

```
// initializing_arrays1.cpp
class Point
{
public:
   Point()   // Default constructor.
   {
   }
   Point( int, int )   // Construct from two ints
   {
   }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
   Point( 3, 3 )     // Use int, int constructor.
};

int main()
{
}
```

Le premier élément de `aPoint` est construit en utilisant le constructeur `Point( int, int )`. Les deux éléments restants sont construits à l'aide du constructeur par défaut.

Les tableaux membres statiques (qu'ils soient **`const`** ou non) peuvent être initialisés dans leurs définitions (en dehors de la déclaration de classe). Par exemple :

C++

```
// initializing_arrays2.cpp
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];
};

const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"   };
int main()
{
}
```

Accès aux éléments du tableau

Vous pouvez accéder à des éléments individuels d'un tableau à l'aide de l'opérateur d'indice de tableau (`[ ]`). Si vous utilisez le nom d'un tableau unidimensionnel sans indice, il est évalué en tant que pointeur vers le premier élément du tableau.

C++

```
// using_arrays.cpp
int main() {
   char chArray[10];
   char *pch = chArray;   // Evaluates to a pointer to the first element.
   char   ch = chArray[0];   // Evaluates to the value of the first element.
   ch = chArray[3];   // Evaluates to the value of the fourth element.
}
```

Lorsque vous utilisez des tableaux multidimensionnels, vous pouvez utiliser différentes combinaisons dans des expressions.

C++

```
// using_arrays_2.cpp
// compile with: /EHsc /W1
#include <iostream>
using namespace std;
int main() {
   double multi[4][4][3];   // Declare the array.
   double (*p2multi)[3];
   double (*p1multi);

   cout << multi[3][2][2] << "\n";   // C4700 Use three subscripts.
   p2multi = multi[3];               // Make p2multi point to
                                     // fourth "plane" of multi.
   p1multi = multi[3][2];            // Make p1multi point to
                                     // fourth plane, third row
                                     // of multi.
}
```

Dans le code précédent, `multi` est un tableau tridimensionnel de type **`double`**. Le `p2multi` pointeur pointe vers un tableau de type **`double`** trois. Dans cet exemple, le tableau est utilisé avec un, deux et trois indices. Bien qu'il soit plus courant de spécifier tous les indices, comme dans l'instruction `cout` , il est parfois utile de sélectionner un sous-ensemble spécifique d'éléments de tableau, comme indiqué dans les instructions qui suivent `cout`.

Opérateur d'indice de surcharge

Comme d'autres opérateurs, l'opérateur d'indice (`[]`) peut être redéfini par l'utilisateur. Le comportement par défaut de l'opérateur d'indice, s'il n'est pas surchargé, est de combiner le nom d'un tableau et l'indice à l'aide de la méthode suivante :

```
*((array_name) + (subscript))
```

Comme dans tous les autres types de pointeurs, la mise à l'échelle est effectuée automatiquement pour ajuster la taille du type. La valeur résultante n'est pas *n* octets de l'origine de `array_name`; au lieu de cela, il s'agit du *nième*élément du tableau. Pour plus d'informations sur cette conversion, consultez Opérateurs additifs.

De même, pour des tableaux multidimensionnels, l'adresse est dérivée à l'aide de la méthode suivante :

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))
```

Tableaux dans les expressions

Lorsqu'un identificateur d'un type de tableau apparaît dans une expression autre que **`sizeof`**, l'adresse de (`&`) ou l'initialisation d'une référence, il est converti en pointeur vers le premier élément de tableau. Par exemple :

C++

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

Le pointeur `psz` pointe vers le premier élément du tableau `szError1`. Les tableaux, contrairement aux pointeurs, ne sont pas des valeurs l modifiables. C'est pourquoi l'affectation suivante est illégale :

C++

```
szError1 = psz;
```

Voir aussi

std::array

Références (C++)

Article • 03/04/2023

Une référence, comme un pointeur, stocke l'adresse d'un objet situé ailleurs dans la mémoire. Contrairement à un pointeur, une référence après son initialisation ne peut pas être définie pour faire référence à un autre objet ni prendre la valeur null. Il existe deux types de références : les références lvalue qui font référence à une variable nommée et les références rvalue qui font référence à un objet temporaire. L'opérateur & signifie une référence lvalue et l'opérateur && une référence rvalue ou une référence universelle (rvalue ou lvalue) en fonction du contexte.

Les références peuvent être déclarées à l'aide de la syntaxe suivante :

> [*storage-class-specifiers*] [*cv-qualifiers*] *spécificateurs de type* [*ms-modifier*] *déclarateur* [=*expression*];

Tout déclarateur valide spécifiant une référence peut être utilisé. La syntaxe simplifiée suivante s'applique, sauf si la référence est une référence à un type de fonction ou à un type tableau :

> [*storage-class-specifiers*] [*cv-qualifiers*] *type-specifiers* [**&** ou **&&**] [*cv-qualifiers*] *identifier* [=*expression*];

Les références sont déclarées à l'aide de la séquence suivante :

1. Les spécificateurs de déclaration :
   - Spécificateur de classe de stockage facultatif.
   - Facultatifs `const` et/ou `volatile` qualificateurs.
   - Spécificateur de type : nom d'un type.
2. Déclarateur :
   - Modificateur spécifique à Microsoft facultatif. Pour plus d'informations, consultez Modificateurs spécifiques à Microsoft.
   - Opérateur **&** ou **&&** opérateur.
   - Facultatifs `const` et/ou `volatile` qualifers.
   - L'identificateur.

3. Initialiseur facultatif.

Les formes de déclarateur plus complexes pour les pointeurs vers des tableaux et des fonctions s'appliquent également aux références aux tableaux et aux fonctions. Pour plus d'informations, consultez Pointeurs.

Plusieurs déclarateurs et initialiseurs peuvent apparaître dans une liste séparée par des virgules après un spécificateur de déclaration unique. Par exemple :

```
C++

int &i;
int &i, &j;
```

Les références, les pointeurs et les objets peuvent être déclarés ensemble :

```
C++

int &ref, *ptr, k;
```

Une référence contient l'adresse d'un objet, mais se comporte syntaxiquement comme un objet.

Dans le programme suivant, notez que le nom de l'objet, `s`, et la référence à l'objet, `SRef`, peuvent être utilisés de la même manière dans les programmes :

Exemple

```
C++

// references.cpp
#include <stdio.h>
struct S {
   short i;
};

int main() {
   S  s;   // Declare the object.
   S& SRef = s;   // Declare and initialize the reference.
   s.i = 3;

   printf_s("%d\n", s.i);
   printf_s("%d\n", SRef.i);

   SRef.i = 4;
   printf_s("%d\n", s.i);
```

```
    printf_s("%d\n", SRef.i);
}
```

Output

```
3
3
4
4
```

Voir aussi

Arguments de fonction de type référence

Retourne la fonction de type référence

Références aux pointeurs

Déclarateur de référence Lvalue : `&`

Article • 16/06/2023

Contient l'adresse d'un objet mais se comporte syntaxiquement comme un objet.

Syntaxe

`lvalue-reference-type-id`:

`type-specifier-seq` **`&`** *`attribute-specifier-seq`*$_{opt}$ *`ptr-abstract-declarator`*$_{opt}$

Notes

Vous pouvez considérer une référence lvalue comme un autre nom d'un objet. Une déclaration de référence lvalue se compose d'une liste facultative de spécificateurs suivie d'un déclarateur de référence. Une référence doit être initialisée et ne peut pas être modifiée.

Tout objet dont l'adresse peut être convertie en un type pointeur donné peut également être converti en type référence similaire. Par exemple, tout objet dont l'adresse peut être convertie en type `char *` peut également être converti en type `char &`.

Ne confondez pas les déclarations de référence avec l'utilisation de l'opérateur address-of. Lorsque *l'identificateur* `&` est précédé d'un type, tel que `int` ou `char`, *l'identificateur* est déclaré en tant que référence au type. Lorsque `&` *l'identificateur* n'est pas précédé d'un type, l'utilisation est celle de l'opérateur address-of.

Exemple

L'exemple suivant illustre le déclarateur de référence en déclarant un objet `Person` et une référence à cet objet. `rFriend` étant une référence à `myFriend`, la mise à jour de l'une ou l'autre variable modifie le même objet.

C++

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
```

```
{
    char* Name;
    short Age;
};

int main()
{
   // Declare a Person object.
   Person myFriend;

   // Declare a reference to the Person object.
   Person& rFriend = myFriend;

   // Set the fields of the Person object.
   // Updating either variable changes the same object.
   myFriend.Name = "Bill";
   rFriend.Age = 40;

   // Print the fields of the Person object to the console.
   cout << rFriend.Name << " is " << myFriend.Age << endl;
}
```

Output

```
Bill is 40
```

Voir aussi

Informations de référence

Arguments de fonction de type référence

Retours de fonction de type référence

Références aux pointeurs

Déclarateur de référence Rvalue : `&&`

Article • 03/04/2023

Contient une référence à une expression rvalue.

Syntaxe

`rvalue-reference-type-id`:

`type-specifier-seq` **`&&`** *`attribute-specifier-seq`*$_{opt}$ *`ptr-abstract-declarator`*$_{opt}$

Notes

Les références rvalue vous permettent de différencier une lvalue d'une rvalue. Les références Lvalue et rvalue sont similaires sur le plan syntaxique et sémantique, mais elles suivent des règles légèrement différentes. Pour plus d'informations sur lvalues et rvalues, consultez Lvalues et Rvalues. Pour plus d'informations sur les références lvalue, consultez Lvalue Reference Declarator : &.

Les sections suivantes décrivent comment les références rvalue prennent en charge l'implémentation de la *sémantique de déplacement et letransfert parfait*.

Sémantique de mouvement

Les références Rvalue prennent en charge l'implémentation de la *sémantique de déplacement*, qui peut augmenter considérablement les performances de vos applications. La sémantique de déplacement vous permet d'écrire du code qui transfère des ressources (telles que la mémoire allouée de manière dynamique) d'un objet vers un autre. La sémantique de déplacement fonctionne, car elle permet le transfert de ressources à partir d'objets temporaires: ceux qui ne peuvent pas être référencés ailleurs dans le programme.

Pour implémenter la sémantique de déplacement, vous fournissez généralement un *constructeur de déplacement et éventuellement* un opérateur d'affectation de déplacement (**`operator=`**) à votre classe. Les opérations de copie et d'assignation dont les sources sont des rvalues profitent ensuite automatiquement de la sémantique de déplacement. Contrairement au constructeur de copie par défaut, le compilateur ne fournit pas de constructeur de déplacement par défaut. Pour plus d'informations sur l'écriture et l'utilisation d'un constructeur de déplacement, consultez Déplacer les constructeurs et les opérateurs d'affectation de déplacement.

Vous pouvez aussi surcharger des fonctions et des opérateurs ordinaires pour tirer parti de la sémantique de déplacement. Visual Studio 2010 introduit la sémantique de déplacement dans la bibliothèque standard C++. Par exemple, la `string` classe implémente des opérations qui utilisent la sémantique de déplacement. Prenons l'exemple suivant qui concatène plusieurs chaînes et affiche le résultat :

C++

```
// string_concatenation.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s = string("h") + "e" + "ll" + "o";
   cout << s << endl;
}
```

Avant Visual Studio 2010, chaque appel à `operator+` alloue et retourne un nouvel objet temporaire `string` (une rvalue). `operator+` ne peut pas ajouter une chaîne à l'autre, car il ne sait pas si les chaînes sources sont des lvalues ou des rvalues. Si les chaînes sources sont toutes deux des valeurs lvalues, elles peuvent être référencées ailleurs dans le programme et ne doivent donc pas être modifiées. Vous pouvez modifier `operator+` pour prendre rvalues à l'aide de références rvalue, qui ne peuvent pas être référencées ailleurs dans le programme. Avec cette modification, `operator+` vous pouvez maintenant ajouter une chaîne à une autre. La modification réduit considérablement le nombre d'allocations de mémoire dynamiques que la `string` classe doit effectuer. Pour plus d'informations sur la `string` classe, consultez basic_string Classe.

La sémantique de déplacement aide également lorsque le compilateur ne peut pas utiliser l'optimisation de la valeur de retour (RVO) ou l'optimisation de la valeur de retour nommée (NRVO). Dans ces cas-là, le compilateur appelle le constructeur de déplacement si le type le définit.

Pour mieux comprendre la sémantique de déplacement, prenez comme exemple l'insertion d'un élément dans un objet `vector`. Si la capacité de l'objet `vector` est dépassée, l'objet `vector` doit réallouer suffisamment de mémoire pour ses éléments, puis copier chaque élément dans un autre emplacement de mémoire pour faire de la place pour l'élément inséré. Lorsqu'une opération d'insertion copie un élément, elle crée d'abord un nouvel élément. Ensuite, il appelle le constructeur de copie pour copier les données de l'élément précédent vers le nouvel élément. Enfin, il détruit l'élément précédent. La sémantique de déplacement vous permet de déplacer des objets

directement sans avoir à effectuer des opérations coûteuses d'allocation de mémoire et de copie.

Pour tirer parti de la sémantique de déplacement dans l'exemple `vector`, vous pouvez écrire un constructeur de déplacement pour déplacer des données d'un objet vers un autre.

Pour plus d'informations sur l'introduction de la sémantique de déplacement dans la bibliothèque C++ Standard dans Visual Studio 2010, consultez Bibliothèque C++ Standard.

Transfert parfait

Le transfert parfait réduit le besoin en fonctions surchargées et permet d'éviter le problème de transfert. Le *problème de transfert* peut se produire lorsque vous écrivez une fonction générique qui prend des références comme paramètres. Si elle transmet (ou *transfère*) ces paramètres à une autre fonction, par exemple, si elle prend un paramètre de type `const T&`, la fonction appelée ne peut pas modifier la valeur de ce paramètre. Si la fonction générique prend un paramètre de type `T&`, la fonction ne peut pas être appelée à l'aide d'une valeur rvalue (par exemple, un objet temporaire ou un littéral entier).

Normalement, pour résoudre ce problème, vous devez fournir des versions surchargées de la fonction générique qui acceptent `T&` et `const T&` pour chacun de ses paramètres. Par conséquent, le nombre de fonctions surchargées augmente de façon exponentielle avec le nombre de paramètres. Les références Rvalue vous permettent d'écrire une version d'une fonction qui accepte des arguments arbitraires. Ensuite, cette fonction peut les transférer vers une autre fonction comme si l'autre fonction avait été appelée directement.

Prenez l'exemple suivant qui déclare quatre types, `W`, `X`, `Y` et `Z`. Le constructeur pour chaque type prend une combinaison différente de **`const`** références non-lvalue **`const`** comme paramètres.

C++

```
struct W
{
   W(int&, int&) {}
};

struct X
{
   X(const int&, int&) {}
```

```
};

struct Y
{
   Y(int&, const int&) {}
};

struct Z
{
   Z(const int&, const int&) {}
};
```

Supposez que vous souhaitez écrire une fonction générique qui génère des objets. L'exemple suivant montre une manière d'écrire cette fonction :

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
   return new T(a1, a2);
}
```

L'exemple suivant montre un appel valide à la fonction `factory` :

C++

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

Toutefois, l'exemple suivant ne contient pas d'appel valide à la `factory` fonction. C'est parce que `factory` prend les références lvalue qui sont modifiables comme paramètres, mais elle est appelée à l'aide de rvalues :

C++

```
Z* pz = factory<Z>(2, 2);
```

Normalement, pour résoudre ce problème, vous devez créer une version surchargée de la fonction `factory` pour chaque combinaison de paramètres `A&` et `const A&`. Les références rvalue vous permettent d'écrire une version de la fonction `factory`, comme indiqué dans l'exemple suivant :

C++

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
   return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

Cet exemple utilise des références rvalue comme paramètres de la fonction `factory`. L'objectif de la std::forward fonction est de transférer les paramètres de la fonction de fabrique au constructeur de la classe de modèle.

L'exemple suivant illustre la fonction `main` qui utilise la fonction modifiée `factory` pour créer des instances des classes `W`, `X`, `Y` et `Z`. La fonction `factory` modifiée transfère ses paramètres (lvalues ou rvalues) au constructeur de classe approprié.

C++

```
int main()
{
   int a = 4, b = 5;
   W* pw = factory<W>(a, b);
   X* px = factory<X>(2, b);
   Y* py = factory<Y>(a, 2);
   Z* pz = factory<Z>(2, 2);

   delete pw;
   delete px;
   delete py;
   delete pz;
}
```

Propriétés des références rvalue

Vous pouvez surcharger une fonction pour qu'elle prenne une référence lvalue et une référence rvalue.

En surchargeant une fonction pour prendre une `const` référence lvalue ou une référence rvalue, vous pouvez écrire du code qui fait la distinction entre les objets non modifiables (lvalues) et les valeurs temporaires modifiables (rvalues). Vous pouvez passer un objet à une fonction qui prend une référence rvalue, sauf si l'objet est marqué comme `const`. L'exemple suivant illustre la fonction `f`, qui est surchargée pour accepter une référence lvalue et une référence rvalue. La fonction `main` appelle `f` avec des lvalues et une rvalue.

C++

```
// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
   // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
   cout << "In f(const MemoryBlock&). This version can't modify the
parameter." << endl;
}

void f(MemoryBlock&&)
{
   cout << "In f(MemoryBlock&&). This version can modify the parameter." <<
endl;
}

int main()
{
   MemoryBlock block;
   f(block);
   f(MemoryBlock());
}
```

Cet exemple produit la sortie suivante :

Output

```
In f(const MemoryBlock&). This version can't modify the parameter.
In f(MemoryBlock&&). This version can modify the parameter.
```

Dans cet exemple, le premier appel à `f` passe une variable locale (une lvalue) comme argument. Le deuxième appel à `f` passe un objet temporaire comme argument. Étant donné que l'objet temporaire ne peut pas être référencé ailleurs dans le programme, l'appel se lie à la version surchargée de `f` qui prend une référence rvalue, qui est libre de modifier l'objet.

Le compilateur traite une référence rvalue nommée comme une lvalue et une référence rvalue sans nom comme une rvalue.

Les fonctions qui prennent une référence rvalue en tant que paramètre traitent le paramètre comme une valeur lvalue dans le corps de la fonction. Le compilateur traite

une référence rvalue nommée comme une valeur lvalue. C'est parce qu'un objet nommé peut être référencé par plusieurs parties d'un programme. Il est dangereux d'autoriser plusieurs parties d'un programme à modifier ou supprimer des ressources de cet objet. Par exemple, si plusieurs parties d'un programme tentent de transférer des ressources à partir du même objet, seul le premier transfert réussit.

L'exemple suivant illustre la fonction `g`, qui est surchargée pour accepter une référence lvalue et une référence rvalue. La fonction `f` accepte une référence rvalue comme paramètre (une référence rvalue nommée) et retourne une référence rvalue (une référence rvalue sans nom). Dans l'appel à `g` à partir de `f`, la résolution de surcharge sélectionne la version de `g` qui accepte une référence lvalue, car le corps de `f` traite son paramètre comme une lvalue. Dans l'appel à `g` à partir de `main`, la résolution de surcharge sélectionne la version de `g` qui accepte une référence rvalue car `f` retourne une référence rvalue.

C++

```
// named-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
   // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
   cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
   cout << "In g(MemoryBlock&&)." << endl;
}

MemoryBlock&& f(MemoryBlock&& block)
{
   g(block);
   return move(block);
}

int main()
{
   g(f(MemoryBlock()));
}
```

Cet exemple produit la sortie suivante :

```
C++
```

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

Dans l'exemple, la `main` fonction passe une valeur rvalue à `f`. Le corps de `f` traite son paramètre nommé comme une lvalue. L'appel de `f` à `g` lie le paramètre à une référence lvalue (première version surchargée de `g`).

- **Vous pouvez convertir une lvalue en référence rvalue.**

La fonction bibliothèque std::move C++ Standard vous permet de convertir un objet en référence rvalue à cet objet. Vous pouvez également utiliser le **`static_cast`** mot clé pour convertir une valeur lvalue en référence rvalue, comme illustré dans l'exemple suivant :

```
C++
```

```
// cast-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
   // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
   cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
   cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
   MemoryBlock block;
   g(block);
   g(static_cast<MemoryBlock&&>(block));
}
```

Cet exemple produit la sortie suivante :

C++

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

Les modèles de fonctions déduisent leurs types d'argument template puis utilisent les règles de réduction de référence.

Un modèle de fonction qui transmet (ou *transfère*) ses paramètres à une autre fonction est un modèle courant. Il est important de comprendre comment fonctionne la déduction de type de modèle pour les modèles de fonction qui prennent des références rvalue.

Si l'argument de fonction est une rvalue, le compilateur déduit l'argument comme étant une référence rvalue. Par exemple, supposons que vous transmettiez une référence rvalue à un objet de type `X` à un modèle de fonction qui prend type `T&&` comme paramètre. La déduction de l'argument de modèle déduit `T` comme étant `X`, le paramètre a donc le type `X&&`. Si l'argument de fonction est une lvalue ou **`const`** lvalue, le compilateur déduit son type comme une référence lvalue ou **`const`** une référence lvalue de ce type.

L'exemple suivant déclare un modèle de structure, puis le spécialise pour différents types de références. La fonction `print_type_and_value` accepte une référence rvalue comme paramètre et la transfère à la version spécialisée appropriée de la méthode `S::print`. La fonction `main` illustre les différentes façons d'appeler la méthode `S::print`.

C++

```
// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

template<typename T> struct S;

// The following structures specialize S by
// lvalue reference (T&), const lvalue reference (const T&),
// rvalue reference (T&&), and const rvalue reference (const T&&).
// Each structure provides a print method that prints the type of
// the structure and its parameter.

template<typename T> struct S<T&> {
   static void print(T& t)
   {
      cout << "print<T&>: " << t << endl;
```

```
    }
};

template<typename T> struct S<const T&> {
    static void print(const T& t)
    {
        cout << "print<const T&>: " << t << endl;
    }
};

template<typename T> struct S<T&&> {
    static void print(T&& t)
    {
        cout << "print<T&&>: " << t << endl;
    }
};

template<typename T> struct S<const T&&> {
    static void print(const T&& t)
    {
        cout << "print<const T&&>: " << t << endl;
    }
};

// This function forwards its parameter to a specialized
// version of the S type.
template <typename T> void print_type_and_value(T&& t)
{
    S<T&&>::print(std::forward<T>(t));
}

// This function returns the constant string "fourth".
const string fourth() { return string("fourth"); }

int main()
{
    // The following call resolves to:
    // print_type_and_value<string&>(string& && t)
    // Which collapses to:
    // print_type_and_value<string&>(string& t)
    string s1("first");
    print_type_and_value(s1);

    // The following call resolves to:
    // print_type_and_value<const string&>(const string& && t)
    // Which collapses to:
    // print_type_and_value<const string&>(const string& t)
    const string s2("second");
    print_type_and_value(s2);

    // The following call resolves to:
    // print_type_and_value<string&&>(string&& t)
    print_type_and_value(string("third"));

    // The following call resolves to:
```

```
    // print_type_and_value<const string&&>(const string&& t)
    print_type_and_value(fourth());
}
```

Cet exemple produit la sortie suivante :

```
C++

print<T&>: first
print<const T&>: second
print<T&&>: third
print<const T&&>: fourth
```

Pour résoudre chaque appel à la `print_type_and_value` fonction, le compilateur effectue d'abord la déduction d'argument de modèle. Le compilateur applique ensuite des règles de réduction de référence lorsqu'il remplace les types de paramètres par les arguments de modèle déduits. Par exemple, le passage de la variable locale `s1` à la fonction `print_type_and_value` provoque la production par le compilateur de la signature de fonction suivante :

```
C++

print_type_and_value<string&>(string& && t)
```

Le compilateur utilise des règles de réduction de référence pour réduire la signature :

```
C++

print_type_and_value<string&>(string& t)
```

Cette version de la fonction `print_type_and_value` transfère ensuite son paramètre à la version spécialisée appropriée de la méthode `S::print`.

Le tableau suivant résume les règles de réduction de référence pour la déduction du type d'argument template :

Type développé	Type réduit
`T& &`	`T&`
`T& &&`	`T&`
`T&& &`	`T&`
`T&& &&`	`T&&`

La déduction d'argument template est un élément important de l'implémentation du transfert parfait. La section Transfert parfait décrit plus en détail le transfert parfait.

Récapitulatif

Les références rvalue différencient les lvalues des rvalues. Pour améliorer les performances de vos applications, elles peuvent éliminer le besoin d'allocations de mémoire et d'opérations de copie inutiles. Ils vous permettent également d'écrire une fonction qui accepte des arguments arbitraires. Cette fonction peut les transférer à une autre fonction comme si l'autre fonction avait été appelée directement.

Voir aussi

Expressions avec opérateurs unaires
Déclarateur de référence Lvalue : &
Lvalues et Rvalues
Déplacer des constructeurs et des opérateurs d'affectation de déplacement (C++)
Bibliothèque C++ standard

Arguments de fonction de type référence

Article • 16/06/2023

Il est souvent plus efficace de passer des références que de grands objets à des fonctions. Cela permet au compilateur de passer l'adresse de l'objet tout en maintenant la syntaxe qui aurait été utilisée pour accéder à l'objet. Prenons l'exemple suivant, qui utilise la structure `Date` :

C++

```
// reference_type_function_arguments.cpp
#include <iostream>

struct Date
{
    short Month;
    short Day;
    short Year;
};

// Create a date of the form DDDYYYY (day of year, year)
// from a Date.
long DateOfYear( Date& date )
{
    static int cDaysInMonth[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long dateOfYear = 0;

    // Add in days for months already elapsed.
    for ( int i = 0; i < date.Month - 1; ++i )
        dateOfYear += cDaysInMonth[i];

    // Add in days for this month.
    dateOfYear += date.Day;

    // Check for leap year.
    if ( date.Month > 2 &&
        (( date.Year % 100 != 0 || date.Year % 400 == 0 ) &&
        date.Year % 4 == 0 ))
        dateOfYear++;

    // Add in year.
    dateOfYear *= 10000;
    dateOfYear += date.Year;

    return dateOfYear;
}
```

```
int main()
{
    Date date{ 8, 27, 2018 };
    long dateOfYear = DateOfYear(date);
    std::cout << dateOfYear << std::endl;
}
```

Le code précédent indique que les membres d'une structure transmise par référence sont accessibles à l'aide de l'opérateur de sélection des membres (.) au lieu de l'opérateur de sélection de membre de pointeur (->).

Bien que les arguments passés en tant que types de référence observent la syntaxe des types autres que pointeurs, ils conservent une caractéristique importante des types de pointeurs : ils sont modifiables sauf s'ils sont déclarés comme `const`. Comme l'objectif du code précédent n'est pas de modifier l'objet `date`, un prototype de fonction plus approprié est :

C++

```
long DateOfYear( const Date& date );
```

Ce prototype garantit que la fonction `DateOfYear` ne modifiera pas son argument.

Toute fonction prototype comme prenant un type de référence peut accepter un objet du même type à sa place, car il existe une conversion standard de *typename* en *typename***&**.

Voir aussi

Informations de référence

Retours de fonction de type référence

Article • 03/04/2023

Les fonctions peuvent être déclarées pour retourner un type référence. Il existe deux raisons d'effectuer une déclaration :

- Les informations retournées sont un objet suffisamment grand pour que retourner une référence soit plus efficace que retourner une copie.
- Le type de la fonction doit être une l-value.
- L'objet référencé ne sort pas de la portée lors du retour de la fonction.

Tout comme il peut être plus efficace de passer des objets volumineux *à* des fonctions par référence, il peut également être plus efficace de retourner des objets volumineux *à partir de* fonctions par référence. Le protocole de retour par référence élimine le besoin de copier l'objet dans un emplacement temporaire avant de le retourner.

Les types de retour par référence peuvent également être utiles lorsque la fonction doit prendre une l-value. La plupart des opérateurs surchargés appartiennent à cette catégorie, notamment l'opérateur d'assignation. Les opérateurs surchargés sont couverts dans Opérateurs surchargés.

Exemple

Prenons l'exemple de `Point` :

C++

```
// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
// Define "accessor" functions as
//  reference types.
unsigned& x();
unsigned& y();
private:
// Note that these are declared at class scope:
unsigned obj_x;
unsigned obj_y;
```

```
};

unsigned& Point :: x()
{
return obj_x;
}
unsigned& Point :: y()
{
return obj_y;
}

int main()
{
Point ThePoint;
// Use x() and y() as l-values.
ThePoint.x() = 7;
ThePoint.y() = 9;

// Use x() and y() as r-values.
cout << "x = " << ThePoint.x() << "\n"
<< "y = " << ThePoint.y() << "\n";
}
```

Sortie

Output

```
x = 7
y = 9
```

Notez que les fonctions `x` et `y` sont déclarées comme retournant des types référence. Ces fonctions peuvent être utilisées des deux côtés d'une instruction d'assignation.

Notez également que, dans main, l'objet Point reste dans la portée et ses membres de référence sont par conséquent toujours actifs et accessibles en toute sécurité.

Les déclarations des types référence doivent contenir des initialiseurs sauf dans les cas suivants :

- Déclaration explicite **`extern`**
- Déclaration d'une classe membre
- Déclaration sans classe
- Déclaration d’un argument d’une fonction ou du type de retour pour une fonction

Attention en retournant l'adresse d'une variable locale

Si vous déclarez un objet dans la portée locale, cet objet sera détruit lors du retour de la fonction. Si la fonction retourne une référence à cet objet, cette référence est susceptible de provoquer une violation d'accès lors de l'exécution si l'appelant tente d'utiliser la référence null.

C++

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

Le compilateur émet un avertissement dans ce cas : `warning C4172: returning address of local variable or temporary`. Dans des programmes simples, il est possible qu'occasionnellement aucune violation d'accès ne se produise si la référence est accessible par l'appelant avant le remplacement de l'emplacement de mémoire. C'est le fruit du hasard. Tenez compte de l'avertissement.

Voir aussi

Informations de référence

Références aux pointeurs

Article • 03/04/2023

Les références aux pointeurs peuvent être déclarées à peu près de la même façon que les références aux objets. Une référence à un pointeur est une valeur modifiable qui est utilisée comme un pointeur normal.

Exemple

Cet exemple de code montre la différence entre l'utilisation d'un pointeur vers un pointeur et d'une référence à un pointeur.

Les fonctions `Add1` et `Add2` sont fonctionnellement équivalentes, bien qu'elles ne soient pas appelées de la même façon. La différence est que `Add1` utilise une double indirection, mais `Add2` utilise la commodité d'une référence à un pointeur.

C++

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library namespace
using namespace std;

enum {
   sizeOfBuffer = 132
};

// Define a binary tree structure.
struct BTree {
   char  *szText;
   BTree *Left;
   BTree *Right;
};

// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

int main( int argc, char *argv[] ) {
   // Usage message
   if( argc < 2 ) {
```

```
      cerr << "Usage: " << argv[0] << " [1 | 2]" << "\n";
      cerr << "\nwhere:\n";
      cerr << "1 uses double indirection\n";
      cerr << "2 uses a reference to a pointer.\n";
      cerr << "\nInput is from stdin. Use ^Z to terminate input.\n";
      return 1;
   }

   char *szBuf = new char[sizeOfBuffer];
   if (szBuf == NULL) {
      cerr << "Out of memory!\n";
      return -1;
   }

   // Read a text file from the standard input device and
   //  build a binary tree.
   while( !cin.eof() )
   {
      cin.get( szBuf, sizeOfBuffer, '\n' );
      cin.get();

      if ( strlen( szBuf ) ) {
         switch ( *argv[1] ) {
            // Method 1: Use double indirection.
            case '1':
               Add1( &btRoot, szBuf );
               break;
            // Method 2: Use reference to a pointer.
            case '2':
               Add2( btRoot, szBuf );
               break;
            default:
               cerr << "Illegal value '"
                  << *argv[1]
                  << "' supplied for add method.\n"
                     << "Choose 1 or 2.\n";
               return -1;
         }
      }
   }
   // Display the sorted list.
   PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTree* MybtRoot ) {
   // Traverse the left branch of the tree recursively.
   if ( MybtRoot->Left )
      PrintTree( MybtRoot->Left );

   // Print the current node.
   cout << MybtRoot->szText << "\n";

   // Traverse the right branch of the tree recursively.
   if ( MybtRoot->Right )
```

```
      PrintTree( MybtRoot->Right );
}

// Add1: Add a node to the binary tree.
//       Uses double indirection.
int Add1( BTree **Root, char *szToAdd ) {
   if ( (*Root) == 0 ) {
      (*Root) = new BTree;
      (*Root)->Left = 0;
      (*Root)->Right = 0;
      (*Root)->szText = new char[strlen( szToAdd ) + 1];
      strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
      return 1;
   }
   else {
      if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
         return Add1( &((*Root)->Left), szToAdd );
      else
         return Add1( &((*Root)->Right), szToAdd );
   }
}

// Add2: Add a node to the binary tree.
//       Uses reference to pointer
int Add2( BTree*& Root, char *szToAdd ) {
   if ( Root == 0 ) {
      Root = new BTree;
      Root->Left = 0;
      Root->Right = 0;
      Root->szText = new char[strlen( szToAdd ) + 1];
      strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
      return 1;
   }
   else {
      if ( strcmp( Root->szText, szToAdd ) > 0 )
         return Add2( Root->Left, szToAdd );
      else
         return Add2( Root->Right, szToAdd );
   }
}
```

Output

```
Usage: references_to_pointers.exe [1 | 2]

where:
1 uses double indirection
2 uses a reference to a pointer.

Input is from stdin. Use ^Z to terminate input.
```

Voir aussi

Informations de référence

Pointeurs (C++)

Article • 16/06/2023

Un pointeur est une variable qui stocke l'adresse mémoire d'un objet. Les pointeurs sont largement utilisés dans C et C++ à trois fins main :

- pour allouer de nouveaux objets sur le tas,
- pour passer des fonctions à d'autres fonctions
- pour itérer sur des éléments dans des tableaux ou d'autres structures de données.

Dans la programmation de style C, *des pointeurs bruts* sont utilisés pour tous ces scénarios. Toutefois, les pointeurs bruts sont la source de nombreuses erreurs de programmation graves. Par conséquent, leur utilisation est fortement déconseillée, sauf si elles offrent un avantage significatif en matière de performances et qu'il n'existe aucune ambiguïté quant au *pointeur propriétaire* responsable de la suppression de l'objet. C++ moderne fournit des *pointeurs intelligents* pour l'allocation d'objets, *des itérateurs* pour parcourir des structures de données et des *expressions lambda* pour passer des fonctions. En utilisant ces fonctionnalités de langage et de bibliothèque au lieu de pointeurs bruts, vous allez rendre votre programme plus sûr, plus facile à déboguer et plus simple à comprendre et à gérer. Pour plus d'informations, consultez Pointeurs intelligents, itérateurs et expressions lambda .

Contenu de cette section

- Pointeurs bruts
- Pointeurs const et volatiles
- Opérateurs new et delete
- Pointeurs intelligents
- Procédure : Créer et utiliser des instances unique_ptr
- Procédure : Créer et utiliser des instances shared_ptr
- Procédure : Créer et utiliser des instances weak_ptr
- Procédure : Créer et utiliser des instances CComPtr et CComQIPtr

Voir aussi

Itérateurs
Expressions lambda

Pointeurs bruts (C++)

Article • 16/06/2023

Un *pointeur* est un type de variable. Il stocke l'adresse d'un objet en mémoire et est utilisé pour accéder à cet objet. Un *pointeur brut* est un pointeur dont la durée de vie n'est pas contrôlée par un objet d'encapsulation, tel qu'un pointeur intelligent. Un pointeur brut peut se voir attribuer l'adresse d'une autre variable non pointeur, ou une valeur de nullptr. Un pointeur qui n'a pas reçu de valeur contient des données aléatoires.

Un pointeur peut également être *déréférencé* pour récupérer la valeur de l'objet vers lequel il pointe. *L'opérateur d'accès membre* fournit l'accès aux membres d'un objet.

C++

```
int* p = nullptr; // declare pointer and initialize it
                  // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

Un pointeur peut pointer vers un objet typé ou vers `void`. Lorsqu'un programme alloue un objet sur le tas en mémoire, il reçoit l'adresse de cet objet sous la forme d'un pointeur. Ces pointeurs sont appelés *pointeurs propriétaires*. Un pointeur propriétaire (ou une copie de celui-ci) doit être utilisé pour libérer explicitement l'objet alloué au tas lorsqu'il n'est plus nécessaire. Le fait de ne pas libérer la mémoire entraîne une *fuite de mémoire* et rend cet emplacement de mémoire indisponible pour tout autre programme sur l'ordinateur. La mémoire allouée à l'aide `new` de doit être libérée à l'aide de `delete` (ou `delete[]`). Pour plus d'informations, consultez new opérateurs et delete.

C++

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

Un pointeur (s'il n'est pas déclaré comme `const`) peut être incrémenté ou décrémenté pour pointer vers un autre emplacement en mémoire. Cette opération est appelée *arithmétique de pointeur*. Il est utilisé dans la programmation de style C pour itérer sur des éléments dans des tableaux ou d'autres structures de données. Un `const` pointeur ne peut pas être fait pour pointer vers un autre emplacement de mémoire, et dans ce

sens est similaire à une référence. Pour plus d'informations, consultez const pointeurs et volatile.

C++

```
    // declare a C-style string. Compiler adds terminating '\0'.
    const char* str = "Hello world";

    const int c = 1;
    const int* pconst = &c; // declare a non-const pointer to const int
    const int c2 = 2;
    pconst = &c2;  // OK pconst itself isn't const
    const int* const pconst2 = &c;
    // pconst2 = &c2; // Error! pconst2 is const.
```

Sur les systèmes d'exploitation 64 bits, un pointeur a une taille de 64 bits. La taille du pointeur d'un système détermine la quantité de mémoire adressable qu'il peut avoir. Toutes les copies d'un pointeur pointent vers le même emplacement de mémoire. Les pointeurs (ainsi que les références) sont largement utilisés en C++ pour passer des objets plus volumineux vers et depuis des fonctions. Il est souvent plus efficace de copier l'adresse d'un objet que de copier l'objet entier. Lors de la définition d'une fonction, spécifiez les paramètres de pointeur comme `const` , sauf si vous envisagez que la fonction modifie l'objet . En général, `const` les références sont la méthode préférée pour passer des objets à des fonctions, sauf si la valeur de l'objet peut être `nullptr`.

Les pointeurs vers les fonctions permettent de passer des fonctions à d'autres fonctions. Ils sont utilisés pour les « rappels » dans la programmation de style C. C++ moderne utilise des expressions lambda à cet effet.

Initialisation et accès aux membres

L'exemple suivant montre comment déclarer, initialiser et utiliser un pointeur brut. Il est initialisé à l'aide `new` de pour pointer un objet alloué sur le tas, que vous devez explicitement `delete`. L'exemple montre également quelques-uns des dangers associés aux pointeurs bruts. (N'oubliez pas que cet exemple est de programmation de style C et non de C++moderne!)

C++

```
#include <iostream>
#include <string>

class MyClass
{
public:
```

```
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

// Accepts a MyClass pointer
void func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}

// Accepts a MyClass object
void func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}


int main()
{
    // Use the * operator to declare a pointer type
    // Use new to allocate and initialize memory
    MyClass* pmc = new MyClass{ 108, "Nick" };

    // Prints the memory address. Usually not what you want.
    std:: cout << pmc << std::endl;

    // Copy the pointed-to object by dereferencing the pointer
    // to access the contents of the memory location.
    // mc is a separate object, allocated here on the stack
    MyClass mc = *pmc;

    // Declare a pointer that points to mc using the addressof operator
    MyClass* pcopy = &mc;

    // Use the -> operator to access the object's public members
    pmc->print(); // "Nick, 108"

    // Copy the pointer. Now pmc and pmc2 point to same object!
    MyClass* pmc2 = pmc;

    // Use copied pointer to modify the original object
    pmc2->name = "Erika";
    pmc->print(); // "Erika, 108"
    pmc2->print(); // "Erika, 108"

    // Pass the pointer to a function.
```

```
    func_A(pmc);
    pmc->print(); // "Erika, 3"
    pmc2->print(); // "Erika, 3"

    // Dereference the pointer and pass a copy
    // of the pointed-to object to a function
    func_B(*pmc);
    pmc->print(); // "Erika, 3" (original not modified by function)

    delete(pmc); // don't forget to give memory back to operating system!
   // delete(pmc2); //crash! memory location was already deleted
}
```

Arithmétique du pointeur et tableaux

Les pointeurs et les tableaux sont étroitement liés. Lorsqu'un tableau est passé par valeur à une fonction, il est passé en tant que pointeur vers le premier élément. L'exemple suivant illustre les propriétés importantes suivantes des pointeurs et des tableaux :

- L'opérateur `sizeof` retourne la taille totale en octets d'un tableau
- Pour déterminer le nombre d'éléments, divisez le nombre total d'octets par la taille d'un élément
- Lorsqu'un tableau est passé à une fonction, il *se décompose* en un type pointeur
- Lorsque l'opérateur `sizeof` est appliqué à un pointeur, il retourne la taille du pointeur, par exemple, 4 octets sur x86 ou 8 octets sur x64

C++

```
#include <iostream>

void func(int arr[], int length)
{
    // returns pointer size. not useful here.
    size_t test = sizeof(arr);

    for(int i = 0; i < length; ++i)
    {
        std::cout << arr[i] << " ";
    }
}

int main()
{

    int i[5]{ 1,2,3,4,5 };
    // sizeof(i) = total bytes
    int j = sizeof(i) / sizeof(i[0]);
```

```
    func(i,j);
}
```

Certaines opérations arithmétiques peuvent être utilisées sur des pointeurs non-pour`const` les faire pointer vers un autre emplacement de mémoire. Les pointeurs sont incrémentés et décrémentés à l'aide des `++`opérateurs , `+=`et `--` `--` . Cette technique peut être utilisée dans des tableaux et est particulièrement utile dans les mémoires tampons de données non typées. Un `void*` est incrémenté par la taille d'un **`char`** (1 octet). Un pointeur typé est incrémenté par la taille du type vers lequel il pointe.

L'exemple suivant montre comment l'arithmétique de pointeur peut être utilisée pour accéder à des pixels individuels dans une bitmap sur Windows. Notez l'utilisation de **`new`** et **`delete`**, et de l'opérateur de déréférencement.

C++

```
#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{

    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    constexpr int bufferSize = 30000;
    unsigned char* buffer = new unsigned char[bufferSize];

    BITMAPFILEHEADER bf;
    bf.bfType = 0x4D42;
    bf.bfSize = header.biSize + 14 + bufferSize;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

    // Create a gray square with a 2-pixel wide outline.
    unsigned char* begin = &buffer[0];
    unsigned char* end = &buffer[0] + bufferSize;
    unsigned char* p = begin;
    constexpr int pixelWidth = 3;
    constexpr int borderWidth = 2;

    while (p < end)
```

```
    {
            // Is top or bottom edge?
        if ((p < begin + header.biWidth * pixelWidth * borderWidth)
            || (p > end - header.biWidth * pixelWidth * borderWidth)
            // Is left or right edge?
            || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth *
pixelWidth)
            || (p - begin) % (header.biWidth * pixelWidth) >
((header.biWidth - borderWidth) * pixelWidth))
        {
            *p = 0x0; // Black
        }
        else
        {
            *p = 0xC3; // Gray
        }
        p++; // Increment one byte sizeof(unsigned char).
    }

    ofstream wf(R"(box.bmp)", ios::out | ios::binary);

    wf.write(reinterpret_cast<char*>(&bf), sizeof(bf));
    wf.write(reinterpret_cast<char*>(&header), sizeof(header));
    wf.write(reinterpret_cast<char*>(begin), bufferSize);

    delete[] buffer; // Return memory to the OS.
    wf.close();
}
```

`void*` Pointeurs

Un pointeur vers `void` pointe simplement vers un emplacement de mémoire brute. Parfois, il est nécessaire d'utiliser `void*` des pointeurs, par exemple lors du passage entre du code C++ et des fonctions C.

Lorsqu'un pointeur typé est converti en pointeur `void` , le contenu de l'emplacement de mémoire est inchangé. Toutefois, les informations de type sont perdues, de sorte que vous ne pouvez pas effectuer d'opérations d'incrémentation ou de décrémentation. Un emplacement de mémoire peut être converti, par exemple, de `MyClass*` à `void*` et à nouveau en `MyClass*`. Ces opérations sont intrinsèquement sujettes aux erreurs et nécessitent une grande attention auxvoid erreurs. C++ moderne décourage l'utilisation de `void` pointeurs dans presque toutes les circonstances.

C++

```
//func.c
void func(void* data, int length)
{
```

```
    char* c = (char*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    void func(void* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

int main()
{
    MyClass* mc = new MyClass{10, "Marian"};
    void* p = static_cast<void*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"
    delete(mc);

    // use operator new to allocate untyped memory block
    void* pvoid = operator new(1000);
    char* pchar = static_cast<char*>(pvoid);
    for(char* c = pchar; c < pchar + 1000; ++c)
    {
        *c = 0x00;
    }
    func(pvoid, 1000);
    char ch = static_cast<char*>(pvoid)[0];
    std::cout << ch << std::endl; // 'A'
    operator delete(pvoid);
}
```

Pointeurs désignant des fonctions

Dans la programmation de style C, les pointeurs de fonction sont principalement utilisés pour passer des fonctions à d'autres fonctions. Cette technique permet à l'appelant de

personnaliser le comportement d'une fonction sans la modifier. En C++moderne, les expressions lambda offrent la même fonctionnalité avec une plus grande sécurité de type et d'autres avantages.

Une déclaration de pointeur de fonction spécifie la signature que la fonction pointée doit avoir :

C++

```
// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
void (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);
```

L'exemple suivant montre une fonction `combine` qui prend comme paramètre toute fonction qui accepte un `std::string` et retourne un `std::string`. Selon la fonction passée à `combine`, elle ajoute ou ajoute une chaîne.

C++

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s)
{
    return base.append(" ").append(s);
}

string prepend(string s)
{
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a))
{
    return (*g)(s);
}

int main()
```

```
{
    cout << combine("from MSVC", append) << "\n";
    cout << combine("Good morning and", prepend) << "\n";
}
```

Voir aussi

Opérateur d’indirectionde pointeurs intelligents : *

Opérateur d’adresse : &

Bienvenue dans C++

Pointeurs const et volatile

Article • 16/06/2023

Les mots clés const et volatile changent la façon dont les pointeurs sont traités. Le `const` mot clé spécifie que le pointeur ne peut pas être modifié après l'initialisation ; le pointeur est protégé contre toute modification par la suite.

Le `volatile` mot clé spécifie que la valeur associée au nom qui suit peut être modifiée par des actions autres que celles de l'application utilisateur. Par conséquent, le `volatile` mot clé est utile pour déclarer des objets en mémoire partagée accessibles par plusieurs processus ou zones de données globales utilisées pour la communication avec des routines de service d'interruption.

Lorsqu'un nom est déclaré comme `volatile`, le compilateur recharge la valeur à partir de la mémoire chaque fois que le programme y accède. Cela réduit considérablement les optimisations possibles. Toutefois, lorsque l'état d'un objet peut changer de manière inattendue, c'est la seule façon de garantir des performances prévisibles du programme.

Pour déclarer l'objet pointé par le pointeur en tant que `const` ou `volatile`, utilisez une déclaration de la forme :

C++

```
const char *cpch;
volatile char *vpch;
```

Pour déclarer la valeur du pointeur(autrement dit, l'adresse réelle stockée dans le pointeur) en tant que `const` ou `volatile`, utilisez une déclaration au format :

C++

```
char * const pchc;
char * volatile pchv;
```

Le langage C++ empêche les affectations qui autoriseraient la modification d'un objet ou d'un pointeur déclaré comme `const`. De telles assignations supprimeraient les informations avec lesquelles l'objet ou le pointeur a été déclaré, ce qui entraînerait la violation de l'objectif de la déclaration d'origine. Prenons les déclarations suivantes :

C++

```
const char cch = 'A';
char ch = 'B';
```

Étant donné les déclarations précédentes de deux objets (`cch`, de type **const char** et `ch`, de type **char),** les déclarations/initialisations suivantes sont valides :

C++

```
const char *pch1 = &cch;
const char *const pch4 = &cch;
const char *pch5 = &ch;
char *pch6 = &ch;
char *const pch7 = &ch;
const char *const pch8 = &ch;
```

Les déclarations/initialisations suivantes sont erronées.

C++

```
char *pch2 = &cch;   // Error
char *const pch3 = &cch;   // Error
```

La déclaration de `pch2` déclare un pointeur par l'intermédiaire duquel un objet constant peut être modifié et par conséquent désactivé. La déclaration de `pch3` spécifie que le pointeur est constant, et non l'objet ; la déclaration n'est pas autorisée pour la même raison que la `pch2` déclaration n'est pas autorisée.

Les huit assignations suivantes indiquent l'assignation par l'intermédiaire du pointeur et la modification de la valeur du pointeur pour les déclarations précédentes ; pour le moment, supposons que l'initialisation est correcte de `pch1` à `pch8`.

C++

```
*pch1 = 'A';  // Error: object declared const
pch1 = &ch;   // OK: pointer not declared const
*pch2 = 'A';  // OK: normal pointer
pch2 = &ch;   // OK: normal pointer
*pch3 = 'A';  // OK: object not declared const
pch3 = &ch;   // Error: pointer declared const
*pch4 = 'A';  // Error: object declared const
pch4 = &ch;   // Error: pointer declared const
```

Les pointeurs déclarés en tant que **`volatile`**, ou en tant que mélange de **`const`** et **`volatile`**, obéissent aux mêmes règles.

Les pointeurs vers `const` des objets sont souvent utilisés dans les déclarations de fonction comme suit :

C++

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char
*strSource );
```

L'instruction précédente déclare une fonction, strcpy_s, où deux des trois arguments sont de type pointeur vers `char`. Étant donné que les arguments sont passés par référence et non par valeur, la fonction serait libre de modifier à la fois `strDestination` et `strSource` si `strSource` n'étaient pas déclarés comme `const`. La déclaration de `strSource` as `const` garantit que l'appelant ne `strSource` peut pas être modifié par la fonction appelée.

> ⓘ **Notes**
>
> Étant donné qu'il existe une conversion standard de *typename** en `const` *typename**, il est légal de passer un argument de type `char *` à **strcpy_s**. Toutefois, l'inverse n'est pas vrai; aucune conversion implicite n'existe pour supprimer l'attribut `const` d'un objet ou d'un pointeur.

Un `const` pointeur d'un type donné peut être affecté à un pointeur du même type. Toutefois, un pointeur qui ne peut pas `const` être affecté à un `const` pointeur. L'exemple de code suivant montre des assignations correctes et incorrectes :

C++

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
pObject = cpObject;
cpObject = pObject;   // C3892
}
```

L'exemple suivant montre comment déclarer un objet comme const si vous avez un pointeur désignant un pointeur vers un objet.

C++

```
// const_pointer2.cpp
struct X {
   X(int i) : m_i(i) { }
   int m_i;
};

int main() {
   // correct
   const X cx(10);
   const X * pcx = &cx;
   const X ** ppcx = &pcx;

   // also correct
   X const cx2(20);
   X const * pcx2 = &cx2;
   X const ** ppcx2 = &pcx2;
}
```

Voir aussi

Pointeurs pointeursbruts

Opérateurs `new` et `delete`

Article • 16/06/2023

C++ prend en charge l'allocation dynamique et la désallocation d'objets à l'aide des new opérateurs et delete . Ces opérateurs allouent *de la mémoire* pour les objets d'un pool appelé *magasin libre* (également appelé tas). L'opérateur `new` appelle la fonction operator newspéciale et l'opérateur `delete` appelle la fonction operator deletespéciale .

Pour obtenir la liste des fichiers de bibliothèque dans la bibliothèque runtime C et la bibliothèque standard C++, consultez Fonctionnalités de la bibliothèque CRT.

Opérateur `new`

Le compilateur traduit une instruction telle que celle-ci en un appel à la fonction `operator new`:

C++

```
char *pch = new char[BUFFER_SIZE];
```

Si la demande concerne zéro octet de stockage, `operator new` retourne un pointeur vers un objet distinct. Autrement dit, des appels répétés pour `operator new` retourner différents pointeurs.

Si la mémoire est insuffisante pour la demande d'allocation, `operator new` lève une `std::bad_alloc` exception. Ou bien, elle retourne `nullptr` si vous avez utilisé le formulaire `new(std::nothrow)` de *placement* ou si vous avez lié dans la prise en charge sans levée`operator new`. Pour plus d'informations, consultez Comportement d'échec d'allocation.

Les deux étendues pour `operator new` les fonctions sont décrites dans le tableau suivant.

Étendue des `operator new` fonctions

Opérateur	Étendue
`::operator new`	Global
class-name`::operator new`	Classe

Le premier argument de `operator new` doit être de type `size_t`, et le type de retour est toujours `void*`.

La fonction globale `operator new` est appelée lorsque l'opérateur `new` est utilisé pour allouer des objets de types intégrés, des objets de type classe qui ne contiennent pas de fonctions définies par `operator new` l'utilisateur et des tableaux d'un type quelconque. Lorsque l'opérateur `new` est utilisé pour allouer des objets d'un type de classe où un `operator new` est défini, celui de `operator new` cette classe est appelé.

Une `operator new` fonction définie pour une classe est une fonction membre statique (qui ne peut pas être virtuelle) qui masque la fonction globale `operator new` pour les objets de ce type de classe. Considérez le cas où `new` est utilisé pour allouer et définir la mémoire sur une valeur donnée :

C++

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};
void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}
// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
   Blanks *a5 = new(0xa5) Blanks;
   return a5 != 0;
}
```

L'argument fourni entre parenthèses à `new` est passé à `Blanks::operator new` en tant qu'argument `chInit` . Toutefois, la fonction globale `operator new` est masquée, ce qui entraîne la génération d'une erreur dans le code suivant :

C++

```
Blanks *SomeBlanks = new Blanks;
```

Le compilateur prend en charge le tableau `new` de membres et `delete` les opérateurs dans une déclaration de classe. Par exemple :

C++

```
class MyClass
{
public:
   void * operator new[] (size_t)
   {
      return 0;
   }
   void   operator delete[] (void*)
   {
   }
};

int main()
{
   MyClass *pMyClass = new MyClass[5];
   delete [] pMyClass;
}
```

Comportement d'échec d'allocation

La `new` fonction de la bibliothèque standard C++ prend en charge le comportement spécifié dans la norme C++ depuis C++98. Quand la mémoire est insuffisante pour une demande d'allocation, `operator new` lève une std::bad_alloc exception.

Le code C++ plus ancien a renvoyé un pointeur Null pour une allocation ayant échoué. Si vous avez du code qui attend la version sans levée de `new`, liez votre programme avec `nothrownew.obj`. Le `nothrownew.obj` fichier remplace global `operator new` par une version qui retourne `nullptr` en cas d'échec d'une allocation. `operator new` ne lève `std::bad_alloc` plus . Pour plus d'informations sur `nothrownew.obj` et d'autres fichiers d'options de l'éditeur de liens, consultez Options de liaison.

Vous ne pouvez pas combiner le code qui recherche des exceptions de global `operator new` avec le code qui recherche les pointeurs Null dans la même application. Toutefois, vous pouvez toujours créer une classe locale `operator new` qui se comporte différemment. Cette possibilité signifie que le compilateur doit agir de manière défensive par défaut et inclure des vérifications pour les retours de pointeur null dans

les `new` appels. Pour plus d'informations sur la façon d'optimiser ces vérifications du compilateur, consultez /Zc:throwingnew.

Gestion d'une mémoire insuffisante

La façon dont vous testez une allocation ayant échoué à partir d'une `new` expression varie selon que vous utilisez le mécanisme d'exception standard ou si vous utilisez un `nullptr` retour. Standard C++ s'attend à ce qu'un allocateur lève ou `std::bad_alloc` une classe dérivée de `std::bad_alloc`. Vous pouvez gérer une exception comme indiqué dans cet exemple :

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   try {
      int *pI = new int[BIG_NUMBER];
   }
   catch (bad_alloc& ex) {
      cout << "Caught bad_alloc: " << ex.what() << endl;
      return -1;
   }
}
```

Lorsque vous utilisez la `nothrow` forme de `new`, vous pouvez tester un échec d'allocation, comme indiqué dans cet exemple :

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   int *pI = new(nothrow) int[BIG_NUMBER];
   if ( pI == nullptr ) {
      cout << "Insufficient memory" << endl;
      return -1;
   }
}
```

Vous pouvez tester l'échec de l'allocation de mémoire lorsque vous avez utilisé *`nothrownew.obj`* le fichier pour remplacer global `operator new` , comme indiqué ici :

C++

```
#include <iostream>
#include <new>
using namespace std;
#define BIG_NUMBER 10000000000LL
int main() {
   int *pI = new int[BIG_NUMBER];
   if ( !pI ) {
      cout << "Insufficient memory" << endl;
      return -1;
   }
}
```

Vous pouvez fournir un gestionnaire pour les demandes d'allocation de mémoire ayant échoué. Il est possible d'écrire une routine de récupération personnalisée pour gérer un tel échec. Il peut, par exemple, libérer de la mémoire réservée, puis autoriser l'allocation à s'exécuter à nouveau. Pour plus d'informations, consultez _set_new_handler.

Opérateur `delete`

La mémoire allouée dynamiquement à l'aide de l'opérateur `new` peut être libérée à l'aide de l'opérateur `delete` . L'opérateur delete appelle la `operator delete` fonction, qui libère de la mémoire dans le pool disponible. L'utilisation de l'opérateur `delete` entraîne également l'appel du destructeur de classe (le cas échéant).

Il existe des fonctions globales et étendues à la `operator delete` classe. `operator delete` Une seule fonction peut être définie pour une classe donnée ; si elle est définie, elle masque la fonction globale `operator delete`. La fonction globale `operator delete` est toujours appelée pour les tableaux de n'importe quel type.

Fonction globale `operator delete` . Il existe deux formes pour les fonctions globales `operator delete` et membres `operator delete` de classe :

C++

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Une seule des deux formes précédentes peut être présente pour une classe donnée. Le premier formulaire prend un seul argument de type `void *`, qui contient un pointeur vers l'objet à libérer. La deuxième forme, la désallocation dimensionnée, prend deux arguments : le premier est un pointeur vers le bloc de mémoire à libérer, et le second

est le nombre d'octets à libérer. Le type de retour des deux formulaires est `void` (`operator delete` ne peut pas retourner de valeur).

L'objectif du deuxième formulaire est d'accélérer la recherche de la catégorie de taille correcte de l'objet à supprimer. Ces informations ne sont souvent pas stockées près de l'allocation elle-même et sont probablement mises en cache. Le deuxième formulaire est utile lorsqu'une `operator delete` fonction d'une classe de base est utilisée pour supprimer un objet d'une classe dérivée.

La `operator delete` fonction étant statique, elle ne peut pas être virtuelle. La `operator delete` fonction obéit au contrôle d'accès, comme décrit dans Member-Access Control.

L'exemple suivant montre les fonctions définies par `operator new` l'utilisateur et `operator delete` conçues pour journaliser les allocations et les désallocations de la mémoire :

C++

```
#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0;  // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
   static int fInOpNew = 0;   // Guard flag.

   if ( fLogMemory && !fInOpNew ) {
      fInOpNew = 1;
      clog << "Memory block " << ++cBlocksAllocated
          << " allocated for " << stAllocateBlock
          << " bytes\n";
      fInOpNew = 0;
   }
   return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
   static int fInOpDelete = 0;   // Guard flag.
   if ( fLogMemory && !fInOpDelete ) {
      fInOpDelete = 1;
      clog << "Memory block " << cBlocksAllocated--
          << " deallocated\n";
      fInOpDelete = 0;
   }

   free( pvMem );
}
```

```
int main( int argc, char *argv[] ) {
   fLogMemory = 1;   // Turn logging on
   if( argc > 1 )
      for( int i = 0; i < atoi( argv[1] ); ++i ) {
         char *pMem = new char[10];
         delete[] pMem;
      }
   fLogMemory = 0;  // Turn logging off.
   return cBlocksAllocated;
}
```

Le code précédent peut être utilisé pour détecter les « fuites de mémoire », c'est-à-dire la mémoire allouée sur le magasin libre, mais jamais libérée. Pour détecter les fuites, les opérateurs globaux **`new`** et **`delete`** sont redéfinis pour compter l'allocation et la désallocation de la mémoire.

Le compilateur prend en charge le tableau **`new`** de membres et **`delete`** les opérateurs dans une déclaration de classe. Par exemple :

C++

```
// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X  {
public:
   void * operator new[] (size_t) {
      return 0;
   }
   void operator delete[] (void*) {}
};

void f() {
   X *pX = new X[5];
   delete [] pX;
}
```

Pointeurs intelligents (C++ moderne)

Article • 16/06/2023

Dans la programmation C++ moderne, la bibliothèque standard comprend *des pointeurs intelligents*, qui sont utilisés pour garantir que les programmes sont exempts de fuites de mémoire et de ressources et qu'ils sont sans exception.

Utilisations pour les pointeurs intelligents

Les pointeurs intelligents sont définis dans l'espace de `std` noms dans le fichier d'en-tête <mémoire> . Elles sont cruciales pour l'idiome de programmation RAII ou *Resource Acquisition Is Initialization* . L'objectif principal de cet idiome est de garantir que l'acquisition des ressources se produit en même temps que l'initialisation de l'objet, afin que toutes les ressources de l'objet soient créées et préparées en une seule ligne de code. En pratique, le grand principe de RAII est de donner la propriété de toute ressource allouée par tas, par exemple la mémoire allouée dynamiquement ou les handles d'objet système, à un objet alloué par la pile dont le destructeur contient le code de suppression ou de libération de la ressource ainsi que le code de nettoyage associé.

Dans la plupart des cas, lorsque vous initialisez un pointeur brut ou un handle de ressource pour pointer vers une ressource réelle, passez immédiatement le pointeur à un pointeur intelligent. En C++ moderne, les pointeurs bruts sont utilisés uniquement dans les petits blocs de code à portée limitée, les boucles ou les fonctions d'assistance lorsque les performances sont essentielles et qu'il n'existe aucune chance de confusion en matière de propriété.

L'exemple suivant compare une déclaration de pointeur brut à une déclaration de pointeur intelligent.

C++

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}
```

```
void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

Comme indiqué dans l'exemple, un pointeur intelligent est un modèle de classe que vous déclarez dans la pile et que vous initialisez en utilisant un pointeur brut qui pointe vers un objet alloué par tas. Une fois le pointeur intelligent initialisé, il possède le pointeur brut. Cela signifie que le pointeur intelligent est chargé de supprimer la mémoire que le pointeur brut spécifie. Le destructeur du pointeur intelligent contient l'appel à supprimer, et comme le pointeur intelligent est déclaré sur la pile, son destructeur est appelé lorsque le pointeur intelligent sort de son étendue, même si une exception est levée à un emplacement situé plus haut dans la pile.

Accédez au pointeur encapsulé en utilisant des opérateurs de pointeur familiers, `->` et `*`, que la classe de pointeur intelligent surcharge pour retourner un pointeur brut encapsulé.

L'idiome de pointeur intelligent C++ ressemble à la création d'objets dans les langages tels que C# : vous créez l'objet, puis vous laissez au système le soin de le supprimer au moment approprié. La différence vient du fait qu'aucun récupérateur de mémoire distinct ne s'exécute en arrière-plan ; la mémoire est gérée via les règles de portée C++ standard afin que l'environnement d'exécution soit plus rapide et plus efficace.

> **ⓘ Important**
>
> Veillez à toujours créer les pointeurs intelligents sur une ligne distincte de code et jamais dans une liste de paramètres, afin qu'une fuite de ressource subtile ne se produise pas suite à certaines règles d'allocation de liste de paramètres.

L’exemple suivant montre comment un `unique_ptr` type de pointeur intelligent de la bibliothèque standard C++ peut être utilisé pour encapsuler un pointeur vers un objet volumineux.

```
C++
```

```
class LargeObject
{
public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}
void SmartPointerDemo()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);

} //pLarge is deleted automatically when function block goes out of scope.
```

L'exemple illustre les étapes essentielles suivantes pour utiliser les pointeurs intelligents.

1. Déclarez le pointeur intelligent comme variable automatique (locale). (N'utilisez pas l'expression `new` ou `malloc` sur le pointeur intelligent lui-même.)

2. Dans le paramètre de type, spécifiez le type pointé du pointeur encapsulé.

3. Passez un pointeur brut à un `new`objet -ed dans le constructeur de pointeur intelligent. (Certaines fonctions utilitaires ou certains constructeurs de pointeur intelligent le font à votre place.)

4. Utilisez les opérateurs `->` et `*` surchargés pour accéder à l'objet.

5. Laissez le pointeur intelligent supprimer l'objet.

Les pointeurs intelligents sont conçus pour être aussi efficaces que possible, aussi bien en termes de mémoire que de performances. Par exemple, la seule donnée membre dans `unique_ptr` est le pointeur encapsulé. Cela signifie que `unique_ptr` a exactement la même taille que ce pointeur, soit quatre octets ou huit octets. L'accès au pointeur encapsulé à l'aide du pointeur intelligent surchargé * et -> n'est pas beaucoup plus lent que l'accès direct aux pointeurs bruts.

Les pointeurs intelligents ont leurs propres fonctions membres, qui sont accessibles à l'aide de la notation « point ». Par exemple, certains pointeurs intelligents de la bibliothèque C++ Standard ont une fonction membre de réinitialisation qui libère la propriété du pointeur. C'est utile lorsque vous souhaitez libérer la mémoire possédée

par le pointeur intelligent avant que celui-ci ne passe hors de portée, comme illustré dans l'exemple suivant.

C++

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...

}
```

Les pointeurs intelligents permettent généralement d'accéder directement à leur pointeur brut. Les pointeurs intelligents de la bibliothèque C++ Standard ont une `get` fonction membre à cet effet et `CComPtr` ont un membre de classe publique `p` . En assurant l'accès direct au pointeur sous-jacent, vous pouvez utiliser le pointeur intelligent pour gérer la mémoire dans votre propre code et continuer à passer le pointeur brut au code qui ne prend pas en charge les pointeurs intelligents.

C++

```
void SmartPointerDemo4()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass raw pointer to a legacy API
    LegacyLargeObjectFunction(pLarge.get());
}
```

Types de pointeurs intelligents

La section suivante résume les différents genres de pointeurs intelligents disponibles dans l'environnement de programmation Windows, et indique quand les utiliser.

Pointeurs intelligents de la bibliothèque C++ Standard

Utilisez ces pointeurs intelligents comme premier choix pour encapsuler les pointeurs vers les objets C++ anciens ordinaires (POCO).

- `unique_ptr`
  Autorise exactement un propriétaire du pointeur sous-jacent. À utiliser comme option par défaut pour POCO à moins que vous ne soyez certain d'avoir besoin de `shared_ptr`. Peut être déplacé vers un nouveau propriétaire, mais pas copié ou partagé. Remplace `auto_ptr`, qui est déconseillé. Comparez à `boost::scoped_ptr`. `unique_ptr` est petit et efficace; la taille est un pointeur et prend en charge les références rvalue pour une insertion et une récupération rapides à partir de collections de bibliothèques C++ Standard. Fichier d'en-tête : `<memory>`. Pour plus d'informations, consultez How to: Create and Use unique_ptr Instances and unique_ptr Class.

- `shared_ptr`
  Pointeur intelligent de références comptabilisées. À utiliser lorsque vous souhaitez affecter un pointeur brut à plusieurs propriétaires, par exemple, lorsque vous retournez la copie d'un pointeur à partir d'un conteneur, mais que vous souhaitez conserver l'original. Le pointeur brut n'est pas supprimé tant que tous les propriétaires `shared_ptr` ne sont pas hors de portée ou n'ont pas abandonné la propriété. La taille contient deux pointeurs ; un pour l'objet et un pour le bloc de contrôle partagé qui contient le nombre de références. Fichier d'en-tête : `<memory>`. Pour plus d'informations, consultez How to: Create and Use shared_ptr Instances and shared_ptr Class.

- `weak_ptr`
  Pointeur intelligent spécifique à utiliser en collaboration avec `shared_ptr`. `weak_ptr` fournit l'accès à un objet appartenant à une ou plusieurs instances `shared_ptr`, mais ne participe pas au décompte de références. À utiliser lorsque vous souhaitez observer un objet, mais n'avez pas besoin qu'il demeure actif. Obligatoire dans certains cas pour interrompre les références circulaires entre les instances `shared_ptr`. Fichier d'en-tête : `<memory>`. Pour plus d'informations, consultez How to: Create and Use weak_ptr Instances and weak_ptr Class.

Pointeurs intelligents pour les objets COM (programmation Windows classique)

Lorsque vous travaillez avec les objets COM, encapsulez les pointeurs d'interface dans un type de pointeur intelligent approprié. La bibliothèque ATL (Active Template Library) définit plusieurs pointeurs intelligents pour différents besoins. Vous pouvez également utiliser le type pointeur intelligent `_com_ptr_t`, que le compilateur utilise lorsqu'il crée des classes wrapper à partir de fichiers .tlb. C'est le meilleur choix lorsque vous ne souhaitez pas inclure les fichiers d'en-tête ATL.

CComPtr, classe
Procédez ainsi à moins que vous ne puissiez pas utiliser ATL. Effectue un décompte des références à l'aide des méthodes `AddRef` et `Release`. Pour plus d'informations, consultez Guide pratique pour créer et utiliser des instances CComPtr et CComQIPtr.

CComQIPtr, classe
Ressemble à `CComPtr`, mais fournit également une syntaxe simplifiée pour appeler `QueryInterface` sur les objets COM. Pour plus d'informations, consultez Guide pratique pour créer et utiliser des instances CComPtr et CComQIPtr.

CComHeapPtr, classe
Pointeur intelligent vers des objets qui utilisent `CoTaskMemFree` pour libérer de la mémoire.

CComGITPtr, classe
Pointeur intelligent pour les interfaces obtenues à partir de la table GIT (Global Interface Table).

classe _com_ptr_t
Ressemble à `CComQIPtr` en termes de fonctionnalités mais ne dépend pas d'en-têtes ATL.

Pointeurs intelligents ATL pour les objets POCO

En plus des pointeurs intelligents pour les objets COM, ATL définit également des pointeurs intelligents et des collections de pointeurs intelligents, pour les anciens objets C++ simples (POCO). Dans la programmation Windows classique, ces types sont des alternatives utiles aux collections bibliothèque standard C++, en particulier lorsque la portabilité du code n'est pas requise ou lorsque vous ne souhaitez pas mélanger les modèles de programmation de la bibliothèque C++ Standard et d'ATL.

CAutoPtr, classe
Pointeur intelligent qui applique une propriété unique en transférant la propriété lors de la copie. Comparable à la classe `std::auto_ptr` déconseillée.

CHeapPtr, classe

Pointeur intelligent pour les objets alloués à l'aide de la fonction malloc C.

CAutoVectorPtr, classe

Pointeur intelligent pour les tableaux alloués à l'aide de `new[]`.

CAutoPtrArray, classe

Classe qui encapsule un tableau d'éléments `CAutoPtr`.

CAutoPtrList, classe

Classe qui encapsule les méthodes de manipulation d'une liste de nœuds `CAutoPtr`.

Voir aussi

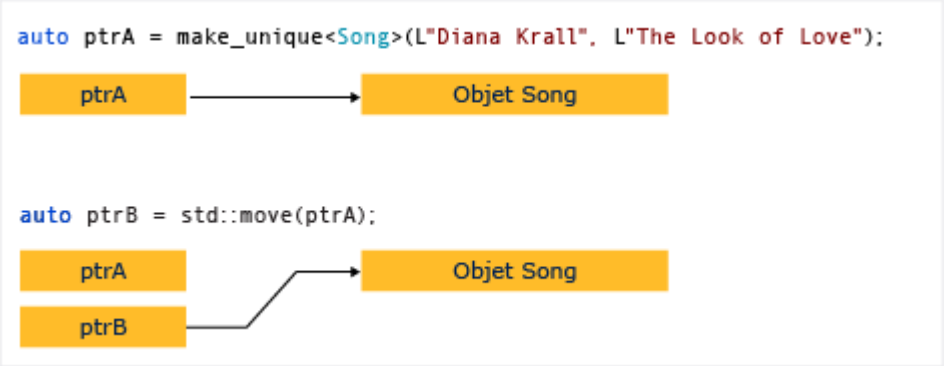
Pointeurs
Informations de référence sur le langage C++
Bibliothèque C++ standard

Procédure : Créer et utiliser des instances unique_ptr

Article • 16/06/2023

Un unique_ptr ne partage pas son pointeur. Il ne peut pas être copié vers un autre `unique_ptr`, passé par valeur à une fonction, ou utilisé dans un algorithme de bibliothèque standard C++ qui nécessite d'effectuer des copies. Un `unique_ptr` peut uniquement être déplacé. Cela signifie que la propriété de la ressource mémoire est transférée à un autre `unique_ptr` et que le `unique_ptr` d'origine ne le possède plus. Il est recommandé de restreindre un objet à un seul propriétaire, car la multiplicité des propriétaires ajoute de la complexité à la logique du programme. Par conséquent, lorsque vous avez besoin d'un pointeur intelligent pour un objet C++ simple, utilisez `unique_ptr`et lorsque vous construisez un `unique_ptr`, utilisez la fonction d'assistance make_unique .

Le diagramme suivant illustre le transfert de propriété entre deux instances `unique_ptr`.



`unique_ptr` est défini dans l'en-tête `<memory>` de la bibliothèque C++ Standard. Il est exactement aussi efficace qu'un pointeur brut et peut être utilisé dans les conteneurs de bibliothèque standard C++. L'ajout d'instances `unique_ptr` aux conteneurs de bibliothèque standard C++ est efficace, car le constructeur de déplacement du `unique_ptr` élimine la nécessité d'une opération de copie.

Exemple 1

L'exemple suivant montre comment créer des instances `unique_ptr` et les passer entre des fonctions.

C++

```
unique_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring&
title)
{
    // Implicit move operation into the variable that stores the result.
    return make_unique<Song>(artist, title);
}

void MakeSongs()
{
    // Create a new unique_ptr with a new object.
    auto song = make_unique<Song>(L"Mr. Children", L"Namonaki Uta");

    // Use the unique_ptr.
    vector<wstring> titles = { song->title };

    // Move raw pointer from one unique_ptr to another.
    unique_ptr<Song> song2 = std::move(song);

    // Obtain unique_ptr from function that returns by value.
    auto song3 = SongFactory(L"Michael Jackson", L"Beat It");
}
```

Ces exemples illustrent cette caractéristique de base de `unique_ptr` : il peut être déplacé, mais pas copié. Le « déplacement » transfère la propriété à un nouvel `unique_ptr` et réinitialise l'ancien `unique_ptr`.

Exemple 2

L'exemple suivant montre comment créer des instances `unique_ptr` et les utiliser dans un vecteur.

C++

```
void SongVector()
{
    vector<unique_ptr<Song>> songs;

    // Create a few new unique_ptr<Song> instances
    // and add them to vector using implicit move semantics.
    songs.push_back(make_unique<Song>(L"B'z", L"Juice"));
    songs.push_back(make_unique<Song>(L"Namie Amuro", L"Funky Town"));
    songs.push_back(make_unique<Song>(L"Kome Kome Club", L"Kimi ga Iru Dake
de"));
    songs.push_back(make_unique<Song>(L"Ayumi Hamasaki", L"Poker Face"));

    // Pass by const reference when possible to avoid copying.
    for (const auto& song : songs)
    {
        wcout << L"Artist: " << song->artist << L"   Title: " << song->title
```

```
<< endl;
    }
}
```

Dans la boucle de plage pour, notez que est `unique_ptr` passé par référence. Si vous essayez d'effectuer un passage par valeur ici, le compilateur génère une erreur, car le constructeur de copie `unique_ptr` est supprimé.

Exemple 3

L'exemple suivant montre comment initialiser un `unique_ptr` qui est un membre de classe.

C++

```
class MyClass
{
private:
    // MyClass owns the unique_ptr.
    unique_ptr<ClassFactory> factory;
public:

    // Initialize by using make_unique with ClassFactory default
constructor.
    MyClass() : factory (make_unique<ClassFactory>())
    {
    }

    void MakeClass()
    {
        factory->DoSomething();
    }
};
```

Exemple 4

Vous pouvez utiliser make_unique pour créer un `unique_ptr` dans un tableau, mais vous ne pouvez pas utiliser `make_unique` pour initialiser les éléments du tableau.

C++

```
// Create a unique_ptr to an array of 5 integers.
auto p = make_unique<int[]>(5);

// Initialize the array.
```

```
for (int i = 0; i < 5; ++i)
{
    p[i] = i;
    wcout << p[i] << endl;
}
```

Pour obtenir d'autres exemples, consultez make_unique.

Voir aussi

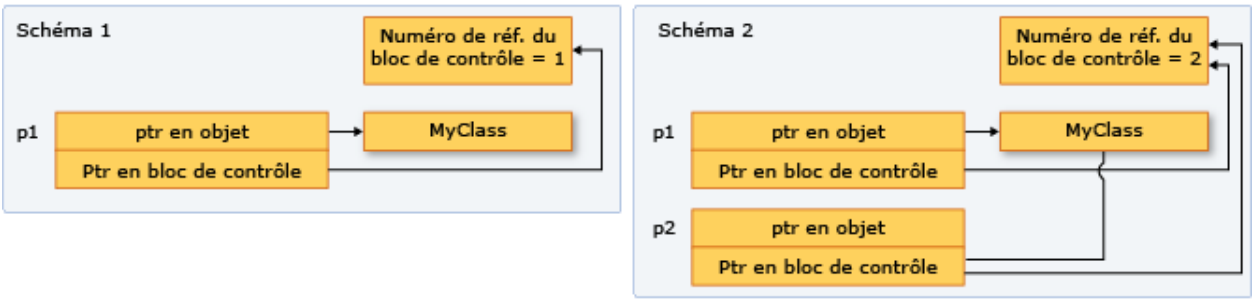
Pointeurs intelligents (C++moderne)
make_unique

Guide pratique pour créer et utiliser des instances shared_ptr

Article • 26/05/2023

Le type `shared_ptr` est un pointeur intelligent de la bibliothèque standard C++ conçu pour des scénarios dans lesquels plusieurs propriétaires peuvent devoir gérer la durée de vie de l'objet en mémoire. Après avoir initialisé `shared_ptr`, vous pouvez le copier, le passer par valeur dans des arguments de fonction et l'assigner à d'autres instances `shared_ptr`. Toutes les instances pointent vers le même objet et partagent l'accès à un "bloc de contrôle" qui incrémente et décrémente le nombre de références chaque fois qu'un nouvel `shared_ptr` est ajouté, est hors de portée ou est réinitialisé. Lorsque le nombre de références atteint zéro, le bloc de contrôle supprime la ressource mémoire et lui-même.

L'illustration suivante représente plusieurs instances `shared_ptr` qui pointent vers un emplacement de mémoire.



Exemple de configuration

Tous les exemples qui suivent supposent que vous avez inclus les en-têtes nécessaires et déclaré les types nécessaires, comme illustré ici :

C++

```
// shared_ptr-examples.cpp
// The following examples assume these declarations:
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

struct MediaAsset
{
    virtual ~MediaAsset() = default; // make it polymorphic
```

```
};

struct Song : public MediaAsset
{
    std::wstring artist;
    std::wstring title;
    Song(const std::wstring& artist_, const std::wstring& title_) :
        artist{ artist_ }, title{ title_ } {}
};

struct Photo : public MediaAsset
{
    std::wstring date;
    std::wstring location;
    std::wstring subject;
    Photo(
        const std::wstring& date_,
        const std::wstring& location_,
        const std::wstring& subject_) :
        date{ date_ }, location{ location_ }, subject{ subject_ } {}
};

using namespace std;

int main()
{
    // The examples go here, in order:
    // Example 1
    // Example 2
    // Example 3
    // Example 4
    // Example 6
}
```

Exemple 1

Quand c'est possible, utilisez la fonction make_shared pour créer un objet `shared_ptr` quand la ressource mémoire est créée pour la première fois. `make_shared` est protégé contre les exceptions. Elle utilise le même appel pour allouer de la mémoire pour le bloc de contrôle et pour la ressource, ce qui réduit la charge liées à la construction. Si vous n'utilisez pas `make_shared`, vous devez utiliser une expression **`new`** explicite pour créer l'objet avant de le passer au constructeur `shared_ptr`. L'exemple suivant indique différentes façons de déclarer et d'initialiser `shared_ptr` avec un nouvel objet.

C++

```
// Use make_shared function when possible.
auto sp1 = make_shared<Song>(L"The Beatles", L"Im Happy Just to Dance With
```

```
You");

// Ok, but slightly less efficient.
// Note: Using new expression as constructor argument
// creates no named variable for other code to access.
shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

// When initialization must be separate from declaration, e.g. class
members,
// initialize with nullptr to make your programming intent explicit.
shared_ptr<Song> sp5(nullptr);
//Equivalent to: shared_ptr<Song> sp5;
//...
sp5 = make_shared<Song>(L"Elton John", L"I'm Still Standing");
```

Exemple 2

L'exemple suivant montre comment déclarer et initialiser les instances `shared_ptr` qui prennent en charge la propriété partagée d'un objet qui a déjà été alloué par un autre `shared_ptr`. Supposons que `sp2` est un `shared_ptr` initialisé.

C++

```
//Initialize with copy constructor. Increments ref count.
auto sp3(sp2);

//Initialize via assignment. Increments ref count.
auto sp4 = sp2;

//Initialize with nullptr. sp7 is empty.
shared_ptr<Song> sp7(nullptr);

// Initialize with another shared_ptr. sp1 and sp2
// swap pointers as well as ref counts.
sp1.swap(sp2);
```

Exemple 3

`shared_ptr` est également utile dans les conteneurs de la bibliothèque standard C++ quand vous utilisez des algorithmes qui copient des éléments. Vous pouvez encapsuler des éléments dans un `shared_ptr`, puis le copier dans d'autres conteneurs à condition que la mémoire sous-jacente soit valide tant que cela est nécessaire, et pas plus longtemps. L'exemple suivant montre comment utiliser les algorithmes `remove_copy_if` sur des instances `shared_ptr` au sein d'un vecteur.

```
```

C++

```
vector<shared_ptr<Song>> v {
  make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"),
  make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"),
  make_shared<Song>(L"Thalía", L"Entre El Mar y Una Estrella")
};

vector<shared_ptr<Song>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2), [] (shared_ptr<Song>
s)
{
    return s->artist.compare(L"Bob Dylan") == 0;
});

for (const auto& s : v2)
{
    wcout << s->artist << L":" << s->title << endl;
}
```

Exemple 4

Utilisez `dynamic_pointer_cast`, `static_pointer_cast` et `const_pointer_cast`, pour caster un `shared_ptr`. Ces fonctions sont semblables aux opérateurs **`dynamic_cast`**, **`static_cast`** et **`const_cast`**. L'exemple suivant indique comment tester le type dérivé de chaque élément dans un vecteur de `shared_ptr` de classes de base, puis copier les éléments et afficher les informations les concernant.

C++

```
vector<shared_ptr<MediaAsset>> assets {
  make_shared<Song>(L"Himesh Reshammiya", L"Tera Surroor"),
  make_shared<Song>(L"Penaz Masani", L"Tu Dil De De"),
  make_shared<Photo>(L"2011-04-06", L"Redmond, WA", L"Soccer field at
Microsoft.")
};

vector<shared_ptr<MediaAsset>> photos;

copy_if(assets.begin(), assets.end(), back_inserter(photos), []
(shared_ptr<MediaAsset> p) -> bool
{
    // Use dynamic_pointer_cast to test whether
    // element is a shared_ptr<Photo>.
    shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(p);
    return temp.get() != nullptr;
});

for (const auto&  p : photos)
```

```
{
    // We know that the photos vector contains only
    // shared_ptr<Photo> objects, so use static_cast.
    wcout << "Photo location: " << (static_pointer_cast<Photo>(p))->location
<< endl;
}
```

Exemple 5

Vous pouvez passer `shared_ptr` à une autre fonction des façons suivantes :

- Passez `shared_ptr` par valeur. Il appelle le constructeur de copie, incrémente le nombre de références et fait de l'appelé un propriétaire. Cette opération induit une petite surcharge, qui peut être significative selon le nombre d'objets `shared_ptr` que vous passez. Utilisez cette option quand le contrat de code implicite ou explicite entre l'appelant et l'appelé nécessite que l'appelé soit un propriétaire.

- Passez `shared_ptr` par référence ou référence const. Dans ce cas, le nombre de références n'est pas incrémenté, et l'appelé peut accéder au pointeur tant que l'appelant n'est pas en dehors de l'étendue. Sinon, l'appelé peut décider de créer un `shared_ptr` basé sur la référence, et de devenir un propriétaire partagé. Utilisez cette option lorsque l'appelant n'a pas connaissance de l'appelé, ou lorsque vous devez passer un `shared_ptr` et que vous voulez empêcher l'opération de copie pour des raisons de performance.

- Passez le pointeur ou une référence sous-jacent(e) à l'objet sous-jacent. Ceci permet à l'appelé d'utiliser l'objet, mais ne lui permet pas de partager la propriété ou d'étendre la durée de vie. Si l'appelé crée un `shared_ptr` à partir du pointeur brut, le nouveau `shared_ptr` est indépendant de l'original et ne contrôle pas la ressource sous-jacente. Utilisez cette option lorsque le contrat entre l'appelant et l'appelé spécifie clairement que l'appelant conserve la propriété de la durée de vie de `shared_ptr`.

- Quand vous décidez comment passer un `shared_ptr`, déterminez si l'appelé doit partager la propriété de la ressource sous-jacente. Un "propriétaire" est un objet ou une fonction qui peut garder la ressource sous-jacente active tant qu'il en a besoin. Si l'appelant doit garantir que l'appelé peut prolonger la vie du pointeur au delà de la durée de vie de la fonction, utilisez le première option. Si vous ne vous préoccupez pas de si l'appelé étend la durée de vie ou non, alors effectuez une transmission par référence et laissez l'appelé la copier ou non.

- Si vous devez donner un accès à la fonction helper au pointeur sous-jacent, et que vous savez que cette fonction helper utilise simplement le pointeur et retourne avant la fonction appelante, cette fonction ne doit pas partager la propriété du pointeur sous-jacent. Elle doit simplement accéder au pointeur dans la durée de vie du `shared_ptr` de l'appelant. Dans ce cas, cela ne pose pas de problème de passer le `shared_ptr` par référence, ou de passer le pointeur brut ou une référence à l'objet sous-jacent. Passer de cette façon fournit un léger avantage en termes de performances et peut également vous aider à exprimer votre intention de programmation.

- Parfois, par exemple dans un `std::vector<shared_ptr<T>>`, vous devrez peut-être passer chaque `shared_ptr` à une expression Lambda ou un objet fonction nommé. Si l'expression lambda ou la fonction ne stocke pas le pointeur, passez le `shared_ptr` par référence pour éviter d'appeler le constructeur de copie pour chaque élément.

C++

```
void use_shared_ptr_by_value(shared_ptr<int> sp);

void use_shared_ptr_by_reference(shared_ptr<int>& sp);
void use_shared_ptr_by_const_reference(const shared_ptr<int>& sp);

void use_raw_pointer(int* p);
void use_reference(int& r);

void test() {
    auto sp = make_shared<int>(5);

    // Pass the shared_ptr by value.
    // This invokes the copy constructor, increments the reference count,
and makes the callee an owner.
    use_shared_ptr_by_value(sp);

    // Pass the shared_ptr by reference or const reference.
    // In this case, the reference count isn't incremented.
    use_shared_ptr_by_reference(sp);
    use_shared_ptr_by_const_reference(sp);

    // Pass the underlying pointer or a reference to the underlying object.
    use_raw_pointer(sp.get());
    use_reference(*sp);

    // Pass the shared_ptr by value.
    // This invokes the move constructor, which doesn't increment the
reference count
    // but in fact transfers ownership to the callee.
    use_shared_ptr_by_value(move(sp));
}
```

Exemple 6

L'exemple suivant montre comment `shared_ptr` surcharge différents opérateurs de comparaison pour activer des comparaisons de pointeur dans la mémoire qui est possédée par les instances `shared_ptr`.

C++

```
// Initialize two separate raw pointers.
// Note that they contain the same values.
auto song1 = new Song(L"Village People", L"YMCA");
auto song2 = new Song(L"Village People", L"YMCA");

// Create two unrelated shared_ptrs.
shared_ptr<Song> p1(song1);
shared_ptr<Song> p2(song2);

// Unrelated shared_ptrs are never equal.
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;
wcout << "p1 == p2 = " << std::boolalpha <<(p1 == p2) << endl;

// Related shared_ptr instances are always equal.
shared_ptr<Song> p3(p2);
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;
```

Voir aussi

Pointeurs intelligents (C++moderne)

Procédure : Créer et utiliser des instances weak_ptr

Article • 03/04/2023

Parfois, un objet doit stocker un moyen d'accéder à l'objet sous-jacent d'un shared_ptr sans entraîner l'incrémentation du nombre de références. En règle générale, cette situation se produit lorsque vous avez des références cycliques entre `shared_ptr` des instances.

La meilleure conception consiste à éviter la propriété partagée des pointeurs chaque fois que vous le pouvez. Toutefois, si vous devez avoir la propriété partagée des `shared_ptr` instances, évitez les références cycliques entre elles. Lorsque les références cycliques sont inévitables, voire préférables pour une raison quelconque, utilisez weak_ptr pour donner à un ou plusieurs propriétaires une référence faible à un autre `shared_ptr`. À l'aide d'un `weak_ptr`, vous pouvez créer un qui joint un `shared_ptr` à un ensemble existant d'instances associées, mais uniquement si la ressource mémoire sous-jacente est toujours valide. Un `weak_ptr` lui-même ne participe pas au comptage des références et, par conséquent, il ne peut pas empêcher le nombre de références de passer à zéro. Toutefois, vous pouvez utiliser un `weak_ptr` pour essayer d'obtenir une nouvelle copie du `shared_ptr` avec lequel il a été initialisé. Si la mémoire a déjà été supprimée, l'opérateur `weak_ptr`bool de retourne `false`. Si la mémoire est toujours valide, le nouveau pointeur partagé incrémente le nombre de références et garantit que la mémoire sera valide tant que la variable reste dans l'étendue `shared_ptr` .

Exemple

L'exemple de code suivant montre un cas où `weak_ptr` est utilisé pour garantir la suppression correcte des objets qui ont des dépendances circulaires. Lorsque vous examinez l'exemple, supposons qu'il a été créé uniquement après l'examen d'autres solutions. Les `Controller` objets représentent un aspect d'un processus de machine, et ils fonctionnent indépendamment. Chaque contrôleur doit être en mesure d'interroger les status des autres contrôleurs à tout moment, et chacun d'eux contient un privé `vector<weak_ptr<Controller>>` à cet effet. Chaque vecteur contient une référence circulaire et, par conséquent, `weak_ptr` les instances sont utilisées à la place de `shared_ptr`.

```
C++
```

```cpp
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Controller
{
public:
   int Num;
   wstring Status;
   vector<weak_ptr<Controller>> others;
   explicit Controller(int i) : Num(i), Status(L"On")
   {
      wcout << L"Creating Controller" << Num << endl;
   }

   ~Controller()
   {
      wcout << L"Destroying Controller" << Num << endl;
   }

   // Demonstrates how to test whether the
   // pointed-to memory still exists or not.
   void CheckStatuses() const
   {
      for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp) {
         auto p = wp.lock();
         if (p)
         {
            wcout << L"Status of " << p->Num << " = " << p->Status << endl;
         }
         else
         {
            wcout << L"Null object" << endl;
         }
      });
   }
};

void RunTest()
{
   vector<shared_ptr<Controller>> v{
       make_shared<Controller>(0),
       make_shared<Controller>(1),
       make_shared<Controller>(2),
       make_shared<Controller>(3),
       make_shared<Controller>(4),
   };

   // Each controller depends on all others not being deleted.
   // Give each controller a pointer to all the others.
```

```
   for (int i = 0; i < v.size(); ++i)
   {
      for_each(v.begin(), v.end(), [&v, i](shared_ptr<Controller> p) {
         if (p->Num != i)
         {
            v[i]->others.push_back(weak_ptr<Controller>(p));
            wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
         }
      });
   }

   for_each(v.begin(), v.end(), [](shared_ptr<Controller> &p) {
      wcout << L"use_count = " << p.use_count() << endl;
      p->CheckStatuses();
   });
}

int main()
{
   RunTest();
   wcout << L"Press any key" << endl;
   char ch;
   cin.getline(&ch, 1);
}
```

Output

```
Creating Controller0
Creating Controller1
Creating Controller2
Creating Controller3
Creating Controller4
push_back to v[0]: 1
push_back to v[0]: 2
push_back to v[0]: 3
push_back to v[0]: 4
push_back to v[1]: 0
push_back to v[1]: 2
push_back to v[1]: 3
push_back to v[1]: 4
push_back to v[2]: 0
push_back to v[2]: 1
push_back to v[2]: 3
push_back to v[2]: 4
push_back to v[3]: 0
push_back to v[3]: 1
push_back to v[3]: 2
push_back to v[3]: 4
push_back to v[4]: 0
push_back to v[4]: 1
push_back to v[4]: 2
push_back to v[4]: 3
use_count = 1
```

```
Status of 1 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 3 = On
Destroying Controller0
Destroying Controller1
Destroying Controller2
Destroying Controller3
Destroying Controller4
Press any key
```

En tant qu'expérience, modifiez le vecteur `others` pour qu'il soit un `vector<shared_ptr<Controller>>`, puis dans la sortie, notez qu'aucun destructeur n'est appelé lors `RunTest` des retours.

Voir aussi

Pointeurs intelligents (C++moderne)

Procédure : Créer et utiliser des instances CComPtr et CComQIPtr

Article • 03/04/2023

En programmation Windows classique, les bibliothèques sont souvent implémentées en tant qu'objets COM (Component Object Model), ou plus précisément, en tant que serveurs COM. De nombreux composants du système d'exploitation Windows sont implémentés en tant que serveurs COM, et de nombreux collaborateurs fournissent des bibliothèques sous cette forme. Pour plus d'informations sur les bases du modèle COM, consultez Component Object Model (COM).

Lorsque vous instanciez un objet COM, enregistrez le pointeur d'interface dans un pointeur intelligent COM, qui effectue un décompte de références en utilisant des appels à `AddRef` et `Release` dans le destructeur. Si vous utilisez la bibliothèque ATL (Active Template Library) ou la bibliothèque MFC (Microsoft Foundation Class), utilisez le pointeur intelligent `CComPtr` . Si vous n'utilisez pas la bibliothèque ATL ou MFC, utilisez `_com_ptr_t`. Étant donné qu'aucun COM n'équivaut à `std::unique_ptr`, utilisez les pointeurs intelligents à la fois pour les scénarios à un seul propriétaire et ceux à plusieurs propriétaires. `CComPtr` et `ComQIPtr` prennent tous deux en charge les opérations de déplacement qui ont des références rvalue.

Exemple : CComPtr

L'exemple suivant montre comment utiliser `CComPtr` pour instancier un objet COM et obtenir des pointeurs sur ses interfaces. Notez que la fonction membre `CComPtr::CoCreateInstance` est utilisée pour créer l'objet COM, au lieu de la fonction Win32 qui porte le même nom.

C++

```
void CComPtrDemo()
{

    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr<IGraphBuilder> pGraph;

    // Use its member function CoCreateInstance to
    // create the COM object and obtain the IGraphBuilder pointer.
    hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
    if(FAILED(hr)){ /*... handle hr error*/ }
```

```
    // Use the overloaded -> operator to call the interface methods.
    hr = pGraph->RenderFile(L"C:\\Users\\Public\\Music\\Sample Music\\Sleep
Away.mp3", NULL);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Declare a second smart pointer and use it to
    // obtain another interface from the object.
    CComPtr<IMediaControl> pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr<IMediaEvent> pEvent;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pEvent));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the second interface.
    hr = pControl->Run();
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the third interface.
    long evCode = 0;
    pEvent->WaitForCompletion(INFINITE, &evCode);

    CoUninitialize();

    // Let the smart pointers do all reference counting.
}
```

`CComPtr` et ses parents font partie de l'ATL et sont définis dans <atlcomcli.h>. `_com_ptr_t` est déclaré dans <comip.h>. Le compilateur crée des spécialisations de `_com_ptr_t` lorsqu'il génère des classes wrapper pour les bibliothèques de types.

Exemple : CComQIPt

La bibliothèque ATL fournit également `CComQIPtr`, qui a une syntaxe plus simple pour interroger un objet COM en vue de récupérer une interface supplémentaire. Toutefois, nous vous recommandons `CComPtr` car il fait tout ce que `CComQIPtr` peut faire et est sémantiquement plus cohérent avec les pointeurs d'interface COM bruts. Si vous utilisez `CComPtr` pour effectuer une requête à une interface, le nouveau pointeur d'interface est placé dans un paramètre de sortie. Si l'appel échoue, un HRESULT est retourné, qui est le modèle COM standard. Avec `CComQIPtr`, la valeur retournée est le pointeur lui-même, et si l'appel échoue, la valeur de retour HRESULT interne est inaccessible. Les deux lignes suivantes indiquent comment les mécanismes de gestion des erreurs dans `CComPtr` et `CComQIPtr` diffèrent.

C++

```
// CComPtr with error handling:
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// CComQIPtr with error handling
CComQIPtr<IMediaEvent> pEvent = pControl;
if(!pEvent){ /*... handle NULL pointer error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }
```

Exemple : IDispatch

`CComPtr` fournit une spécialisation pour IDispatch qui lui permet de stocker les pointeurs vers les composants d'automation COM et d'appeler des méthodes sur l'interface à l'aide de la liaison tardive. `CComDispatchDriver` est un typedef pour `CComQIPtr<IDispatch, &IIDIDispatch>`, qui peut être converti implicitement en `CComPtr<IDispatch>`. Par conséquent, lorsque l'un de ces trois noms apparaît dans le code, il correspond à `CComPtr<IDispatch>`. L'exemple suivant montre comment obtenir un pointeur vers le modèle objet Microsoft Word en utilisant `CComPtr<IDispatch>`.

C++

```
void COMAutomationSmartPointerDemo()
{

    CComPtr<IDispatch> pWord;
    CComQIPtr<IDispatch, &IID_IDispatch> pqi = pWord;
    CComDispatchDriver pDriver = pqi;

    HRESULT hr;
    _variant_t pOutVal;

    CoInitialize(NULL);
    hr = pWord.CoCreateInstance(L"Word.Application", NULL,
CLSCTX_LOCAL_SERVER);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Make Word visible.
    hr = pWord.PutPropertyByName(_bstr_t("Visible"),  &_variant_t(1));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Get the Documents collection and store it in new CComPtr
    hr = pWord.GetPropertyByName(_bstr_t("Documents"), &pOutVal);
```

```
    if(FAILED(hr)){ /*... handle hr error*/ }

    CComPtr<IDispatch> pDocuments = pOutVal.pdispVal;

    // Use Documents to open a document
    hr = pDocuments.Invoke1 (_bstr_t("Open"),
&_variant_t("c:\\users\\public\\documents\\sometext.txt"),&pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CoUninitialize();
}
```

Voir aussi

Pointeurs intelligents (C++moderne)

Gestion des exceptions dans MSVC

Article • 16/06/2023

Une exception est une condition d'erreur, éventuellement en dehors du contrôle du programme, qui empêche le programme de se poursuivre selon son chemin d'exécution normal. Certaines opérations, notamment la création d'objets, l'entrée/sortie de fichier et les appels de fonction effectués à partir d'autres modules, sont toutes des sources potentielles d'exceptions, même lorsque votre programme s'exécute correctement. Un code robuste vous permet d'anticiper les exceptions et de les gérer. Pour détecter les erreurs logiques, utilisez des assertions plutôt que des exceptions (consultez Utilisation d'assertions).

Types d'exceptions

Le compilateur Microsoft C++ (MSVC) prend en charge trois types de gestion des exceptions :

- Gestion des exceptions C++

  Pour la plupart des programmes C++, vous devez utiliser la gestion des exceptions C++. Il est sécurisé de type et garantit que les destructeurs d'objets sont appelés pendant le déroulement de la pile.

- Gestion structurée des exceptions

  Windows fournit son propre mécanisme d'exception, appelé gestion structurée des exceptions (SEH). Il n'est pas recommandé pour la programmation C++ ou MFC. Utilisez SEH uniquement dans les programmes C non MFC.

- Exceptions MFC

  Depuis la version 3.0, MFC utilise des exceptions C++. Il prend toujours en charge ses anciennes macros de gestion des exceptions, qui sont similaires aux exceptions C++ dans la forme. Pour obtenir des conseils sur le mélange de macros MFC et d'exceptions C++, consultez Exceptions : utilisation de macros MFC et d'exceptions C++.

Utilisez une option de compilateur /EH pour spécifier le modèle de gestion des exceptions à utiliser dans un projet C++. La gestion des exceptions C++ standard (**/EHsc**) est la valeur par défaut dans les nouveaux projets C++ dans Visual Studio.

Nous vous déconseillons de mélanger les mécanismes de gestion des exceptions. Par exemple, n'utilisez pas d'exceptions C++ avec une gestion structurée des exceptions. L'utilisation de la gestion des exceptions C++ rend exclusivement votre code plus portable et vous permet de gérer les exceptions de n'importe quel type. Pour plus d'informations sur les inconvénients de la gestion structurée des exceptions, consultez Gestion structurée des exceptions.

Contenu de cette section

- Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs
- Comment concevoir la sécurité des exceptions
- Comment établir une interface entre le code exceptionnel et le code non exceptionnel
- Instructions try, catch et throw
- Comment les blocs catch sont évalués
- Exceptions et déroulement de la pile
- Spécifications des exceptions
- noexcept
- Exceptions C++ non prises en charge
- Mélange d'exceptions C (structurées) et C++
- Gestion structurée des exceptions (C/C++)

Voir aussi

Informations de référence
sur le langage C++ Gestion des
exceptions x64Gestion des exceptions (C++/CLI et C++/CX)

Meilleures pratiques C++ modernes pour les exceptions et la gestion des erreurs

Article • 16/06/2023

Dans le C++moderne, dans la plupart des scénarios, la meilleure façon de signaler et de gérer les erreurs logiques et les erreurs d'exécution consiste à utiliser des exceptions. Cela est particulièrement vrai lorsque la pile peut contenir plusieurs appels de fonction entre la fonction qui détecte l'erreur et la fonction qui a le contexte pour gérer l'erreur. Les exceptions fournissent un moyen formel et bien défini pour le code qui détecte les erreurs de transmettre les informations dans la pile des appels.

Utiliser des exceptions pour du code exceptionnel

Les erreurs de programme sont souvent divisées en deux catégories : les erreurs logiques provoquées par des erreurs de programmation, par exemple une erreur « index hors limites ». Et, les erreurs d'exécution qui sont indépendantes du contrôle du programmeur, par exemple, une erreur « service réseau indisponible ». Dans la programmation de style C et dans COM, le rapport d'erreurs est géré en retournant une valeur qui représente un code d'erreur ou un code status pour une fonction particulière, ou en définissant une variable globale que l'appelant peut éventuellement récupérer après chaque appel de fonction pour voir si des erreurs ont été signalées. Par exemple, la programmation COM utilise la valeur de retour HRESULT pour communiquer les erreurs à l'appelant. Et l'API Win32 a la `GetLastError` fonction permettant de récupérer la dernière erreur signalée par la pile des appels. Dans les deux cas, il appartient à l'appelant de reconnaître le code et d'y répondre de manière appropriée. Si l'appelant ne gère pas explicitement le code d'erreur, le programme peut se bloquer sans avertissement. Ou bien, il peut continuer à s'exécuter à l'aide de données incorrectes et produire des résultats incorrects.

Les exceptions sont préférées en C++ moderne pour les raisons suivantes :

- Une exception force le code appelant à reconnaître une condition d'erreur et à la gérer. Les exceptions non gérées arrêtent l'exécution du programme.
- Une exception saute au point de la pile des appels qui peut gérer l'erreur. Les fonctions intermédiaires peuvent laisser l'exception se propager. Ils n'ont pas

besoin de coordonner avec d'autres couches.

- Le mécanisme de déroulement de la pile d'exceptions détruit tous les objets dans l'étendue après la levée d'une exception, selon des règles bien définies.
- Une exception permet une séparation propre entre le code qui détecte l'erreur et le code qui gère l'erreur.

L'exemple simplifié suivant montre la syntaxe nécessaire pour lever et intercepter des exceptions en C++.

C++

```
#include <stdexcept>
#include <limits>
#include <iostream>

using namespace std;

void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
        throw invalid_argument("MyFunc argument too large.");
    //...
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}
```

Les exceptions en C++ ressemblent à celles des langages tels que C# et Java. Dans le `try` bloc, si une exception est *levée* , elle est *interceptée* par le premier bloc associé `catch` dont le type correspond à celui de l'exception. En d'autres termes, l'exécution passe de l'instruction `throw` à l'instruction `catch` . Si aucun bloc catch utilisable n'est trouvé, `std::terminate` est appelé et le programme se ferme. En C++, n'importe quel type peut être levée ; Toutefois, nous vous recommandons de lever un type qui dérive directement ou indirectement de `std::exception`. Dans l'exemple précédent, le type

d'exception, invalid_argument, est défini dans la bibliothèque standard dans le fichier d'en-tête <stdexcept> . C++ ne fournit pas ou ne nécessite pas de `finally` bloc pour s'assurer que toutes les ressources sont libérées si une exception est levée. L'idiome d'acquisition de ressources est initialisation (RAII), qui utilise des pointeurs intelligents, fournit les fonctionnalités requises pour le nettoyage des ressources. Pour plus d'informations, consultez Guide pratique pour la conception de la sécurité des exceptions. Pour plus d'informations sur le mécanisme de déroulement de pile C++, consultez Exceptions et déroulement de la pile.

Instructions de base

La gestion robuste des erreurs est difficile dans n'importe quel langage de programmation. Bien que les exceptions fournissent plusieurs fonctionnalités qui prennent en charge une bonne gestion des erreurs, elles ne peuvent pas effectuer tout le travail à votre place. Pour tirer parti des avantages du mécanisme d'exception, gardez les exceptions à l'esprit lorsque vous concevez votre code.

- Utilisez des assertions pour case activée d'erreurs qui ne devraient jamais se produire. Utilisez des exceptions pour case activée en cas d'erreurs qui peuvent se produire, par exemple des erreurs de validation d'entrée sur des paramètres de fonctions publiques. Pour plus d'informations, consultez la section Exceptions et assertions .

- Utilisez des exceptions lorsque le code qui gère l'erreur est séparé du code qui détecte l'erreur par un ou plusieurs appels de fonction intermédiaires. Déterminez s'il faut utiliser des codes d'erreur à la place dans les boucles critiques en matière de performances, lorsque le code qui gère l'erreur est étroitement couplé au code qui la détecte.

- Pour chaque fonction qui peut lever ou propager une exception, fournissez l'une des trois garanties d'exception : la garantie forte, la garantie de base ou la garantie nothrow (noexcept). Pour plus d'informations, consultez Guide pratique pour la conception de la sécurité des exceptions.

- Lèvez des exceptions par valeur, interceptez-les par référence. N'attrapez pas ce que vous ne pouvez pas gérer.

- N'utilisez pas de spécifications d'exception, qui sont dépréciées en C++11. Pour plus d'informations, consultez la section Spécifications et noexceptspécifications des exceptions.

- Utilisez des types d'exception de bibliothèque standard lorsqu'ils s'appliquent. Dérivez des types d'exceptions personnalisés de la exception hiérarchie de classes .
- N'autorisez pas les exceptions à s'échapper des destructeurs ou des fonctions de désallocation de la mémoire.

Exceptions et performances

Le mécanisme d'exception a un coût de performances minimal si aucune exception n'est levée. Si une exception est levée, le coût du parcours et du déroulement de la pile est à peu près comparable au coût d'un appel de fonction. Des structures de données supplémentaires sont nécessaires pour suivre la pile des appels après l'entrée d'un `try` bloc, et des instructions supplémentaires sont nécessaires pour dérouler la pile si une exception est levée. Toutefois, dans la plupart des scénarios, le coût en matière de performances et d'empreinte mémoire n'est pas significatif. L'effet négatif des exceptions sur les performances est susceptible d'être significatif uniquement sur les systèmes à mémoire limitée. Ou, dans les boucles critiques pour les performances, où une erreur est susceptible de se produire régulièrement et où il existe un couplage étroit entre le code pour la gérer et le code qui la signale. Dans tous les cas, il est impossible de connaître le coût réel des exceptions sans profilage et mesure. Même dans les rares cas où le coût est important, vous pouvez le mettre en balance par rapport à l'exactitude accrue, à la facilité de maintenance et à d'autres avantages fournis par une stratégie d'exception bien conçue.

Exceptions et assertions

Les exceptions et les assertions sont deux mécanismes distincts permettant de détecter les erreurs d'exécution dans un programme. Utilisez `assert` des instructions pour tester les conditions pendant le développement qui ne doivent jamais être vraies si tout votre code est correct. Il est inutile de gérer une telle erreur à l'aide d'une exception, car l'erreur indique que quelque chose dans le code doit être corrigé. Il ne représente pas une condition à partir de laquelle le programme doit récupérer au moment de l'exécution. Un `assert` arrête l'exécution au niveau de l'instruction afin que vous puissiez inspecter l'état du programme dans le débogueur. Une exception continue l'exécution à partir du premier gestionnaire catch approprié. Utilisez des exceptions pour case activée conditions d'erreur qui peuvent se produire au moment de l'exécution, même si votre code est correct, par exemple, « fichier introuvable » ou « mémoire insuffisante ». Les exceptions peuvent gérer ces conditions, même si la récupération génère simplement un message dans un journal et met fin au programme. Toujours case activée arguments aux fonctions publiques à l'aide d'exceptions. Même si

votre fonction est sans erreur, vous n'avez peut-être pas un contrôle total sur les arguments qu'un utilisateur peut lui passer.

Exceptions C++ par rapport aux exceptions Windows SEH

Les programmes C et C++ peuvent utiliser le mécanisme de gestion structurée des exceptions (SEH) dans le système d'exploitation Windows. Les concepts dans SEH ressemblent à ceux des exceptions C++, sauf que SEH utilise les `__try` constructions , `__except` et `__finally` au lieu de `try` et `catch`. Dans le compilateur Microsoft C++ (MSVC), les exceptions C++ sont implémentées pour SEH. Toutefois, lorsque vous écrivez du code C++, utilisez la syntaxe d'exception C++.

Pour plus d'informations sur SEH, consultez Gestion structurée des exceptions (C/C++).

Spécifications d'exception et `noexcept`

Les spécifications d'exception ont été introduites en C++ pour spécifier les exceptions qu'une fonction peut lever. Toutefois, les spécifications d'exceptions se sont avérées problématiques dans la pratique et sont dépréciées dans l'ébauche de norme C++11. Nous vous recommandons de ne pas utiliser `throw` de spécifications d'exception à l'exception de `throw()`, ce qui indique que la fonction n'autorise aucune exception à échapper. Si vous devez utiliser les spécifications d'exception du formulaire `throw( type-name )` déprécié, la prise en charge de MSVC est limitée. Pour plus d'informations, consultez Spécifications d'exception (throw). Le `noexcept` spécificateur est introduit dans C++11 comme alternative préférée à `throw()`.

Voir aussi

Guide pratique pour faire l'interface entre du code exceptionnel et du code non exceptionnel
Informations de référence sur le langage C++
Bibliothèque C++ standard

Procédure : concevoir pour la sécurité des exceptions

Article • 16/06/2023

Un des avantages du mécanisme d'exception est que l'exécution, associée aux données sur l'exception, passe directement de l'instruction qui lève l'exception à la première instruction catch qui la gère. Le gestionnaire peut concerner n'importe quels niveaux de la pile des appels. Les fonctions qui sont appelées entre l'instruction try et l'instruction throw n'ont pas besoin de connaître quoi que ce soit concernant l'exception levée. Toutefois, elles doivent être conçues afin qu'elles puissent être "inopinément" mises hors de portée lorsqu'une exception peut se propager en remontant. Il faut donc veiller à ne pas laisser des objets, de la mémoire perdue, ou des structures de données partiellement créés qui seraient inutilisables.

Techniques de base

Une stratégie fiable de gestion des exceptions requiert une pensée attentionnée et doit faire partie du processus de création. En général, la plupart des exceptions sont détectées et levées dans les couches inférieures d'un module de logiciel, mais généralement ces couches ne disposent pas d'un contexte suffisant pour gérer l'erreur ou pour transmettre un message aux utilisateurs finaux. Dans les couches intermédiaires, les fonctions peuvent intercepter et lever à nouveau une exception lorsqu'elles sont chargées d'inspecter l'objet exception, ou elles peuvent avoir des informations supplémentaires utiles à fournir au niveau supérieur qui interceptera finalement l'exception. Une fonction doit intercepter et "avaler" une exception uniquement si elle peut être récupérée entièrement. Dans de nombreux cas, le comportement correct dans les couches intermédiaires est de laisser une exception se propager vers le haut de la pile des appels. Même dans la couche la plus haute, il peut s'avérer nécessaire de laisser une exception non gérée terminer un programme si l'exception laisse le programme dans un état dans lequel son exactitude ne peut pas être garantie.

Peu importe comment une fonction gère une exception, afin de garantir qu'elle soit protégée contre les exceptions, elle doit être conçue selon les principes de base suivants.

Conserver les classes de ressources simples

Lorsque vous encapsulez la gestion manuelle des ressources dans des classes, utilisez une classe qui ne fait rien d’autre que gérer une seule ressource. En gardant la classe

simple, vous réduisez le risque d'introduire des fuites de ressources. Utilisez des pointeurs intelligents lorsque cela est possible, comme illustré dans l'exemple suivant. Cet exemple est intentionnellement artificiel et simpliste afin de mettre en évidence les différences lorsque `shared_ptr` est utilisé.

C++

```
// old-style new/delete version
class NDResourceClass {
private:
    int*   m_p;
    float* m_q;
public:
    NDResourceClass() : m_p(0), m_q(0) {
        m_p = new int;
        m_q = new float;
    }

    ~NDResourceClass() {
        delete m_p;
        delete m_q;
    }
    // Potential leak! When a constructor emits an exception,
    // the destructor will not be invoked.
};

// shared_ptr version
#include <memory>

using namespace std;

class SPResourceClass {
private:
    shared_ptr<int> m_p;
    shared_ptr<float> m_q;
public:
    SPResourceClass() : m_p(new int), m_q(new float) { }
    // Implicitly defined dtor is OK for these members,
    // shared_ptr will clean up and avoid leaks regardless.
};

// A more powerful case for shared_ptr

class Shape {
    // ...
};

class Circle : public Shape {
    // ...
};

class Triangle : public Shape {
    // ...
```

```
};

class SPShapeResourceClass {
private:
    shared_ptr<Shape> m_p;
    shared_ptr<Shape> m_q;
public:
    SPShapeResourceClass() : m_p(new Circle), m_q(new Triangle) { }
};
```

Utiliser l'idiome RAII pour gérer les ressources

Pour garantir la sécurité contre les exceptions, une fonction doit s'assurer que les objets qu'elle a alloués à l'aide `malloc` de ou `new` sont détruits, et que toutes les ressources telles que les handles de fichier sont fermées ou libérées même si une exception est levée. L'idiome RAII (*Resource Acquisition Is Initialization*) lie la gestion de ces ressources à la durée de vie des variables automatiques. Lorsqu'une fonction est hors de portée, soit en retournant normalement ou alors suite à une exception, les destructeurs pour toutes les variables automatiques entièrement construites sont appelés. Un objet wrapper RAII tel qu'un pointeur intelligent appelle la fonction appropriée, de suppression ou de fermeture, dans son destructeur. Dans un code protégée contre les exceptions, il est extrêmement important de passer la propriété de chaque ressource immédiatement à un certain type d'objet RAII. Notez que les `vector`classes , `string`, `make_shared`, `fstream`et similaires gèrent l'acquisition de la ressource pour vous. Toutefois, `unique_ptr` et les constructions traditionnelles `shared_ptr` sont spéciales, car l'acquisition de ressources est effectuée par l'utilisateur au lieu de l'objet ; par conséquent, elles comptent comme *une destruction de ressource* , mais sont douteuses en tant que RAII.

Les trois garanties d'exception

En règle générale, la sécurité des exceptions est abordée en fonction des trois garanties d'exception qu'une fonction peut fournir : la *garantie sans défaillance*, la *garantie forte* et la *garantie de base*.

Garantie sans échec

La garantie sans échec est la garantie la plus puissante qu'une fonction puisse fournir. Elle indique que la fonction ne lèvera pas d'exception ou n'autorisera aucune propagation. Toutefois, vous ne pouvez pas de manière fiable fournir une telle garantie sauf si (a) vous savez que toutes les fonctions que cette fonction appelle sont également

sans échec, ou (b) vous savez que toutes les exceptions levées seront interceptées avant qu'elles n'atteignent cette fonction, ou (c) vous savez comment intercepter et gérer correctement toutes les exceptions pouvant atteindre cette fonction.

La garantie forte et la garantie de base reposent sur l'hypothèse que les destructeurs sont sans échec. Tous les conteneurs et types de la bibliothèque standard garantissent que leurs destructeurs ne lèvent pas. Il existe également une exigence inverse : La bibliothèque standard requiert que les types définis par l'utilisateur qui lui sont fournis, par exemple, les arguments de modèle, aient des destructeurs non lanceurs.

Garantie forte

La garantie forte déclare que si une fonction est hors de portée suite à une exception, il n'y aura pas de fuite de mémoire et l'état du programme ne sera pas modifié. Une fonction qui fournit une garantie forte est essentiellement une transaction qui comporte une sémantique de validation ou de restauration : soit il réussit complètement, soit il n'a aucun effet.

Garantie de base

La garantie de base est la plus faible des trois. Toutefois, elle peut être le meilleur choix lorsqu'une garantie forte est trop coûteuse en termes de consommation de mémoire ou de performances. La garantie de base indique que si une exception se produit, aucune fuite de mémoire n'a lieu et l'objet est toujours dans un état utilisable bien que les données peuvent avoir été modifiées.

Classes sans risque d'exception

Une classe peut garantir sa propre protection contre les exception, même lorsqu'elle est consommée par des fonctions potentiellement dangereuses, en évitant d'elle-même d'être partiellement construite ou partiellement détruite. Si un constructeur de classe s'arrête avant la fin, l'objet n'est pas créé et son destructeur ne sera jamais appelé. Bien que les variables automatiques qui sont initialisées avant l'exception auront leurs destructeurs appelés, la mémoire ou les ressources allouées dynamiquement qui ne sont pas gérées par un pointeur intelligent ou une variable automatique similaire seront inutilisables.

Les types intégrés sont tous sans échec, et les types de la bibliothèque standard prennent en charge la garantie de base au minimum. Suivez ces indications pour tout type défini par l'utilisateur qui doit être protégé contre les exceptions :

- Utilisez des pointeurs intelligents ou d'autres wrappers de type RAII pour gérer toutes les ressources. Évitez la fonctionnalité de gestion des ressources dans votre destructeur de classe, car le destructeur ne sera pas appelé si le constructeur lève une exception. Toutefois, si la classe est un gestionnaire de ressources dédié qui contrôle une seule ressource, il est acceptable d'utiliser le destructeur pour gérer des ressources.

- Comprenez qu'une exception levée dans un constructeur de classe de base ne peut pas être avalée dans un constructeur de classe dérivé. Si vous souhaitez traduire et lever à nouveau l'exception de classe de base dans un constructeur dérivé, utilisez une fonction try block.

- Déterminez s'il faut stocker tous les états de classe dans un membre de données qui est encapsulé dans un pointeur intelligent, en particulier si une classe a un concept d'« initialisation qui est autorisée à échouer ». Bien que C++ autorise les membres de données non initialisés, il ne prend pas en charge les instances de classe non initialisées ou partiellement initialisées. Un constructeur doit soit réussir, soit échouer ; aucun objet n'est créé si le constructeur ne s'exécute pas jusqu'à la fin.

- Ne permettez à aucune exception d'échapper à un destructeur. Un axiome de base C++ est que les destructeurs ne doivent jamais permettre à une exception de se propager vers le haut de la pile des appels. Si un destructeur doit effectuer une opération qui pourrait être un levage d'exception, il doit le faire dans un bloc Try Catch et avaler l'exception. La bibliothèque standard fournit cette garantie pour tous les destructeurs qu'elle définit.

Voir aussi

Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs
Procédure : interface entre code exceptionnel et code non exceptionnel

Guide pratique pour interfacer le code exceptionnel et le code non exceptionnel

Article • 16/06/2023

Cet article explique comment implémenter une gestion cohérente des exceptions dans le code C++, et comment traduire des exceptions vers et à partir de codes d'erreur aux limites des exceptions.

Parfois, le code C++ doit s'interfacer avec du code qui n'utilise pas d'exceptions (code non exceptionnel). Une telle interface est appelée *limite d'exception*. Par exemple, vous pouvez appeler la fonction Win32 `CreateFile` dans votre programme C++. `CreateFile` ne lève pas d'exceptions. Au lieu de cela, il définit des codes d'erreur qui peuvent être récupérés par la `GetLastError` fonction. Si votre programme C++ n'est pas trivial, vous préférez probablement avoir une stratégie cohérente de gestion des erreurs basée sur les exceptions. Et, vous ne voulez probablement pas abandonner les exceptions simplement parce que vous interfacez avec du code non exceptionnel. Vous ne souhaitez pas non plus mélanger les stratégies d'erreur basées sur les exceptions et les stratégies d'erreur non basées sur les exceptions dans votre code C++.

Appeler des fonctions non exceptionnelles à partir de C++

Lorsque vous appelez une fonction non exceptionnelle depuis C++, l'idée est d'encapsuler cette fonction dans une fonction C++ qui détecte les erreurs et lève éventuellement une exception. Lorsque vous concevez une telle fonction de wrapper, déterminez d'abord le type de garantie d'exception à fournir : noexcept, strong ou basic. Ensuite, créez la fonction de manière à ce que toutes les ressources, par exemple, les handles de fichiers, soient correctement libérées si une exception est levée. En règle générale, cela signifie que vous utilisez des pointeurs intelligents ou des gestionnaires de ressources similaires pour posséder les ressources. Pour plus d'informations sur les considérations relatives à la conception, consultez Guide pratique pour la conception pour la sécurité des exceptions.

Exemple

L'exemple suivant illustre les fonctions C++ qui utilisent les fonctions Win32 `CreateFile` et `ReadFile` en interne pour ouvrir et lire deux fichiers. La classe `File` est un wrapper RAII (Resource Acquisition Is Initialization) pour les handles de fichiers. Son constructeur détecte une condition « fichier introuvable » et lève une exception pour propager l'erreur dans la pile d'appels de l'exécutable C++ (dans cet exemple, la `main()` fonction). Si une exception est levée après qu'un objet `File` est entièrement construit, le destructeur appelle automatiquement `CloseHandle` pour libérer le handle de fichiers. (Si vous préférez, vous pouvez utiliser la classe ATL (Active Template Library) `CHandle` dans ce même but, ou une `unique_ptr` avec une fonction de suppression personnalisée.) Les fonctions qui appellent les API Win32 et CRT détectent les erreurs, puis lèvent les exceptions C++ à l'aide de la fonction définie `ThrowLastErrorIf` localement, qui à son tour utilise la `Win32Exception` classe dérivée de la `runtime_error` classe . Toutes les fonctions de cet exemple fournissent une garantie d'exception forte : si une exception est levée à un moment quelconque de ces fonctions, aucune ressource n'est divulguée et aucun état du programme n'est modifié.

C++

```
// compile with: /EHsc
#include <Windows.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <string>
#include <limits>
#include <stdexcept>

using namespace std;

string FormatErrorMessage(DWORD error, const string& msg)
{
    static const int BUFFERLENGTH = 1024;
    vector<char> buf(BUFFERLENGTH);
    FormatMessageA(FORMAT_MESSAGE_FROM_SYSTEM, 0, error, 0, buf.data(),
        BUFFERLENGTH - 1, 0);
    return string(buf.data()) + "   ("  + msg  + ")";
}

class Win32Exception : public runtime_error
{
private:
    DWORD m_error;
public:
    Win32Exception(DWORD error, const string& msg)
        : runtime_error(FormatErrorMessage(error, msg)), m_error(error) { }

    DWORD GetErrorCode() const { return m_error; }
};
```

```
void ThrowLastErrorIf(bool expression, const string& msg)
{
    if (expression) {
        throw Win32Exception(GetLastError(), msg);
    }
}

class File
{
private:
    HANDLE m_handle;

    // Declared but not defined, to avoid double closing.
    File& operator=(const File&);
    File(const File&);
public:
    explicit File(const string& filename)
    {
        m_handle = CreateFileA(filename.c_str(), GENERIC_READ,
FILE_SHARE_READ,
            nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, nullptr);
        ThrowLastErrorIf(m_handle == INVALID_HANDLE_VALUE,
            "CreateFile call failed on file named " + filename);
    }

    ~File() { CloseHandle(m_handle); }

    HANDLE GetHandle() { return m_handle; }
};

size_t GetFileSizeSafe(const string& filename)
{
    File fobj(filename);
    LARGE_INTEGER filesize;

    BOOL result = GetFileSizeEx(fobj.GetHandle(), &filesize);
    ThrowLastErrorIf(result == FALSE, "GetFileSizeEx failed: " + filename);

    if (filesize.QuadPart < (numeric_limits<size_t>::max)()) {
        return filesize.QuadPart;
    } else {
        throw;
    }
}

vector<char> ReadFileVector(const string& filename)
{
    File fobj(filename);
    size_t filesize = GetFileSizeSafe(filename);
    DWORD bytesRead = 0;

    vector<char> readbuffer(filesize);

    BOOL result = ReadFile(fobj.GetHandle(), readbuffer.data(),
```

```
readbuffer.size(),
        &bytesRead, nullptr);
    ThrowLastErrorIf(result == FALSE, "ReadFile failed: " + filename);

    cout << filename << " file size: " << filesize << ", bytesRead: "
        << bytesRead << endl;

    return readbuffer;
}

bool IsFileDiff(const string& filename1, const string& filename2)
{
    return ReadFileVector(filename1) != ReadFileVector(filename2);
}

#include <iomanip>

int main ( int argc, char* argv[] )
{
    string filename1("file1.txt");
    string filename2("file2.txt");

    try
    {
        if(argc > 2) {
            filename1 = argv[1];
            filename2 = argv[2];
        }

        cout << "Using file names " << filename1 << " and " << filename2 <<
endl;

        if (IsFileDiff(filename1, filename2)) {
            cout << "+++ Files are different." << endl;
        } else {
            cout<< "=== Files match." << endl;
        }
    }
    catch(const Win32Exception& e)
    {
        ios state(nullptr);
        state.copyfmt(cout);
        cout << e.what() << endl;
        cout << "Error code: 0x" << hex << uppercase << setw(8) <<
setfill('0')
            << e.GetErrorCode() << endl;
        cout.copyfmt(state); // restore previous formatting
    }
}
```

Appeler du code exceptionnel à partir d'un code non exceptionnel

Les fonctions C++ déclarées comme `extern "C"` peuvent être appelées par les programmes C. Les serveurs COM C++ peuvent être consommés par du code écrit dans n'importe quel nombre de langues différentes. Lorsque vous implémentez des fonctions publiques prenant en charge les exceptions en C++ pour qu'elles soient appelées par du code non exceptionnel, la fonction C++ ne doit pas autoriser les exceptions à se propager à l'appelant. Ces appelants n'ont aucun moyen d'intercepter ou de gérer les exceptions C++. Le programme peut se terminer, fuiter des ressources ou provoquer un comportement non défini.

Nous recommandons à votre `extern "C"` fonction C++ d'intercepter spécifiquement chaque exception qu'elle sait gérer et, le cas échéant, de convertir l'exception en code d'erreur que l'appelant comprend. Si toutes les exceptions potentielles ne sont pas connues, la fonction C++ doit avoir un bloc `catch(...)` comme dernier gestionnaire. Dans ce cas, il est préférable de signaler une erreur irrécupérable à l'appelant, car votre programme peut être dans un état inconnu et irrécupérable.

L'exemple suivant montre une fonction qui suppose que toute exception pouvant être levée est un `Win32Exception` type d'exception ou dérivé de `std::exception`. La fonction intercepte toute exception de ces types et propage les informations d'erreur sous la forme d'un code d'erreur Win32 à l'appelant.

C++

```
BOOL DiffFiles2(const string& file1, const string& file2)
{
    try
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return FALSE;
        }
        return TRUE;
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }

    catch(std::exception& e)
    {
```

```
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return FALSE;
}
```

Lorsque vous convertissez des exceptions en codes d'erreur, il existe un problème potentiel : les codes d'erreur ne contiennent souvent pas la richesse des informations qu'une exception peut stocker. Pour résoudre ce problème, vous pouvez fournir un `catch` bloc pour chaque type d'exception spécifique qui peut être levée et effectuer une journalisation pour enregistrer les détails de l'exception avant qu'elle ne soit convertie en code d'erreur. Cette approche peut créer du code répétitif si plusieurs fonctions utilisent toutes le même ensemble de `catch` blocs. Une bonne façon d'éviter la répétition du code consiste à refactoriser ces blocs en une fonction d'utilitaire privée qui implémente le `try` et `catch` bloque et accepte un objet de fonction appelé dans le `try` bloc. Dans chaque fonction publique, passez le code à la fonction utilitaire en tant qu'expression lambda.

C++

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}
```

L'exemple suivant montre comment écrire l'expression lambda qui définit le foncteur. Une expression lambda est souvent plus facile à lire inline que le code qui appelle un objet de fonction nommé.

C++

```
bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
```

```
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}
```

Pour plus d'informations sur les expressions lambda, voir Expressions lambda.

Appeler du code exceptionnel via du code non exceptionnel à partir d'un code exceptionnel

Il est possible, mais non recommandé, de lever des exceptions dans le code ignorant les exceptions. Par exemple, votre programme C++ peut appeler une bibliothèque qui utilise des fonctions de rappel que vous fournissez. Dans certaines circonstances, vous pouvez lever des exceptions des fonctions de rappel dans le code non exceptionnel que votre appelant d'origine peut gérer. Toutefois, les circonstances dans lesquelles les exceptions peuvent fonctionner avec succès sont strictes. Vous devez compiler le code de la bibliothèque de manière à préserver la sémantique de déroulement de la pile. Le code ignorant l'exception ne peut rien faire qui puisse intercepter l'exception C++. En outre, le code de bibliothèque entre l'appelant et le rappel ne peut pas allouer de ressources locales. Par exemple, le code qui ne prend pas en charge les exceptions ne peut pas avoir de locaux qui pointent vers la mémoire du tas allouée. Ces ressources sont divulguées lorsque la pile est dissociée.

Ces conditions doivent être remplies pour lever des exceptions dans le code qui ne prend pas en compte les exceptions :

- Vous pouvez générer le chemin du code entier dans le code qui ne prend pas en charge les exceptions à l'aide de `/EHs`,
- Il n'y a pas de ressources allouées localement qui peuvent s'échapper lorsque la pile est déwound,
- Le code n'a `__except` pas de gestionnaires d'exceptions structurés qui interceptent toutes les exceptions.

Étant donné que la levée d'exceptions sur un code non exceptionnel est sujette aux erreurs et peut entraîner des problèmes de débogage difficiles, nous ne vous recommandons pas.

Voir aussi

Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs
Procédure : concevoir pour la sécurité des exceptions

Instructions try, throw et catch (C++)

Article • 03/04/2023

Pour implémenter la gestion des exceptions en C++, vous utilisez `try` des expressions , `throw` et `catch` .

Tout d'abord, utilisez un `try` bloc pour inclure une ou plusieurs instructions susceptibles de lever une exception.

Une `throw` expression signale qu'une condition exceptionnelle (souvent une erreur) s'est produite dans un `try` bloc. Vous pouvez utiliser un objet de n'importe quel type comme opérande d'une `throw` expression. En général, cet objet est utilisé pour transmettre des informations sur l'erreur. Dans la plupart des cas, nous vous recommandons d'utiliser la std::exception classe ou l'une des classes dérivées définies dans la bibliothèque standard. Si l'une d'elles n'est pas appropriée, nous vous recommandons de dériver votre propre classe d'exception à partir de `std::exception`.

Pour gérer les exceptions qui peuvent être levées, implémentez un ou plusieurs `catch` blocs immédiatement après un `try` bloc. Chaque `catch` bloc spécifie le type d'exception qu'il peut gérer.

Cet exemple montre un `try` bloc et ses gestionnaires. Supposons que `GetNetworkResource()` acquiert des données via une connexion réseau et que les deux types d'exception sont des classes définies par l'utilisateur qui dérivent de `std::exception`. Notez que les exceptions sont interceptées par `const` référence dans l'instruction `catch` . Nous vous recommandons de lever des exceptions par valeur et de les intercepter par référence const.

Exemple

C++

```
MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
```

```
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

Remarques

Le code après la `try` clause est la section protégée du code. L'expression `throw`*lève*(c'est-à-dire, déclenche) une exception. Le bloc de code après la `catch` clause est le gestionnaire d'exceptions. Il s'agit du gestionnaire qui *intercepte* l'exception levée si les types dans les `throw` expressions et `catch` sont compatibles. Pour obtenir la liste des règles qui régissent la correspondance de type dans `catch` les blocs, consultez Comment les blocs catch sont évalués. Si l'instruction `catch` spécifie des points de suspension (...) au lieu d'un type, le `catch` bloc gère chaque type d'exception. Lorsque vous compilez avec l'option /EHa , il peut s'agir d'exceptions structurées en C et d'exceptions asynchrones générées par le système ou générées par l'application, telles que la protection de la mémoire, la division par zéro et les violations à virgule flottante. Étant donné que `catch` les blocs sont traités dans l'ordre du programme pour trouver un type correspondant, un gestionnaire de points de suspension doit être le dernier gestionnaire du bloc associé `try` . Utilisez `catch(...)` avec précaution ; n'autorisez pas un programme à continuer, sauf si le bloc catch sait comment gérer l'exception spécifique qui est interceptée. En général, un bloc `catch(...)` est utilisé pour enregistrer des erreurs et effectuer un nettoyage spécial avant l'arrêt de l'exécution du programme.

Une `throw` expression qui n'a pas d'opérande fait renaître l'exception en cours de traitement. Nous vous recommandons ce formulaire lors de la nouvelle croissance de l'exception, car cela conserve les informations de type polymorphe de l'exception d'origine. Une telle expression ne doit être utilisée que dans un `catch` gestionnaire ou dans une fonction appelée à partir d'un `catch` gestionnaire. L'objet d'exception rethrown est l'objet d'exception d'origine, et non une copie.

C++

```
try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions - dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

Voir aussi

Meilleures pratiques C++ modernes pour les exceptions et la gestion des erreurs

Mots clés

Exceptions C++ non gérées

__uncaught_exception

Mode d'évaluation des blocs Catch (C++)

Article • 03/04/2023

C++ vous permet de lever des exceptions de tout type, bien qu'en général il soit recommandé de lever des types dérivés de std::exception. Une exception C++ peut être interceptée par un `catch` gestionnaire qui spécifie le même type que l'exception levée, ou par un gestionnaire qui peut intercepter n'importe quel type d'exception.

Si le type de l'exception levée est une classe, qui possède également une classe (ou des classes) de base, elle peut être interceptée par des gestionnaires qui acceptent les classes de base du type de l'exception, ainsi que des références aux bases du type de l'exception. Notez que lorsqu'une exception est interceptée par une référence, elle est associée à l'objet exception levée réel. Dans le cas contraire, il s'agit d'une copie (similaire à un argument pour une fonction).

Lorsqu'une exception est levée, elle peut être interceptée par les types de `catch` gestionnaires suivants :

- Gestionnaire capable d'accepter tout type (à l'aide de la syntaxe avec points de suspension).
- Gestionnaire qui accepte le même type que l'objet d'exception ; car il s'agit d'une copie et `const` `volatile` les modificateurs sont ignorés.
- Gestionnaire qui accepte une référence au même type que l'objet exception.
- Gestionnaire qui accepte une référence à une `const` forme ou `volatile` du même type que l'objet d'exception.
- Gestionnaire qui accepte une classe de base du même type que l'objet exception ; puisqu'il s'agit d'une copie et `const` `volatile` que les modificateurs sont ignorés. Le `catch` gestionnaire d'une classe de base ne doit pas précéder le `catch` gestionnaire de la classe dérivée.
- Gestionnaire qui accepte une référence à une classe de base du même type que l'objet exception.
- Gestionnaire qui accepte une référence à une `const` forme ou `volatile` d'une classe de base du même type que l'objet exception.

- Gestionnaire qui accepte un pointeur vers lequel un objet pointeur levé peut être converti via des règles de conversion de pointeur standard.

L'ordre dans `catch` lequel les gestionnaires apparaissent est significatif, car les gestionnaires d'un bloc donné `try` sont examinés dans l'ordre de leur apparence. Par exemple, il est incorrect de placer le gestionnaire d'une classe de base avant le gestionnaire d'une classe dérivée. Une fois qu'un gestionnaire correspondant `catch` est trouvé, les gestionnaires suivants ne sont pas examinés. Par conséquent, un gestionnaire de points de suspension `catch` doit être le dernier gestionnaire de son `try` bloc. Par exemple :

C++

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

Dans cet exemple, le gestionnaire de points de suspension `catch` est le seul gestionnaire examiné.

Voir aussi

Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs

Exceptions et déroulement de pile en C++

Article • 03/04/2023

Dans le mécanisme d'exception C++, le contrôle passe de l'instruction Throw à la première instruction Catch qui peut gérer le type levé. Lorsque l'instruction catch est atteinte, toutes les variables automatiques qui se trouvent dans l'étendue entre les instructions throw et catch sont détruites dans un processus appelé déroulement *de la pile*. L'exécution du déroulement de pile se déroule comme suit :

1. Le contrôle atteint l'instruction `try` par exécution séquentielle normale. La section protégée du `try` bloc est exécutée.

2. Si aucune exception n'est levée pendant l'exécution de la section protégée, les `catch` clauses qui suivent le `try` bloc ne sont pas exécutées. L'exécution se poursuit au niveau de l'instruction après la dernière `catch` clause qui suit le bloc associé `try` .

3. Si une exception est levée pendant l'exécution de la section protégée ou dans une routine que la section protégée appelle directement ou indirectement, un objet exception est créé à partir de l'objet créé par l'opérande `throw` . (Cela implique qu'un constructeur de copie peut être impliqué.) À ce stade, le compilateur recherche une `catch` clause dans un contexte d'exécution plus élevé qui peut gérer une exception du type levée, ou un `catch` gestionnaire capable de gérer n'importe quel type d'exception. Les `catch` gestionnaires sont examinés dans l'ordre de leur apparition après le `try` bloc. Si aucun gestionnaire approprié n'est trouvé, le bloc englobant `try` dynamique suivant est examiné. Ce processus se poursuit jusqu'à ce que le bloc englobant le plus à l'extérieur `try` soit examiné.

4. Si aucun gestionnaire correspondant ne parvient à être trouvé, ou si une exception se produit pendant le processus de déroulement mais avant que le gestionnaire n'obtienne le contrôle, la fonction runtime `terminate` prédéfinie est appelée. Si une autre exception se produit après la levée de l'exception mais avant le début du déroulement, la fonction `terminate` est appelée.

5. Si un gestionnaire correspondant `catch` est trouvé et qu'il intercepte par valeur, son paramètre formel est initialisé en copiant l'objet exception. Si l'interception s'effectue par référence, le paramètre est initialisé pour faire référence à l'objet exception. Une fois le paramètre formel initialisé, le processus de déroulement de

la pile démarre. Cela implique la destruction de tous les objets automatiques qui ont été entièrement construits, mais pas encore détruits, entre le début du `try` bloc associé au `catch` gestionnaire et le site de levée de l'exception. La destruction se produit dans l'ordre inverse de la construction. Le `catch` gestionnaire est exécuté et le programme reprend l'exécution après le dernier gestionnaire, c'est-à-dire à la première instruction ou construction qui n'est pas un `catch` gestionnaire. Le contrôle peut uniquement entrer un `catch` gestionnaire par le biais d'une exception levée, jamais par le biais d'une `goto` instruction ou d'une `case` étiquette dans une `switch` instruction.

Exemple de déroulement de pile

L'exemple suivant illustre le déroulement de la pile lorsqu'une exception est levée. L'exécution sur le thread passe de l'instruction Throw dans `C` à l'instruction Catch dans `main` et déroule chaque fonction pendant le processus. Notez l'ordre dans lequel les objets `Dummy` sont créés puis détruits lorsqu'ils passent hors de portée. Notez également qu'aucune fonction ne se termine, sauf `main`, qui contient l'instruction Catch. La fonction `A` ne retourne jamais d'un appel à `B()`, et `B` ne retourne jamais d'un appel à `C()`. Si vous supprimez les marques de commentaire de la définition du pointeur `Dummy` et de l'instruction Delete correspondante et que vous exécutez le programme, le pointeur n'est jamais supprimé. Cela indique ce qui peut se produire lorsque les fonctions ne fournissent pas de garantie d'exception. Pour plus d'informations, consultez Guide pratique pour concevoir des exceptions. Si vous commentez l'instruction Catch, vous pouvez observer ce qui se produit lorsqu'un programme se termine en raison d'une exception non gérée.

C++

```
#include <string>
#include <iostream>
using namespace std;

class MyException{};
class Dummy
{
    public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created
Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s  << MyName <<  endl; }
    string MyName;
    int level;
};
```

```
void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
  //  Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
 //   delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}

/* Output:
    Entering main
    Created Dummy: M
    Copy created Dummy: M
    Entering FunctionA
    Copy created Dummy: A
    Entering FunctionB
    Copy created Dummy: B
    Entering FunctionC
```

```
    Destroyed Dummy: C
    Destroyed Dummy: B
    Destroyed Dummy: A
    Destroyed Dummy: M
    Caught an exception of type: class MyException
    Exiting main.

*/
```

Spécifications d'exception (throw, noexcept) (C++)

Article • 03/04/2023

Les spécifications d'exception sont une fonctionnalité de langage C++ qui indique l'intention du programmeur concernant les types d'exceptions pouvant être propagés par une fonction. Vous pouvez spécifier qu'une fonction peut ou non quitter par une exception à l'aide d'une *spécification d'exception*. Le compilateur peut utiliser ces informations pour optimiser les appels à la fonction et pour arrêter le programme si une exception inattendue échappe à la fonction.

Avant C++17, il existait deux types de spécifications d'exceptions. La *spécification noexcept* était nouvelle dans C++11. Elle spécifie si le jeu d'exceptions potentielles pouvant échapper à la fonction est vide. La *spécification d'exception dynamique*, ou `throw(optional_type_list)` spécification, a été déconseillée en C++11 et supprimée en C++17, à l'exception de `throw()`, qui est un alias pour `noexcept(true)`. Cette spécification d'exception a été conçue pour fournir des informations récapitulatives sur les exceptions qui peuvent être levées d'une fonction, mais dans la pratique, elle a été jugée problématique. La seule spécification d'exception dynamique qui s'est avérée quelque peu utile était la spécification inconditionnelle `throw()` . Par exemple, la déclaration de fonction :

```
C++

void MyFunction(int i) throw();
```

indique au compilateur que la fonction ne lève pas d'exception. Toutefois, en /std:c++14 mode, cela peut entraîner un comportement non défini si la fonction lève une exception. Par conséquent, nous vous recommandons d'utiliser l'opérateur noexcept au lieu de celui ci-dessus :

```
C++

void MyFunction(int i) noexcept;
```

Le tableau suivant récapitule l'implémentation Microsoft C++ des spécifications d'exception :

Spécification d'exception	Signification

Spécification d'exception	Signification

`noexcept`
`noexcept(true)`
`throw()`
	La fonction ne lève pas d'exception. En `/std:c++14` mode (qui est la valeur par défaut) `noexcept` et `noexcept(true)` sont équivalents. Lorsqu'une exception est levée à partir d'une fonction déclarée `noexcept` ou `noexcept(true)`, std::terminate est appelée. Lorsqu'une exception est levée à partir d'une fonction déclarée en `throw()` `/std:c++14` mode, le résultat est un comportement non défini. Aucune fonction spécifique n'est appelée. Il s'agit d'une divergence par rapport à la norme C++14, qui exigeait que le compilateur appelle std::unexpected.
	Visual Studio 2017 version 15.5 et ultérieures : En `/std:c++17` mode , `noexcept`, `noexcept(true)` et `throw()` sont tous équivalents. En `/std:c++17` mode , `throw()` est un alias pour `noexcept(true)`. En `/std:c++17` mode et versions ultérieures, lorsqu'une exception est levée à partir d'une fonction déclarée avec l'une de ces spécifications, std::terminate est appelée comme requis par la norme C++17.

`noexcept(false)`
`throw(...)`
Aucune spécification
	La fonction peut lever une exception de n'importe quel type.

`throw(type)`	(**C++14 et versions antérieures**) La fonction peut lever une exception de type `type`. Le compilateur accepte la syntaxe, mais l'interprète comme `noexcept(false)`. En `/std:c++17` mode et versions ultérieures, le compilateur émet l'avertissement C5040.

Si la gestion des exceptions est utilisée dans une application, il doit y avoir une fonction dans la pile des appels qui gère les exceptions levées avant qu'elles ne quittent l'étendue externe d'une fonction marquée `noexcept`, `noexcept(true)` ou `throw()`. Si des fonctions appelées entre celle qui lève une exception et celle qui gère l'exception sont spécifiées en tant que `noexcept`, `noexcept(true)` (ou `throw()` en `/std:c++17` mode), le programme est arrêté lorsque la fonction noexcept propage l'exception.

Le comportement d'exception d'une fonction dépend des facteurs suivants :

- Quel mode de compilation standard de langage est défini.
- si vous compilez la fonction sous C ou C++ ;
- Option /EH de compilateur que vous utilisez.
- si vous spécifiez de manière explicite la spécification d'exception.

Les spécifications d'exceptions explicites ne sont pas autorisées sur les fonctions C. Une fonction C est supposée ne pas lever d'exceptions sous `/EHsc`, et peut lever des

exceptions structurées sous `/EHs`, `/EHa` ou `/EHac`.

Le tableau suivant récapitule si une fonction C++ peut potentiellement être levée sous différentes options de gestion des exceptions du compilateur :

Fonction	`/EHsc`	`/EHs`	`/EHa`	`/EHac`
Fonction C++ sans spécification d'exception	Oui	Oui	Oui	Oui
Fonction C++ avec `noexcept`, `noexcept(true)` ou `throw()` spécification d'exception	Non	Non	Oui	Oui
Fonction C++ avec `noexcept(false)`, `throw(...)` ou `throw(type)` spécification d'exception	Oui	Oui	Oui	Oui

Exemple

C++

```
// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
   printf_s("in handler\n");
}

void f1(void) throw(int) {
   printf_s("About to throw 1\n");
   if (1)
      throw 1;
}

void f5(void) throw() {
   try {
      f1();
   }
   catch(...) {
      handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//   f1();
// }

void __declspec(nothrow) f2(void) {
   try {
      f1();
```

```
   }
   catch(int) {
      handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
   f1();
}

int main() {
   f2();

   try {
      f4();
   }
   catch(...) {
      printf_s("Caught exception from f4\n");
   }
   f5();
}
```

Output

```
About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler
```

Voir aussi

tryinstructions , throwet catch (C++)

Meilleures pratiques C++ modernes pour les exceptions et la gestion des erreurs

`noexcept` (C++)

Article • 16/06/2023

C++11 : Spécifie si une fonction peut lever des exceptions.

Syntaxe

`noexcept-specifier`:

`noexcept`

`noexcept-expression`

`throw` **`(`** **`)`**

`noexcept-expression`:

`noexcept` **`(`** *`constant-expression`* **`)`**

Paramètres

`constant-expression`

Expression constante de type **`bool`** qui indique si l'ensemble des types d'exceptions potentiels est vide. La version inconditionnelle est équivalente à `noexcept(true)`.

Notes

Un *`noexcept-expression`* est un type de *spécification d'exception* : suffixe d'une déclaration de fonction qui représente un ensemble de types pouvant être mis en correspondance par un gestionnaire d'exceptions pour toute exception qui quitte une fonction. L'opérateur `noexcept(constant_expression)` conditionnel unaire lorsque *`constant_expression`* génère **`true`**, et son synonyme **`noexcept`** inconditionnel , spécifient que l'ensemble des types d'exceptions potentiels qui peuvent quitter une fonction est vide. Autrement dit, la fonction ne lève jamais d'exception et n'autorise jamais la propagation d'une exception en dehors de son étendue. L'opérateur `noexcept(constant_expression)` lorsque *`constant_expression`* génère **`false`**, ou l'absence d'une spécification d'exception (autre que pour un destructeur ou une fonction de désallocation), indique que le jeu d'exceptions potentielles qui peuvent quitter la fonction est l'ensemble de tous les types.

Marquez une fonction comme **`noexcept`** uniquement si toutes les fonctions qu'elle appelle, directement ou indirectement, sont également **`noexcept`** ou **`const`**. Le compilateur ne case activée pas nécessairement tous les chemins de code pour les

exceptions qui peuvent atteindre une `noexcept` fonction. Si une exception quitte l'étendue externe d'une fonction marquée `noexcept`, std::terminate est appelée immédiatement et il n'y a aucune garantie que les destructeurs des objets dans l'étendue seront appelés. Utilisez `noexcept` au lieu du spécificateur `throw()` d'exception dynamique . La *spécification d'exception dynamique*, ou `throw(optional_type_list)` spécification, a été déconseillée en C++11 et supprimée en C++17, à l'exception de `throw()`, qui est un alias pour `noexcept(true)`. Nous vous recommandons de l'appliquer `noexcept` à toute fonction qui n'autorise jamais une exception à se propager dans la pile des appels. Lorsqu'une fonction est déclarée `noexcept`, elle permet au compilateur de générer du code plus efficace dans plusieurs contextes différents. Pour plus d'informations, consultez Spécifications des exceptions.

Exemple

Un modèle de fonction qui copie son argument peut être déclaré `noexcept` à condition que l'objet copié soit un ancien type de données simple (POD). Cette fonction peut être déclarée comme suit :

C++

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
   // ...
}
```

Voir aussi

Meilleures pratiques C++ modernes pour les exceptions et la gestion des erreurs
Spécifications d'exception (throw, noexcept)

Exceptions C++ non gérées

Article • 03/04/2023

Si un gestionnaire correspondant (ou gestionnaire de points de suspension `catch`) est introuvable pour l'exception actuelle, la fonction d'exécution `terminate` prédéfinie est appelée. (Vous pouvez également appeler `terminate` explicitement l'un de vos gestionnaires.) L'action par défaut de `terminate` consiste à appeler `abort`. Si vous souhaitez `terminate` pour appeler une autre fonction dans votre programme avant de quitter l'application, appelez la fonction `set_terminate` avec le nom de la fonction à appeler comme unique argument. Vous pouvez appeler `set_terminate` à tout moment dans votre programme. La `terminate` routine appelle toujours la dernière fonction donnée en tant qu'argument à `set_terminate`.

Exemple

L'exemple suivant lève une exception `char *`, mais ne contient pas de gestionnaire désigné pour intercepter des exceptions de type `char *`. L'appel à `set_terminate` indique à `terminate` d'appeler `term_func`.

C++

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
   cout << "term_func was called by terminate." << endl;
   exit( -1 );
}
int main() {
   try
   {
      set_terminate( term_func );
      throw "Out of memory!"; // No catch handler for this exception
   }
   catch( int )
   {
      cout << "Integer exception raised." << endl;
   }
   return 0;
}
```

Output

```
Output
```

```
term_func was called by terminate.
```

La fonction `term_func` doit terminer le programme ou le thread actuel, de préférence en appelant `exit`. Si elle ne fait pas cela et revient à la place à son appelant, la fonction `abort` est appelée.

Voir aussi

Meilleures pratiques C++ modernes pour la gestion des exceptions et des erreurs

Mélange d'exceptions C (structurées) et C++

Article • 16/06/2023

Si vous souhaitez écrire du code portable, l'utilisation de la gestion structurée des exceptions (SEH) dans un programme C++ n'est pas recommandée. Toutefois, vous pouvez parfois vouloir compiler à l'aide /EHa d'exceptions structurées et de code source C++, et avoir besoin d'une certaine facilité pour gérer les deux types d'exceptions. Étant donné qu'un gestionnaire d'exceptions structurées n'a aucun concept d'objets ou d'exceptions typées, il ne peut pas gérer les exceptions levées par le code C++. Toutefois, les gestionnaires C++ `catch` peuvent gérer les exceptions structurées. La syntaxe de gestion des exceptions C++ (`try`, `throw`, `catch`) n'est pas acceptée par le compilateur C, mais la syntaxe de gestion des exceptions structurée (`__try`, `__except`, `__finally`) est prise en charge par le compilateur C++.

Consultez _set_se_translator pour plus d'informations sur la façon de gérer les exceptions structurées en tant qu'exceptions C++.

Si vous combinez des exceptions structurées et C++, tenez compte des problèmes potentiels suivants :

- Les exceptions C++ et les exceptions structurées ne peuvent pas être mélangées dans la même fonction.
- Les gestionnaires d'arrêt (`__finally` blocs) sont toujours exécutés, même pendant le déroulement après la levée d'une exception.
- La gestion des exceptions C++ peut intercepter et conserver la sémantique de déroulement dans tous les modules compilés avec les options du /EH compilateur, qui activent la sémantique de déroulement.
- Il peut y avoir des situations dans lesquelles les fonctions de destructeur ne sont pas appelées pour tous les objets. Par exemple, une exception structurée peut se produire lors d'une tentative d'appel de fonction via un pointeur de fonction non initialisé. Si les paramètres de fonction sont des objets construits avant l'appel, les destructeurs de ces objets ne sont pas appelés pendant le déroulement de la pile.

Étapes suivantes

- Utilisation setjmp ou longjmp dans des programmes C++

Consultez plus d'informations sur l'utilisation de `setjmp` et `longjmp` dans les programmes C++.

- Gérer les exceptions structurées en C++

  Consultez des exemples de façons d'utiliser C++ pour gérer les exceptions structurées.

Voir aussi

Meilleures pratiques C++ modernes pour les exceptions et la gestion des erreurs

Utilisation de setjmp et longjmp

Article • 16/06/2023

Lorsque setjmp et longjmp sont utilisés ensemble, ils fournissent un moyen d'exécuter un non local `goto`. Ils sont généralement utilisés dans le code C pour passer le contrôle d'exécution à la gestion des erreurs ou code de récupération dans une routine précédemment appelée sans utiliser les conventions d'appel ou de retour standard.

> ⊗ **Attention**
>
> Étant donné que `setjmp` et `longjmp` ne prennent pas en charge la destruction correcte des objets frame de pile de manière portable entre les compilateurs C++, et parce qu'ils peuvent dégrader les performances en empêchant l'optimisation sur les variables locales, nous ne recommandons pas leur utilisation dans les programmes C++. Nous vous recommandons d'utiliser `try` les constructions et `catch` à la place.

Si vous décidez d'utiliser `setjmp` et `longjmp` dans un programme C++, incluez <également setjmp.h> ou <setjmpex.h> pour garantir une interaction correcte entre les fonctions et la gestion structurée des exceptions (SEH) ou C++.

Section spécifique à Microsoft

Si vous utilisez une option /EH pour compiler du code C++, les destructeurs pour les objets locaux sont appelés pendant le déroulement de la pile. Toutefois, si vous utilisez **/EHs** ou **/EHsc** pour compiler, et si l'une de vos fonctions n'utilise aucun appelexcept`longjmp`, le déroulement du destructeur pour cette fonction peut ne pas se produire, en fonction de l'état de l'optimiseur.

Dans le code portable, lorsqu'un `longjmp` appel est exécuté, la destruction correcte des objets basés sur des images n'est pas explicitement garantie par la norme et peut ne pas être prise en charge par d'autres compilateurs. Pour vous informer, au niveau d'avertissement 4, un appel à `setjmp` provoque l'avertissement C4611 : l'interaction entre « _setjmp » et la destruction d'objets C++ n'est pas portable.

FIN de la section spécifique à Microsoft

Voir aussi

Mélange d'exceptions C (structurées) et C++

Gérer les exceptions structurées en C++

Article • 16/06/2023

La principale différence entre la gestion des exceptions structurées C (SEH) et la gestion des exceptions C++ est que le modèle de gestion des exceptions C++ traite les types, tandis que le modèle de gestion des exceptions structurées C traite des exceptions d'un type ; plus précisément, `unsigned int`. Autrement dit, les exceptions C sont identifiées par une valeur entière non signée, tandis que les exceptions C++ sont identifiées par le type de données. Lorsqu'une exception structurée est levée en C, chaque gestionnaire possible exécute un filtre qui examine le contexte d'exception C et détermine s'il faut accepter l'exception, la transmettre à un autre gestionnaire ou l'ignorer. Lorsqu'une exception est levée en C++, elle peut être de n'importe quelle type.

Une deuxième différence est que le modèle de gestion des exceptions structurées C est appelé *asynchrone*, car les exceptions se produisent secondairement au flux normal de contrôle. Le mécanisme de gestion des exceptions C++ est entièrement *synchrone*, ce qui signifie que les exceptions se produisent uniquement lorsqu'elles sont levées.

Lorsque vous utilisez l'option de compilateur /EHs ou /EHsc , aucun gestionnaire d'exceptions C++ ne gère les exceptions structurées. Ces exceptions sont gérées uniquement par `__except` des gestionnaires d'exceptions structurés ou `__finally` des gestionnaires d'arrêt structurés. Pour plus d'informations, consultez Gestion structurée des exceptions (C/C++).

Sous l'option du compilateur /EHa , si une exception C est déclenchée dans un programme C++, elle peut être gérée par un gestionnaire d'exceptions structuré avec son filtre associé ou par un gestionnaire C++ `catch` , selon la valeur la plus proche dynamique du contexte d'exception. Par exemple, cet exemple de programme C++ déclenche une exception C à l'intérieur d'un contexte C++ `try` :

Exemple - Intercepter une exception C dans un bloc catch C++

C++

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHa
#include <iostream>

using namespace std;
void SEHFunc( void );
```

```
int main() {
   try {
      SEHFunc();
   }
   catch( ... ) {
      cout << "Caught a C exception."<< endl;
   }
}

void SEHFunc() {
   __try {
      int x, y = 0;
      x = 5 / y;
   }
   __finally {
      cout << "In finally." << endl;
   }
}
```

Output

```
In finally.
Caught a C exception.
```

Classes wrapper d'exception C

Dans un exemple simple comme ci-dessus, l'exception C ne peut être interceptée que par des points de suspension (...) `catch` Gestionnaire. Aucune information sur le type ou la nature de l'exception n'est communiquée au gestionnaire. Bien que cette méthode fonctionne, dans certains cas, vous pouvez définir une transformation entre les deux modèles de gestion des exceptions afin que chaque exception C soit associée à une classe spécifique. Pour en transformer une, vous pouvez définir une classe « wrapper » d'exception C, qui peut être utilisée ou dérivée de afin d'attribuer un type de classe spécifique à une exception C. Ainsi, chaque exception C peut être gérée séparément par un gestionnaire C++ `catch` spécifique, au lieu de toutes les exceptions dans un seul gestionnaire.

Votre classe wrapper peut avoir une interface constituée de certaines fonctions membres qui déterminent la valeur de l'exception et qui accèdent aux informations de contexte d'exception étendues fournies par le modèle d'exception C. Vous pouvez également définir un constructeur par défaut et un constructeur qui accepte un `unsigned int` argument (pour fournir la représentation d'exception C sous-jacente) et un constructeur de copie au niveau du bit. Voici une implémentation possible d'une classe wrapper d'exception C :

C++

```
// exceptions_Exception_Handling_Differences2.cpp
// compile with: /c
class SE_Exception {
private:
   SE_Exception() {}
   SE_Exception( SE_Exception& ) {}
   unsigned int nSE;
public:
   SE_Exception( unsigned int n ) : nSE( n ) {}
   ~SE_Exception() {}
   unsigned int getSeNumber() {
      return nSE;
   }
};
```

Pour utiliser cette classe, installez une fonction de traduction d'exception C personnalisée appelée par le mécanisme de gestion des exceptions interne chaque fois qu'une exception C est levée. Dans votre fonction de traduction, vous pouvez lever toute exception typée (peut-être un `SE_Exception` type ou un type de classe dérivé de `SE_Exception`) qui peut être interceptée par un gestionnaire C++ `catch` correspondant approprié. À la place, la fonction de traduction peut retourner, ce qui indique qu'elle n'a pas géré l'exception. Si la fonction de traduction elle-même déclenche une exception C, terminate est appelé.

Pour spécifier une fonction de traduction personnalisée, appelez la fonction _set_se_translator avec le nom de votre fonction de traduction comme argument unique. La fonction de traduction que vous écrivez est appelée une fois pour chaque appel de fonction sur la pile qui contient `try` des blocs. Il n'existe aucune fonction de traduction par défaut ; si vous n'en spécifiez pas en appelant **_set_se_translator**, l'exception C ne peut être interceptée que par un gestionnaire de `catch` points de suspension.

Exemple : utiliser une fonction de traduction personnalisée

Par exemple, le code suivant installe une fonction de traduction personnalisée, puis lève une exception C qui est encapsulée par la classe `SE_Exception` :

C++

```
// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHa
```

```
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
   SE_Exception() {}
   unsigned int nSE;
public:
   SE_Exception( SE_Exception& e) : nSE(e.nSE) {}
   SE_Exception(unsigned int n) : nSE(n) {}
   ~SE_Exception() {}
   unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}
```

Output

```
In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094
```

Voir aussi

Mélange d'exceptions C (structurées) et d'exceptions C++

Structured Exception Handling (C/C++)

Article • 16/06/2023

La gestion structurée des exceptions (SEH) est une extension Microsoft de C et C++ pour gérer correctement certaines situations de code exceptionnelles, telles que les erreurs matérielles. Bien que Windows et Microsoft C++ prennent en charge SEH, nous vous recommandons d'utiliser la gestion des exceptions C++ standard ISO dans le code C++. Cela rend votre code plus portable et plus flexible. Toutefois, pour conserver le code existant ou pour des types particuliers de programmes, vous devrez peut-être toujours utiliser SEH.

Spécifique à Microsoft :

Grammaire

> *try-except-statement* :
>
> **__try** *compound-statement* **__except** **(** *filter-expression* **)** *compound-statement*
>
> *try-finally-statement* :
>
> **__try** *compound-statement* **__finally** *compound-statement*

Notes

Avec SEH, vous pouvez vous assurer que les ressources, telles que les blocs de mémoire et les fichiers, sont correctement libérées en cas d'arrêt inattendu de l'exécution. Vous pouvez également gérer des problèmes spécifiques, par exemple une mémoire insuffisante, en utilisant du code structuré concis qui ne repose pas sur `goto` des instructions ou des tests complexes des codes de retour.

Les try-except instructions et try-finally mentionnées dans cet article sont des extensions Microsoft pour les langages C et C++. Elles prennent en charge SEH en permettant aux applications de prendre le contrôle d'un programme après des événements qui termineraient sinon son exécution. Bien que la gestion SEH fonctionne avec des fichiers sources C++, elle n'est pas spécifiquement conçue pour C++. Si vous utilisez SEH dans un programme C++ que vous compilez à l'aide de l'option /EHa ou /EHsc , des destructeurs pour les objets locaux sont appelés, mais un autre comportement d'exécution peut ne pas correspondre à ce que vous attendez. Pour obtenir une illustration, consultez l'exemple plus loin dans cet article. Dans la plupart des cas, au lieu de SEH, nous vous recommandons d'utiliser la gestion des exceptions C++ standard

ISO. En utilisant la gestion des exceptions C++, vous pouvez garantir que votre code est plus portable et gérer les exceptions de tout type.

Si vous avez du code C qui utilise SEH, vous pouvez le combiner avec du code C++ qui utilise la gestion des exceptions C++. Pour plus d'informations, consultez Gérer les exceptions structurées en C++.

Il existe deux mécanismes de gestion SEH :

- Des gestionnaires d'exceptions, ou `__except` des blocs, qui peuvent répondre à l'exception ou l'ignorer en fonction de la *`filter-expression`* valeur. Pour plus d'informations, consultez try-except instruction.
- Gestionnaires d'arrêt, ou `__finally` blocs, qui sont toujours appelés, qu'une exception provoque l'arrêt ou non. Pour plus d'informations, consultez try-finally instruction.

Ces deux types de gestionnaires sont distincts, mais sont étroitement liés par le biais d'un processus appelé déroulement *de la pile*. Lorsqu'une exception structurée se produit, Windows recherche le gestionnaire d'exceptions le plus récemment installé qui est actuellement actif. Le gestionnaire peut effectuer l'une des trois opérations suivantes :

- Impossible de reconnaître l'exception et de passer le contrôle à d'autres gestionnaires (`EXCEPTION_CONTINUE_SEARCH`).
- Reconnaissez l'exception, mais ignorez-la (`EXCEPTION_CONTINUE_EXECUTION`).
- Reconnaître l'exception et la gérer (`EXCEPTION_EXECUTE_HANDLER`).

Le gestionnaire d'exceptions qui reconnaît l'exception peut ne pas être dans la fonction qui s'exécutait quand l'exception s'est produite. Il peut se trouver dans une fonction beaucoup plus élevée sur la pile. La fonction en cours d'exécution et toutes les autres fonctions sur le frame de pile sont terminées. Au cours de ce processus, la pile est *dissociée*. Autrement dit, les variables non statiques locales des fonctions terminées sont effacées de la pile.

Tout en déroulant la pile, le système d'exploitation appelle tous les gestionnaires de terminaisons que vous avez écrits pour chaque fonction. En utilisant un gestionnaire d'arrêt, vous propre des ressources qui, sinon, resteraient ouvertes en raison d'un arrêt anormal. Si vous avez entré une section critique, vous pouvez la quitter dans le gestionnaire d'arrêt. Lorsque le programme va s'arrêter, vous pouvez effectuer d'autres tâches de nettoyage, telles que la fermeture et la suppression de fichiers temporaires.

Étapes suivantes

- Écriture d'un gestionnaire d'exceptions
- Écriture d'un gestionnaire des arrêts
- Gérer les exceptions structurées en C++

Exemple

Comme indiqué précédemment, les destructeurs pour les objets locaux sont appelés si vous utilisez SEH dans un programme C++ et que vous le compilez à l'aide de l'option `/EHa` ou `/EHsc` . Toutefois, le comportement pendant l'exécution peut ne pas être celui que vous attendez si vous utilisez également des exceptions C++. Cet exemple illustre ces différences de comportement.

C++

```
#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}
```

```
int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\n");
    }

    return 0;
}
```

Si vous utilisez `/EHsc` pour compiler ce code mais que la macro `CPPEX` de contrôle de test local n'est pas définie, le `TestClass` destructeur ne s'exécute pas. Une sortie classique ressemble à ceci :

Output

```
Triggering SEH exception
Executing SEH __except block
```

Si vous utilisez `/EHsc` pour compiler le code et `CPPEX` est défini à l'aide `/DCPPEX` de (afin qu'une exception C++ soit levée), le `TestClass` destructeur s'exécute et la sortie ressemble à ceci :

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Si vous utilisez `/EHa` pour compiler le code, le `TestClass` destructeur s'exécute si une exception a été levée à l'aide d'une expression C++ `throw` standard ou à l'aide de SEH. Autrement dit, si `CPPEX` est défini ou non. Une sortie classique ressemble à ceci :

Output

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Pour plus d'informations, consultez /EH (Modèle de gestion des exceptions).

FIN de la section spécifique à Microsoft

Voir aussi

Gestion des exceptions

Mots clés

<exception>

Gestion des erreurs et des exceptions

Gestion structurée des exceptions (Windows)

Écriture d'un gestionnaire d'exceptions

Article • 03/04/2023

Les gestionnaires d'exceptions sont généralement utilisés pour répondre à des erreurs spécifiques. Vous pouvez utiliser la syntaxe de gestion des exceptions pour filtrer les exceptions autres que celles que vous savez gérer. D'autres exceptions doivent être passées à d'autres gestionnaires (éventuellement dans la bibliothèque Runtime ou dans le système d'exploitation) écrits pour rechercher ces exceptions spécifiques.

Les gestionnaires d'exceptions utilisent l'instruction try-except.

Sur quels éléments souhaitez-vous obtenir des informations supplémentaires ?

- Instruction try-except
- Écriture d'un filtre d'exception
- Déclenchement d'exceptions logicielles
- Exceptions matérielles
- Restrictions sur les gestionnaires d'exceptions

Voir aussi

Structured Exception Handling (C/C++)

Instruction `try-except`

Article • 16/06/2023

L'instruction `try-except` est une extension **spécifique à Microsoft** qui prend en charge la gestion structurée des exceptions dans les langages C et C++.

C++

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

Grammaire

`try-except-statement`:

`__try` *`compound-statement`* **`__except (`** *`expression`* **`)`** *`compound-statement`*

Notes

L'instruction `try-except` est une extension Microsoft des langages C et C++. Il permet aux applications cibles de prendre le contrôle lorsque des événements se produisent qui mettent normalement fin à l'exécution du programme. Ces événements sont appelés *exceptions structurées*, ou *exceptions* en abrégé. Le mécanisme qui traite ces exceptions est appelé *gestion structurée des exceptions* (SEH).

Pour plus d'informations, consultez l'instruction try-finally.

Les exceptions peuvent être matérielles ou logicielles. La gestion structurée des exceptions est utile même lorsque les applications ne peuvent pas récupérer complètement à partir d'exceptions matérielles ou logicielles. SEH permet d'afficher des informations d'erreur et d'intercepter l'état interne de l'application pour faciliter le diagnostic du problème. Il est particulièrement utile pour les problèmes intermittents qui ne sont pas faciles à reproduire.

ⓘ Notes

La gestion structurée des exceptions fonctionne avec Win32 pour les fichiers sources C et C++. Toutefois, il n'est pas spécialement conçu pour C++. Vous pouvez vous assurer que votre code est plus portable en utilisant la gestion des exceptions C++. En outre, la gestion des exceptions C++ est plus souple, car elle permet de traiter des exceptions de tout type. Pour les programmes C++, nous vous recommandons d'utiliser les instructions natives de gestion des exceptions C++ : **try, catch et throw** .

L'instruction composée après la `__try` clause est le *corps* ou la section *protégée* . L'expression `__except` est également appelée expression de *filtre* . Sa valeur détermine la façon dont l'exception est gérée. L'instruction composée après la clause `__except` constitue le gestionnaire d'exceptions. Le gestionnaire spécifie les actions à effectuer si une exception est levée lors de l'exécution de la section body. L'exécution se déroule comme suit :

1. La section protégée est exécutée.
2. Si aucune exception ne se produit pendant l'exécution de la section protégée, l'exécution se poursuit à l'instruction située après la clause `__except`.
3. Si une exception se produit pendant l'exécution de la section protégée, ou dans une routine que la section protégée appelle, l'expression `__except` est évaluée. Il existe trois valeurs possibles :
    - `EXCEPTION_CONTINUE_EXECUTION` (-1) L'exception est ignorée. Poursuivre l'exécution au point où l'exception s'est produite.
    - `EXCEPTION_CONTINUE_SEARCH` (0) L'exception n'est pas reconnue. Poursuivre la recherche d'un gestionnaire dans la pile, en premier pour qu'il contienne des instructions `try-except`, puis pour les gestionnaires avec la priorité la plus élevée suivante.
    - `EXCEPTION_EXECUTE_HANDLER` (1) L'exception est reconnue. Transférez le contrôle au gestionnaire d'exceptions en exécutant l'instruction `__except` composée, puis poursuivez l'exécution après le `__except` bloc.

L'expression `__except` est évaluée en tant qu'expression C. Elle est limitée à une valeur unique, à l'opérateur d'expression conditionnelle ou à l'opérateur virgule. Si un traitement plus étendu est requis, l'expression peut appeler une routine qui retourne l'une des trois valeurs répertoriées ci-dessus.

Chaque application peut avoir son propre gestionnaire d'exceptions.

Il n'est pas valide d'accéder à une `__try` instruction, mais de sauter en dehors d'une instruction. Le gestionnaire d'exceptions n'est pas appelé si un processus est terminé au milieu de l'exécution d'une `try-except` instruction.

Pour la compatibilité avec les versions précédentes, **_try**, **_except** et **_leave** sont des synonymes de `__try`, `__except` et `__leave` sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Le mot clé `__leave`

La `__leave` mot clé est valide uniquement dans la section protégée d'une `try-except` instruction, et son effet est de passer à la fin de la section protégée. L'exécution se poursuit à la première instruction située après le gestionnaire d'exceptions.

Une `goto` instruction peut également sortir de la section protégée et ne dégrade pas les performances comme dans une instruction **try-finally** . Cela est dû au fait que le déroulement de la pile ne se produit pas. Toutefois, nous vous recommandons d'utiliser le `__leave` mot clé plutôt qu'une `goto` instruction. La raison en est que vous êtes moins susceptible de faire une erreur de programmation si la section protégée est volumineuse ou complexe.

Fonctions intrinsèques de gestion des exceptions structurées

La gestion structurée des exceptions fournit deux fonctions intrinsèques qui peuvent être utilisées avec l'instruction `try-except` : GetExceptionCode et GetExceptionInformation.

`GetExceptionCode` retourne le code (entier 32 bits) de l'exception.

La fonction `GetExceptionInformation` intrinsèque retourne un pointeur vers une structure EXCEPTION_POINTERS contenant des informations supplémentaires sur l'exception. Ce pointeur vous permet d'accéder à l'état de l'ordinateur qui existait au moment d'une exception matérielle. La structure est comme suit :

C++

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Les types de `PEXCEPTION_RECORD` pointeur et `PCONTEXT` sont définis dans le fichier <include winnt.h>, et `_EXCEPTION_RECORD` et `_CONTEXT` sont définis dans le fichier <include excpt.h>

Vous pouvez utiliser `GetExceptionCode` dans le gestionnaire d'exceptions. Toutefois, vous pouvez utiliser `GetExceptionInformation` uniquement dans l'expression de filtre d'exception. Les informations vers laquelle il pointe se situent généralement sur la pile et ne sont plus disponibles lorsque le contrôle est transféré au gestionnaire d'exceptions.

La fonction intrinsèque AbnormalTermination est disponible dans un gestionnaire de terminaison. Elle retourne 0 si le corps de l'instruction **try-finally** se termine séquentiellement. Dans tous les autres cas, elle retourne 1.

<excpt.h> définit d'autres noms pour ces intrinsèques :

`GetExceptionCode` équivaut à `_exception_code`

`GetExceptionInformation` équivaut à `_exception_info`

`AbnormalTermination` équivaut à `_abnormal_termination`

Exemple

C++

```
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}

int main()
```

```
{
    int* p = 0x00000000;   // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13;    // causes an access violation exception;
        }
        __finally
        {
            puts("in finally. termination: ");
            puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
        }
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        puts("in except");
    }
    puts("world");
}
```

Sortie

Output

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
        abnormal
in except
world
```

Voir aussi

Écriture d'un gestionnaire d'exceptions
Structured Exception Handling (C/C++)
Mots clés

Écriture d’un filtre d’exception

Article • 03/04/2023

Vous pouvez gérer une exception en accédant au niveau du gestionnaire d'exceptions ou en reprenant l'exécution. Au lieu d’utiliser le code du gestionnaire d’exceptions pour gérer l’exception et tomber, vous pouvez utiliser une expression *de filtre* pour propre le problème. Ensuite, en retournant `EXCEPTION_CONTINUE_EXECUTION` (-1), vous pouvez reprendre le flux normal sans effacer la pile.

> ⓘ **Notes**
>
> Certaines exceptions ne peuvent pas être continuées. Si *le filtre* prend la valeur -1 pour une telle exception, le système déclenche une nouvelle exception. Lorsque vous appelez **RaiseException**, vous déterminez si l’exception continuera.

Par exemple, le code suivant utilise un appel de fonction dans l’expression *de filtre* : cette fonction gère le problème, puis retourne -1 pour reprendre le flux normal de contrôle :

C++

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
   int Eval_Exception( int );

   __try {}

   __except ( Eval_Exception( GetExceptionCode( ))) {
      ;
   }

}
void ResetVars( int ) {}
int Eval_Exception ( int n_except ) {
   if ( n_except != STATUS_INTEGER_OVERFLOW &&
      n_except != STATUS_FLOAT_OVERFLOW )   // Pass on most exceptions
   return EXCEPTION_CONTINUE_SEARCH;

   // Execute some code to clean up problem
   ResetVars( 0 );   // initializes data to 0
   return EXCEPTION_CONTINUE_EXECUTION;
}
```

Il est judicieux d'utiliser un appel de fonction dans l'expression *de filtre* chaque fois que *le filtre* doit effectuer quelque chose de complexe. L'évaluation de l'expression provoque l'exécution de la fonction, dans ce cas, `Eval_Exception`.

Notez l'utilisation de GetExceptionCode pour déterminer l'exception. Cette fonction doit être appelée à l'intérieur de l'expression de filtre de l'instruction **`__except`** . `Eval_Exception` ne peut pas appeler `GetExceptionCode`, mais le code d'exception doit lui être transmis.

Ce gestionnaire passe le contrôle à un autre gestionnaire sauf si l'exception est un dépassement d'entier ou de virgule flottante. Si tel est le cas, le gestionnaire appelle une fonction (`ResetVars` n'est qu'un exemple, pas une fonction API) pour réinitialiser des variables globales. Le **`__except`** bloc d'instructions, qui est vide dans cet exemple, ne peut jamais être exécuté, car `Eval_Exception` ne retourne `EXCEPTION_EXECUTE_HANDLER` jamais (1).

Utilisation d'un appel de fonction est une bonne technique à usage général pour traiter des expressions de filtre complexes. Il y a deux autres fonctionnalités de langage C utiles :

- l'opérateur conditionnel ;
- l'opérateur virgule ;

L'opérateur conditionnel est fréquemment utile ici. Il peut être utilisé pour case activée pour un code de retour spécifique, puis retourner l'une des deux valeurs différentes. Par exemple, le filtre dans le code suivant reconnaît l'exception uniquement si l'exception est `STATUS_INTEGER_OVERFLOW`:

```
C++

__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

Dans ce cas, l'objectif de l'opérateur conditionnel consiste principalement à assurer la clarté, car le code suivant produit les mêmes résultats :

```
C++

__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

L'opérateur conditionnel est plus utile dans les situations où vous pouvez souhaiter que le filtre évalue à -1, `EXCEPTION_CONTINUE_EXECUTION`.

L’opérateur virgule vous permet d’exécuter plusieurs expressions dans l’ordre. Elle retourne ensuite la valeur de la dernière expression. Par exemple, le code suivant enregistre le code d'exception dans une variable puis le teste :

```
C++
```

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

Voir aussi

Écriture d’un gestionnaire d’exceptions
Structured Exception Handling (C/C++)

Déclenchement d’exceptions logicielles

Article • 03/04/2023

Certaines des sources les plus courantes d'erreurs de programme ne sont pas marquées en tant qu'exceptions par le système. Par exemple, si vous essayez d'allouer un bloc de mémoire mais que la mémoire disponible est insuffisante, la fonction runtime ou API ne déclenche pas d'exception mais retourne un code d'erreur.

Toutefois, vous pouvez traiter n’importe quelle condition comme une exception en détectant cette condition dans votre code, puis en la signalant en appelant la fonction RaiseException . Le fait de marquer les erreurs de cette manière vous permet de tirer parti de la gestion structurée des exceptions pour tout type d'erreur d'exécution.

Pour utiliser la gestion structurée des exceptions avec des erreurs :

- Définissez votre propre code d'exception pour l'événement.
- Appelez `RaiseException` lorsque vous détectez un problème.
- Utilisez les filtres de gestion des exceptions pour tester le code d'exception que vous avez défini.

Le <fichier winerror.h> affiche le format des codes d’exception. Pour veiller à ne pas définir un code qui est en conflit avec un code d'exception existant, définissez le troisième bit le plus significatif à 1. Les quatre bits les plus significatifs doivent être définis comme indiqué dans le tableau suivant.

Bits	Paramètre binaire recommandé	Description
31-30	11	Ces deux bits décrivent la status de base du code : 11 = erreur, 00 = réussite, 01 = information, 10 = avertissement.
29	1	Bit client. Affectez-lui la valeur 1 pour les codes définis par l'utilisateur.
28	0	Bit réservé. (Laissez la valeur 0.)

Vous pouvez définir les deux premiers bits à un paramètre autre 11 binaire, bien que le paramètre « erreur » soit appropriée pour la plupart des exceptions. Veillez à définir les bits 29 et 28 comme indiqué dans le tableau précédent.

Le code d’erreur résultant doit donc avoir les quatre bits les plus élevés définis sur E hexadécimal. Par exemple, les définitions suivantes définissent des codes d’exception

qui n'entrent pas en conflit avec les codes d'exception Windows. (Toutefois, vous pouvez être amené à vérifier quels codes sont utilisés par les DLL tierces.)

```
C++

#define STATUS_INSUFFICIENT_MEM       0xE0000001
#define STATUS_FILE_BAD_FORMAT        0xE0000002
```

Après avoir défini un code d'exception, vous pouvez l'utiliser pour lever une exception. Par exemple, le code suivant déclenche l'exception `STATUS_INSUFFICIENT_MEM` en réponse à un problème d'allocation de mémoire :

```
C++

lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0);
```

Si vous souhaitez déclencher simplement une exception, vous pouvez définir les trois derniers paramètres à 0. Les trois derniers paramètres sont utiles pour transmettre des informations supplémentaires et définir un indicateur qui empêche les gestionnaires de poursuivre l'exécution. Pour plus d'informations, consultez la fonction RaiseException dans le Kit de développement logiciel (SDK) Windows.

Dans vos filtres de gestion des exceptions, vous pouvez ensuite tester les codes que vous avez définis. Par exemple :

```
C++

__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
        GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

Voir aussi

Écriture d'un gestionnaire d'exceptions
Gestion structurée des exceptions (C/C++)

Exceptions matérielles

Article • 03/04/2023

La plupart des exceptions standard identifiées par le système d'exploitation sont des exceptions définies par le matériel. Windows identifie quelques exceptions logicielles de bas niveau, mais celles-ci sont généralement gérées de façon optimale par le système d'exploitation.

Windows mappe les erreurs matérielles des différents processeurs aux codes d'exception figurant dans cette section. Dans certains cas, un processeur peut générer uniquement un sous-ensemble de ces exceptions. Windows prétraite les informations sur l'exception et émet le code d'exception approprié.

Les exceptions matérielles identifiées par Windows sont résumées dans le tableau suivant :

Code d'exception	**Raison de l'exception**
STATUS_ACCESS_VIOLATION	Lecture ou écriture à un emplacement mémoire inaccessible.
STATUS_BREAKPOINT	Rencontre d'un point d'arrêt défini par le matériel ; utilisé uniquement par les débogueurs.
STATUS_DATATYPE_MISALIGNMENT	Lecture ou écriture dans les données à une adresse qui n'est pas correctement alignée ; par exemple, des entités 16 bits doivent être alignées sur des limites de 2 octets. (Non applicable aux processeurs Intel 80*x*86.)
STATUS_FLOAT_DIVIDE_BY_ZERO	Division du type à virgule flottante par 0,0.
STATUS_FLOAT_OVERFLOW	Dépassement de l'exposant positif maximal du type à virgule flottante.
STATUS_FLOAT_UNDERFLOW	Dépassement de la grandeur de l'exposant négatif le plus bas du type à virgule flottante.
STATUS_FLOATING_RESEVERED_OPERAND	Utilisation d'un format à virgule flottante réservé (utilisation non valide de format).
STATUS_ILLEGAL_INSTRUCTION	Tentative d'exécution d'un code d'instruction non défini par le processeur.
STATUS_PRIVILEGED_INSTRUCTION	Exécution d'une instruction non autorisée dans le mode actuel de l'ordinateur.

Code d'exception	Raison de l'exception
STATUS_INTEGER_DIVIDE_BY_ZERO	Division d'un type entier par 0.
STATUS_INTEGER_OVERFLOW	Tentative d'une opération qui dépasse la plage de l'entier.
STATUS_SINGLE_STEP	Exécution d'une instruction en mode pas à pas ; utilisé uniquement par les débogueurs.

Un grand nombre des exceptions répertoriées dans le tableau précédent sont destinées à être gérées par les débogueurs, le système d'exploitation ou un autre code de bas niveau. À l'exception des erreurs relatives aux entiers et aux nombres à virgule flottante, votre code ne doit pas gérer ces erreurs. Ainsi, vous devez généralement utiliser le filtre de gestion des exceptions pour ignorer les exceptions (valeur 0). Sinon, vous risquez d'empêcher les mécanismes de niveau inférieur de réagir de manière appropriée. Vous pouvez toutefois prendre les précautions appropriées contre l'effet potentiel de ces erreurs de bas niveau en écrivant des gestionnaires d'arrêt.

Voir aussi

Écriture d'un gestionnaire d'exceptions
Structured Exception Handling (C/C++)

Restrictions sur les gestionnaires d’exceptions

Article • 16/06/2023

La principale limitation à l’utilisation de gestionnaires d’exceptions dans le code est que vous ne pouvez pas utiliser d’instruction `goto` pour accéder à un bloc d’instructions `__try` . À la place, vous devez entrer dans le bloc d'instruction via le flux de contrôle normal. Vous pouvez sortir d’un `__try` bloc d’instructions et imbriquer des gestionnaires d’exceptions comme vous le souhaitez.

Voir aussi

Écriture d’un gestionnaire d’exceptions
Structured Exception Handling (C/C++)

Écriture d'un gestionnaire des terminaisons

Article • 03/04/2023

Contrairement à un gestionnaire d'exceptions, un gestionnaire de terminaisons est toujours exécuté, que le bloc de code protégé soit terminé normalement ou non. L'objectif unique du gestionnaire de terminaisons doit être de garantir que les ressources, telles que la mémoire, les handles et les fichiers, sont fermées correctement indépendamment de la façon dont se termine l'exécution d'une section de code.

Les gestionnaires de terminaisons utilisent l'instruction try-finally.

Sur quels éléments souhaitez-vous obtenir des informations supplémentaires ?

- L'instruction try-finally
- Nettoyage des ressources
- Minutage des actions dans la gestion des exceptions
- Restrictions sur les gestionnaires d’arrêts

Voir aussi

Structured Exception Handling (C/C++)

Instruction `try-finally`

Article • 11/04/2023

L'instruction `try-finally` est une extension **spécifique à Microsoft** qui prend en charge la gestion structurée des exceptions dans les langages C et C++.

Syntaxe

La syntaxe suivante décrit l'instruction `try-finally` :

C++

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

Grammaire

> *try-finally-statement*:
>
> **__try** *compound-statement* **__finally** *compound-statement*

L'instruction `try-finally` est une extension Microsoft des langages C et C++ qui permettent aux applications cibles de garantir l'exécution du code de nettoyage lorsque l'exécution d'un bloc de code est interrompue. Le nettoyage se compose de tâches telles que la désallocation de mémoire, la fermeture de fichiers et la libération des handles de fichiers. L'instruction `try-finally` est particulièrement utile pour les routines qui ont plusieurs endroits où un contrôle est effectué pour une erreur qui peut provoquer un retour prématuré de la routine.

Pour obtenir des informations connexes et un exemple de code, consultez try-except Instruction. Pour plus d'informations sur la gestion structurée des exceptions en général, consultez Gestion structurée des exceptions. Pour plus d'informations sur la gestion des exceptions dans les applications managées avec C++/CLI, consultez Gestion des exceptions sous /clr.

> ⓘ **Notes**
>
> La gestion structurée des exceptions fonctionne avec Win32 pour les fichiers sources C et C++. Toutefois, elle n'est pas conçue spécifiquement pour C++. Vous pouvez vous assurer que votre code est plus portable en utilisant la gestion des exceptions C++. En outre, la gestion des exceptions C++ est plus souple, car elle permet de traiter des exceptions de tout type. Pour les programmes C++, il est recommandé d'utiliser le mécanisme de gestion des exceptions C++ (**tryinstructions , catchet throw**).

L'instruction composée après la clause `__try` est la section protégée. L'instruction composée après la clause `__finally` est le gestionnaire de terminaisons. Le gestionnaire spécifie un ensemble d'actions qui s'exécutent lorsque la section protégée est quittée, qu'elle quitte la section protégée par une exception (arrêt anormal) ou par un automne standard (arrêt normal).

Le contrôle atteint une instruction `__try` par exécution séquentielle simple (passage). Lorsque le contrôle entre dans , `__try`son gestionnaire associé devient actif. Si le flux de contrôle atteint la fin du bloc try, l'exécution se produit de la façon suivante :

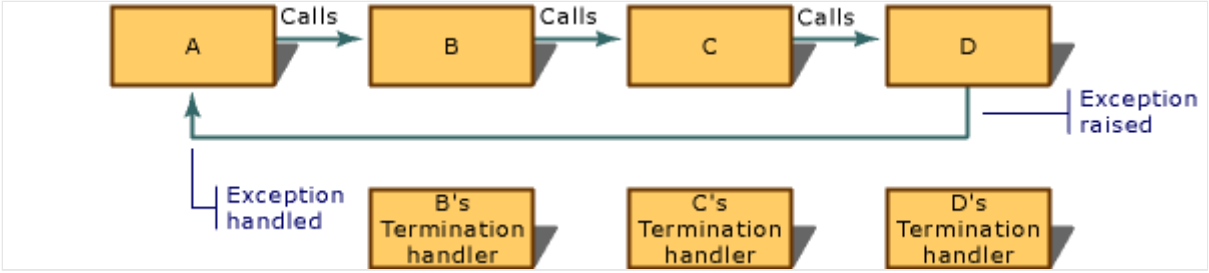
1. Le gestionnaire de terminaisons est appelé.

2. Lorsque le gestionnaire de terminaisons se termine, l'exécution reprend après l'instruction `__finally`. Toutefois, la section protégée se termine (par exemple, par le biais d'un `goto` corps ou d'une `return` instruction hors du corps protégé), le gestionnaire de terminaison est exécuté *avant* que le flux de contrôle ne quitte la section protégée.

   Une `__finally` instruction ne bloque pas la recherche d'un gestionnaire d'exceptions approprié.

Si une exception se produit dans le `__try` bloc, le système d'exploitation doit trouver un gestionnaire pour l'exception, sinon le programme échoue. Si un gestionnaire est trouvé, tous les `__finally` blocs sont exécutés et l'exécution reprend dans le gestionnaire.

Par exemple, supposons qu'une série d'appels de fonction lie la fonction A à la fonction D, comme indiqué dans l'illustration suivante. Chaque fonction a un gestionnaire de terminaisons. Si une exception est levée dans la fonction D et gérée dans A, les gestionnaires de terminaisons sont appelés dans l'ordre suivant à mesure que le système déroule la pile : D, C, B.



Ordre d'exécution du gestionnaire d'arrêt

> ⓘ Notes
>
> Le comportement de try-finally est différent de certains autres langages qui prennent en charge l'utilisation de `finally`, comme C#. Un seul `__try` peut avoir l'un ou l'autre, mais pas les deux, de `__finally` et `__except`. Si les deux doivent être utilisés conjointement, une instruction try-except externe doit entourer l'instruction try-finally interne. Les règles qui spécifient le moment d'exécution de chaque blocs sont également différentes.

Pour la compatibilité avec les versions précédentes, , et sont synonymes de `__try`, `__finally` et `__leave`, sauf si l'option /Za du compilateur (Désactiver les extensions de langage) est spécifiée. `_leave` `_finally` `_try`

Mot clé __leave

La `__leave` mot clé est valide uniquement dans la section protégée d'une `try-finally` instruction, et son effet est de passer à la fin de la section protégée. L'exécution continue à la première instruction dans le gestionnaire de terminaisons.

Une `goto` instruction peut également sortir de la section protégée, mais elle dégrade les performances, car elle appelle le déroulement de la pile. L'instruction `__leave` est plus efficace, car elle ne provoque pas le déroulement de la pile.

Arrêt anormal

La sortie d'une `try-finally` instruction à l'aide de la fonction d'exécution longjmp est considérée comme une interruption anormale. Il n'est pas conforme de sauter dans une instruction `__try`, mais il est conforme d'en sortir. Toutes les `__finally` instructions actives entre le point de départ (arrêt normal du `__try` bloc) et la destination (le `__except` bloc qui gère l'exception) doivent être exécutées. Cela s'appelle un *déroulement local*.

Si un `__try` bloc est arrêté prématurément pour une raison quelconque, y compris un saut hors du bloc, le système exécute le bloc associé dans `__finally` le cadre du processus de déroulement de la pile. Dans ce cas, la AbnormalTermination fonction retourne si elle est `true` appelée à partir du `__finally` bloc ; sinon, elle retourne `false`.

Le gestionnaire de terminaison n'est pas appelé si un processus est tué au milieu de l'exécution d'une `try-finally` instruction.

FIN de la section spécifique à Microsoft

Voir aussi

Écriture d'un gestionnaire des arrêts
Structured Exception Handling (C/C++)
Mots clés
Syntaxe du gestionnaire de terminaison

Nettoyage des ressources

Article • 16/06/2023

Pendant l'exécution du gestionnaire d'arrêt, vous ne savez peut-être pas quelles ressources ont été acquises avant l'appel du gestionnaire d'arrêt. Il est possible que le `__try` bloc d'instruction ait été interrompu avant l'acquisition de toutes les ressources, de sorte que toutes les ressources n'ont pas été ouvertes.

Pour être sûr, vous devez case activée pour voir quelles ressources sont ouvertes avant de procéder au nettoyage de la gestion de l'arrêt. Voici une procédure recommandée :

1. Initialiser les handles sur NULL.
2. Dans le bloc d'instructions `__try` , acquérir des ressources. Les descripteurs sont définis sur des valeurs positives à mesure que la ressource est acquise.
3. Dans le `__finally` bloc d'instructions, libérez chaque ressource dont le handle ou la variable d'indicateur correspondant est différent de zéro ou non NULL.

Exemple

Par exemple, le code suivant utilise un gestionnaire d'arrêt pour fermer trois fichiers et libérer un bloc de mémoire. Ces ressources ont été acquises dans le bloc d'instructions `__try` . Avant de nettoyer une ressource, le code vérifie d'abord si la ressource a été acquise.

C++

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
   FILE  *fp1 = NULL,
         *fp2 = NULL,
         *fp3 = NULL;
   LPVOID lpvoid = NULL;
   errno_t err;

   __try {
      lpvoid = malloc( BUFSIZ );

      err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
```

```
      err = fopen_s(&fp2, "NAMES.DAT", "w+" );
      err = fopen_s(&fp3, "CARS.DAT", "w+" );
   }
   __finally {
      if ( fp1 )
         fclose( fp1 );
      if ( fp2 )
         fclose( fp2 );
      if ( fp3 )
         fclose( fp3 );
      if ( lpvoid )
         free( lpvoid );
   }
}

int main() {
   fileOps();
}
```

Voir aussi

Écriture d'un gestionnaire des arrêts

Structured Exception Handling (C/C++)

Synchronisation de la gestion des exceptions : Un résumé

Article • 03/04/2023

Un gestionnaire d'arrêt est exécuté quelle que soit la façon dont le `__try` bloc d'instruction est arrêté. Les causes incluent le saut en dehors du `__try` bloc, une `longjmp` instruction qui transfère le contrôle hors du bloc et le déroulement de la pile en raison de la gestion des exceptions.

> ⓘ **Notes**
>
> Le compilateur Microsoft C++ prend en charge deux formes d'instructions `setjmp` et `longjmp` . La version rapide ignore la gestion du bloc de fin mais est plus efficace. Pour utiliser cette version, incluez le fichier <setjmp.h>. L'autre version prend en charge la gestion du bloc de fin, comme il est décrit dans le paragraphe précédent. Pour utiliser cette version, incluez le fichier <setjmpex.h>. L'augmentation des performances de la version rapide dépend de la configuration matérielle.

Le système d'exploitation exécute tous les gestionnaires de terminaisons dans l'ordre approprié avant que tout autre code soit exécuté, notamment le corps d'un gestionnaire d'exceptions.

Lorsque la cause de l'interruption est une exception, le système doit d'abord exécuter la partie de filtre d'un ou plusieurs gestionnaires d'exceptions avant de décider ce qui doit être terminé. L'ordre des événements est le suivant :

1. une exception est levée.
2. Le système examine la hiérarchie des gestionnaires d'exceptions actifs et exécute le filtre du gestionnaire avec la priorité la plus élevée. Il s'agit du gestionnaire d'exceptions le plus récemment installé et le plus profondément imbriqué, en passant par les blocs et les appels de fonction.
3. Si ce filtre réussit le contrôle (retourne 0), le processus se poursuit jusqu'à ce qu'un filtre qui ne passe pas le contrôle soit trouvé.
4. Si ce filtre retourne -1, l'exécution continue là où l'exception a été levée et aucune résiliation n'a lieu.
5. Si le filtre retourne 1, les événements suivants se produisent :

- Le système déroule la pile : il efface toutes les trames de pile entre l'emplacement où l'exception a été levée et le cadre de pile qui contient le gestionnaire d'exceptions.
- Au fur et à mesure que la pile est déroulée, chaque gestionnaire de terminaisons figurant sur la pile est exécuté.
- Le gestionnaire d'exceptions lui-même est exécuté.
- La ligne de code prend la main après la fin de ce gestionnaire d'exceptions.

Voir aussi

Écriture d'un gestionnaire des arrêts
Structured Exception Handling (C/C++)

Restrictions sur les gestionnaires de terminaison

Article • 03/04/2023

Vous ne pouvez pas utiliser une `goto` instruction pour accéder à un bloc d'instructions `__try` ou à un bloc d'instructions `__finally` . À la place, vous devez entrer dans le bloc d'instruction via le flux de contrôle normal. (Vous pouvez toutefois sortir d'un bloc d'instruction `__try` .) En outre, vous ne pouvez pas imbriquer un gestionnaire d'exceptions ou un gestionnaire de terminaison à l'intérieur d'un `__finally` bloc.

Certains types de code autorisés dans un gestionnaire d'arrêt produisent des résultats douteux. Vous devez donc les utiliser avec précaution, le cas échéant. L'une est une `goto` instruction qui sort d'un bloc d'instructions `__finally` . Si le bloc s'exécute dans le cadre d'une arrêt normal, rien d'inhabituel ne se produit. Mais si le système déroule la pile, ce déroulement s'arrête. Ensuite, la fonction actuelle prend le contrôle comme s'il n'y avait pas d'arrêt anormal.

Une `return` instruction à l'intérieur d'un bloc d'instructions `__finally` présente à peu près la même situation. Le contrôle retourne à l'appelant immédiat de la fonction qui contient le gestionnaire de terminaison. Si le système déroulait la pile, ce processus est arrêté. Ensuite, le programme continue comme si aucune exception n'avait été levée.

Voir aussi

Écriture d'un gestionnaire des arrêts
Structured Exception Handling (C/C++)

Transport des exceptions entre les threads

Article • 02/05/2023

Le compilateur Microsoft C++ (MSVC) prend en charge *le transport d'une exception* d'un thread vers un autre. Le transport des exceptions vous permet d'intercepter une exception dans un thread, puis faire en sorte que l'exception semble levée dans un thread différent. Par exemple, utilisez cette fonction pour écrire une application multithread où le thread principal gère toutes les exceptions levées par les threads secondaires. Le gestion des exceptions est utile principalement aux développeurs qui créent des bibliothèques ou systèmes de programmation parallèle. Pour implémenter des exceptions de transport, MSVC fournit le type exception_ptr et les fonctions current_exception, rethrow_exception et make_exception_ptr .

Syntaxe

C++

```
namespace std
{
   typedef unspecified exception_ptr;
   exception_ptr current_exception();
   void rethrow_exception(exception_ptr p);
   template<class E>
       exception_ptr make_exception_ptr(E e) noexcept;
}
```

Paramètres

`unspecified`

Une classe interne non spécifiée utilisée pour implémenter le type `exception_ptr`.

`p`

Un objet `exception_ptr` qui référence une exception.

`E`

Une classe représentant une exception.

`e`

Une instance de la classe de paramètre `E`.

Valeur de retour

La fonction `current_exception` retourne un objet `exception_ptr` qui référence l'exception actuellement en cours. Si aucune exception n'est en cours, la fonction retourne un `exception_ptr` objet qui n'est associé à aucune exception.

La `make_exception_ptr` fonction retourne un `exception_ptr` objet qui fait référence à l'exception spécifiée par le `e` paramètre .

Notes

Scénario

Imaginez que vous souhaitiez créer une application capable d'évoluer afin de gérer une quantité de travail variable. Pour atteindre cet objectif, vous concevez une application multithread où un thread principal initial crée autant de threads secondaires nécessaires pour effectuer le travail. Les threads secondaires aident le thread principal à gérer les ressources, équilibrer les charges et améliorer le débit. Lorsque vous distribuez du travail, l'application multithread fonctionne mieux qu'une application à un seul thread.

Toutefois, si un thread secondaire lève une exception, il faut que le thread principal la gère. C'est parce que votre application doit gérer des exceptions de façon cohérente et unifiée, indépendamment du nombre de threads secondaires.

Solution

Pour gérer le scénario précédent, la norme C++ prend en charge le transport d'une exception entre les threads. Si un thread secondaire lève une exception, cette exception devient *l'exception actuelle*. Par analogie avec le monde réel, l'exception actuelle est dite *en vol*. L'exception actuelle est en cours de vol entre le moment où elle est levée et le gestionnaire d'exceptions qui l'intercepte retourne.

Le thread secondaire peut intercepter l'exception actuelle dans un **`catch`** bloc, puis appeler la `current_exception` fonction pour stocker l'exception dans un `exception_ptr` objet. L'objet `exception_ptr` doit être disponible pour le thread secondaire et le thread principal. Par exemple, l'objet `exception_ptr` peut être une variable globale dont l'accès est contrôlé par un mutex. Le terme *transport d'une exception* signifie qu'une exception d'un thread peut être convertie en un formulaire accessible par un autre thread.

Ensuite, le thread principal appelle la fonction `rethrow_exception`, qui extrait puis lève l'exception de l'objet `exception_ptr`. Lorsque l'exception est levée, elle devient l'exception actuelle dans le thread principal. Autrement dit, l'exception semble provenir du thread principal.

Enfin, le thread principal peut intercepter l'exception actuelle dans un **`catch`** bloc, puis la traiter ou la lever dans un gestionnaire d'exceptions de niveau supérieur. Sinon, le thread principal peut ignorer l'exception et permettre la fin du processus.

La plupart des applications n'ont pas besoin de transporter les exceptions entre les threads. Toutefois, cette fonctionnalité est utile dans un système informatique parallèle, car le système peut diviser le travail parmi les threads secondaires, les processeurs ou les cœurs. Dans un environnement informatique parallèle, un thread unique dédié peut gérer toutes les exceptions dans les threads secondaires et peut présenter un modèle cohérent de gestion des exceptions pour toute application.

Pour plus d'informations sur la proposition de comité de normalisation C++, recherchez sur Internet le numéro de document N2179, intitulé « Language Support for Transporting Exceptions between Threads ».

Modèles de gestion des exceptions et options de compilateur

Le modèle de gestion des exceptions de votre application détermine s'il peut intercepter et déplacer une exception. Visual C++ prend en charge trois modèles pour gérer les exceptions C++ : gestion des exceptions C++ standard ISO, gestion structurée des exceptions (SEH) et exceptions CLR (Common Language Runtime). Utilisez les options du compilateur et /clr pour spécifier le /EH modèle de gestion des exceptions de votre application.

Seule la combinaison suivante des options du compilateur et des instructions de programmation peut transporter une exception. D'autres combinaisons ne peuvent pas intercepter les exceptions ou intercepter mais ne peuvent pas transporter les exceptions.

- L'option **`/EHa`** du compilateur et l'instruction **`catch`** peuvent transporter les exceptions SEH et C++.
- Les **`/EHa`** options du compilateur , **`/EHs`** et et **`/EHsc`** l'instruction **`catch`** peuvent transporter des exceptions C++.
- L'option **`/clr`** du compilateur et l'instruction **`catch`** peuvent transporter des exceptions C++. L'option **`/clr`** du compilateur implique la spécification de l'option

`/EHa` . Le compilateur ne prend pas en charge le transport des exceptions managées. En effet, les exceptions managées, qui sont dérivées de la classe System.Exception, sont déjà des objets que vous pouvez déplacer d'un thread à l'autre à l'aide des fonctionnalités du Common Language Runtime.

> **ⓘ Important**
>
> Nous vous recommandons de spécifier l'option du `/EHsc` compilateur et d'intercepter uniquement les exceptions C++. Vous vous exposez à une menace de sécurité si vous utilisez l'option du `/EHa` compilateur ou `/clr` et une `catch` instruction avec une *déclaration d'exception* de points de suspension (`catch(...)`). Vous avez probablement l'intention d'utiliser l'instruction `catch` pour capturer quelques exceptions spécifiques. Toutefois, l'instruction `catch(...)` capture toutes les exceptions C++ et SEH, notamment les exceptions inattendues qui devraient s'avérer irrécupérables. Si vous ignorez ou traitez mal une exception inattendue, le code malveillant peut utiliser cette possibilité pour fragiliser la sécurité de votre programme.

Usage

Les sections suivantes décrivent comment transporter des exceptions à l'aide du `exception_ptr` type et des `current_exception` fonctions , `rethrow_exception` et `make_exception_ptr` .

exception_ptr type

Utilisez un objet `exception_ptr` pour référencer l'exception actuelle ou une instance d'une exception spécifiée par l'utilisateur. Dans l'implémentation Microsoft, une exception est représentée par une EXCEPTION_RECORD structure. Chaque objet `exception_ptr` inclut un champ de référence d'exception qui pointe vers une copie de la structure `EXCEPTION_RECORD` qui représente l'exception.

Lorsque vous déclarez une `exception_ptr` variable, la variable n'est associée à aucune exception. Autrement dit, son champ de référence d'exception est NULL. Ce type d'objet `exception_ptr` est appelé *exception_ptr null*.

Utilisez la fonction `current_exception` ou `make_exception_ptr` pour assigner une exception à un objet `exception_ptr`. Lorsque vous assignez une exception à une variable `exception_ptr`, le champ de référence de l'exception de la variable pointe vers une

copie de l'exception. Si la mémoire est insuffisante pour copier l'exception, le champ de référence d'exception pointe vers une copie d'une exception std::bad_alloc. Si la `current_exception` fonction ou `make_exception_ptr` ne peut pas copier l'exception pour une autre raison, la fonction appelle la fonction d'arrêt pour quitter le processus en cours.

Malgré son nom, un `exception_ptr` objet n'est pas lui-même un pointeur. Il n'obéit pas à la sémantique du pointeur et ne peut pas être utilisé avec les opérateurs d'accès aux membres du pointeur (`->`) ou d'indirection (`*`). L'objet `exception_ptr` n'a aucune donnée membre ou fonction membre publique.

Comparaisons

Vous pouvez utiliser les opérateurs égal (`==`) et non égal (`!=`) pour comparer deux objets `exception_ptr`. Les opérateurs ne comparent pas la valeur binaire (modèle de bits) des `EXCEPTION_RECORD` structures qui représentent les exceptions. À la place, les opérateurs comparent les adresses dans le domaine de référence d'exception des objets `exception_ptr`. Par conséquent, une valeur null `exception_ptr` et la valeur NULL sont égales.

current_exception fonction

Appelez la `current_exception` fonction dans un **`catch`** bloc. Si une exception est en cours de vol et que le **`catch`** bloc peut intercepter l'exception, la `current_exception` fonction retourne un `exception_ptr` objet qui fait référence à l'exception. Sinon, la fonction retourne un objet `exception_ptr` null.

Détails

La fonction `current_exception` capture l'exception en vol même si l'instruction **`catch`** spécifie une instruction exception-declaration.

Le destructeur de l'exception actuelle est appelé à la fin du **`catch`** bloc si vous ne ressinez pas l'exception. Toutefois, même si vous appelez la fonction `current_exception` dans le destructeur, celle-ci retourne un objet `exception_ptr` qui référence l'exception actuelle.

Les appels successifs à la fonction `current_exception` retournent des objets `exception_ptr` qui font référence à des copies de l'exception actuelle. Ainsi, les objets se

comparent comme étant inégaux, car ils font référence à des copies différentes, même si les copies ont la même valeur binaire.

Exceptions SEH

Si vous utilisez l'option du `/EHa` compilateur, vous pouvez intercepter une exception SEH dans un bloc C++ `catch` . La fonction `current_exception` retourne un objet `exception_ptr` qui référence l'exception SEH. Et la `rethrow_exception` fonction lève l'exception SEH si vous l'appelez avec l'objet transporté `exception_ptr` comme argument.

La `current_exception` fonction retourne une valeur null `exception_ptr` si vous l'appelez dans un gestionnaire de terminaison SEH `__finally` , un gestionnaire d'exceptions `__except` ou l'expression de `__except` filtre.

Une exception transportée ne prend pas en charge les exceptions imbriquées. Une exception imbriquée se produit lorsqu'une autre exception est levée pendant la gestion d'une exception. Si vous interceptez une exception imbriquée, les données membres `EXCEPTION_RECORD.ExceptionRecord` pointent vers une chaîne de structures `EXCEPTION_RECORD` qui décrivent les exceptions associées. La `current_exception` fonction ne prend pas en charge les exceptions imbriquées, car elle retourne un `exception_ptr` objet dont `ExceptionRecord` le membre de données est zéro.

Si vous interceptez une exception SEH, vous devez gérer la mémoire référencée par tout pointeur dans le tableau de données membres `EXCEPTION_RECORD.ExceptionInformation`. Vous devez vous assurer que la mémoire est valide pendant la durée de vie de l'objet `exception_ptr` correspondant, et que cette mémoire est libérée lorsque l'objet `exception_ptr` est supprimé.

Vous pouvez utiliser les fonctions de traduction d'exceptions structurées avec la fonctionnalité de transport d'exceptions. Si une exception SEH est traduite en une exception C++, la fonction `current_exception` retourne `exception_ptr` qui référence l'exception traduite au lieu de l'exception SEH originale. La `rethrow_exception` fonction lève l'exception traduite, et non l'exception d'origine. Pour plus d'informations sur les fonctions de traduction se, consultez _set_se_translator.

rethrow_exception fonction

Après avoir enregistré une exception interceptée dans un objet `exception_ptr`, le thread principal peut traiter l'objet. Dans votre thread principal, appelez la fonction

`rethrow_exception` avec l'objet `exception_ptr` en tant qu'argument. La fonction `rethrow_exception` extrait l'exception de l'objet `exception_ptr` puis lève l'exception dans le contexte du thread principal. Si le paramètre *p* de la `rethrow_exception` fonction est null `exception_ptr`, la fonction lève std::bad_exception.

L'exception extraite est maintenant l'exception actuelle dans le thread principal, et vous pouvez la gérer comme n'importe quelle autre exception. Si vous interceptez l'exception, vous pouvez la gérer immédiatement ou utiliser une **`throw`** instruction pour l'envoyer à un gestionnaire d'exceptions de niveau supérieur. Sinon, ne faites rien et laissez le gestionnaire d'exceptions systèmes par défaut terminer le processus.

make_exception_ptr, fonction

La fonction `make_exception_ptr` prend une instance de classe comme argument puis retourne un `exception_ptr` qui référence l'instance. Généralement, vous spécifiez un objet de classe exception comme argument pour la fonction `make_exception_ptr`, bien que tout objet de classe puisse être l'argument.

Appeler la `make_exception_ptr` fonction revient à lever une exception C++, à l'intercepter dans un **`catch`** bloc, puis à appeler la `current_exception` fonction pour retourner un `exception_ptr` objet qui fait référence à l'exception. L'implémentation Microsoft de la fonction `make_exception_ptr` est plus efficace que le fait de lever puis d'intercepter une exception.

Une application n'a généralement pas besoin de la `make_exception_ptr` fonction et nous déconseillons son utilisation.

Exemple

L'exemple suivant transporte une exception C++ standard et une exception C++ personnalisée d'un thread à un autre.

C++

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;
```

```
// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int           aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );

// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};
int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;

    // Create secondary threads.
    for( int i=0; i < THREADCOUNT; i++ )
    {
        aArg[i] = i;
        aThread[i] = CreateThread(
            NULL,       // Default security attributes.
            0,          // Default stack size.
            (LPTHREAD_START_ROUTINE) ThrowExceptions,
            (LPVOID) &aArg[i], // Thread function argument.
            0,          // Default creation flags.
            &ThreadID); // Receives thread identifier.
        if( aThread[i] == NULL )
        {
            printf("CreateThread error: %d\n", GetLastError());
            return -1;
        }
    }

    // Wait for all threads to terminate.
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
    // Close thread handles.
    for( int i=0; i < THREADCOUNT; i++ ) {
        CloseHandle(aThread[i]);
    }

    // Rethrow and catch the transported exceptions.
    for ( int i = 0; i < THREADCOUNT; i++ ) {
        try {
            if (aException[i] == NULL) {
                printf("exception_ptr %d: No exception was transported.\n",
i);
            }
            else {
                rethrow_exception( aException[i] );
            }
        }
        catch( const invalid_argument & ) {
            printf("exception_ptr %d: Caught an invalid_argument
```

```
exception.\n", i);
        }
        catch( const myException & ) {
            printf("exception_ptr %d: Caught a  myException exception.\n",
i);
        }
    }
}
// Each thread throws an exception depending on its thread
// function argument, and then ends.
DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.
            // In practice, your application performs an operation that
            // implicitly throws an exception.
            throw invalid_argument("A C++ exception.");
        }
        catch ( const invalid_argument & ) {
            aException[x] = current_exception();
        }
    }
    else {
        // User-defined exception.
        aException[x] = make_exception_ptr( myException() );
    }
    return TRUE;
}
```

Output

```
exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a  myException exception.
```

Spécifications

En-tête: `<exception>`

Voir aussi

Gestion des exceptions/EH (modèle de gestion des exceptions)

/clr (Compilation Common Language Runtime)

Assertion et messages fournis par l'utilisateur (C++)

Article • 03/04/2023

Le langage C++ prend en charge trois mécanismes de gestion des erreurs qui vous aident à déboguer votre application : la directive #error, la mot clé static_assert et la macro d'assertion, _assert, _wassert macro. Ces trois mécanismes génèrent des messages d'erreur et deux d'entre eux testent également les assertions logicielles. Une assertion logicielle spécifie une condition supposée être vraie en un point particulier de votre programme. Si une assertion au moment de la compilation échoue, le compilateur génère un message de diagnostic et une erreur de compilation. Si une assertion d'exécution échoue, le système d'exploitation génère un message de diagnostic et ferme votre application.

Notes

La durée de vie de votre application se compose d'une phase de prétraitement, de compilation et d'exécution. Chaque mécanisme de gestion des erreurs accède aux informations de débogage disponibles pendant l'une de ces phases. Pour déboguer efficacement, sélectionnez le mécanisme qui fournit des informations pertinentes concernant cette phase :

- La directive #error est en vigueur au moment du prétraitement. Elle génère sans condition un message spécifié par l'utilisateur et provoque l'échec de la compilation avec une erreur. Le message peut contenir du texte manipulé par des directives de préprocesseur, mais aucune expression obtenue n'est évaluée.

- La déclaration static_assert est en vigueur au moment de la compilation. Elle teste une assertion logicielle représentée par une expression intégrale spécifiée par l'utilisateur qui peut être convertie en valeur booléenne. Si l'expression est évaluée à la valeur zéro (false), le compilateur génère le message spécifié par l'utilisateur et la compilation échoue avec une erreur.

  La déclaration `static_assert` est particulièrement utile pour le débogage des modèles, car des arguments template peuvent être inclus dans l'expression spécifiée par l'utilisateur.

- La macro assert, _assert, _wassert est en vigueur au moment de l'exécution. Elle évalue une expression spécifiée par l'utilisateur, et si le résultat est zéro, le système génère un message de diagnostic et ferme votre application. De nombreuses

autres macros, telles que_ASSERT et _ASSERTE, ressemblent à cette macro, mais émettent des messages de diagnostic définis par le système ou définis par l'utilisateur.

Voir aussi

directive #error (C/C++)
assert Macro, _assert, _wassert
macros _ASSERT, _ASSERTE _ASSERT_EXPR
static_assert
_STATIC_ASSERT Macro
Modèles

static_assert

Article • 16/06/2023

Teste une assertion logicielle au moment de la compilation. Si l'expression constante spécifiée est `false`, le compilateur affiche le message spécifié, le cas échéant, et la compilation échoue avec l'erreur C2338 ; sinon, la déclaration n'a aucun effet.

Syntaxe

```
static_assert( constant-expression, string-literal );

static_assert( constant-expression ); // C++17 (Visual Studio 2017 and
later)
```

Paramètres

expression constante
Expression constante intégrale pouvant être convertie en expression booléenne. Si l'expression évaluée est zéro (false), le paramètre *string-literal* s'affiche et la compilation échoue avec une erreur. Si l'expression est différente de zéro (true), la `static_assert` déclaration n'a aucun effet.

string-literal
Message qui s'affiche si le paramètre *d'expression constante* est égal à zéro. Le message est une chaîne de caractères dans le jeu de caractères de base du compilateur ; c'est-à-dire, pas des caractères multioctets ou larges.

Notes

Le paramètre *d'expression constante* d'une `static_assert` déclaration représente une *assertion logicielle*. Une assertion logicielle spécifie une condition supposée être vraie en un point particulier de votre programme. Si la condition est true, la `static_assert` déclaration n'a aucun effet. Si la condition est false, l'assertion échoue, le compilateur affiche le message dans le paramètre *string-literal* et la compilation échoue avec une erreur. Dans Visual Studio 2017 et versions ultérieures, le paramètre string-literal est facultatif.

La `static_assert` déclaration teste une assertion logicielle au moment de la compilation. En revanche, les fonctions Assert Macro et _assert et _wassert testent une assertion logicielle au moment de l'exécution et entraînent un coût d'exécution dans l'espace ou le temps. La `static_assert` déclaration est particulièrement utile pour le débogage des modèles, car les arguments de modèle peuvent être inclus dans le paramètre *d'expression constante* .

Le compilateur examine la `static_assert` déclaration pour rechercher des erreurs de syntaxe lorsque la déclaration est rencontrée. Le compilateur évalue immédiatement le paramètre *d'expression constante* s'il ne dépend pas d'un paramètre de modèle. Sinon, le compilateur évalue le paramètre *d'expression constante* lorsque le modèle est instancié. Par conséquent, le compilateur peut publier un message de diagnostic une fois la déclaration rencontrée, puis à nouveau lorsque le modèle est instancié.

Vous pouvez utiliser le `static_assert` mot clé au niveau de l'espace de noms, de la classe ou de l'étendue de bloc. (le `static_assert` mot clé est techniquement une déclaration, même si elle n'introduit pas de nouveau nom dans votre programme, car elle peut être utilisée dans l'étendue de l'espace de noms.)

Description de avec étendue d'espace de `static_assert` noms

Dans l'exemple suivant, la déclaration a l'étendue de l'espace `static_assert` de noms. Comme le compilateur connaît la taille du type `void *`, l'expression est évaluée immédiatement.

Exemple : `static_assert` avec étendue d'espace de noms

C++

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not
supported.");
```

Description de avec l'étendue de `static_assert` classe

Dans l'exemple suivant, la déclaration a une `static_assert` étendue de classe. vérifie `static_assert` qu'un paramètre de modèle est un type POD (*Plain Old Data*). Le compilateur examine la `static_assert` déclaration lorsqu'elle est déclarée, mais n'évalue pas le paramètre *d'expression constante* tant que le `basic_string` modèle de classe n'est pas instancié dans `main()`.

Exemple : `static_assert` avec étendue de classe

C++

```
#include <type_traits>
#include <iosfwd>
namespace std {
template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
    static_assert(std::is_pod<CharT>::value,
                  "Template argument CharT must be a POD type in class
template basic_string");
    // ...
    };
}

struct NonPOD {
    NonPOD(const NonPOD &) {}
    virtual ~NonPOD() {}
};

int main()
{
    std::basic_string<char> bs;
}
```

Description de avec étendue de `static_assert` bloc

Dans l'exemple suivant, la déclaration a une `static_assert` étendue de bloc. vérifie `static_assert` que la taille de la structure VMPage est égale à la taille de page de la mémoire virtuelle du système.

Exemple : `static_assert` à l'étendue du bloc

C++

```
#include <sys/param.h> // defines PAGESIZE
class VMMClient {
public:
    struct VMPage { // ...
           };
    int check_pagesize() {
    static_assert(sizeof(VMPage) == PAGESIZE,
        "Struct VMPage must be the same size as a system virtual memory
page.");
    // ...
    }
// ...
};
```

Voir aussi

Assertion et messages fournis par l'utilisateur (C++)

directive #error (C/C++)

assert Macro, _assert, _wassert

Modèles

Jeu de caractères ASCII

Déclarations et définitions

Vue d'ensemble des modules dans C++

Article • 24/04/2023

C++20 introduit des *modules*, une solution moderne qui transforme les bibliothèques et les programmes C++ en composants. Un *module* est un ensemble de fichiers de code source compilés indépendamment des fichiers sources (ou plus précisément des unités de traduction qui les importent). Les modules éliminent ou réduisent la plupart des problèmes associés à l'utilisation des fichiers d'en-tête. Ils réduisent souvent les temps de compilation. Les macros, les directives de préprocesseur et les noms non exportés déclarés dans un module ne sont pas visibles en dehors du module. Ils n'ont aucun effet sur la compilation de l'unité de traduction qui importe le module. Vous pouvez importer des modules dans n'importe quel ordre sans vous soucier des redéfinitions de macros. Les déclarations dans l'unité de traduction d'importation ne participent pas à la résolution de surcharge ou à la recherche de noms dans le module importé. Une fois qu'un module est compilé une seule fois, les résultats sont stockés dans un fichier binaire qui décrit tous les types, fonctions et modèles exportés. Le compilateur peut traiter ce fichier beaucoup plus rapidement qu'un fichier d'en-tête. De plus, le compilateur peut le réutiliser à chaque endroit où le module est importé dans un projet.

Vous pouvez utiliser des modules côte à côte avec des fichiers d'en-tête. Un fichier source C++ peut également former `import` des modules et des `#include` fichiers d'en-tête. Dans certains cas, vous pouvez importer un fichier d'en-tête en tant que module, ce qui est plus rapide que d'utiliser `#include` pour le traiter avec le préprocesseur. Nous vous recommandons d'utiliser autant que possible des modules dans les nouveaux projets plutôt que des fichiers d'en-tête. Pour les projets existants de plus grande taille en cours de développement actif, essayez de convertir des en-têtes hérités en modules. Basez votre adoption sur si vous obtenez une réduction significative des temps de compilation.

Pour comparer les modules à d'autres façons d'importer la bibliothèque standard, consultez Comparer les unités d'en-tête, les modules et les en-têtes précompilés.

Activer les modules dans le compilateur Microsoft C++

À compter de Visual Studio 2022 version 17.1, les modules standard C++20 sont entièrement implémentés dans le compilateur Microsoft C++.

Avant sa spécification par la norme C++20, Microsoft disposait d'une prise en charge expérimentale des modules. Le compilateur a également pris en charge l'importation de

modules de bibliothèque standard prédéfinis, décrits ci-dessous.

À compter de Visual Studio 2022 version 17.5, l'importation de la bibliothèque standard en tant que module est standardisée et entièrement implémentée dans le compilateur Microsoft C++. Cette section décrit l'ancienne méthode expérimentale, qui est toujours prise en charge. Pour plus d'informations sur la nouvelle façon standardisée d'importer la bibliothèque standard à l'aide de modules, consultez Importer la bibliothèque standard C++ à l'aide de modules.

Vous pouvez utiliser la fonctionnalité modules pour créer des modules à partition unique et pour importer les modules bibliothèque standard fournis par Microsoft. Pour activer la prise en charge des modules de bibliothèque standard, compilez avec /experimental:module et /std:c++latest. Dans un projet Visual Studio, cliquez avec le bouton droit sur le nœud du projet dans **Explorateur de solutions** et choisissez **Propriétés**. Définissez la liste déroulante **Configuration** sur **Toutes les configurations**, puis choisissez **Propriétés**> de configuration**C/C++**>**Language**>**Enable C++ Modules C++ (expérimental)** .

Un module et le code qui l'utilise doivent être compilés avec les mêmes options de compilateur.

Utiliser la bibliothèque standard C++ en tant que modules (expérimental)

Cette section décrit l'implémentation expérimentale, qui est toujours prise en charge. La nouvelle façon standardisée de consommer la bibliothèque standard C++ en tant que modules est décrite dans Importer la bibliothèque standard C++ à l'aide de modules.

En important la bibliothèque standard C++ en tant que modules plutôt que de l'inclure dans les fichiers d'en-tête, vous pouvez potentiellement accélérer les temps de compilation en fonction de la taille de votre projet. La bibliothèque expérimentale est divisée en modules nommés suivants :

- `std.regex` fournit le contenu de l'en-tête `<regex>`
- `std.filesystem` fournit le contenu de l'en-tête `<filesystem>`
- `std.memory` fournit le contenu de l'en-tête `<memory>`
- `std.threading` fournit le contenu des en-têtes `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>`et `<thread>`
- `std.core` fournit tout le reste de la bibliothèque standard C++

Pour utiliser ces modules, ajoutez une déclaration d'importation en haut du fichier de code source. Par exemple :

C++

```
import std.core;
import std.regex;
```

Pour utiliser les modules Microsoft Standard Library, compilez votre programme avec /EHsc les options et /MD .

Exemple de base

L'exemple suivant montre une définition de module simple dans un fichier source appelé *`Example.ixx`*. L'extension `.ixx` est requise pour les fichiers d'interface de module dans Visual Studio. Dans cet exemple, le fichier d'interface contient à la fois la définition de la fonction et la déclaration. Toutefois, vous pouvez également placer les définitions dans un ou plusieurs fichiers d'implémentation de module distincts, comme illustré dans un exemple ultérieur. L'instruction `export module Example;` indique que ce fichier est l'interface principale d'un module appelé `Example`. Le **`export`** modificateur sur `f()` indique que cette fonction est visible quand `Example` est importé par un autre programme ou module. Le module fait référence à un espace de noms `Example_NS`.

C++

```
// Example.ixx
export module Example;

#define ANSWER 42

namespace Example_NS
{
   int f_internal() {
        return ANSWER;
      }

   export int f() {
      return f_internal();
   }
}
```

Le fichier *`MyProgram.cpp`* utilise la **`import`** déclaration pour accéder au nom exporté par `Example`. Le nom `Example_NS` est visible ici, mais pas tous ses membres. En outre, la macro `ANSWER` n'est pas visible.

C++

```
// MyProgram.cpp
import Example;
import std.core;

using namespace std;

int main()
{
   cout << "The result of f() is " << Example_NS::f() << endl; // 42
   // int i = Example_NS::f_internal(); // C2039
   // int j = ANSWER; //C2065
}
```

La `import` déclaration ne peut apparaître qu'à l'étendue globale.

Grammaire du module

module-name:
  module-name-qualifier-seq opt *identifier*

module-name-qualifier-seq:
  identifier **.**
  module-name-qualifier-seq *identifier* **.**

module-partition:
  : *module-name*

module-declaration:
  export Opter **module** *module-name* *module-partition* Opter *attribute-specifier-seq* Opter **;**

module-import-declaration:
  export Opter **import** *module-name* *attribute-specifier-seq* Opter **;**
  export Opter **import** *module-partition* *attribute-specifier-seq* Opter **;**
  export Opter **import** *header-name* *attribute-specifier-seq* Opter **;**

Implémentation de modules

Une *interface de module* exporte le nom du module et tous les espaces de noms, types, fonctions, etc. qui composent l'interface publique du module. Une *implémentation de module* définit les éléments exportés par le module. Dans sa forme la plus simple, un

module peut se composer d'un fichier unique qui combine l'interface du module et l'implémentation. Vous pouvez également placer les implémentations dans un ou plusieurs fichiers d'implémentation de module distincts, de la même façon que `.h` les fichiers et `.cpp` sont utilisés.

Pour les modules plus volumineux, vous pouvez fractionner des parties du module en sous-modules appelés *partitions*. Chaque partition se compose d'un fichier d'interface de module qui exporte un nom de partition de module. Une partition peut également avoir un ou plusieurs fichiers d'implémentation de partition. Le module dans son ensemble a une *interface de module principale*, l'interface publique du module qui peut également importer et exporter les interfaces de partition.

Un module se compose d'une ou plusieurs *unités de module*. Une unité de module est une unité de traduction (un fichier source) qui contient une déclaration de module. Il existe plusieurs types d'unités de module :

- Une *unité d'interface de module* est une unité de module qui exporte un nom de module ou un nom de partition de module. Une unité d'interface de module a `export module` dans sa déclaration de module.

- Une *unité d'implémentation de module* est une unité de module qui n'exporte pas de nom de module ou de partition de module. Comme son nom l'indique, il est utilisé pour implémenter un module.

- Une *unité d'interface de module principale* est une unité d'interface de module qui exporte le nom du module. Il ne doit y avoir qu'une seule unité d'interface de module principal dans un module.

- Une *unité d'interface de partition de module* est une unité d'interface de module qui exporte un nom de partition de module.

- Une *unité d'implémentation de partition de module* est une unité d'implémentation de module qui a un nom de partition de module dans sa déclaration de module, mais pas `export` de mot clé.

Le **`export`** mot clé est utilisé uniquement dans les fichiers d'interface. Un fichier d'implémentation peut être **`import`** un autre module, mais ne peut pas **`export`** avoir de noms. Les fichiers d'implémentation peuvent avoir n'importe quelle extension.

Modules, espaces de noms et recherche dépendante des arguments

Les règles pour les espaces de noms dans les modules sont les mêmes que dans tout autre code. Si une déclaration au sein d'un espace de noms est exportée, l'espace de noms englobant (à l'exclusion des membres non exportés) est également exporté implicitement. Si un espace de noms est explicitement exporté, toutes les déclarations dans cette définition d'espace de noms sont exportées.

Lorsqu'il effectue une recherche dépendante des arguments pour les résolutions de surcharge dans l'unité de traduction d'importation, le compilateur considère les fonctions déclarées dans la même unité de traduction (y compris les interfaces de module) que l'emplacement où le type des arguments de la fonction est défini.

Partitions de module

Une partition de module est similaire à un module, sauf qu'elle partage la propriété de toutes les déclarations dans l'ensemble du module. Tous les noms exportés par les fichiers d'interface de partition sont importés et réexporté par le fichier d'interface primaire. Le nom d'une partition doit commencer par le nom du module suivi d'un signe deux-points. Les déclarations dans l'une des partitions sont visibles dans l'ensemble du module. Aucune précaution particulière n'est nécessaire pour éviter les erreurs d'une règle de définition (ODR). Vous pouvez déclarer un nom (fonction, classe, etc.) dans une partition et le définir dans une autre. Un fichier d'implémentation de partition commence comme suit :

```
C++

module Example:part1;
```

Le fichier d'interface de partition commence comme suit :

```
C++

export module Example:part1;
```

Pour accéder aux déclarations dans une autre partition, une partition doit l'importer, mais elle ne peut utiliser que le nom de la partition, pas le nom du module :

```
C++

module Example:part2;
import :part1;
```

L'unité d'interface principale doit importer et réexporter tous les fichiers de partition d'interface du module comme suit :

```
C++
```

```
export import :part1;
export import :part2;
...
```

L'unité d'interface principale peut importer des fichiers d'implémentation de partition, mais ne peut pas les exporter. Ces fichiers ne sont pas autorisés à exporter des noms. Cette restriction permet à un module de conserver les détails d'implémentation internes au module.

Modules et fichiers d'en-tête

Vous pouvez inclure des fichiers d'en-tête dans un fichier source de module en plaçant la `#include` directive avant la déclaration de module. Ces fichiers sont considérés comme étant dans le *fragment de module global*. Un module peut uniquement voir les noms dans le fragment de module global qui se trouvent dans les en-têtes qu'il inclut explicitement. Le fragment de module global contient uniquement les symboles utilisés.

```
C++
```

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

Vous pouvez utiliser un fichier d'en-tête traditionnel pour contrôler les modules importés :

```
C++
```

```
// MyProgram.h
import std.core;
#ifdef DEBUG_LOGGING
import std.filesystem;
#endif
```

Fichiers d'en-tête importés

Certains en-têtes sont suffisamment autonomes pour pouvoir être introduits à l'aide de la `import` mot clé. La main différence entre un en-tête importé et un module importé est que toutes les définitions de préprocesseur dans l'en-tête sont visibles dans le programme d'importation immédiatement après l'instruction `import`.

C++

```
import <vector>;
import "myheader.h";
```

Voir aussi

module, import, export
Tutoriel sur les modules nommés
Comparer les unités d'en-tête, les modules et les en-têtes précompilés

`module`, `import`, `export`

Article • 16/06/2023

Les `module` déclarations , `import` et `export` sont disponibles en C++20 et nécessitent le commutateur du /experimental:module compilateur avec /std:c++20 ou une version ultérieure (par `/std:c++latest` exemple). Pour plus d'informations, consultez Vue d'ensemble des modules en C++.

`module`

Placez une `module` déclaration au début d'un fichier d'implémentation de module pour spécifier que le contenu du fichier appartient au module nommé.

C++

```
module ModuleA;
```

`export`

Utilisez une `export module` déclaration pour le fichier d'interface principal du module, qui doit avoir l'extension `.ixx`:

C++

```
export module ModuleA;
```

Dans un fichier d'interface, utilisez le `export` modificateur sur les noms destinés à faire partie de l'interface publique :

C++

```
// ModuleA.ixx

export module ModuleA;

namespace ModuleA_NS
{
   export int f();
   export double d();
   double internal_f(); // not exported
}
```

Les noms non exportés ne sont pas visibles pour le code qui importe le module :

```
C++

//MyProgram.cpp

import ModuleA;

int main() {
  ModuleA_NS::f(); // OK
  ModuleA_NS::d(); // OK
  ModuleA_NS::internal_f(); // Ill-formed: error C2065: 'internal_f':
undeclared identifier
}
```

Le `export` mot clé peut ne pas apparaître dans un fichier d'implémentation de module. Quand `export` est appliqué à un nom d'espace de noms, tous les noms de l'espace de noms sont exportés.

`import`

Utilisez une `import` déclaration pour rendre les noms d'un module visibles dans votre programme. La `import` déclaration doit apparaître après la `module` déclaration et après toutes `#include` les directives, mais avant toutes les déclarations dans le fichier.

```
C++

module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

Notes

Les deux `import` et `module` sont traités comme des mots clés uniquement lorsqu'ils apparaissent au début d'une ligne logique :

```
C++
```

```
// OK:
module ;
module module-name
import :
import <
import "
import module-name
export module ;
export module module-name
export import :
export import <
export import "
export import module-name

// Error:
int i; module ;
```

Section spécifique à Microsoft

Dans Microsoft C++, les jetons et `module` sont toujours des `import` identificateurs et jamais des mots clés lorsqu'ils sont utilisés comme arguments d'une macro.

Exemple

C++

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

Mettre fin à Microsoft Spécifique

Voir aussi

Vue d'ensemble des modules dans C++

Tutoriel : Importer la bibliothèque standard C++ à l'aide de modules à partir de la ligne de commande

Article • 08/06/2023

Découvrez comment importer la bibliothèque standard C++ à l'aide de modules de bibliothèque C++. Cela permet une compilation beaucoup plus rapide et est plus robuste que l'utilisation de fichiers d'en-tête ou d'unités d'en-tête ou d'en-têtes précompilés (PCH).

Dans ce tutoriel, découvrez les points suivants :

- Comment importer la bibliothèque standard en tant que module à partir de la ligne de commande.
- Avantages en matière de performances et de facilité d'utilisation des modules.
- Les deux modules `std` de bibliothèque standard et `std.compat` la différence entre eux.

Prérequis

Ce didacticiel nécessite Visual Studio 2022 17.5 ou version ultérieure.

> ⚠ **Avertissement**
>
> Ce tutoriel concerne une fonctionnalité d'aperçu. La fonctionnalité est susceptible d'être modifiée d'une version préliminaire à l'autre. Vous ne devez pas utiliser les fonctionnalités d'aperçu dans le code de production.

Présentation des modules de bibliothèque standard

Les fichiers d'en-tête souffrent d'une sémantique qui change en fonction des définitions de macros, de l'ordre dans lequel vous les incluez, et ils ralentissent la compilation. Les modules résolvent ces problèmes.

Il est désormais possible d'importer la bibliothèque standard en tant que module au lieu d'un enchevêtrement de fichiers d'en-tête. Cela est beaucoup plus rapide et plus

robuste que l'inclusion de fichiers d'en-tête ou d'unités d'en-tête ou d'en-têtes précompilés (PCH).

La bibliothèque standard C++23 introduit deux modules nommés : `std` et `std.compat`.

- `std` exporte les déclarations et les noms définis dans l'espace de noms `std`de bibliothèque standard C++, tels que `std::vector` et `std::sort`. Il exporte également le contenu des en-têtes de wrapper C tels que `<cstdio>` et `<cstdlib>`, qui fournissent des fonctions telles que `std::printf()`. Les fonctions C définies dans *l'espace de noms global*, telles que `::printf()`, ne sont pas exportées. Cela améliore la situation où l'inclusion d'un en-tête de wrapper C comme `<cstdio>`*également* inclus des fichiers d'en-tête C comme `stdio.h`, qui a apporté les versions de l'espace de noms global C. Ce n'est pas un problème si vous importez `std`.
- `std.compat` exporte tout dans `std` et ajoute les espaces de noms globaux du runtime C tels que `::printf`, `::fopen`, `::size_t`, `::strlen`, et ainsi de suite. Le `std.compat` module facilite l'utilisation de codebases qui font référence à de nombreuses fonctions/types de runtime C dans l'espace de noms global.

Le compilateur importe l'ensemble de la bibliothèque standard lorsque vous utilisez `import std;` ou et `import std.compat;` le fait plus rapidement que d'apporter un seul fichier d'en-tête. Autrement dit, il est plus rapide d'intégrer l'ensemble de la bibliothèque standard avec `import std;` (ou `import std.compat`) que `#include <vector>`dans , par exemple.

Étant donné que les modules nommés n'exposent pas de macros, les macros telles que , , , `va_arg`et d'autres ne sont pas disponibles lorsque vous importez `std` ou `std.compat`. `offsetof` `errno` `assert` Pour plus d'informations sur les solutions de contournement, consultez Considérations relatives aux modules nommés de la bibliothèque standard .

À propos des modules C++

Les fichiers d'en-tête sont la façon dont les déclarations et les définitions ont été partagées entre les fichiers sources en C++. Avant les modules de bibliothèque standard, vous incluiez la partie de la bibliothèque standard dont vous aviez besoin avec une directive comme `#include <vector>`. Les fichiers d'en-tête sont fragiles et difficiles à composer, car leur sémantique peut changer en fonction de l'ordre dans lequel vous les incluez ou si certaines macros sont définies ou non. Ils ralentissent également la compilation, car ils sont retraités par chaque fichier source qui les inclut.

C++20 introduit une alternative moderne appelée *modules*. En C++23, nous avons pu capitaliser sur la prise en charge des modules pour introduire des modules nommés pour représenter la bibliothèque standard.

Comme les fichiers d'en-tête, les modules vous permettent de partager des déclarations et des définitions entre les fichiers sources. Mais contrairement aux fichiers d'en-tête, les modules ne sont pas fragiles et sont plus faciles à composer, car leur sémantique ne change pas en raison des définitions de macros ou de l'ordre dans lequel vous les importez. Le compilateur peut traiter les modules beaucoup plus rapidement qu'il ne peut traiter `#include` des fichiers, et utilise également moins de mémoire au moment de la compilation. Les modules nommés n'exposent pas de définitions de macros ni de détails d'implémentation privée.

Pour plus d'informations sur les modules, consultez Vue d'ensemble des modules en C++ Cet article traite également de l'utilisation de la bibliothèque standard C++ en tant que modules, mais utilise une méthode plus ancienne et expérimentale.

Cet article présente la nouvelle et meilleure façon d'utiliser la bibliothèque standard. Pour plus d'informations sur les autres façons d'utiliser la bibliothèque standard, consultez Comparer les unités d'en-tête, les modules et les en-têtes précompilés.

Importer la bibliothèque standard avec `std`

Les exemples suivants montrent comment utiliser la bibliothèque standard en tant que module à l'aide du compilateur de ligne de commande. Pour plus d'informations sur la procédure à suivre dans l'IDE Visual Studio, consultez Générer des modules de bibliothèque STANDARD ISO C++23.

L'instruction `import std;` ou `import std.compat;` importe la bibliothèque standard dans votre application. Mais tout d'abord, vous devez compiler la bibliothèque standard nommée modules. Les étapes suivantes montrent comment procéder.

Exemple : `std`

1. Ouvrez une invite de commandes x86 Native Tools pour VS : dans le menu **Démarrer** de Windows, tapez *x86 native* et l'invite doit apparaître dans la liste des applications. Vérifiez que l'invite concerne Visual Studio 2022 preview version 17.5 ou ultérieure. Vous obtiendrez des erreurs si vous utilisez la mauvaise version de l'invite. Les exemples utilisés dans ce tutoriel concernent l'interpréteur de commandes CMD. Si vous utilisez PowerShell, remplacez par `"$Env:VCToolsInstallDir\modules\std.ixx"` `"%VCToolsInstallDir\modules\std.ixx"`.

2. Créez un répertoire, tel que `%USERPROFILE%\source\repos\STLModules`, et faites-en le répertoire actif. Si vous choisissez un répertoire auquel vous n'avez pas accès en écriture, vous obtiendrez des erreurs telles que l'impossibilité d'ouvrir le fichier `std.ifc` de sortie .

3. Compilez le `std` module nommé avec la commande suivante :

   ```
   dos
   ```

   ```
   cl /std:c++latest /EHsc /nologo /W4 /MTd /c
   "%VCToolsInstallDir%\modules\std.ixx"
   ```

   Si vous obtenez des erreurs, vérifiez que vous utilisez la version correcte de l'invite de commandes. Si vous rencontrez toujours des problèmes, veuillez signaler un bogue dans Visual Studio Developer Community .

   Vous devez compiler le `std` module nommé à l'aide des mêmes paramètres de compilateur que vous envisagez d'utiliser avec le code qui l'importe. Si vous disposez d'une solution multi-projets, vous pouvez compiler la bibliothèque standard nommée module une fois, puis y faire référence à partir de tous vos projets à l'aide de l'option /reference du compilateur.

   À l'aide de la commande du compilateur précédente, le compilateur génère deux fichiers :

   - `std.ifc` est la représentation binaire compilée de l'interface de module nommée que le compilateur consulte pour traiter l'instruction `import std;` . Il s'agit d'un artefact au moment de la compilation uniquement. Il n'est pas fourni avec votre application.
   - `std.obj` contient l'implémentation du module nommé. Ajoutez `std.obj` à la ligne de commande lorsque vous compilez l'exemple d'application pour lier statiquement les fonctionnalités que vous utilisez de la bibliothèque standard à votre application.

   Les commutateurs de ligne de commande clés dans cet exemple sont les suivants :

Commutateur	Signification
/std:c++:latest	Utilisez la dernière version de la bibliothèque et du standard du langage C++. Bien que la prise en charge des modules soit disponible sous `/std:c++20`, vous avez besoin de la bibliothèque standard la plus récente pour obtenir la prise en charge des modules nommés de bibliothèque standard.

Commutateur	Signification
/EHsc	Utilisez la gestion des exceptions C++, à l'exception des fonctions marquées `extern "C"`.
/MTd	Utilisez la bibliothèque d'exécution de débogage multithread statique.
/c	Compilez sans liaison.

4. Pour essayer d'importer la `std` bibliothèque, commencez par créer un fichier nommé `importExample.cpp` avec le contenu suivant :

```
C++
```

```
// requires /std:c++latest

import std;

int main()
{
    std::cout << "Import the STL library for best performance\n";
    std::vector<int> v{5, 5, 5};
    for (const auto& e : v)
    {
        std::cout << e;
    }
}
```

   Dans le code précédent, `import std;` remplace `#include <vector>` et `#include <iostream>`. L'instruction `import std;` rend toute la bibliothèque standard disponible avec une instruction. Si vous êtes préoccupé par les performances, il peut être utile de savoir que l'importation de l'ensemble de la bibliothèque standard est souvent beaucoup plus rapide que le traitement d'un seul fichier d'en-tête de bibliothèque standard tel que `#include <vector>`.

5. Compilez l'exemple à l'aide de la commande suivante dans le même répertoire que l'étape précédente :

```
dos
```

```
cl /std:c++latest /EHsc /nologo /W4 /MTd importExample.cpp std.obj
```

   Nous n'avons pas eu à spécifier `std.ifc` sur la ligne de commande, car le compilateur recherche automatiquement le `.ifc` fichier correspondant au nom du module spécifié par une `import` instruction. Lorsque le compilateur rencontre `import std;`, il trouve `std.ifc` , car nous l'avons placé dans le même répertoire

que le code source, ce qui nous soulage de la nécessité de le spécifier sur la ligne de commande. Si le `.ifc` fichier se trouve dans un répertoire différent du code source, utilisez le commutateur du /reference compilateur pour y faire référence.

Si vous générez un seul projet, vous pouvez combiner les étapes de création du module nommé de bibliothèque `std` standard et l'étape de génération de votre application en ajoutant `"%VCToolsInstallDir%\modules\std.ixx"` à la ligne de commande. Veillez à le placer avant tous les `.cpp` fichiers qui la consomment. Par défaut, le nom de l'exécutable de sortie est extrait du premier fichier d'entrée. Utilisez l'option du `/Fe` compilateur pour spécifier le nom de fichier de sortie souhaité. Ce tutoriel montre la compilation du `std` module nommé en tant qu'étape distincte, car vous n'avez besoin de générer qu'une seule fois la bibliothèque standard nommée module, puis vous pouvez y faire référence à partir de votre projet ou de plusieurs projets. Mais il peut être pratique de tout générer ensemble, comme le montre cette ligne de commande :

dos

```
cl /FeimportExample /std:c++latest /EHsc /nologo /W4 /MTd
"%VCToolsInstallDir%\modules\std.ixx" importExample.cpp
```

Étant donné la ligne de commande précédente, le compilateur produit un exécutable nommé `importExample.exe`. Lorsque vous l'exécutez, il produit la sortie suivante :

Sortie

```
Import the STL library for best performance
555
```

Les commutateurs de ligne de commande de touche dans cet exemple sont identiques à ceux de l'exemple précédent, sauf que le `/c` commutateur n'est pas utilisé.

Importer la bibliothèque standard et les fonctions C globales avec `std.compat`

La bibliothèque standard C++ inclut la bibliothèque standard ISO C. Le `std.compat` module fournit toutes les fonctionnalités du `std` module telles que `std::vector`, `std::cout`, `std::printf`, `std::scanf`, et ainsi de suite. Mais il fournit également les

versions d'espace de noms globaux de ces fonctions telles que `::printf`, `::scanf`, `::fopen`, `::size_t`et ainsi de suite.

Le `std.compat` module nommé est une couche de compatibilité fournie pour faciliter la migration du code existant qui fait référence aux fonctions runtime C dans l'espace de noms global. Si vous souhaitez éviter d'ajouter des noms à l'espace de noms global, utilisez `import std;`. Si vous devez faciliter la migration d'un codebase qui utilise de nombreuses fonctions d'exécution C non qualifiées (c'est-à-dire un espace de noms global), utilisez `import std.compat;`. Cela fournit les noms du runtime C de l'espace de noms global afin que vous n'ayez pas à qualifier toutes les mentions de nom global avec `std::`. Si vous n'avez pas de code existant qui utilise les fonctions du runtime C de l'espace de noms global, vous n'avez pas besoin d'utiliser `import std.compat;`. Si vous appelez uniquement quelques fonctions du runtime C dans votre code, il peut être préférable d'utiliser `import std;` et de qualifier les quelques noms de runtime C d'espace de noms globaux qui en ont besoin avec `std::`. Par exemple : `std::printf()`. Si vous voyez une erreur comme `error C3861: 'printf': identifier not found` lorsque vous essayez de compiler votre code, envisagez d'utiliser `import std.compat;` pour importer les fonctions du runtime C de l'espace de noms global.

Exemple : `std.compat`

Avant de pouvoir utiliser `import std.compat;` dans votre code, vous devez compiler le module `std.compat.ixx`nommé . Les étapes sont similaires à celles de la génération du `std` module nommé. Les étapes incluent la génération du `std` module nommé en premier, car `std.compat` cela dépend de celui-ci :

1. Ouvrez une invite de commandes Native Tools pour VS : dans le menu **Démarrer** de Windows, tapez *x86 natif* et l'invite doit apparaître dans la liste des applications. Vérifiez que l'invite est destinée à Visual Studio 2022 preview version 17.5 ou ultérieure. Vous obtiendrez des erreurs du compilateur si vous utilisez la version incorrecte de l'invite.

2. Créez un répertoire pour essayer cet exemple, par `%USERPROFILE%\source\repos\STLModules`exemple , et définissez-le comme répertoire actif. Si vous choisissez un répertoire auquel vous n'avez pas accès en écriture, vous obtenez des erreurs telles que l'impossibilité d'ouvrir le fichier `std.ifc`de sortie .

3. Compilez les `std` modules nommés et `std.compat` avec la commande suivante :

   ```
   dos
   ```

```
cl /std:c++latest /EHsc /nologo /W4 /MTd /c
"%VCToolsInstallDir%\modules\std.ixx"
"%VCToolsInstallDir%\modules\std.compat.ixx"
```

Vous devez compiler `std` et `std.compat` utiliser les mêmes paramètres de compilateur que ceux que vous envisagez d'utiliser avec le code qui les importe. Si vous disposez d'une solution à plusieurs projets, vous pouvez les compiler une seule fois, puis vous y référer à partir de tous vos projets à l'aide de l'option de /reference compilateur.

Si vous obtenez des erreurs, vérifiez que vous utilisez la version correcte de l'invite de commandes. Si vous rencontrez toujours des problèmes, signalez un bogue dans Visual Studio Developer Community. Bien que cette fonctionnalité soit toujours en préversion, vous trouverez une liste des problèmes connus sous Le problème 1694 des unités d'en-tête et des modules de la bibliothèque Standard.

Le compilateur génère quatre fichiers des deux étapes précédentes :

- `std.ifc` est l'interface de module nommée binaire compilée que le compilateur consulte pour traiter l'instruction `import std;` . Le compilateur consulte `std.ifc` également pour traiter `import std.compat;` , car `std.compat` s'appuie sur `std`. Il s'agit d'un artefact au moment de la compilation uniquement. Il n'est pas fourni avec votre application.
- `std.obj` contient l'implémentation de la bibliothèque standard.
- `std.compat.ifc` est l'interface de module nommée binaire compilée que le compilateur consulte pour traiter l'instruction `import std.compat;` . Il s'agit d'un artefact au moment de la compilation uniquement. Il n'est pas fourni avec votre application.
- `std.compat.obj` contient l'implémentation. Toutefois, la majeure partie de l'implémentation est fournie par `std.obj`. Ajoutez `std.obj` à la ligne de commande lorsque vous compilez l'exemple d'application pour lier statiquement les fonctionnalités que vous utilisez à partir de la bibliothèque standard à votre application.

4. Pour essayer d'importer la `std.compat` bibliothèque, créez un fichier nommé `stdCompatExample.cpp` avec le contenu suivant :

C++

```
import std.compat;

int main()
```

```
{
    printf("Import std.compat to get global names like printf()\n");

    std::vector<int> v{5, 5, 5};
    for (const auto& e : v)
    {
        printf("%i", e);
    }
}
```

Dans le code précédent, `import std.compat;` remplace `#include <cstdio>` et `#include <vector>`. L'instruction `import std.compat;` rend la bibliothèque standard et les fonctions du runtime C disponibles avec une seule instruction. Si vous êtes préoccupé par les performances, les performances des modules sont telles que l'importation de ce module nommé, qui inclut la bibliothèque standard C++ et les fonctions d'espace de noms globaux de la bibliothèque runtime C, est plus rapide que même le traitement d'un seul include comme `#include <vector>`.

5. Compilez l'exemple à l'aide de la commande suivante :

```
dos

cl /std:c++latest /EHsc /nologo /W4 /MTd stdCompatExample.cpp std.obj
std.compat.obj
```

Nous n'avons pas eu à spécifier `std.compat.ifc` sur la ligne de commande, car le compilateur recherche automatiquement le `.ifc` fichier qui correspond au nom du module dans une `import` instruction. Lorsque le compilateur rencontre `import std.compat;` , il trouve `std.compat.ifc` , car nous l'avons placé dans le même répertoire que le code source, ce qui nous soulage de la nécessité de le spécifier sur la ligne de commande. Si le `.ifc` fichier se trouve dans un autre répertoire que le code source, utilisez le commutateur du /reference compilateur pour y faire référence.

Même si nous importons `std.compat`, vous devez également établir un lien par rapport `std.obj` à, car c'est là que repose `std.compat` l'implémentation de bibliothèque standard.

Si vous créez un seul projet, vous pouvez combiner l'étape de génération des modules nommés et de la `std` bibliothèque standard avec l'étape de génération de votre application en ajoutant `"%VCToolsInstallDir%\modules\std.ixx"` et `"%VCToolsInstallDir%\modules\std.compat.ixx"` (dans cet ordre) à la ligne de `std.compat` commande. Veillez à les placer avant tous `.cpp` les fichiers qui les

consomment et spécifiez `/Fe` de nommer le généré `exe` comme indiqué dans cet exemple :

dos

```
cl /FestdCompatExample /std:c++latest /EHsc /nologo /W4 /MTd
"%VCToolsInstallDir%\modules\std.ixx"
"%VCToolsInstallDir%\modules\std.compat.ixx" stdCompatExample.cpp
```

Je les ai présentés en tant qu'étapes distinctes dans ce didacticiel, car vous n'avez besoin de générer la bibliothèque standard nommée modules qu'une seule fois, puis vous pouvez y faire référence à partir de votre projet ou de plusieurs projets. Mais il peut être pratique de les créer tous à la fois.

La commande du compilateur précédente produit un exécutable nommé `stdCompatExample.exe`. Lorsque vous l'exécutez, elle produit la sortie suivante :

Sortie

```
Import std.compat to get global names like printf()
555
```

Considérations relatives aux modules nommés de bibliothèque standard

Le contrôle de version pour les modules nommés est le même que pour les en-têtes. Les `.ixx` fichiers de module nommés cohabitent avec les en-têtes, par exemple : `"%VCToolsInstallDir%\modules\std.ixx`, qui se résout `C:\Program Files\Microsoft Visual Studio\2022\Preview\VC\Tools\MSVC\14.35.32019\modules\std.ixx` en dans la version des outils utilisés au moment de la rédaction de cet article. Sélectionnez la version du module nommé à utiliser de la même façon que vous choisissez la version du fichier d'en-tête à utiliser par le répertoire à partir duquel vous faites référence.

Ne mélangez pas et ne faites pas correspondre les unités d'en-tête d'importation et les modules nommés. Par exemple, ne et `import <vector>;` `import std;` dans le même fichier.

Ne mélangez pas et ne faites pas correspondre les fichiers d'en-tête de bibliothèque standard C++ et les modules nommés. Par exemple, ne et `#include <vector>` `import std;` dans le même fichier. Toutefois, vous pouvez inclure des en-têtes C et importer des modules nommés dans le même fichier. Par exemple, vous pouvez `import std;` et

`#include <math.h>` dans le même fichier. N'incluez pas la version de la bibliothèque standard C++, `<cmath>`.

Vous n'avez pas à vous défendre contre l'importation d'un module plusieurs fois : aucun protection d'en-tête n'est `#ifndef` nécessaire. Le compilateur sait quand il a déjà importé un module nommé et ignore les tentatives en double.

Si vous devez utiliser la `assert()` macro, alors `#include <assert.h>`.

Si vous devez utiliser la `errno` macro, `#include <errno.h>`. Étant donné que les modules nommés n'exposent pas de macros, il s'agit de la solution de contournement si vous devez case activée pour les erreurs à partir de `<math.h>`, par exemple.

Les macros telles que `NAN`, `INFINITY`et `INT_MIN` sont définies par `<limits.h>`, que vous pouvez inclure. Toutefois, si vous `import std;` pouvez utiliser `numeric_limits<double>::quiet_NaN()` et `numeric_limits<double>::infinity()` au lieu de `NAN` et `INFINITY`, et `std::numeric_limits<int>::min()` au lieu de `INT_MIN`.

Résumé

Dans ce tutoriel, vous avez importé la bibliothèque standard à l'aide de modules. Ensuite, découvrez comment créer et importer vos propres modules dans le didacticiel modules nommés en C++.

Voir aussi

Comparer les unités d'en-tête, les modules et les en-têtes précompilés
Vue d'ensemble des modules dans C++
Présentation des modules C++ dans Visual Studio
Déplacement d'un projet vers C++ nommé Modules

Didacticiel sur les modules nommés (C++)

Article • 03/04/2023

Ce tutoriel concerne la création de modules C++20. Les modules remplacent les fichiers d'en-tête. Vous allez découvrir comment les modules sont une amélioration des fichiers d'en-tête.

Dans ce tutoriel, vous allez apprendre à effectuer les actions suivantes :

- Créer et importer un module
- Créer une unité d'interface de module principal
- Créer un fichier de partition de module
- Créer un fichier d'implémentation d'unité de module

Prérequis

Ce didacticiel nécessite Visual Studio 2022 17.1.0 ou version ultérieure.

Vous pouvez obtenir des erreurs IntelliSense lors de l'utilisation de l'exemple de code dans ce didacticiel. Le travail sur le moteur IntelliSense rattrape le compilateur. Les erreurs IntelliSense peuvent être ignorées et n'empêchent pas la génération de l'exemple de code. Pour suivre la progression du travail IntelliSense, consultez ce problème.

Présentation des modules C++

Les fichiers d'en-tête sont la façon dont les déclarations et les définitions sont partagées entre les fichiers sources en C++. Les fichiers d'en-tête sont fragiles et difficiles à composer. Elles peuvent être compilées différemment selon l'ordre dans lequel vous les incluez, ou selon les macros définies ou non définies. Ils peuvent ralentir le temps de compilation, car ils sont retraités pour chaque fichier source qui les inclut.

C++20 introduit une approche moderne de la création de programmes C++ : *modules*.

Comme les fichiers d'en-tête, les modules vous permettent de partager des déclarations et des définitions entre les fichiers sources. Mais contrairement aux fichiers d'en-tête, les modules ne divulguent pas les définitions de macros ou les détails d'implémentation privée.

Les modules sont plus faciles à composer, car leur sémantique ne change pas en raison des définitions de macros ou de ce qui a été importé, de l'ordre des importations, etc. Ils facilitent également le contrôle de ce qui est visible par les consommateurs.

Les modules fournissent des garanties de sécurité supplémentaires que les fichiers d'en-tête ne le font pas. Le compilateur et l'éditeur de liens fonctionnent ensemble pour éviter d'éventuels problèmes de collision de noms et fournir des garanties de règle de définition (ODR) plus fortes.

Un modèle de propriété forte évite les conflits entre les noms au moment du lien, car l'éditeur de liens attache les noms exportés au module qui les exporte. Ce modèle permet au compilateur Microsoft Visual C++ d'empêcher un comportement non défini provoqué par la liaison de différents modules qui signalent des noms similaires dans le même programme. Pour plus d'informations, consultez Propriété forte.

Un module est constitué d'un ou plusieurs fichiers de code source compilés dans un fichier binaire. Le fichier binaire décrit tous les types, fonctions et modèles exportés dans le module. Lorsqu'un fichier source importe un module, le compilateur lit le fichier binaire qui contient le contenu du module. La lecture du fichier binaire est beaucoup plus rapide que le traitement d'un fichier d'en-tête. En outre, le fichier binaire est réutilisé par le compilateur chaque fois que le module est importé, ce qui permet de gagner encore plus de temps. Étant donné qu'un module est généré une fois plutôt qu'à chaque fois qu'il est importé, la durée de génération peut être réduite, parfois de façon spectaculaire.

Plus important encore, les modules n'ont pas les problèmes de fragilité que les fichiers d'en-tête. L'importation d'un module ne modifie pas la sémantique du module ni la sémantique d'un autre module importé. Les macros, les directives de préprocesseur et les noms non exportés déclarés dans un module ne sont pas visibles par le fichier source qui l'importe. Vous pouvez importer des modules dans n'importe quel ordre et cela ne modifie pas la signification des modules.

Les modules peuvent être utilisés côte à côte avec les fichiers d'en-tête. Cette fonctionnalité est pratique si vous migrez une base de code pour utiliser des modules, car vous pouvez le faire par étapes.

Dans certains cas, un fichier d'en-tête peut être importé en tant qu'unité d'en-tête plutôt qu'en tant `#include` que fichier. Les unités d'en-tête sont l'alternative recommandée aux fichiers d'en-tête précompilés (PCH). Ils sont plus faciles à configurer et à utiliser que les fichiers PCH partagés , mais ils offrent des avantages de performances similaires. Pour plus d'informations, consultez Procédure pas à pas : générer et importer des unités d'en-tête dans Microsoft Visual C++.

Votre code peut consommer automatiquement des modules dans le même projet, ou n'importe quel projet référencé, en utilisant des références de projet à projet à des projets de bibliothèque statique.

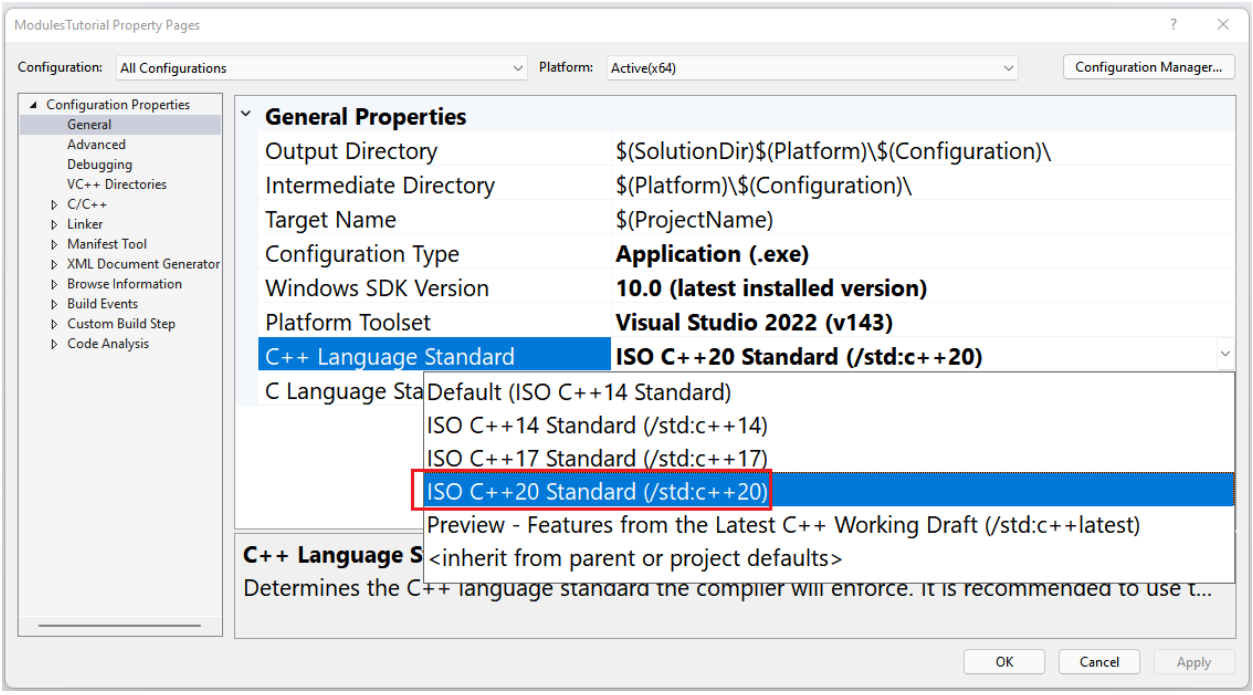
Créer le projet

À mesure que nous créons un projet simple, nous allons examiner différents aspects des modules. Le projet implémente une API à l'aide d'un module au lieu d'un fichier d'en-tête.

Dans Visual Studio 2022 ou version ultérieure, choisissez **Créer un projet** , puis le type de projet **Application console** (pour C++). Si ce type de projet n'est pas disponible, vous n'avez peut-être pas sélectionné la charge de travail **Développement Desktop avec C++** lors de l'installation de Visual Studio. Vous pouvez utiliser la Visual Studio Installer pour ajouter la charge de travail C++.

Donnez au nouveau projet le nom `ModulesTutorial` et créez le projet.

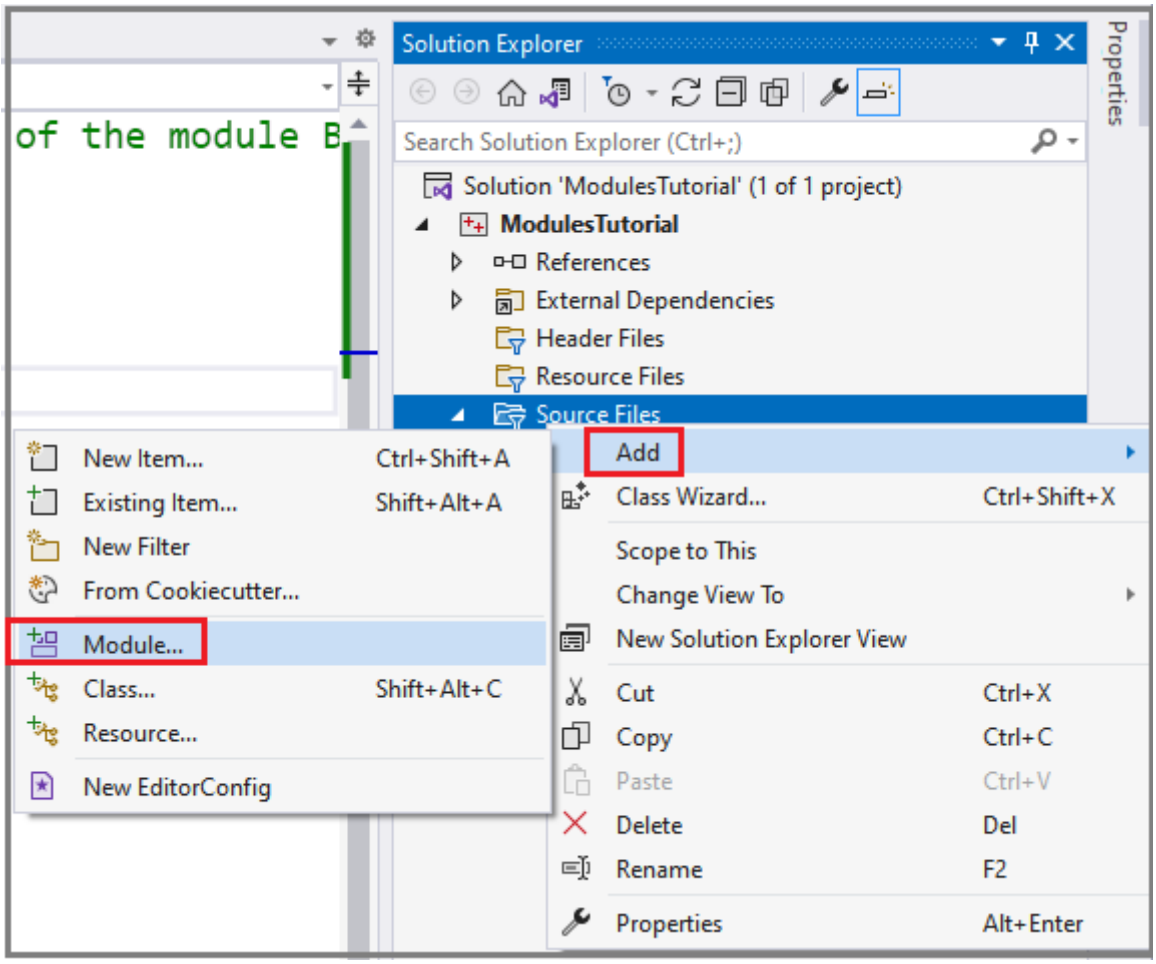
Étant donné que les modules sont une fonctionnalité C++20, utilisez l'option du /std:c++20 compilateur ou /std:c++latest . Dans le **Explorateur de solutions**, cliquez avec le bouton droit sur le nom `ModulesTutorial` du projet , puis choisissez **Propriétés**. Dans la boîte de dialogue Pages de propriétés du projet, remplacez **Configuration** par **Toutes les configurations** et **Plateforme** par **Toutes les plateformes**. Sélectionnez **Propriétés**> de configuration**Générales** dans le volet d'arborescence à gauche. Sélectionnez la propriété **C++ Language Standard** . Utilisez la liste déroulante pour modifier la valeur de la propriété en **ISO C++20 Standard (/std:c++20).** Sélectionnez **OK** pour accepter la modification.



Créer l'unité d'interface du module principal

Un module se compose d'un ou plusieurs fichiers. L'un de ces fichiers doit être ce que l'on appelle *l'unité d'interface du module principal*. Il définit ce que le module exporte ; autrement dit, ce que les importateurs du module verront. Il ne peut y avoir qu'une seule unité d'interface de module principal par module.

Pour ajouter une unité d'interface de module principal, dans **Explorateur de solutions**, cliquez avec le bouton droit sur **Fichiers sources**, puis sélectionnez **Ajouter un>module**.



Dans la boîte de dialogue **Ajouter un nouvel élément** qui s'affiche, donnez le nom `BasicPlane.Figures.ixx` au nouveau module, puis choisissez **Ajouter**.

Le contenu par défaut du fichier de module créé comporte deux lignes :

```
C++

export module BasicPlane;

export void MyFunc();
```

Les `export module` mots clés de la première ligne déclarent que ce fichier est une unité d'interface de module. Il y a un point subtil ici : pour chaque module nommé, il doit y

avoir exactement une unité d'interface de module sans partition de module spécifiée. Cette unité de module est appelée *unité d'interface de module principale*.

L'unité d'interface de module principale est l'endroit où vous déclarez les fonctions, les types, les modèles, les autres modules et les partitions de module à exposer lorsque les fichiers sources importent le module. Un module peut se composer de plusieurs fichiers, mais seul le fichier d'interface de module principal identifie les éléments à exposer.

Remplacez le contenu du `BasicPlane.Figures.ixx` fichier par :

```
C++

export module BasicPlane.Figures; // the export module keywords mark this
file as a primary module interface unit
```

Cette ligne identifie ce fichier en tant qu'interface de module principale et donne au module un nom : `BasicPlane.Figures`. Le point dans le nom du module n'a pas de signification particulière pour le compilateur. Une période peut être utilisée pour communiquer la façon dont votre module est organisé. Si vous avez plusieurs fichiers de module qui fonctionnent ensemble, vous pouvez utiliser des points pour indiquer une séparation des problèmes. Dans ce tutoriel, nous allons utiliser des points pour indiquer différentes zones fonctionnelles de l'API.

Ce nom est également d'où provient le « named » dans « named module ». Les fichiers qui font partie de ce module utilisent ce nom pour s'identifier dans le module nommé. Un module nommé est la collection d'unités de module portant le même nom de module.

Nous devrions parler de l'API que nous allons implémenter pendant un instant avant d'aller plus loin. Cela a un impact sur les choix que nous faisons ensuite. L'API représente différentes formes. Nous allons uniquement fournir quelques formes dans cet exemple : `Point` et `Rectangle`. `Point` est destiné à être utilisé dans le cadre de formes plus complexes, telles que `Rectangle`.

Pour illustrer certaines fonctionnalités des modules, nous allons factoriser cette API en plusieurs parties. Un élément sera l'API `Point` . L'autre partie sera `Rectangle`. Imaginez que cette API deviendra quelque chose de plus complexe. La division est utile pour séparer les problèmes ou faciliter la maintenance du code.

Jusqu'à présent, nous avons créé l'interface de module principal qui exposera cette API. Nous allons maintenant générer l'API `Point` . Nous voulons qu'il fait partie de ce module. Pour des raisons d'organisation logique et d'efficacité de build potentielle, nous

voulons que cette partie de l'API soit facilement compréhensible par elle-même. Pour ce faire, nous allons créer un fichier de *partition de module* .

Un fichier de partition de module est un élément, ou un composant, d'un module. Ce qui le rend unique, c'est qu'il peut être traité comme une pièce individuelle du module, mais uniquement dans le module. Les partitions de module ne peuvent pas être consommées en dehors d'un module. Les partitions de module sont utiles pour diviser l'implémentation de module en éléments gérables.

Lorsque vous importez une partition dans le module principal, toutes ses déclarations deviennent visibles pour le module principal, qu'elles soient exportées ou non. Les partitions peuvent être importées dans n'importe quelle interface de partition, interface de module principal ou unité de module qui appartient au module nommé.

Créer un fichier de partition de module

`Point` partition de module

Pour créer un fichier de partition de module, dans le **Explorateur de solutions** cliquez avec le bouton droit sur **Fichiers sources**, puis sélectionnez **Ajouter un**>**module**. Nommez le fichier `BasicPlane.Figures-Point.ixx` et choisissez **Ajouter**.

Étant donné qu'il s'agit d'un fichier de partition de module, nous avons ajouté un trait d'union et le nom de la partition au nom du module. Cette convention aide le compilateur dans le cas de la ligne de commande, car le compilateur utilise des règles de recherche de nom basées sur le nom du module pour rechercher le fichier compilé `.ifc` pour la partition. De cette façon, vous n'avez pas besoin de fournir des arguments de ligne de commande explicites `/reference` pour rechercher les partitions qui appartiennent au module. Il est également utile pour organiser les fichiers qui appartiennent à un module par nom, car vous pouvez facilement voir quels fichiers appartiennent à quels modules.

Remplacez le contenu du champ `BasicPlane.Figures-Point.ixx` avec :

C++

```
export module BasicPlane.Figures:Point; // defines a module partition,
Point, that's part of the module BasicPlane.Figures

export struct Point
{
    int x, y;
};
```

Si le fichier commence par `export module`. Ces mots clés sont également la façon dont l'interface du module principal commence. Ce qui différencie ce fichier est le signe deux-points (`:`) suivant le nom du module, suivi du nom de la partition. Cette convention de nommage identifie le fichier en tant que *partition de module*. Étant donné qu'elle définit l'interface de module pour une partition, elle n'est pas considérée comme l'interface du module principal.

Le nom `BasicPlane.Figures:Point` identifie cette partition comme faisant partie du module `BasicPlane.Figures`. (N'oubliez pas que le point dans le nom n'a pas de signification particulière pour le compilateur). Le signe deux-points indique que ce fichier contient une partition de module nommée `Point` qui appartient au module `BasicPlane.Figures`. Nous pouvons importer cette partition dans d'autres fichiers qui font partie de ce module nommé.

Dans ce fichier, le `export` mot clé est `struct Point` visible pour les consommateurs.

`Rectangle` partition de module

La partition suivante que nous allons définir est `Rectangle`. Créez un autre fichier de module en suivant les mêmes étapes que précédemment : dans **Explorateur de solutions**, cliquez avec le bouton droit sur **Fichiers sources**, puis sélectionnez **Ajouter un>module**. Nommez le fichier *`BasicPlane.Figures-Rectangle.ixx`* et sélectionnez **Ajouter**.

Remplacez le contenu du champ *`BasicPlane.Figures-Rectangle.ixx`* avec :

C++

```
export module BasicPlane.Figures:Rectangle; // defines the module partition
Rectangle

import :Point;

export struct Rectangle // make this struct visible to importers
{
    Point ul, lr;
};

// These functions are declared, but will
// be defined in a module implementation file
export int area(const Rectangle& r);
export int height(const Rectangle& r);
export int width(const Rectangle& r);
```

Le fichier commence `export module BasicPlane.Figures:Rectangle;` par lequel déclare une partition de module qui fait partie du module `BasicPlane.Figures`. Le `:Rectangle` ajouté au nom du module le définit comme une partition du module `BasicPlane.Figures`. Il peut être importé individuellement dans l'un des fichiers de module qui font partie de ce module nommé.

Ensuite, `import :Point;` montre comment importer une partition de module. L'instruction `import` rend tous les types, fonctions et modèles exportés dans la partition de module visibles pour le module. Vous n'avez pas besoin de spécifier le nom du module. Le compilateur sait que ce fichier appartient au `BasicPlane.Figures` module en raison du `export module BasicPlane.Figures:Rectangle;` en haut du fichier.

Ensuite, le code exporte la définition et les `struct Rectangle` déclarations de certaines fonctions qui retournent différentes propriétés du rectangle. Le `export` mot clé indique s'il faut rendre visible ce qui précède pour les consommateurs du module. Il est utilisé pour rendre les fonctions `area`, `height` et `width` visibles en dehors du module.

Toutes les définitions et déclarations d'une partition de module sont visibles par l'unité de module d'importation, qu'elles aient le `export` mot clé ou non. Le `export` mot clé détermine si la définition, la déclaration ou le typedef seront visibles en dehors du module lorsque vous exportez la partition dans l'interface du module principal.

Les noms sont rendus visibles par les consommateurs d'un module de plusieurs manières :

- Placez le mot clé `export` devant chaque type, fonction, etc., que vous souhaitez exporter.
- Si vous placez `export` devant un espace de noms, par exemple `export namespace N { ... }`, tout ce qui est défini dans les accolades est exporté. Mais si vous définissez `namespace N { struct S {...};}` ailleurs dans le module , il `struct S` n'est pas disponible pour les consommateurs du module. Elle n'est pas disponible, car cette déclaration d'espace de noms n'est pas préfacée par `export`, même s'il existe un autre espace de noms portant le même nom.
- Si un type, une fonction, etc., ne doit pas être exporté, omettez le `export` mot clé. Il sera visible pour d'autres fichiers qui font partie du module, mais pas pour les importateurs du module.
- Utilisez `module :private;` pour marquer le début de la partition de module privé. La partition de module privé est une section du module où les déclarations ne sont visibles que pour ce fichier. Ils ne sont pas visibles pour les fichiers qui importent ce module ou d'autres fichiers qui font partie de ce module. Considérez-le comme

une section qui est statique localement dans le fichier. Cette section est visible uniquement dans le fichier.
- Pour rendre visible une partition de module ou de module importée, utilisez `export import`. Un exemple est présenté dans la section suivante.

Composer les partitions de module

Maintenant que nous avons défini les deux parties de l'API, réunissons-les afin que les fichiers qui importent ce module puissent y accéder dans leur ensemble.

Toutes les partitions de module doivent être exposées dans le cadre de la définition de module à laquelle elles appartiennent. Les partitions sont exposées dans l'interface du module principal. Ouvrez le *`BasicPlane.Figures.ixx`* fichier, qui définit l'interface du module principal. Remplacez son contenu par :

C++

```
export module BasicPlane.Figures; // keywords export module marks this as a primary module interface unit

export import :Point; // bring in the Point partition, and export it to consumers of this module
export import :Rectangle; // bring in the Rectangle partition, and export it to consumers of this module
```

Les deux lignes qui commencent `export import` par sont nouvelles ici. Lorsqu'ils sont combinés comme ceci, ces deux mots clés indiquent au compilateur d'importer le module spécifié et de le rendre visible pour les consommateurs de ce module. Dans ce cas, le signe deux-points (`:`) dans le nom du module indique que nous importons une partition de module.

Les noms importés n'incluent pas le nom complet du module. Par exemple, la `:Point` partition a été déclarée en tant que `export module BasicPlane.Figures:Point`. Pourtant, ici, nous importons `:Point`. Étant donné que nous sommes dans le fichier d'interface du module principal pour le module `BasicPlane.Figures`, le nom du module est implicite et seul le nom de la partition est spécifié.

Jusqu'à présent, nous avons défini l'interface du module principal, qui expose la surface d'API que nous voulons mettre à disposition. Mais nous avons uniquement déclaré, non défini, `area()`, `height()` ou `width()`. Nous allons ensuite créer un fichier d'implémentation de module.

Créer un fichier d'implémentation d'unité de module

Les fichiers d'implémentation d'unités de module ne se terminent pas par une `.ixx` extension. Il s'agit de fichiers normaux `.cpp` . Ajoutez un fichier d'implémentation d'unité de module en créant un fichier source avec un clic droit dans le **Explorateur de solutions** sur **Fichiers sources**, sélectionnez **Ajouter un>nouvel élément**, puis sélectionnez **Fichier C++ (.cpp)**. Donnez au nouveau fichier le nom `BasicPlane.Figures-Rectangle.cpp`, puis choisissez **Ajouter**.

La convention de nommage du fichier d'implémentation de la partition de module suit la convention de nommage d'une partition. Mais il a une `.cpp` extension, car il s'agit d'un fichier d'implémentation.

Remplacez le contenu du `BasicPlane.Figures-Rectangle.cpp` fichier par :

C++

```
module;

// global module fragment area. Put #include directives here

module BasicPlane.Figures:Rectangle;

int area(const Rectangle& r) { return width(r) * height(r); }
int height(const Rectangle& r) { return r.ul.y - r.lr.y; }
int width(const Rectangle& r) { return r.lr.x - r.ul.x; }
```

Ce fichier commence `module;` par qui introduit une zone spéciale du module appelée *fragment de module global*. Il précède le code du module nommé et est l'endroit où vous pouvez utiliser des directives de préprocesseur telles que `#include`. Le code du fragment de module global n'est pas détenu ou exporté par l'interface du module.

Lorsque vous incluez un fichier d'en-tête, vous ne souhaitez généralement pas qu'il soit traité comme une partie exportée du module. Vous incluez généralement le fichier d'en-tête comme détail d'implémentation qui ne doit pas faire partie de l'interface du module. Il peut y avoir des cas avancés où vous voulez le faire, mais en général ce n'est pas le cas. Aucune métadonnées distinctes (`.ifc` fichiers) n'est générée pour `#include` les directives dans le fragment de module global. Les fragments de module globaux fournissent un bon emplacement pour inclure des fichiers d'en-tête tels que `windows.h`, ou sur Linux, `unistd.h`.

Le fichier d'implémentation de module que nous créons n'inclut aucune bibliothèque, car il n'en a pas besoin dans le cadre de son implémentation. Mais si c'était le cas, c'est dans ce domaine que les `#include` directives iraient.

La ligne `module BasicPlane.Figures:Rectangle;` indique que ce fichier fait partie du module `BasicPlane.Figures` nommé . Le compilateur intègre automatiquement les types et fonctions exposés par l'interface du module principal dans ce fichier. Une unité d'implémentation de module n'a pas le `export` mot clé avant le `module` mot clé dans sa déclaration de module.

Voici la définition des fonctions `area()`, `height()` et `width()`. Ils ont été déclarés dans la `Rectangle` partition dans *`BasicPlane.Figures-Rectangle.ixx`*. Étant donné que l'interface de module principale de ce module a importé les `Point` partitions de module et `Rectangle` , ces types sont visibles ici dans le fichier d'implémentation d'unité de module. Une fonctionnalité intéressante des unités d'implémentation de module : le compilateur rend automatiquement tous les éléments de l'interface principale du module correspondant visibles pour le fichier. Non `imports <module-name>` n'est pas nécessaire.

Tout ce que vous déclarez dans une unité d'implémentation n'est visible que par le module auquel il appartient.

Importation du module

Nous allons maintenant utiliser le module que nous avons défini. Ouvrez le fichier *`ModulesTutorial.cpp`* . Il a été créé automatiquement dans le cadre du projet. Il contient actuellement la fonction `main()`. Remplacez son contenu par :

C++

```
#include <iostream>

import BasicPlane.Figures;

int main()
{
    Rectangle r{ {1,8}, {11,3} };

    std::cout << "area: " << area(r) << '\n';
    std::cout << "width: " << width(r) << '\n';

    return 0;
}
```

L'instruction `import BasicPlane.Figures;` rend toutes les fonctions et types exportés du `BasicPlane.Figures` module visibles dans ce fichier. Il peut venir avant ou après n'importe quelle `#include` directive.

L'application utilise ensuite les types et les fonctions du module pour générer la zone et la largeur du rectangle défini :

Sortie

```
area: 50
width: 10
```

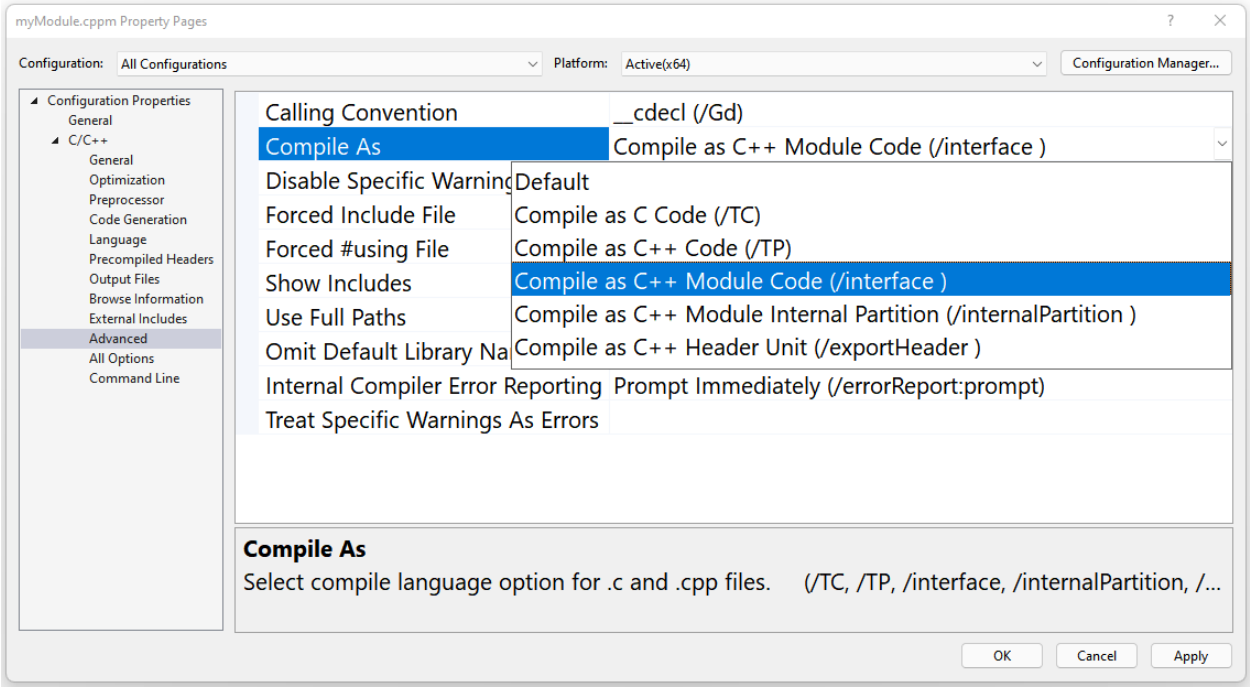
Anatomie d'un module

Examinons maintenant plus en détail les différents fichiers de module.

Interface de module principal

Un module se compose d'un ou plusieurs fichiers. L'un d'eux définit l'interface que les importateurs verront. Ce fichier contient *l'interface du module principal*. Il ne peut y avoir qu'une seule interface de module principal par module. Comme indiqué précédemment, l'unité d'interface de module exportée ne spécifie pas de partition de module.

Il a une `.ixx` extension par défaut. Toutefois, vous pouvez traiter un fichier source avec n'importe quelle extension comme un fichier d'interface de module. Pour ce faire, **définissez la propriété Compiler en tant que** dans l'onglet **Avancé** de la page de propriétés du fichier source sur **Compiler en tant que module (/interface)** :



Le plan de base d'un fichier de définition d'interface de module est le suivant :

C++

```
module; // optional. Defines the beginning of the global module fragment

// #include directives go here but only apply to this file and
// aren't shared with other module implementation files.
// Macro definitions aren't visible outside this file, or to importers.
// import statements aren't allowed here. They go in the module preamble,
below.

export module [module-name]; // Required. Marks the beginning of the module
preamble

// import statements go here. They're available to all files that belong to
the named module
// Put #includes in in the global module fragment, above

// After any import statements, the module purview begins here
// Put exported functions, types, and templates here

module :private; // optional. The start of the private module partition.

// Everything after this point is visible only within this file, and isn't
// visible to any of the other files that belong to the named module.
```

Ce fichier doit commencer par pour `module;` indiquer le début du fragment de module global ou `export module [module-name];` pour indiquer le début du *purview du module*.

Le module purview est l'emplacement où se trouvent les fonctions, les types, les modèles, etc., que vous souhaitez exposer à partir du module.

C'est également là que vous pouvez exposer d'autres modules ou partitions de module via les `export import` mots clés, comme indiqué dans le *`BasicPlane.Figures.ixx`* fichier.

Le fichier d'interface primaire doit exporter directement ou indirectement toutes les partitions d'interface définies pour le module, sinon le programme est mal formé.

La partition de module privé est l'endroit où vous pouvez placer les éléments que vous souhaitez voir uniquement dans ce fichier.

Les unités d'interface de module préfacent le mot clé `module` avec le mot clé `export`.

Pour plus d'informations sur la syntaxe du module, consultez Modules.

Unités d'implémentation de module

Les unités d'implémentation de module appartiennent à un module nommé. Le module nommé auquel ils appartiennent est indiqué par l'instruction `module [module-name]` dans le fichier. Les unités d'implémentation de module fournissent des détails d'implémentation que, pour des raisons d'hygiène du code ou d'autres raisons, vous ne souhaitez pas placer dans l'interface du module principal ou dans un fichier de partition de module.

Les unités d'implémentation de module sont utiles pour fractionner un grand module en morceaux plus petits, ce qui peut accélérer les temps de génération. Cette technique est brièvement abordée dans la section Meilleures pratiques .

Les fichiers d'unité d'implémentation de module ont une *`.cpp`* extension. Le plan de base d'un fichier d'unité d'implémentation de module est le suivant :

C++

```
// optional #include or import statements. These only apply to this file
// imports in the associated module's interface are automatically available
to this file

module [module-name]; // required. Identifies which named module this
implementation unit belongs to

// implementation
```

Fichiers de partition de module

Les partitions de module permettent de regrouper un module en différents éléments, ou *partitions*. Les partitions de module sont destinées à être importées uniquement dans

les fichiers qui font partie du module nommé. Ils ne peuvent pas être importés en dehors du module nommé.

Une partition a un fichier d'interface et aucun ou plusieurs fichiers d'implémentation. Une partition de module partage la propriété de toutes les déclarations dans l'ensemble du module.

Tous les noms exportés par les fichiers d'interface de partition doivent être importés et réexporté (`export import`) par le fichier d'interface primaire. Le nom d'une partition doit commencer par le nom du module, suivi d'un signe deux-points, puis du nom de la partition.

Le plan de base d'un fichier d'interface de partition ressemble à ceci :

C++

```
module; // optional. Defines the beginning of the global module fragment

// This is where #include directives go. They only apply to this file and
aren't shared
// with other module implementation files.
// Macro definitions aren't visible outside of this file or to importers
// import statements aren't allowed here. They go in the module preamble,
below

export module [Module-name]:[Partition name]; // Required. Marks the
beginning of the module preamble

// import statements go here.
// To access declarations in another partition, import the partition. Only
use the partition name, not the module name.
// For example, import :Point;
// #include directives don't go here. The recommended place is in the global
module fragment, above

// export imports statements go here

// after import, export import statements, the module purview begins
// put exported functions, types, and templates for the partition here

module :private; // optional. Everything after this point is visible only
within this file, and isn't
                         // visible to any of the other files that belong to
the named module.
...
```

Bonnes pratiques en matière de modules

Un module et le code qui l'importe doivent être compilés avec les mêmes options de compilateur.

Nommage de module

- Vous pouvez utiliser des points (« . ») dans les noms de vos modules, mais ils n'ont pas de signification particulière pour le compilateur. Utilisez-les pour transmettre du sens aux utilisateurs de votre module. Par exemple, commencez par l'espace de noms supérieur de la bibliothèque ou du projet. Terminez avec un nom qui décrit les fonctionnalités du module. `BasicPlane.Figures` est destiné à transmettre une API pour les plans géométriques, et plus particulièrement les figures qui peuvent être représentées sur un plan.
- Le nom du fichier qui contient l'interface principale du module est généralement le nom du module. Par exemple, étant donné le nom `BasicPlane.Figures` du module , le nom du fichier contenant l'interface principale est nommé *`BasicPlane.Figures.ixx`*.
- Le nom d'un fichier de partition de module est généralement `<primary-module-name>-<module-partition-name>` l'emplacement où le nom du module est suivi d'un trait d'union (« - ») puis du nom de la partition. Par exemple : *`BasicPlane.Figures-Rectangle.ixx`*

Si vous générez à partir de la ligne de commande et que vous utilisez cette convention de nommage pour les partitions de module, vous n'avez pas besoin d'ajouter `/reference` explicitement pour chaque fichier de partition de module. Le compilateur les recherche automatiquement en fonction du nom du module. Le nom du fichier de partition compilé (se terminant par une *`.ifc`* extension) est généré à partir du nom du module. Considérez le nom `BasicPlane.Figures:Rectangle` du module : le compilateur prévoit que le fichier de partition compilé correspondant pour `Rectangle` est nommé `BasicPlane.Figures-Rectangle.ifc`. Le compilateur utilise ce schéma de nommage pour faciliter l'utilisation des partitions de module en recherchant automatiquement les fichiers d'unité d'interface pour les partitions.

Vous pouvez les nommer à l'aide de votre propre convention. Mais vous devez ensuite spécifier les arguments correspondants /reference au compilateur de ligne de commande.

Modules de facteur

Utilisez des fichiers d'implémentation de module et des partitions pour factoriser votre module pour faciliter la maintenance du code et accélérer les temps de compilation.

Par exemple, le déplacement de l'implémentation d'un module hors du fichier de définition d'interface de module vers un fichier d'implémentation de module signifie que les modifications apportées à l'implémentation n'entraînent pas nécessairement la recompilation de chaque fichier qui importe le module (sauf si vous avez `inline` des implémentations).

Les partitions de module facilitent la factoriser logiquement un module volumineux. Ils peuvent être utilisés pour améliorer le temps de compilation afin que les modifications apportées à une partie de l'implémentation n'entraînent pas la recompilation de tous les fichiers du module.

Résumé

Dans ce tutoriel, vous avez découvert les principes de base des modules C++20. Vous avez créé une interface de module principal, défini une partition de module et créé un fichier d'implémentation de module.

Voir aussi

Vue d'ensemble des modules dans C++
module, import, export mots clés
Présentation des modules C++ dans Visual Studio
Modules C++20 pratiques et l'avenir des outils autour des modules C++
Déplacement d'un projet vers C++ nommé Modules
Procédure pas à pas : Générer et importer des unités d'en-tête dans Microsoft Visual C++

Modèles (C++)

Article • 16/06/2023

Les modèles sont la base de la programmation générique en C++. En tant que langage fortement typé, C++ exige que toutes les variables aient un type spécifique, soit déclaré explicitement par le programmeur, soit déduit par le compilateur. Toutefois, de nombreuses structures et algorithmes de données se ressemblent, quel que soit le type sur lequel ils fonctionnent. Les modèles vous permettent de définir les opérations d'une classe ou d'une fonction et permettent à l'utilisateur de spécifier les types concrets sur lesquels ces opérations doivent fonctionner.

Définition et utilisation de modèles

Un modèle est une construction qui génère un type ou une fonction ordinaire au moment de la compilation en fonction des arguments que l'utilisateur fournit pour les paramètres du modèle. Par exemple, vous pouvez définir un modèle de fonction comme suit :

C++

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Le code ci-dessus décrit un modèle pour une fonction générique avec un paramètre *de* type unique T, dont la valeur de retour et les paramètres d'appel (lhs et rhs) sont tous de ce type. Vous pouvez nommer un paramètre de type comme vous le souhaitez, mais par convention, les lettres majuscules uniques sont les plus couramment utilisées. *T* est un paramètre de modèle ; le `typename` mot clé indique que ce paramètre est un espace réservé pour un type. Lorsque la fonction est appelée, le compilateur remplace chaque instance de par l'argument de `T` type concret spécifié par l'utilisateur ou déduit par le compilateur. Le processus dans lequel le compilateur génère une classe ou une fonction à partir d'un modèle est appelé *instanciation de modèle* ; `minimum<int>` est une instanciation du modèle `minimum<T>`.

Ailleurs, un utilisateur peut déclarer une instance du modèle spécialisé pour int. Supposons que get_a() et get_b() sont des fonctions qui retournent un int :

C++

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

Toutefois, comme il s'agit d'un modèle de fonction et que le compilateur peut déduire le type de `T` à partir des arguments *a* et *b*, vous pouvez l'appeler comme une fonction ordinaire :

C++

```
int i = minimum(a, b);
```

Lorsque le compilateur rencontre cette dernière instruction, il génère une nouvelle fonction dans laquelle chaque occurrence de *T* dans le modèle est remplacée par **`int`**:

C++

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Les règles relatives à la façon dont le compilateur effectue la déduction de type dans les modèles de fonction sont basées sur les règles des fonctions ordinaires. Pour plus d'informations, consultez Résolution de surcharge des appels de modèle de fonction.

Paramètres de type

Dans le `minimum` modèle ci-dessus, notez que le paramètre de type *T* n'est qualifié en aucune façon tant qu'il n'est pas utilisé dans les paramètres d'appel de fonction, où les qualificateurs const et référence sont ajoutés.

Il n'existe aucune limite pratique au nombre de paramètres de type. Séparez plusieurs paramètres par des virgules :

C++

```
template <typename T, typename U, typename V> class Foo{};
```

Le mot clé **`class`** équivaut à **`typename`** dans ce contexte. Vous pouvez exprimer l'exemple précédent comme suit :

```
```

C++

```
template <class T, class U, class V> class Foo{};
```

Vous pouvez utiliser l'opérateur de points de suspension (...) pour définir un modèle qui prend un nombre arbitraire de paramètres de type zéro ou plus :

C++

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

Tout type intégré ou défini par l'utilisateur peut être utilisé comme argument de type. Par exemple, vous pouvez utiliser std::vector dans la bibliothèque standard pour stocker des variables de type `int`, , `double`, std::string, `MyClass`, `const MyClass*`, `MyClass&`, et ainsi de suite. La principale restriction lors de l'utilisation de modèles est qu'un argument de type doit prendre en charge toutes les opérations appliquées aux paramètres de type. Par exemple, si nous appelons `minimum` en utilisant `MyClass` comme dans cet exemple :

C++

```
class MyClass
{
public:
    int num;
    std::wstring description;
};

int main()
{
    MyClass mc1 {1, L"hello"};
    MyClass mc2 {2, L"goodbye"};
    auto result = minimum(mc1, mc2); // Error! C2678
}
```

Une erreur du compilateur sera générée, car `MyClass` ne fournit pas de surcharge pour l'opérateur < .

Il n'est pas nécessaire que les arguments de type d'un modèle particulier appartiennent tous à la même hiérarchie d'objets, bien que vous puissiez définir un modèle qui applique une telle restriction. Vous pouvez combiner des techniques orientées objet

avec des modèles ; par exemple, vous pouvez stocker un dérivé* dans un vecteur<Base*>. Notez que les arguments doivent être des pointeurs

```
C++
```

```
vector<MyClass*> vec;
   MyDerived d(3, L"back again", time(0));
   vec.push_back(&d);

   // or more realistically:
   vector<shared_ptr<MyClass>> vec2;
   vec2.push_back(make_shared<MyDerived>());
```

Les exigences de base que `std::vector` les autres conteneurs de bibliothèque standard imposent aux éléments de `T` sont d'être `T` assignables à la copie et constructibles de copie.

Paramètres non de type

Contrairement aux types génériques dans d'autres langages tels que C# et Java, les modèles C++ prennent en charge les *paramètres non de type*, également appelés paramètres de valeur. Par exemple, vous pouvez fournir une valeur intégrale constante pour spécifier la longueur d'un tableau, comme dans cet exemple qui est similaire à la classe std::array dans la bibliothèque Standard :

```
C++
```

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Notez la syntaxe dans la déclaration de modèle. La `size_t` valeur est transmise en tant qu'argument de modèle au moment de la compilation et doit être **`const`** ou une **`constexpr`** expression. Vous l'utilisez comme suit :

```
C++
```

```
MyArray<MyClass*, 10> arr;
```

D'autres types de valeurs, y compris les pointeurs et les références, peuvent être transmis en tant que paramètres non de type. Par exemple, vous pouvez passer un pointeur vers une fonction ou un objet de fonction pour personnaliser une opération à l'intérieur du code du modèle.

Déduction de type pour les paramètres de modèle autres que le type

Dans Visual Studio 2017 et versions ultérieures, et en mode ou version `/std:c++17` ultérieure, le compilateur déduit le type d'un argument de modèle non de type qui est déclaré avec `auto`:

C++

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;      // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;   // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;    // v3 == 'a', decltype(v3) is char
```

Modèles en tant que paramètres de modèle

Un modèle peut être un paramètre de modèle. Dans cet exemple, MyClass2 a deux paramètres de modèle : un paramètre typename *T* et un paramètre de modèle *Arr* :

C++

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

Étant donné que le paramètre *Arr* lui-même n'a pas de corps, ses noms de paramètres ne sont pas nécessaires. En fait, il s'agit d'une erreur de faire référence aux noms de typename ou de paramètres de classe *d'Arr* à partir du corps de `MyClass2`. Pour cette raison, les noms de paramètres de type *d'Arr* peuvent être omis, comme indiqué dans cet exemple :

C++

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

Arguments de modèle par défaut

Les modèles de classe et de fonction peuvent avoir des arguments par défaut. Lorsqu'un modèle a un argument par défaut, vous pouvez le laisser non spécifié lorsque vous l'utilisez. Par exemple, le modèle std::vector a un argument par défaut pour l'allocateur :

C++

```
template <class T, class Allocator = allocator<T>> class vector;
```

Dans la plupart des cas, la classe std::allocator par défaut est acceptable. Vous utilisez donc un vecteur comme suit :

C++

```
vector<int> myInts;
```

Mais si nécessaire, vous pouvez spécifier un allocateur personnalisé comme suit :

C++

```
vector<int, MyAllocator> ints;
```

Pour plusieurs arguments template, tous les arguments après le premier argument par défaut doivent avoir des arguments par défaut.

Lorsque vous utilisez un modèle dont tous les paramètres sont par défaut, utilisez des crochets vides :

C++

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};
...
```

```
int main()
{
    Bar<> bar; // use all default type arguments
}
```

Spécialisation de modèle

Dans certains cas, il n'est pas possible ou souhaitable pour un modèle de définir exactement le même code pour n'importe quel type. Par exemple, vous pouvez définir un chemin de code à exécuter uniquement si l'argument de type est un pointeur, un std::wstring ou un type dérivé d'une classe de base particulière. Dans ce cas, vous pouvez définir une *spécialisation* du modèle pour ce type particulier. Lorsqu'un utilisateur instancie le modèle avec ce type, le compilateur utilise la spécialisation pour générer la classe, et pour tous les autres types, le compilateur choisit le modèle plus général. Les spécialisations dans lesquelles tous les paramètres sont spécialisés sont *des spécialisations complètes*. Si seuls certains paramètres sont spécialisés, il s'agit d'une *spécialisation partielle*.

C++

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

Un modèle peut avoir n'importe quel nombre de spécialisations tant que chaque paramètre de type spécialisé est unique. Seuls les modèles de classe peuvent être partiellement spécialisés. Toutes les spécialisations complètes et partielles d'un modèle doivent être déclarées dans le même espace de noms que le modèle d'origine.

Pour plus d'informations, consultez Spécialisation de modèle.

`typename`

Article • 16/06/2023

Dans les définitions de modèle, `typename` fournit au compilateur un indicateur indiquant qu'un identificateur inconnu est un type. Dans les listes de paramètres de modèle, il est utilisé pour spécifier un paramètre de type.

Syntaxe

> `typename` *`identifier`* `;`

Notes

Le `typename` mot clé doit être utilisé si un nom dans une définition de modèle est un nom qualifié qui dépend d'un argument de modèle ; il est facultatif si le nom qualifié n'est pas dépendant. Pour plus d'informations, consultez Modèles et résolution de noms.

`typename` peut être utilisé par n'importe quel type n'importe où dans une déclaration ou une définition de modèle. Il n'est pas autorisé dans la liste des classes de base, sauf en tant qu'argument de modèle pour une classe de base de modèle.

C++

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType>  // typename OK.
{};
```

Le `typename` mot clé peut également être utilisé à la place de dans les listes de paramètres de `class` modèle. Par exemple, les instructions suivantes sont sémantiquement équivalentes :

C++

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Exemple

C++

```
// typename.cpp
template<class T> class X
{
   typename T::Y m_y;   // treat Y as a type
};

int main()
{
}
```

Voir aussi

Modèles
Mots clés

Modèles de classe

Article • 03/04/2023

Cet article décrit les règles spécifiques aux modèles de classe C++.

Fonctions membres des modèles de classe

Les fonctions membres peuvent être définies à l'intérieur ou à l'extérieur d'un modèle de classe. Ils sont définis comme des modèles de fonction s'ils sont définis en dehors du modèle de classe.

C++

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T*  pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}
```

Comme pour toute fonction membre de classe de modèle, la définition de la fonction membre constructeur de la classe inclut deux fois la liste d'arguments de modèle.

Les fonctions membres peuvent elles-mêmes être des modèles de fonction et spécifier des paramètres supplémentaires, comme dans l'exemple suivant.

C++

```
// member_templates.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

Modèles de classes imbriquées

Les modèles peuvent être définis dans des classes ou des modèles de classe, auquel cas ils sont appelés modèles membres. Les modèles membres qui sont des classes sont appelés modèles de classe imbriqués. Les modèles membres qui sont des fonctions sont abordés dans Modèles de fonction membre.

Les modèles de classe imbriqués sont déclarés en tant que modèles de classe dans la portée de la classe externe. Ils peuvent être définis à l'intérieur ou à l'extérieur de la classe englobante.

Le code suivant illustre un modèle de classe imbriqué à l'intérieur d'une classe ordinaire.

C++

```
// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{

   template <class T>
   struct Y
   {
      T m_t;
      Y(T t): m_t(t) { }
   };
```

```
   Y<int> yInt;
   Y<char> yChar;

public:
   X(int i, char c) : yInt(i), yChar(c) { }
   void print()
   {
      cout << yInt.m_t << " " << yChar.m_t << endl;
   }
};

int main()
{
   X x(1, 'a');
   x.print();
}
```

Le code suivant utilise des paramètres de type de modèle imbriqué pour créer des modèles de classe imbriqués :

C++

```
// nested_class_template2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
   template <class U> class Y
   {
      U* u;
   public:
      Y();
      U& Value();
      void print();
      ~Y();
   };

   Y<int> y;
public:
   X(T t) { y.Value() = t; }
   void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
   cout << "X<T>::Y<U>::Y()" << endl;
   u = new U();
```

```
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
   return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
   cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
   cout << "X<T>::Y<U>::~Y()" << endl;
   delete u;
}

int main()
{
   X<int>* xi = new X<int>(10);
   X<char>* xc = new X<char>('c');
   xi->print();
   xc->print();
   delete xi;
   delete xc;
}

/* Output:
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()
*/
```

Les classes locales ne sont pas autorisées à avoir des modèles membres.

Amis de modèle

Les modèles de classe peuvent avoir des amis. Une classe ou un modèle de classe, une fonction ou un modèle de fonction peuvent être des éléments friend d'une classe de modèle. Les éléments friend peuvent également être des spécialisations d'un modèle de classe ou de fonction, mais des spécialisations non partielles.

Dans l'exemple suivant, une fonction friend est définie en tant que modèle de fonction dans le modèle de classe. Ce code génère une version de la fonction friend pour chaque instanciation du modèle. Cette construction est utile si votre fonction friend dépend des mêmes paramètres de modèle que la classe.

C++

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
   T* array;
   int size;

public:
   Array(int sz): size(sz) {
      array = new T[size];
      memset(array, 0, size * sizeof(T));
   }

   Array(const Array& a) {
      size = a.size;
      array = new T[size];
      memcpy_s(array, a.array, sizeof(T));
   }

   T& operator[](int i) {
      return *(array + i);
   }

   int Length() { return size; }

   void print() {
      for (int i = 0; i < size; i++)
         cout << *(array + i) << " ";

      cout << endl;
   }

   template<class T>
   friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
   Array<T>* a = new Array<T>(a1.size + a2.size);
   for (int i = 0; i < a1.size; i++)
      (*a)[i] = *(a1.array + i);

   for (int i = 0; i < a2.size; i++)
```

```
      (*a)[i + a1.size] = *(a2.array + i);

   return a;
}

int main() {
   Array<char> alpha1(26);
   for (int i = 0 ; i < alpha1.Length() ; i++)
      alpha1[i] = 'A' + i;

   alpha1.print();

   Array<char> alpha2(26);
   for (int i = 0 ; i < alpha2.Length() ; i++)
      alpha2[i] = 'a' + i;

   alpha2.print();
   Array<char>*alpha3 = combine(alpha1, alpha2);
   alpha3->print();
   delete alpha3;
}
/* Output:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l
m n o p q r s t u v w x y z
*/
```

L'exemple suivant implique un élément friend possédant une spécialisation de modèle. Une spécialisation de modèle de fonction est automatiquement un élément friend si le modèle de fonction d'origine est un élément friend.

Il est également possible de déclarer uniquement la version spécialisée du modèle en tant qu'ami, comme l'indique le commentaire avant la déclaration friend dans le code suivant. Si vous déclarez une spécialisation en tant qu'ami, vous devez placer la définition de la spécialisation de modèle friend en dehors de la classe de modèle.

C++

```
// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
```

```
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {
        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}
```

```
int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}
/* Output:
10 generic
10 int
*/
```

L'exemple suivant montre un modèle de classe friend déclaré dans un modèle de classe. Le modèle de classe est utilisé ensuite comme argument template pour la classe friend. Les modèles de classe Friend doivent être définis en dehors du modèle de classe dans lequel ils sont déclarés. Toutes les spécialisations ou spécialisations partielles du modèle friend sont également des éléments friend du modèle de classe d'origine.

C++

```
// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
private:
   T* data;
   void InitData(int seed) { data = new T(seed); }
public:
   void print() { cout << *data << endl; }
   template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
   U* GetNewObject(int seed)
   {
      U* pu = new U;
      pu->InitData(seed);
      return pu;
   }
};

int main()
{
```

```
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
/* Output:
65
97
A
a
*/
```

Réutilisation des paramètres de modèle

Les paramètres de modèle peuvent être réutilisés dans la liste des paramètres du modèle. Par exemple, le code suivant est autorisé :

C++

```
// template_specifications2.cpp

class Y
{
};
template<class T, T* pT> class X1
{
};
template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{
}
```

Voir aussi

Modèles

Modèles de fonctions

Article • 16/06/2023

Les modèles de classe définissent une famille de classes connexes basées sur les arguments de type passés à la classe lors de son instanciation. Les modèles de fonction sont semblables aux modèles de classe mais définissent une famille de fonctions. Avec les modèles de fonction, vous pouvez spécifier un ensemble de fonctions basées sur le même code mais agir sur différents types ou différentes classes. Le modèle de fonction suivant permute deux éléments :

C++

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
   T c(a);
   a = b;
   b = c;
}
int main() {
}
```

Ce code définit une famille de fonctions qui permutent les valeurs des arguments. À partir de ce modèle, vous pouvez générer des fonctions qui échangeront `int` et des types, `long` ainsi que des types définis par l'utilisateur. `MySwap` permutera même une classe si le constructeur de copie et l'opérateur d'assignation de cette classe sont correctement définis.

En outre, le modèle de fonction vous empêche d'échanger des objets de différents types, car le compilateur connaît les types des paramètres *a* et *b* au moment de la compilation.

Bien que cette fonction puisse être effectuée par une fonction non basée sur un modèle à l'aide de pointeurs void, la version de modèle est de type sécurisé. Prenons les appels suivants :

C++

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );          //OK
MySwap( j, Hello );      //error
```

Le deuxième appel `MySwap` déclenche une erreur au moment de la compilation, car le compilateur ne peut pas générer de fonction `MySwap` avec des paramètres de types différents. Si des pointeurs void étaient utilisés, les deux appels de fonction se compileraient correctement, mais la fonction ne fonctionnerait pas correctement au moment de l'exécution.

Il est autorisé de spécifier explicitement les arguments template d'un modèle de fonction. Par exemple :

C++

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
   f<char>(j);   // Generate the specialization f(char).
   // If not explicitly specified, f(int) would be deduced.
}
```

Lorsque l'argument template est spécifié explicitement, les conversions implicites normales sont effectuées pour convertir l'argument de fonction vers le type des paramètres de modèle de fonction correspondants. Dans l'exemple ci-dessus, le compilateur est converti en `j` type `char`.

Voir aussi

Modèles
Instanciation de modèle de fonction
Instanciation explicite
Spécialisation explicite des modèles de fonction

Instanciation du modèle de fonction

Article • 03/04/2023

Lorsqu'un modèle de fonction est appelé pour la première fois pour chaque type, le compilateur crée une instanciation. Chaque instanciation est une version de la fonction basée sur un modèle spécialisée pour le type. Cette instanciation est appelée chaque fois que la fonction est utilisée pour le type. Si vous avez plusieurs instanciations identiques, même dans différents modules, une seule copie de l'instanciation finira dans le fichier exécutable.

La conversion des arguments de fonction est autorisée dans les modèles de fonction pour toute paire argument-paramètre où le paramètre ne dépend pas d'un argument template.

Les modèles de fonction peuvent être instanciés de manière explicite en déclarant le modèle avec un type particulier comme argument. Par exemple, le code suivant est autorisé :

C++

```
// function_template_instantiation.cpp
template<class T> void f(T) { }

// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);

// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
}
```

Voir aussi

Modèles de fonction

Instanciation explicite

Article • 16/06/2023

Vous pouvez utiliser l'instanciation explicite pour créer une instanciation d'une classe ou d'une fonction modèle sans l'utiliser dans votre code. Étant donné qu'elle est utile lorsque vous créez des fichiers de bibliothèque (`.lib`) qui utilisent des modèles pour la distribution, les définitions de modèles non étayées ne sont pas placées dans des fichiers objet (`.obj`) .

Exemples

Ce code instancie explicitement les `MyStack` `int` variables et six éléments :

```
C++

template class MyStack<int, 6>;
```

Cette instruction crée une instanciation de `MyStack` sans réserver de stockage pour un objet. Le code est généré pour tous les membres.

La ligne suivante instancie explicitement uniquement la fonction membre du constructeur :

```
C++

template MyStack<int, 6>::MyStack( void );
```

Vous pouvez instancier explicitement des modèles de fonction à l'aide d'un argument de type spécifique pour les redéclarer, comme illustré dans l'exemple d'instanciation de modèle de fonction.

Vous pouvez utiliser le `extern` mot clé pour empêcher l'instanciation automatique des membres. Par exemple :

```
C++

extern template class MyStack<int, 6>;
```

De même, vous pouvez marquer des membres spécifiques comme étant externes et non instanciés :

```
C++
```

```
extern template MyStack<int, 6>::MyStack( void );
```

Vous pouvez utiliser le `extern` mot clé pour empêcher le compilateur de générer le même code d'instanciation dans plusieurs modules d'objet. Vous devez instancier le modèle de fonction à l'aide des paramètres de modèle explicites spécifiés dans au moins un module lié si la fonction est appelée. Sinon, vous obtiendrez une erreur d'éditeur de liens lorsque le programme est généré.

> ⓘ **Notes**
>
> La `extern` mot clé dans la spécialisation s'applique uniquement aux fonctions membres définies en dehors du corps de la classe. Les fonctions définies à l'intérieur de la déclaration de classe sont considérées comme des fonctions inline et sont toujours instanciées.

Voir aussi

Modèles de fonctions

Spécialisation explicite de modèles de fonctions

Article • 03/04/2023

Avec un modèle de fonction, vous pouvez définir un comportement spécial pour un type spécifique en fournissant une spécialisation explicite (substitution) du modèle de fonction pour ce type. Par exemple :

```
C++
```

```
template<> void MySwap(double a, double b);
```

Cette déclaration vous permet de définir une fonction différente pour les `double` variables. Comme les fonctions sans modèle, les conversions de type standard (telles que la promotion d'une variable de type `float` vers `double`) sont appliquées.

Exemple

```
C++
```

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
}

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
}
int main()
{
}
```

Voir aussi

Modèles de fonction

Classement partiel des modèles de fonction (C++)

Article • 16/06/2023

Plusieurs modèles de fonction qui correspondent à la liste d'arguments d'un appel de fonction peuvent être disponibles. C++ définit un classement partiel des modèles de fonction pour spécifier quelle fonction doit être appelée. Le classement est partiel car certains modèles peuvent être considérés comme également spécialisés.

Le compilateur choisit le modèle de fonction le plus spécialisé disponible parmi les correspondances possibles. Par exemple, si un modèle de fonction prend un type `T` et qu'un autre modèle de fonction qui prend `T*` est disponible, la `T*` version est dite plus spécialisée. Il est préférable à la version générique `T` chaque fois que l'argument est un type pointeur, même si les deux sont des correspondances autorisées.

Utilisez le processus suivant pour déterminer si un candidat de modèle de fonction est plus spécialisé :

1. Considérez deux modèles de fonction, `T1` et `T2`.
2. Remplacez les paramètres dans par `T1` un type `X`unique hypothétique .
3. Avec la liste de paramètres dans `T1`, vérifiez si `T2` est un modèle valide pour cette liste de paramètres. Ignorez toutes les conversions implicites.
4. Répétez le même processus avec `T1` et `T2` inversé.
5. Si un modèle est une liste d'arguments de modèle valide pour l'autre modèle, mais que l'inverse n'est pas vrai, ce modèle est considéré comme moins spécialisé que l'autre modèle. Si, à l'étape précédente, les deux modèles forment des arguments valides l'un pour l'autre, ils sont considérés comme tout aussi spécialisés et des résultats d'appel ambigus lorsque vous tentez de les utiliser.
6. Grâce à ces règles :
    a. Une spécialisation de modèle pour un type spécifique est plus spécialisée qu'une spécialisation acceptant un argument de type générique.
    b. Un modèle prenant uniquement `T*` est plus spécialisé qu'un seul prenant uniquement `T`, car un type `X*` hypothétique est un argument valide pour un `T` argument de modèle, mais `X` n'est pas un argument valide pour un `T*` argument de modèle.

c. `const T` est plus spécialisé que `T`, car `const X` est un argument valide pour un `T` argument de modèle, mais `X` n'est pas un argument valide pour un `const T` argument de modèle.

d. `const T*` est plus spécialisé que `T*`, car `const X*` est un argument valide pour un `T*` argument de modèle, mais `X*` n'est pas un argument valide pour un `const T*` argument de modèle.

Exemple

L'exemple suivant fonctionne comme spécifié dans la norme :

C++

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>

template <class T> void f(T) {
   printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
   printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
   printf_s("Even more specialized function for const T*\n");
}

int main() {
   int i =0;
   const int j = 0;
   int *pi = &i;
   const int *cpi = &j;

   f(i);   // Calls less specialized function.
   f(pi);  // Calls more specialized function.
   f(cpi); // Calls even more specialized function.
   // Without partial ordering, these calls would be ambiguous.
}
```

Sortie

Output

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

Voir aussi

Modèles de fonctions

Modèles de fonctions membres

Article • 16/06/2023

Le terme « modèle membre » fait référence aux modèles de fonction membre et aux modèles de classes imbriqués. Les modèles de fonction membre sont des modèles de fonction qui sont membres d'une classe ou d'un modèle de classe.

Les fonctions membres peuvent être des modèles de fonction dans plusieurs contextes. Toutes les fonctions des modèles de classe sont génériques, mais ne sont pas appelées modèles membres ou modèles de fonction membre. Si ces fonctions membres prennent leurs propres arguments de modèle, elles sont considérées comme des modèles de fonction membre.

Exemple : Déclarer des modèles de fonction membre

Les modèles de fonction membre des classes ou modèles de classe non-modèle sont déclarés en tant que modèles de fonction avec leurs paramètres de modèle.

C++

```
// member_function_templates.cpp
struct X
{
   template <class T> void mf(T* t) {}
};

int main()
{
   int i;
   X* x = new X();
   x->mf(&i);
}
```

Exemple : modèle de fonction membre d'un modèle de classe

L'exemple suivant montre un modèle de fonction membre d'un modèle de classe.

C++

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u)
   {
   }
};

int main()
{
}
```

Exemple : Définir des modèles membres en dehors de la classe

C++

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}
```

Exemple : Conversion définie par l'utilisateur avec modèle

Les classes locales ne sont pas autorisées à avoir des modèles membres.

Les modèles de fonction membre ne peuvent pas être des fonctions virtuelles. Et, ils ne peuvent pas remplacer les fonctions virtuelles d'une classe de base lorsqu'elles sont

déclarées avec le même nom qu'une fonction virtuelle de classe de base.

L'exemple suivant montre une conversion définie par l'utilisateur avec modèle :

C++

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
   template <class U> operator S<U>()
   {
      return S<U>();
   }
};

int main()
{
   S<int> s1;
   S<long> s2 = s1;  // Convert s1 using UDC and copy constructs S<long>.
}
```

Voir aussi

Modèles de fonctions

Spécialisation de modèle

Article • 16/06/2023

Les modèles de classe peuvent être partiellement spécialisés et la classe obtenue est toujours un modèle. La spécialisation partielle permet de personnaliser partiellement le code du modèle pour des types spécifiques dans des situations telles que les suivantes :

- Un modèle comporte plusieurs types et seuls certains d'entre eux doivent être spécialisés. Le résultat est un modèle paramétrable sur les types restants.
- Un modèle comporte un seul type, mais une spécialisation est nécessaire pour les types pointeur, référence, pointeur vers un membre ou pointeur fonction. La spécialisation proprement dite est toujours un modèle sur le type désigné ou référencé.

Exemple : Spécialisation partielle des modèles de classe

C++

```
// partial_specialization_of_class_templates.cpp
#include <stdio.h>

template <class T> struct PTS {
   enum {
      IsPointer = 0,
      IsPointerToDataMember = 0
   };
};

template <class T> struct PTS<T*> {
   enum {
      IsPointer = 1,
      IsPointerToDataMember = 0
   };
};

template <class T, class U> struct PTS<T U::*> {
   enum {
      IsPointer = 0,
      IsPointerToDataMember = 1
   };
};

struct S{};

int main() {
```

```
   printf_s("PTS<S>::IsPointer == %d \nPTS<S>::IsPointerToDataMember ==
%d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
   printf_s("PTS<S*>::IsPointer == %d \nPTS<S*>::IsPointerToDataMember ==
%d\n"
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
   printf_s("PTS<int S::*>::IsPointer == %d \nPTS"
            "<int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>::
            IsPointerToDataMember);
}
```

Output

```
PTS<S>::IsPointer == 0
PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1
PTS<S*>::IsPointerToDataMember == 0
PTS<int S::*>::IsPointer == 0
PTS<int S::*>::IsPointerToDataMember == 1
```

Exemple : Spécialisation partielle pour les types de pointeurs

Si vous avez une classe de collection de modèles qui accepte n'importe quel type `T`, vous pouvez créer une spécialisation partielle qui accepte n'importe quel type `T*`de pointeur . Le code suivant illustre un modèle de classe de collection `Bag` et une spécialisation partielle pour les types pointeur dans laquelle la collection déréférence les types pointeur avant de les copier dans le tableau. La collection stocke ensuite les valeurs cibles des pointeurs. Avec le modèle d'origine, seuls les pointeurs proprement dits auraient été stockés dans la collection, laissant les données vulnérables à la suppression ou à la modification. Dans cette version de pointeur spéciale de la collection, du code vérifiant l'existence d'un pointeur null dans la méthode `add` est ajouté.

C++

```
// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
   T* elem;
   int size;
```

```
   int max_size;

public:
   Bag() : elem(0), size(0), max_size(1) {}
   void add(T t) {
      T* tmp;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmp = new T [max_size];
         for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
         tmp[size++] = t;
         delete[] elem;
         elem = tmp;
      }
      else
         elem[size++] = t;
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << elem[i] << " ";
      cout << endl;
   }
};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
   T* elem;
   int size;
   int max_size;

public:
   Bag() : elem(0), size(0), max_size(1) {}
   void add(T* t) {
      T* tmp;
      if (t == NULL) {   // Check for NULL
         cout << "Null pointer!" << endl;
         return;
      }

      if (size + 1 >= max_size) {
         max_size *= 2;
         tmp = new T [max_size];
         for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
         tmp[size++] = *t;  // Dereference
         delete[] elem;
         elem = tmp;
      }
      else
         elem[size++] = *t; // Dereference
   }
```

```
    void print() {
       for (int i = 0; i < size; i++)
          cout << elem[i] << " ";
       cout << endl;
    }
};

int main() {
    Bag<int> xi;
    Bag<char> xc;
    Bag<int*> xp; // Uses partial specialization for pointer types.

    xi.add(10);
    xi.add(9);
    xi.add(8);
    xi.print();

    xc.add('a');
    xc.add('b');
    xc.add('c');
    xc.print();

    int i = 3, j = 87, *p = new int[2];
    *p = 8;
    *(p + 1) = 100;
    xp.add(&i);
    xp.add(&j);
    xp.add(p);
    xp.add(p + 1);
    delete[] p;
    p = NULL;
    xp.add(p);
    xp.print();
}
```

Output

```
10 9 8
a b c
Null pointer!
3 87 8 100
```

Exemple : Définir une spécialisation partielle de sorte qu'un type soit `int`

L'exemple suivant définit une classe de modèle qui accepte des paires de deux types, puis définit une spécialisation partielle de cette classe de modèle spécialisée afin que

l'un des types soit `int`. La spécialisation définit une méthode de tri supplémentaire qui implémente un tri par propagation simple sur l'entier.

C++

```
// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
   Key* keys;
   Value* values;
   int size;
   int max_size;
public:
   Dictionary(int initial_size) :  size(0) {
      max_size = 1;
      while (initial_size >= max_size)
         max_size *= 2;
      keys = new Key[max_size];
      values = new Value[max_size];
   }
   void add(Key key, Value value) {
      Key* tmpKey;
      Value* tmpVal;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmpKey = new Key [max_size];
         tmpVal = new Value [max_size];
         for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
         }
         tmpKey[size] = key;
         tmpVal[size] = value;
         delete[] keys;
         delete[] values;
         keys = tmpKey;
         values = tmpVal;
      }
      else {
         keys[size] = key;
         values[size] = value;
      }
      size++;
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
   }
};
```

```
// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
   int* keys;
   Value* values;
   int size;
   int max_size;
public:
   Dictionary(int initial_size) :  size(0) {
      max_size = 1;
      while (initial_size >= max_size)
         max_size *= 2;
      keys = new int[max_size];
      values = new Value[max_size];
   }
   void add(int key, Value value) {
      int* tmpKey;
      Value* tmpVal;
      if (size + 1 >= max_size) {
         max_size *= 2;
         tmpKey = new int [max_size];
         tmpVal = new Value [max_size];
         for (int i = 0; i < size; i++) {
            tmpKey[i] = keys[i];
            tmpVal[i] = values[i];
         }
         tmpKey[size] = key;
         tmpVal[size] = value;
         delete[] keys;
         delete[] values;
         keys = tmpKey;
         values = tmpVal;
      }
      else {
         keys[size] = key;
         values[size] = value;
      }
      size++;
   }

   void sort() {
      // Sort method is defined.
      int smallest = 0;
      for (int i = 0; i < size - 1; i++) {
         for (int j = i; j < size; j++) {
            if (keys[j] < keys[smallest])
               smallest = j;
         }
         swap(keys[i], keys[smallest]);
         swap(values[i], values[smallest]);
      }
   }

   void print() {
      for (int i = 0; i < size; i++)
         cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
```

```
    }
};

int main() {
    Dictionary<const char*, const char*> dict(10);
    dict.print();
    dict.add("apple", "fruit");
    dict.add("banana", "fruit");
    dict.add("dog", "animal");
    dict.print();

    Dictionary<int, const char*> dict_specialized(10);
    dict_specialized.print();
    dict_specialized.add(100, "apple");
    dict_specialized.add(101, "banana");
    dict_specialized.add(103, "dog");
    dict_specialized.add(89, "cat");
    dict_specialized.print();
    dict_specialized.sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized.print();
}
```

Output

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```

Modèles et résolution de noms

Article • 03/04/2023

Dans les définitions de modèle, il existe trois types de noms :

- les noms déclarés localement, notamment le nom du modèle lui-même et tous les noms déclarés dans la définition de modèle ;
- les noms provenant de la portée englobante située en dehors de la définition de modèle ;
- les noms qui dépendent d'une certaine façon des arguments template, appelés noms dépendants.

Alors que les deux premiers noms concernent aussi des portées de classe et de fonction, des règles spécifiques de résolution de noms sont requises dans les définitions de modèle pour gérer la complexité ajouté des noms dépendants. Cela provient du fait que le compilateur connaît peu ces noms avant que le modèle ne soit instancié car il peut s'agir de types complètement différents selon les arguments template utilisés. Les noms non dépendants sont recherchés selon les règles classiques et au point de définition du modèle. Ces noms, qui sont indépendants des arguments template, sont recherchés une fois pour toutes les spécialisations de modèle. Les noms dépendants ne sont pas recherchés tant que le modèle n'est pas instancié et sont recherchés séparément pour chaque spécialisation.

Un type est dépendant s'il dépend des arguments template. En particulier un type est dépendant s'il s'agit :

- de l'argument template lui-même ;

  ```
  C++
  ```

  ```
  T
  ```

- d'un nom complet avec une qualification comprenant un type dépendant ;

  ```
  C++
  ```

  ```
  T::myType
  ```

- d'un nom complet si la partie non qualifiée identifie un type dépendant ;

  ```
  C++
  ```

```
N::T
```

- d'un type const ou volatile pour lequel le type de base est un type dépendant ;

```
C++

const T
```

- d'un type pointeur, référence, tableau ou pointeur de fonction basé sur un type dépendant :

```
C++

T *, T &, T [10], T (*)()
```

- d'un tableau dont la taille est basée sur un paramètre de modèle :

```
C++

template <int arg> class X {
int x[arg] ; // dependent type
}
```

- d'un type modèle construit à partir d'un modèle de paramètre :

```
C++

T<int>, MyTemplate<T>
```

Dépendance de type et dépendance de valeur

Les noms et les expressions dépendants d'un paramètre de modèle sont classés comme dépendant de type ou dépendant de valeur selon que le paramètre de modèle est un paramètre de type ou un paramètre de valeur. En outre, tous les identificateurs déclarés dans un modèle avec un type dépendant de l'argument template sont considérés comme dépendants de valeur, étant donné qu'un type intégral ou énumération est initialisé avec une expression dépendante de valeur.

Les expressions dépendantes de type et de valeur sont des expressions qui impliquent des variables qui dépendent de type ou de valeur. Ces expressions peuvent avoir une sémantique qui diffère selon les paramètres utilisés pour le modèle.

Voir aussi

Modèles

Résolution du nom pour les types dépendants

Article • 16/06/2023

Utilisez `typename` pour les noms qualifiés dans les définitions de modèle pour indiquer au compilateur que le nom qualifié donné identifie un type. Pour plus d'informations, consultez typename.

C++

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
   void f(typename T::myType* mt) {}
};

class Yarg
{
public:
   struct myType { };
};

int main()
{
   X<Yarg> x;
   x.f(new Yarg::myType());
   printf("Name resolved by using typename keyword.");
}
```

Output

```
Name resolved by using typename keyword.
```

La recherche de noms pour les noms dépendants examine les noms à partir du contexte de la définition de modèle (dans l'exemple suivant, ce contexte est trouvé`myFunction(char)`) et du contexte de l'instanciation du modèle. Dans l'exemple suivant, le modèle est instancié dans main ; par conséquent, le `MyNamespace::myFunction` est visible à partir du point d'instanciation et est choisi comme meilleure correspondance. Si `MyNamespace::myFunction` était renommé, `myFunction(char)` serait appelé à la place.

Tous les noms sont résolus comme s'ils étaient des noms dépendants. Néanmoins, nous vous recommandons d'utiliser des noms complets en cas de conflit éventuel.

C++

```
// template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
   cout << "Char myFunction" << endl;
}

template <class T> class Class1
{
public:
   Class1(T i)
   {
      // If replaced with myFunction(1), myFunction(char)
      // will be called
      myFunction(i);
}
};

namespace MyNamespace
{
   void myFunction(int)
   {
      cout << "Int MyNamespace::myFunction" << endl;
   }
};

using namespace MyNamespace;

int main()
{
   Class1<int>* c1 = new Class1<int>(100);
}
```

Output

Output

```
Int MyNamespace::myFunction
```

Levée des ambiguïtés de modèle

Visual Studio 2012 applique les règles standard C++98/03/11 pour lever l'ambiguïté avec l'mot clé « modèle ». Dans l'exemple suivant, Visual Studio 2010 accepte à la fois les lignes non conformes et les lignes conformes. Visual Studio 2012 accepte uniquement les lignes conformes.

C++

```
#include <iostream>
#include <ostream>
#include <typeinfo>
using namespace std;

template <typename T> struct Allocator {
    template <typename U> struct Rebind {
        typedef Allocator<U> Other;
    };
};

template <typename X, typename AY> struct Container {
    #if defined(NONCONFORMANT)
        typedef typename AY::Rebind<X>::Other AX; // nonconformant
    #elif defined(CONFORMANT)
        typedef typename AY::template Rebind<X>::Other AX; // conformant
    #else
        #error Define NONCONFORMANT or CONFORMANT.
    #endif
};

int main() {
    cout << typeid(Container<int, Allocator<float>>::AX).name() << endl;
}
```

La conformité avec les règles de levée des ambiguïtés est requise car, par défaut, C++ suppose que `AY::Rebind` n'est pas un modèle. Par conséquent, le compilateur interprète le `<` suivant comme un signe Inférieur à. Il doit savoir si `Rebind` est un modèle afin de pouvoir analyser correctement `<` comme crochet angulaire.

Voir aussi

Résolution de noms

Résolution de nom pour les noms déclarés localement

Article • 16/06/2023

Le nom du modèle lui-même peut être qualifié avec ou sans les arguments template. Dans la portée d'un modèle de classe, le nom lui-même fait référence au modèle. Dans la portée d'une spécialisation de modèle ou d'une spécialisation partielle, le nom seul fait référence à la spécialisation ou la spécialisation partielle. D'autres spécialisations ou spécialisations partielles du modèle peuvent également être référencées, avec les arguments template appropriés.

Exemple : Spécialisation et spécialisation partielle

Le code suivant montre que le nom `A` du modèle de classe est interprété différemment dans l'étendue d'une spécialisation ou d'une spécialisation partielle.

C++

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
   A* a1;   // A refers to A<T>
   A<int>* a2;  // A<int> refers to a specialization of A.
   A<T*>* a3;   // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
   A* a4; // A refers to A<T*>.
};

template<> class A<int> {
   A* a5; // A refers to A<int>.
};
```

Exemple : Conflit de noms entre le paramètre de modèle et l'objet

En cas de conflit de noms entre un paramètre de modèle et un autre objet, le paramètre de modèle peut être masqué ou ne peut pas être masqué. Les règles suivantes vous aideront à déterminer la priorité.

Le paramètre de modèle est dans la portée à partir du point où il apparaît la première fois, jusqu'à la fin de la classe ou du modèle de fonction. Si le nom s'affiche à nouveau dans la liste d'arguments template ou dans la liste de classes de base, il fait référence au même type. En C++ standard, aucun autre nom identique au paramètre de modèle ne peut être déclaré dans la même portée. Une extension Microsoft permet de redéfinir le paramètre de modèle dans la portée du modèle. L'exemple suivant montre l'utilisation du paramètre de modèle dans la spécification de base d'un modèle de classe.

C++

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
   // Derived1<int> d;
}
```

Exemple : Définir une fonction membre en dehors du modèle de classe

Lorsque des fonctions membres sont définies en dehors du modèle de classe, un autre nom de paramètre de modèle peut être utilisé. Si la définition de la fonction membre de modèle de classe utilise un nom différent de celui de la déclaration pour le paramètre de modèle et que le nom utilisé dans la définition est en conflit avec un autre membre de la déclaration, le membre dans la déclaration de modèle est prioritaire.

C++

```
// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
   struct Z {
      Z() { cout << "Z::Z()" << endl; }
   };
   void f();
};
```

```
template <class Z>
void C<Z>::f() {
   // Z refers to the struct Z, not to the template arg;
   // Therefore, the constructor for struct Z will be called.
   Z z;
}

int main() {
   C<int> c;
   c.f();
}
```

Output

```
Z::Z()
```

Exemple : Définir un modèle ou une fonction membre en dehors de l'espace de noms

Lors de la définition d'un modèle de fonction ou d'une fonction membre en dehors de l'espace de noms dans lequel le modèle a été déclaré, l'argument de modèle est prioritaire sur les noms des autres membres de l'espace de noms.

C++

```
// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
   void g() { cout << "NS::g" << endl; }

   template <class T> struct C {
      void f();
      void g() { cout << "C<T>::g" << endl; }
   };
};

template <class T>
void NS::C<T>::f() {
   g(); // C<T>::g, not NS::g
};

int main() {
   NS::C<int> c;
   c.f();
}
```

Output

```
C<T>::g
```

Exemple : La classe de base ou le nom du membre masque l'argument de modèle

Dans les définitions qui se trouvent en dehors de la déclaration de classe de modèle, si une classe de modèle a une classe de base qui ne dépend pas d'un argument de modèle et si la classe de base ou l'un de ses membres a le même nom qu'un argument de modèle, la classe de base ou le nom du membre masque l'argument de modèle.

C++

```
// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
   int i;
   void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
   void f();
};

template <class B, int i>
void C<B, i>::f() {
   B b;   // Base class b, not template argument.
   b.print();
   i = 1; // Set base class's i to 1.
}

int main() {
   C<int, 1> c;
   c.f();
   cout << c.i << endl;
}
```

Output

```
Base
1
```

Voir aussi

Résolution de noms

Résolution de surcharge des appels de modèles de fonctions

Article • 16/06/2023

Un modèle de fonction peut surcharger des fonctions non-modèle du même nom. Dans ce scénario, le compilateur tente d'abord de résoudre un appel de fonction à l'aide de la déduction d'argument de modèle pour instancier le modèle de fonction avec une spécialisation unique. Si la déduction de l'argument de modèle échoue, le compilateur prend en compte les surcharges de modèle de fonction instanciées et les surcharges de fonction non-modèle pour résoudre l'appel. Ces autres surcharges sont appelées *ensembles de candidats*. Si la déduction de l'argument de modèle réussit, la fonction générée est comparée aux autres fonctions de l'ensemble de candidats pour déterminer la meilleure correspondance, en suivant les règles de résolution de surcharge. Pour plus d'informations, consultez Surcharge des fonctions.

Exemple : Choisir une fonction sans modèle

Si une fonction sans modèle correspond également à un modèle de fonction, la fonction non-modèle est choisie (sauf si les arguments de modèle ont été explicitement spécifiés), comme dans l'appel `f(1, 1)` dans l'exemple suivant.

C++

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
   cout << "void f(T1, T2)" << endl;
};

int main()
{
   f(1, 1);   // Equally good match; choose the non-template function.
   f('a', 1); // Chooses the function template.
   f<int, int>(2, 2);  // Template arguments explicitly specified.
}
```

Output

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

Exemple : Modèle de fonction de correspondance exacte préféré

L'exemple suivant montre que le modèle de fonction correspondant exactement est préférable si la fonction sans modèle nécessite une conversion.

C++

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
   cout << "void f(T1, T2)" << endl;
};

int main()
{
   long l = 0;
   int i = 0;
   // Call the function template f(long, int) because f(int, int)
   // would require a conversion from long to int.
   f(l, i);
}
```

Output

```
void f(T1, T2)
```

Voir aussi

Résolution de noms

typename

Organisation du code source (modèles C++)

Article • 03/04/2023

Lorsque vous définissez un modèle de classe, vous devez organiser le code source de sorte que les définitions de membres soient visibles par le compilateur en cas de besoin. Vous avez le choix d'utiliser le *modèle d'inclusion* ou le modèle d'*instanciation explicite*. Dans le modèle d'inclusion, vous incluez les définitions de membre dans chaque fichier qui utilise un modèle. Cette approche est la plus simple et offre une flexibilité maximale en termes de types concrets compatibles avec votre modèle. Son inconvénient est qu'il peut augmenter le délai de compilation. Les délais peuvent être importants si un projet ou les fichiers inclus eux-mêmes sont volumineux. Avec l'instanciation explicite, le modèle lui-même instancie les classes ou les membres de classe concrets pour des types spécifiques. Cette approche peut accélérer le délai de compilation, mais elle limite l'utilisation aux seules classes que l'implémenteur de modèle a activées à l'avance. En règle générale, nous vous recommandons d'utiliser le modèle d'inclusion, sauf si le délai de compilation pose problème.

Arrière-plan

Les modèles ne sont pas comme les classes ordinaires dans le sens où le compilateur ne génère pas de code objet pour un modèle ou l'un de ses membres. Il n'y a rien à générer tant que le modèle n'est pas instancié avec des types concrets. Lorsque le compilateur rencontre une instanciation de modèle comme `MyClass<int> mc;` et qu'il n'existe encore aucune classe avec cette signature, il génère une nouvelle classe. Il tente également de générer le code des fonctions d'un membre qui sont utilisées. Si ces définitions se trouvent dans un fichier qui n'est pas #included, directement ou indirectement, dans le fichier .cpp en cours de compilation, le compilateur ne peut pas les voir. Du point de vue du compilateur, il ne s'agit pas nécessairement d'une erreur. Les fonctions peuvent être définies dans une autre unité de traduction où l'éditeur de liens les trouvera. Si l'éditeur de liens ne trouve pas ce code, il génère une erreur *externe non résolue* .

Le modèle d'inclusion

La façon la plus simple et la plus courante de rendre les définitions de modèle visibles dans une unité de traduction consiste à placer les définitions dans le fichier d'en-tête

lui-même. Tout `.cpp` fichier qui utilise le modèle a simplement à `#include` l'en-tête. Cette approche est utilisée dans la bibliothèque Standard.

C++

```
#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    // Full definitions:
    MyArray(){}
    void Print()
    {
        for (const auto v : arr)
        {
            std::cout << v << " , ";
        }
    }

    T& operator[](int i)
   {
       return arr[i];
   }
};
#endif
```

Avec cette approche, le compilateur a accès à la définition de modèle complète et peut instancier des modèles à la demande pour n'importe quel type. Il est simple et relativement facile à entretenir. Mais le modèle d'inclusion présente un certain coût en termes de délai de compilation. Ce coût peut être significatif dans des programmes volumineux, surtout si l'en-tête du modèle lui-même inclut d'autres en-têtes. Chaque `.cpp` fichier qui #includes l'en-tête obtient sa propre copie des modèles de fonction et de toutes les définitions. L'éditeur de liens est généralement en mesure de trier les choses afin que vous ne vous retrouvez pas avec plusieurs définitions pour une fonction, mais cela prend du temps pour effectuer ce travail. Dans les programmes plus petits, ce délai de compilation supplémentaire sera sans doute marginal.

Le modèle d'instanciation explicite

Si le modèle d'inclusion n'est pas viable pour votre projet et que vous connaissez définitivement l'ensemble des types qui seront utilisés pour instancier un modèle, vous pouvez séparer le code du modèle dans un `.h` fichier et `.cpp` et dans le `.cpp` fichier

instancier explicitement les modèles. Cette approche génère du code objet que le compilateur voit lorsqu'il rencontre des instanciations utilisateur.

Vous créez une instanciation explicite à l'aide de la mot clé **`template`** suivie de la signature de l'entité que vous souhaitez instancier. Cette entité peut être un type ou un membre. Si vous instanciez explicitement un type, tous les membres sont instanciés.

Le fichier `MyArray.h` d'en-tête déclare la classe de `MyArray` modèle :

C++

```
//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};
#endif
```

Le fichier `MyArray.cpp` source instancie `template MyArray<double, 5>` explicitement et `template MyArray<string, 5>`:

C++

```
//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << "'";
    }
    cout << endl;
}
```

```
template MyArray<double, 5>;
template MyArray<string, 5>;
```

Dans l'exemple précédent, les instanciations explicites se trouvent au bas du `.cpp` fichier. Un `MyArray` peut être utilisé uniquement pour les **`double`** types ou `String` .

> ⓘ Notes
>
> En C++11, le **`export`** mot clé était déconseillé dans le contexte des définitions de modèle. Dans la pratique, cela a un impact limité car la plupart des compilateurs ne l'ont jamais pris en charge.

Gestion des événements

Article • 16/06/2023

La gestion des événements est principalement prise en charge pour les classes COM (classes C++ qui implémentent des objets COM, généralement à l'aide de classes ATL ou de l'attribut coclass). Pour plus d'informations, consultez Gestion des événements dans COM.

La gestion des événements est également prise en charge pour les classes C++ natives (classes C++ qui n'implémentent pas d'objets COM). La prise en charge de la gestion des événements C++ native est déconseillée et sera supprimée dans une version ultérieure. Pour plus d'informations, consultez Gestion des événements en C++ natif.

> ⓘ **Notes**
>
> Les attributs d'événement en C++ natif ne sont pas compatibles avec C++Standard. Ils ne se compilent pas lorsque vous spécifiez le **/permissive-** mode de conformité.

La gestion des événements prend en charge l'utilisation unique et multithread. Il protège les données contre l'accès multithread simultané. Vous pouvez dériver des sous-classes à partir de classes de source d'événement ou de récepteur. Ces sous-classes prennent en charge l'approvisionnement et la réception d'événements étendus.

Le compilateur Microsoft C++ inclut des attributs et des mots clés pour la déclaration d'événements et de gestionnaires d'événements. Les attributs et les mots clés d'événement peuvent être utilisés dans les programmes CLR et dans les programmes C++ natifs.

Article	Description
event_source	Crée une source d'événement.
event_receiver	Crée un récepteur d'événements (récepteur).
__event	Déclare un événement.
__raise	Met en évidence le site d'appel d'un événement.
__hook	Associe une méthode de gestionnaire à un événement.
__unhook	Dissocie une méthode de gestionnaire d'un événement.

Voir aussi

Informations de référence sur le langage C++

Mots clés

Mot clé `__event`

Article • 03/04/2023

Déclare un événement.

> ⓘ **Notes**
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Syntaxe

```
__event member-function-declarator ;
__event __interface interface-specifier ;
__event data-member-declarator ;
```

Notes

Les mot clé `__event` spécifiques à Microsoft peuvent être appliqués à une déclaration de fonction membre, à une déclaration d'interface ou à une déclaration de membre de données. Toutefois, vous ne pouvez pas utiliser le `__event` mot clé pour qualifier un membre d'une classe imbriquée.

Selon que votre source et votre récepteur d'événements sont C++ natif, COM ou managé (.NET Framework), vous pouvez utiliser les constructions suivantes en tant qu'événements :

C++ natif	COM	Managé (.NET Framework)
fonctions membres	-	method
-	interface	-
-	-	membre de données

Utilisez __hook dans un récepteur d'événements pour associer une fonction membre de gestionnaire à une fonction membre d'événement. Une fois que vous avez créé un événement avec le `__event` mot clé, tous les gestionnaires d'événements liés à cet événement sont ensuite appelés lorsque l'événement est appelé.

Une `__event` déclaration de fonction membre ne peut pas avoir de définition ; une définition est implicitement générée, de sorte que la fonction membre d'événement peut être appelée comme s'il s'agissait d'une fonction membre ordinaire.

> ⓘ Notes
>
> Une classe ou un struct avec modèle ne peut pas contenir d'événements.

Événements natifs

Les événements natifs sont des fonctions membres. Le type de retour est généralement `HRESULT` ou `void`, mais peut être n'importe quel type intégral, y compris un `enum`. Lorsqu'un événement utilise un type de retour intégral, une condition d'erreur est définie lorsqu'un gestionnaire d'événements retourne une valeur différente de zéro. Dans ce cas, l'événement déclenché appelle les autres délégués.

C++

```
// Examples of native C++ events:
__event void OnDblClick();
__event HRESULT OnClick(int* b, char* s);
```

Pour obtenir un exemple de code , consultez Gestion des événements en C++ natif .

Événements COM

Les événements COM sont des interfaces. Les paramètres d'une fonction membre dans une interface source d'événement doivent être *dans* des paramètres, mais ils ne sont pas appliqués de manière rigoureuse. C'est parce qu'un paramètre *out* n'est pas utile lors de la multidiffusion. Un avertissement de niveau 1 est émis si vous utilisez un paramètre *out* .

Le type de retour est généralement `HRESULT` ou `void`, mais peut être n'importe quel type intégral, y compris `enum`. Lorsqu'un événement utilise un type de retour intégral et qu'un gestionnaire d'événements retourne une valeur différente de zéro, il s'agit d'une condition d'erreur. L'événement déclenché annule les appels aux autres délégués. Le compilateur marque automatiquement une interface source d'événement comme un source dans l'IDL généré.

Le __interface mot clé est toujours requis après `__event` pour une source d'événement COM.

```
C++
```

```
// Example of a COM event:
__event __interface IEvent1;
```

Pour obtenir un exemple de code , consultez Gestion des événements dans COM .

Événements managés

Pour plus d'informations sur le codage d'événements dans la nouvelle syntaxe, consultez événement.

Les événements managés sont des membres de données ou des fonctions membres. Lorsqu'il est utilisé avec un événement, le type de retour d'un délégué doit être conforme à la spécification common language. Le type de retour du gestionnaire d'événements doit correspondre à celui du délégué. Pour plus d'informations sur les délégués, consultez Délégués et événements. Si un événement managé est un membre de données, son type doit être un pointeur vers un délégué.

Dans .NET Framework, vous pouvez traiter un membre de données comme s'il s'agissait d'une méthode (autrement dit, la méthode `Invoke` de son délégué correspondant). Pour ce faire, prédéfinisez le type de délégué pour la déclaration d'un membre de données d'événement managé. En revanche, une méthode d'événement managé définit implicitement le délégué managé correspondant s'il n'est pas déjà défini. Par exemple, vous pouvez déclarer une valeur d'événement telle que `OnClick` comme événement de la manière suivante :

```
C++
```

```
// Examples of managed events:
__event ClickEventHandler* OnClick;  // data member as event
__event void OnClick(String* s);  // method as event
```

Lorsque vous déclarez implicitement un événement managé, vous pouvez spécifier `add` des accesseurs et `remove` qui sont appelés lorsque des gestionnaires d'événements sont ajoutés ou supprimés. Vous pouvez également définir la fonction membre qui appelle (déclenche) l'événement à partir de l'extérieur de la classe.

Exemple : événements natifs

C++

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};
```

Exemple : événements COM

C++

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-
B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT MyEvent();
};
[ coclass, uuid("00000000-0000-0000-0000-000000000003"),  event_source(com)
]
class CSource : public IEventSource {
public:
   __event __interface IEventSource;
   HRESULT FireEvent() {
      __raise MyEvent();
      return S_OK;
   }
};
```

Voir aussi

Mots clés

Gestion des événements

event_source

event_receiver

__hook
__unhook
__raise

Mot clé `__hook`

Article • 16/06/2023

Associe une méthode de gestionnaire à un événement.

> ⓘ **Notes**
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Syntaxe

C++

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

Paramètres

`&SourceClass::EventMethod`
Pointeur vers la méthode d'événement à laquelle vous raccordez la méthode de gestionnaire d'événements :

- Événements C++ natifs : *`SourceClass`* est la classe source d'événements et *`EventMethod`* est l'événement.
- Événements COM : *`SourceClass`* est l'interface source d'événement et *`EventMethod`* est l'une de ses méthodes.
- Événements managés : *`SourceClass`* est la classe source d'événements et *`EventMethod`* est l'événement.

interface

Nom de l'interface lié à *receiver*, uniquement pour les récepteurs d'événements COM dans lesquels le *layout_dependent* paramètre de l'attribut event_receiver est **true**.

source

Pointeur vers une instance de la source d'événement. Selon le code type spécifié dans event_receiver, *source* peut être l'un des types suivants :

- Un pointeur d'objet source de l'événement natif.
- Pointeur IUnknown basé sur (source COM).
- Un pointeur d'objet managé (pour les événements managés).

&ReceiverClass::HandlerMethod

Pointeur vers la méthode de gestionnaire d'événements à raccorder à un événement. Le gestionnaire est spécifié en tant que méthode d'une classe ou référence à celle-ci. Si vous ne spécifiez pas le nom de la classe, **__hook** suppose que la classe est celle à partir de laquelle elle est appelée.

- Événements C++ natifs : *ReceiverClass* est la classe de récepteur d'événements et HandlerMethod est le gestionnaire.
- Événements COM : *ReceiverClass* est l'interface du récepteur d'événements et *HandlerMethod* est l'un de ses gestionnaires.
- Événements managés : *ReceiverClass* est la classe de récepteur d'événements et *HandlerMethod* est le gestionnaire.

receiver

(Facultatif) Pointeur vers un instance de la classe du récepteur d'événements. Si vous ne spécifiez pas de récepteur, la valeur par défaut est la classe ou la structure de récepteur dans laquelle **__hook** est appelée.

Utilisation

Peut s'utiliser dans une portée de fonction quelconque, notamment Main, en dehors de la classe de récepteur d'événements.

Notes

Utilisez la fonction `__hook` intrinsèque dans un récepteur d'événements pour associer ou crocheter une méthode de gestionnaire à une méthode d'événement. Le gestionnaire spécifié est ensuite appelé lorsque la source déclenche l'événement spécifié. Vous pouvez raccorder plusieurs gestionnaires à un même événement ou raccorder plusieurs événements à un même gestionnaire.

Il existe deux formes de `__hook`. Vous pouvez utiliser le premier formulaire (quatre arguments) dans la plupart des cas, en particulier pour les récepteurs d'événements COM dans lesquels le paramètre *layout_dependent* de l'attribut event_receiver est `false`.

Dans ce cas, vous n'avez pas besoin de crocheter toutes les méthodes d'une interface avant de déclencher un événement sur l'une des méthodes. Il vous suffit de crocheter la méthode qui gère l'événement. Vous pouvez utiliser la deuxième forme (à deux arguments) de `__hook` uniquement pour un récepteur d'événements COM dans lequel `layout_dependent = true`.

`__hook` retourne une valeur longue. Une valeur de retour différente de zéro indique qu'une erreur s'est produite (les événements managés lèvent une exception).

Le compilateur vérifie l'existence d'un événement et la conformité de la signature d'événement à la signature du délégué.

Vous pouvez appeler `__hook` et `__unhook` en dehors du récepteur d'événements, à l'exception des événements COM.

Une alternative à l'utilisation `__hook` consiste à utiliser l'opérateur +=.

Pour plus d'informations sur le codage d'événements managés dans la nouvelle syntaxe, consultez event.

> ⓘ **Notes**
>
> Une classe ou structure modélisée ne peut pas contenir d'événements.

Exemple

Pour obtenir des exemples, consultez Gestion des événements en C++ natif et Gestion des événements dans COM .

Voir aussi

Mots clés
Gestion des événements
event_source
event_receiver
__event
__unhook
__raise

Mot clé `__raise`

Article • 16/06/2023

Met en évidence le site d'appel d'un événement.

> ⓘ Notes
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Syntaxe

> `__raise` *`method-declarator`* `;`

Notes

À partir du code managé, un événement ne peut être déclenché qu'à partir de la classe où il est défini. Pour plus d'informations, consultez event.

Le mot clé `__raise` provoque l'émission d'une erreur si vous appelez un non-événement.

> ⓘ Notes
>
> Une classe ou structure modélisée ne peut pas contenir d'événements.

Exemple

C++

```
// EventHandlingRef_raise.cpp
struct E {
   __event void func1();
   void func1(int) {}

   void func2() {}

   void b() {
      __raise func1();
      __raise func1(1);  // C3745: 'int Event::bar(int)':
```

```
                        // only an event can be 'raised'
      __raise func2();   // C3745
   }
};

int main() {
   E e;
   __raise e.func1();
   __raise e.func1(1);  // C3745
   __raise e.func2();   // C3745
}
```

Voir aussi

Mots clés

Gestion des événements

__event

__hook

__unhook

Extensions de composants pour .NET et UWP

Mot clé `__unhook`

Article • 16/06/2023

Dissocie une méthode de gestionnaire d'un événement.

> ⓘ **Notes**
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Syntaxe

C++

```
long  __unhook(
   &SourceClass::EventMethod,
   source,
   &ReceiverClass::HandlerMethod
   [, receiver = this]
);

long  __unhook(
   interface,
   source
);

long  __unhook(
   source
);
```

Paramètres

`&SourceClass::EventMethod`
Pointeur vers la méthode d'événement à partir de laquelle vous décrochez la méthode de gestionnaire d'événements :

- Événements C++ natifs : *`SourceClass`* est la classe source d'événements et *`EventMethod`* est l'événement.
- Événements COM : *`SourceClass`* est l'interface source d'événement et *`EventMethod`* est l'une de ses méthodes.

- Événements managés : *`SourceClass`* est la classe source d'événements et *`EventMethod`* est l'événement.

`interface`
Nom de l'interface en cours de décochage du *récepteur*, uniquement pour les récepteurs d'événements COM dans lesquels le paramètre *layout_dependent* de l'attribut event_receiver est **`true`**.

`source`
Pointeur vers une instance de la source d'événement. Selon le code `type` spécifié dans `event_receiver`, *la source* peut être l'un des types suivants :

- Un pointeur d'objet source de l'événement natif.
- Pointeur `IUnknown` basé sur (source COM).
- Un pointeur d'objet managé (pour les événements managés).

`&ReceiverClass::HandlerMethod` Pointeur vers la méthode de gestionnaire d'événements à détacher d'un événement. Le gestionnaire est spécifié en tant que méthode d'une classe ou référence à celle-ci ; si vous ne spécifiez pas le nom de la classe, **`__unhook`** suppose que la classe est celle dans laquelle elle est appelée.

- Événements C++ natifs : *`ReceiverClass`* est la classe de récepteur d'événements et `HandlerMethod` est le gestionnaire.
- Événements COM : *`ReceiverClass`* est l'interface du récepteur d'événements et *`HandlerMethod`* est l'un de ses gestionnaires.
- Événements managés : *`ReceiverClass`* est la classe de récepteur d'événements et *`HandlerMethod`* est le gestionnaire.

`receiver`(facultatif) Pointeur vers un instance de la classe du récepteur d'événements. Si vous ne spécifiez pas de récepteur, la valeur par défaut est la classe ou la structure de récepteur dans laquelle **`__unhook`** est appelée.

Usage

Peut être utilisé dans n'importe quelle étendue de fonction, y compris `main`, en dehors de la classe du récepteur d'événements.

Notes

Utilisez la fonction `__unhook` intrinsèque dans un récepteur d'événements pour dissocier ou « détacher » une méthode de gestionnaire d'une méthode d'événement.

Il existe trois formes de `__unhook`. Vous pouvez utiliser la première forme (à quatre arguments) dans la plupart des cas. Vous pouvez utiliser la deuxième forme (à deux arguments) de `__unhook` uniquement pour un récepteur d'événements COM ; il décochait l'ensemble de l'interface d'événement. Vous pouvez utiliser la troisième forme (un argument) pour déconnecter tous les délégués de la source spécifiée.

Une valeur de retour différente de zéro indique qu'une erreur s'est produite (les événements managés lèvent une exception).

Si vous appelez `__unhook` sur un gestionnaire d'événements et d'événements qui ne sont pas déjà connectés, cela n'aura aucun effet.

Au moment de la compilation, le compilateur vérifie que l'événement existe et effectue la vérification du type de paramètre avec le gestionnaire spécifié.

Vous pouvez appeler `__hook` et `__unhook` en dehors du récepteur d'événements, à l'exception des événements COM.

Une alternative à l'utilisation `__unhook` consiste à utiliser l'opérateur -= .

Pour plus d'informations sur le codage d'événements managés dans la nouvelle syntaxe, consultez événement.

> ⓘ **Notes**
>
> Une classe ou structure modélisée ne peut pas contenir d'événements.

Exemple

Pour obtenir des exemples, consultez Gestion des événements en C++ natif et Gestion des événements dans COM .

Voir aussi

Mots clés
event_source
event_receiver
__event

__hook
__raise

Gestion des événements en mode natif C++

Article • 16/06/2023

Dans la gestion des événements C++ native, vous configurez une source d'événement et un récepteur d'événements à l'aide des attributs event_source et event_receiver , respectivement, en spécifiant `type=native`. Ces attributs permettent aux classes sur lesquelles ils sont appliqués aux événements de déclenchement et de gérer les événements dans un contexte natif non COM.

> ⓘ **Notes**
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Déclaration d'événements

Dans une classe source d'événements, utilisez le __event mot clé sur une déclaration de méthode pour déclarer la méthode en tant qu'événement. Veillez à déclarer la méthode, mais ne la définissez pas. Si c'est le cas, cela génère une erreur de compilateur, car le compilateur définit implicitement la méthode lorsqu'elle est transformée en événement. Les événements natifs peuvent être des méthodes avec zéro, un ou plusieurs paramètres. Le type de retour peut être `void` ou n'importe quel type intégral.

Définition de gestionnaires d'événements

Dans une classe de récepteur d'événements, vous définissez des gestionnaires d'événements. Les gestionnaires d'événements sont des méthodes avec des signatures (types de retour, conventions d'appel et arguments) qui correspondent à l'événement qu'ils vont gérer.

Raccordement de gestionnaires d'événements à des événements

Dans une classe de récepteur d'événements, vous utilisez également la fonction __hook intrinsèque pour associer des événements à des gestionnaires d'événements et __unhook pour dissocier les événements des gestionnaires d'événements. Vous pouvez

raccorder plusieurs événements à un gestionnaire d'événements, ou plusieurs gestionnaires d'événements à un événement.

Événements de déclenchement

Pour déclencher un événement, appelez la méthode déclarée en tant qu'événement dans la classe source d'événement. Si des gestionnaires ont été raccordés à l'événement, les gestionnaires sont appelés.

Code d'événement C++ natif

L'exemple suivant montre comment déclencher un événement en C++ natif. Pour compiler et exécuter l'exemple, consultez les commentaires du code. Pour générer le code dans l'IDE Visual Studio, vérifiez que l'option `/permissive-` est désactivée.

Exemple

Code

C++

```
// evh_native.cpp
// compile by using: cl /EHsc /W3 evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
   __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
   void MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
   }

   void MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
   }

   void hookEvent(CSource* pSource) {
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }
```

```
   void unhookEvent(CSource* pSource) {
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};

int main() {
   CSource source;
   CReceiver receiver;

   receiver.hookEvent(&source);
   __raise source.MyEvent(123);
   receiver.unhookEvent(&source);
}
```

Sortie

Output

```
MyHandler2 was called with value 123.
MyHandler1 was called with value 123.
```

Voir aussi

Gestion des événements

Gestion des événements dans COM

Article • 03/04/2023

Dans la gestion des événements COM, vous configurez une source d'événements et un récepteur d'événements à l'aide event_source des attributs etevent_receiver, respectivement, en spécifiant=`type` `com` . Ces attributs injectent du code approprié pour les interfaces personnalisées, de répartition et doubles. Le code injecté permet aux classes attribuées de déclencher des événements et de gérer des événements via des points de connexion COM.

> **Notes**
>
> Les attributs d'événement en C++ natif sont incompatibles avec le C++ standard. Ils ne sont pas compilés lorsque vous spécifiez le **/permissive-** mode de conformité.

Déclaration d'événements

Dans une classe source d'événements, utilisez le __event mot clé sur une déclaration d'interface pour déclarer les méthodes de cette interface en tant qu'événements. Les événements de cette interface sont déclenchés lorsqu'ils sont appelés comme méthodes d'interface. Les méthodes sur les interfaces d'événement peuvent avoir zéro ou plusieurs paramètres (qui doivent toutes être *dans des* paramètres). Le type de retour peut être void ou un type intégral.

Définition de gestionnaires d'événements

Vous définissez des gestionnaires d'événements dans une classe de récepteur d'événements. Les gestionnaires d'événements sont des méthodes avec des signatures (types de retour, conventions d'appel et arguments) qui correspondent à l'événement qu'ils vont gérer. Pour les événements COM, les conventions d'appel n'ont pas besoin de correspondre. Pour plus d'informations, consultez Événements COM dépendant de la disposition ci-dessous .

Raccordement de gestionnaires d'événements à des événements

Dans une classe de récepteur d'événements, vous utilisez également la fonction __hook intrinsèque pour associer des événements à des gestionnaires d'événements et __unhook pour dissocier les événements des gestionnaires d'événements. Vous pouvez raccorder plusieurs événements à un gestionnaire d'événements, ou plusieurs gestionnaires d'événements à un événement.

> ⓘ **Notes**
>
> En général, il existe deux techniques pour permettre à un récepteur d'événements COM d'accéder aux définitions de l'interface de source d'événements. La première, comme indiqué ci-dessous, consiste à partager un fichier d'en-tête commun. La deuxième consiste à utiliser **#import** avec le `embedded_idl` qualificateur d'importation, afin que la bibliothèque de types source d'événements soit écrite dans le fichier .tlh avec le code généré par l'attribut conservé.

Événements de déclenchement

Pour déclencher un événement, appelez une méthode dans l'interface déclarée avec le **`__event`** mot clé dans la classe source d'événement. Si des gestionnaires ont été raccordés à l'événement, les gestionnaires sont appelés.

Code d'événement COM

L'exemple suivant montre comment déclencher un événement dans une classe COM. Pour compiler et exécuter l'exemple, consultez les commentaires du code.

C++

```
// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
   [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
   [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;
```

Puis le serveur :

C++

```
// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-
B8A1CEC98830") ];

[coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
   __event __interface IEvents;

   HRESULT FireEvent() {
      __raise MyEvent(123);
      return S_OK;
   }
};
```

Puis le client :

C++

```
// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
   HRESULT MyHandler1(int nValue) {
      printf_s("MyHandler1 was called with value %d.\n", nValue);
      return S_OK;
   }

   HRESULT MyHandler2(int nValue) {
      printf_s("MyHandler2 was called with value %d.\n", nValue);
      return S_OK;
   }
```

```
   void HookEvent(IEventSource* pSource) {
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }

   void UnhookEvent(IEventSource* pSource) {
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
      __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
   }
};

int main() {
   // Create COM object
   CoInitialize(NULL);
   {
      IEventSource* pSource = 0;
      HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL,
CLSCTX_ALL, __uuidof(IEventSource), (void **) &pSource);
      if (FAILED(hr)) {
         return -1;
      }

      // Create receiver and fire event
      CReceiver receiver;
      receiver.HookEvent(pSource);
      pSource->FireEvent();
      receiver.UnhookEvent(pSource);
   }
   CoUninitialize();
   return 0;
}
```

Output

Output

```
MyHandler1 was called with value 123.
MyHandler2 was called with value 123.
```

Événements COM dépendants de la disposition

La dépendance aux dispositions représente un problème uniquement dans le cadre de la programmation COM. Dans la gestion des événements natifs et managés, les signatures (type de retour, convention d'appel et arguments) des gestionnaires doivent correspondre à leurs événements, mais les noms des gestionnaires n'ont pas besoin de correspondre à leurs événements.

Toutefois, dans la gestion des événements COM, lorsque vous définissez le `layout_dependent` paramètre de `event_receiver` sur **`true`**, la correspondance de nom et de signature est appliquée. Les noms et les signatures des gestionnaires dans le récepteur d'événements et dans les événements crochets doivent correspondre exactement.

Lorsque `layout_dependent` est défini **`false`** sur , la convention d'appel et la classe de stockage (virtuelle, statique, etc.) peuvent être mélangées et mises en correspondance entre la méthode d'événement de déclenchement et les méthodes de hooking (ses délégués). Il est légèrement plus efficace d'avoir `layout_dependent` = **`true`**.

Par exemple, supposons que `IEventSource` est défini pour disposer des méthodes suivantes :

C++

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Supposez que la source d'événement a la forme suivante :

C++

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
   __event __interface IEvents;

   HRESULT FireEvent() {
      MyEvent1(123);
      MyEvent2(123);
      return S_OK;
   }
};
```

Ensuite, dans le récepteur d'événements, tout gestionnaire raccordé à une méthode dans `IEventSource` doit correspondre au nom et à la signature, comme indiqué :

C++

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
   HRESULT MyEvent1(int nValue) {  // name and signature matches MyEvent1
      ...
   }
   HRESULT MyEvent2(E c, char* pc) {  // signature doesn't match MyEvent2
```

```
      ...
   }
   HRESULT MyHandler1(int nValue) {  // name doesn't match MyEvent1 (or 2)
      ...
   }
   void HookEvent(IEventSource* pSource) {
      __hook(IFace, pSource);  // Hooks up all name-matched events
                               // under layout_dependent = true
      __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1);   // valid
      __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2);   // not valid
      __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
   }
};
```

Voir aussi

Gestion des événements

Modificateurs propres à Microsoft

Article • 03/04/2023

Cette section décrit les extensions C++ spécifiques à Microsoft dans les domaines suivants :

- Adressage basé, pratique d'utiliser un pointeur comme base à partir de laquelle d'autres pointeurs peuvent être décalés
- Conventions d'appel de fonction
- Attributs de classe de stockage étendus déclarés avec le mot clé __declspec
- Le mot clé __w64

mots clés spécifiques à Microsoft

Un grand nombre des mots clés spécifiques à Microsoft peuvent être utilisés pour modifier des déclarateurs afin de former des types dérivés. Pour plus d'informations sur les déclarateurs, consultez Déclarateurs.

Mot clé	Signification	Utilisé pour former des types dérivés ?
__based	Le nom qui suit déclare un décalage de 32 bits par rapport à la base 32 bits contenue dans la déclaration.	Oui
__cdecl	Le nom qui suit utilise les conventions de nommage et d'appel du langage C.	Oui
__declspec	Le nom qui suit spécifie un attribut de classe de stockage spécifique à Microsoft.	Non
__fastcall	Le nom qui suit déclare une fonction qui utilise des registres, lorsqu'ils sont disponibles, à la place de la pile, pour transmettre des arguments.	Oui
__Restreindre	Similaire à __declspec(restreindre), mais pour une utilisation sur des variables.	Non
__stdcall	Le nom qui suit spécifie une fonction qui respecte la convention d'appel standard.	Oui

Mot clé	Signification	Utilisé pour former des types dérivés ?
__w64	Marque un type de données comme étant plus grand sur un compilateur 64 bits.	Non
__unaligned	Spécifie qu'un pointeur désignant un type ou d'autres données n'est pas aligné.	Non
__vectorcall	Le nom qui suit déclare une fonction qui utilise des registres, y compris les registres SSE lorsqu'ils sont disponibles, à la place de la pile, pour transmettre des arguments.	Oui

Voir aussi

Informations de référence sur le langage C++

Adressage basé sur

Article • 03/04/2023

Cette section comprend les rubriques suivantes :

- grammaire __based
- Pointeurs basés

Voir aussi

Modificateurs spécifiques à Microsoft

`__based` Grammaire

Article • 16/06/2023

Section spécifique à Microsoft

L'adressage est utile lorsque vous avez besoin d'exercer un contrôle précis sur le segment dans lequel les objets sont alloués (données statiques et dynamiques).

La seule forme d'adressage de base acceptable dans les compilations 32 bits et 64 bits est « basée sur un pointeur » qui définit un type qui contient un déplacement 32 bits ou 64 bits vers une base 32 bits ou 64 bits ou basée sur `void`.

Grammaire

based-range-modifier:

`__based(` *base-expression* `)`

base-expression:

based-variable *based-abstract-declarator* *segment-name* *segment-cast*

based-variable:

identifier

based-abstract-declarator:

abstract-declarator

base-type:

type-name

FIN de la section spécifique à Microsoft

Voir aussi

Pointeurs basés sur

Pointeurs basés sur (C++)

Article • 03/04/2023

Le `__based` mot clé vous permet de déclarer des pointeurs basés sur des pointeurs (pointeurs qui sont des décalages par rapport aux pointeurs existants). Le `__based` mot clé est spécifique à Microsoft.

Syntaxe

```
type __based( base ) declarator
```

Notes

Les pointeurs basés sur des adresses de pointeur sont la seule forme de l'mot clé `__based` valide dans les compilations 32 bits ou 64 bits. Pour le compilateur Microsoft C/C++ 32 bits, un pointeur based est un décalage de 32 bits par rapport à une base de pointeur 32 bits. Une restriction similaire s'applique pour les environnements 64 bits, où un pointeur based est un décalage de 64 bits par rapport à la base 64 bits.

Les pointeurs basés sur pointeurs peuvent par exemple être utilisés pour des identificateurs persistants qui contiennent des pointeurs. Une liste liée composée de pointeurs basés sur un pointeur peut être enregistrée sur le disque, puis rechargée dans un autre emplacement mémoire, et les pointeurs restent valides. Par exemple :

C++

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
   void __based( vpBuffer ) *vpData;
   struct llist_t __based( vpBuffer ) *llNext;
};
```

Le pointeur `vpBuffer` reçoit l'adresse de la mémoire allouée à un point ultérieur dans le programme. La liste liée est déplacée par rapport à la valeur de `vpBuffer`.

ⓘ Notes

Les identificateurs persistants contenant des pointeurs peuvent également être effectués à l'aide **de fichiers mappés en mémoire**.

Lors du déréférencement d'un pointeur based, la base doit être spécifiée explicitement ou connue implicitement via la déclaration.

Pour la compatibilité avec les versions précédentes, **_based** est synonyme de **__based** , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Exemple

Le code suivant illustre comment modifier un pointeur based en modifiant sa base.

C++

```
// based_pointers2.cpp
// compile with: /EHsc
#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
   pBased = &a1[0];
   pBasedPtr pb = 0;

   cout << *pb << endl;
   cout << *(pb+1) << endl;

   pBased = &a2[0];

   cout << *pb << endl;
   cout << *(pb+1) << endl;
}
```

Output

```
1
2
10
11
```

Voir aussi

Mots clés

alloc_text

Conventions d’appel

Article • 16/06/2023

Le compilateur Visual C/C++ fournit diverses conventions pour appeler des fonctions internes et externes. La compréhension de ces différentes approches peut vous aider à déboguer votre programme et à lier votre code avec des routines en langage assembleur.

Les rubriques se rapportant à ce sujet expliquent les différences entre les conventions d’appel, la façon dont les arguments sont passés et la manière dont les valeurs sont retournées par les fonctions. Elles présentent également les appels de fonction naked, une fonctionnalité avancée qui vous permet d’écrire votre propre code de prologue et d’épilogue.

Pour plus d’informations sur les conventions d’appel pour les processeurs x64, consultez Convention d’appel.

Rubriques de cette section

- Conventions de passage d’arguments et d’affectation de noms (`__cdecl`, `__stdcall`, `__fastcall` et autres)
- Exemple d’appel : prototype de fonction et appel
- Utilisation d'appels de fonction naked pour écrire du code personnalisé de prologue/épilogue
- Coprocesseur à virgule flottante et conventions d’appel
- Conventions d’appel obsolètes

Voir aussi

Modificateurs spécifiques à Microsoft

Passage des arguments et conventions de dénomination

Article • 16/06/2023

Section spécifique à Microsoft

Les compilateurs Microsoft C++ vous permettent de spécifier des conventions pour passer des arguments et retourner des valeurs entre les fonctions et les appelants. Les conventions ne sont pas toutes disponibles sur toutes les plateformes prises en charge et certaines conventions utilisent des implémentations spécifiques à la plateforme. Dans la plupart des cas, les mots clés ou les commutateurs de compilation qui spécifient une convention non prise en charge sur une plateforme spécifique sont ignorés et la convention par défaut de la plateforme est utilisée.

Sur les plateformes x86, tous les arguments sont étendus à 32 bits lorsqu'ils sont passés. Les valeurs de retour sont également élargies à 32 bits et retournées dans le registre EAX, sauf pour les structures de 8 octets, qui sont retournées dans la paire de registres EDX:EAX. Les structures plus grandes sont retournées dans le registre EAX comme pointeurs vers des structures de retour masquées. Les paramètres font l'objet d'un push sur la pile de droite à gauche. Les structures qui ne sont pas des POD ne sont pas retournées dans les registres.

Le compilateur génère du code de prologue et d'épilogue pour enregistrer et restaurer les registres ESI, EDI, EBX et EBP, s'ils sont utilisés dans la fonction.

> **Notes**
>
> Lorsqu'un struct, une union ou une classe est retourné(e) à partir d'une fonction par valeur, toutes les définitions du type doivent être identiques, sinon le programme peut échouer au moment de l'exécution.

Pour plus d'informations sur la définition de votre propre code prolog et épilogue de fonction, consultez Appels de fonction nus.

Pour plus d'informations sur les conventions d'appel par défaut dans le code qui cible les plateformes x64, consultez Convention d'appel x64. Pour plus d'informations sur les problèmes de convention d'appel dans le code qui cible les plateformes ARM, consultez Problèmes courants de migration ARM dans Visual C++.

Les conventions d'appel suivantes sont prises en charge par le compilateur Visual C/C++.

Mot clé	Nettoyage de pile	Passage de paramètres
__cdecl	Appelant	Effectue un push des paramètres sur la pile, dans l'ordre inverse (de droite à gauche)
__clrcall	n/a	Charger les paramètres sur la pile d'expression CLR dans l'ordre (de gauche à droite).
__stdcall	Appelé	Effectue un push des paramètres sur la pile, dans l'ordre inverse (de droite à gauche)
__fastcall	Appelé	Stocké dans les registres, puis fait l'objet d'un push sur la pile
__thiscall	Appelé	Poussée sur la pile ; `this` pointeur stocké dans ECX
__vectorcall	Appelé	Stocké dans les registres, puis fait l'objet d'un push sur la pile dans l'ordre inverse (de droite à gauche)

Pour plus d'informations, consultez Conventions d'appel obsolètes.

FIN de la section spécifique à Microsoft

Voir aussi

Conventions d'appel

__cdecl

Article • 03/04/2023

`__cdecl` est la convention d'appel par défaut pour les programmes C et C++. Étant donné que la pile est nettoyée par l'appelant, elle peut effectuer des `vararg` fonctions. La `__cdecl` convention d'appel crée des exécutables plus volumineux que __stdcall, car elle nécessite que chaque appel de fonction inclue du code de nettoyage de la pile. La liste suivante illustre l'implémentation de cette convention d'appel. Le `__cdecl` modificateur est spécifique à Microsoft.

Élément	Implémentation
Ordre de transmission des arguments	De droite à gauche
Responsabilité de la maintenance de la pile	La fonction appelante enlève les arguments de la pile.
Convention de décoration de nom	Le trait de soulignement (_) est préfixé aux noms, sauf lorsque des fonctions __cdecl qui utilisent une liaison C sont exportées.
Convention de conversion de casse	Aucune conversion de casse n'est effectuée.

> ⓘ Notes
>
> Pour plus d'informations, consultez Noms décorés.

Placez le `__cdecl` modificateur avant une variable ou un nom de fonction. Étant donné que les conventions de nommage et d'appel C sont les conventions par défaut, la seule fois que vous devez utiliser `__cdecl` dans le code x86 est lorsque vous avez spécifié l'option `/Gv` du compilateur (vectorcall), `/Gz` (stdcall) ou `/Gr` (fastcall). L'option du compilateur /Gd force la convention d'appel `__cdecl` .

Sur les processeurs ARM et x64, `__cdecl` est accepté mais généralement ignoré par le compilateur. Par convention sur ARM et x64, les arguments sont passés dans les registres le cas échéant, et les arguments suivants sont passés sur la pile. Dans le code x64, utilisez `__cdecl` pour remplacer l'option du compilateur **/Gv** et utiliser la convention d'appel x64 par défaut.

Pour les fonctions de classe non statiques, si la fonction est définie hors ligne, il n'est pas nécessaire de spécifier le modificateur de convention d'appel dans la définition hors

ligne. En d'autres termes, pour les méthodes membres non statiques de classe, la convention d'appel spécifiée dans le cadre de la déclaration est utilisée par défaut au stade de la définition. Compte tenu de la définition de classe suivante :

```
C++
```

```
struct CMyClass {
   void __cdecl mymethod();
};
```

le code suivant :

```
C++
```

```
void CMyClass::mymethod() { return; }
```

équivaut au code :

```
C++
```

```
void __cdecl CMyClass::mymethod() { return; }
```

Pour la compatibilité avec les versions précédentes, **cdecl** et **_cdecl** sont synonymes de **`__cdecl`** , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Exemple

Dans l'exemple suivant, le compilateur est chargé d'utiliser les conventions de nommage et d'appel du langage C pour la fonction `system`.

```
C++
```

```
// Example of the __cdecl keyword on function
int __cdecl system(const char *);
// Example of the __cdecl keyword on function pointer
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD
flags, ...);
```

Voir aussi

Conventions de passage d’arguments et d’affectation de noms
Mots clés

__clrcall

Article • 16/06/2023

Spécifie qu'une fonction peut uniquement être appelée à partir du code managé. Utilisez **__clrcall** pour toutes les fonctions virtuelles qui seront appelées uniquement à partir du code managé. Toutefois, cette convention d'appel ne peut pas être utilisée pour les fonctions qui sont appelées à partir du code natif. Le modificateur **__clrcall** est spécifique à Microsoft.

Utilisez **__clrcall** pour améliorer les performances lors de l'appel d'une fonction managée à une fonction managée virtuelle ou d'une fonction managée à une fonction managée via un pointeur.

Les points d'entrée sont des fonctions séparées et générées par le compilateur. Si une fonction a des points d'entrée natifs et managés, l'un d'eux est la fonction réelle avec l'implémentation de fonction. L'autre fonction est une fonction séparée (une conversion de code) qui appelle la fonction réelle et qui laisse le Common Langage Runtime effectuer PInvoke. Lorsque vous marquez une fonction comme **__clrcall**, vous indiquez que l'implémentation de la fonction doit être MSIL et que la fonction de point d'entrée native ne sera pas générée.

Lorsque vous prenez l'adresse d'une fonction native si **__clrcall** n'est pas spécifié, le compilateur utilise le point d'entrée natif. **__clrcall** indique que la fonction est gérée et qu'il n'est pas nécessaire de passer du mode managé au mode natif. Dans ce cas, le compilateur utilise le point d'entrée managé.

Lorsque `/clr` (not `/clr:pure` ou `/clr:safe`) est utilisé et **__clrcall** n'est pas utilisé, la prise de l'adresse d'une fonction retourne toujours l'adresse de la fonction de point d'entrée native. Lorsque **__clrcall** est utilisé, la fonction de point d'entrée native n'est pas créée. Vous obtenez donc l'adresse de la fonction managée, et non une fonction thunk de point d'entrée. Pour plus d'informations, consultez Double thunking. Les options du compilateur **/clr:pure** et **/clr:safe** sont déconseillées dans Visual Studio 2015 et non prises en charge dans Visual Studio 2017.

/clr (Common Language Runtime Compilation) implique que toutes les fonctions et pointeurs de fonction sont **__clrcall** et que le compilateur n'autorise pas la marque d'une fonction à l'intérieur de la compilation et d'être marquée autre chose que **__clrcall**. Lorsque **/clr:pure** est utilisé, **__clrcall** ne peut être spécifié que sur les pointeurs de fonction et les déclarations externes.

Vous pouvez appeler directement **__clrcall** fonctions à partir du code C++ existant qui a été compilé à l'aide de **/clr** tant que cette fonction a une implémentation MSIL. **__clrcall**

fonctions ne peuvent pas être appelées directement à partir de fonctions qui ont inline asm et appellent des intrinisics spécifiques au processeur, par exemple, même si ces fonctions sont compilées avec `/clr`.

__clrcall pointeurs de fonction sont uniquement destinés à être utilisés dans le domaine d'application dans lequel ils ont été créés. Au lieu de passer **__clrcall** pointeurs de fonction entre les domaines d'application, utilisez CrossAppDomainDelegate. Pour plus d'informations, consultez Domaines d'application et Visual C++.

Exemples

Notez que lorsqu'une fonction est déclarée avec **__clrcall**, le code est généré si nécessaire ; par exemple, quand la fonction est appelée.

C++

```
// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
   Console::WriteLine("in Func1");
   return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the adddress of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf();   // comment this line and comparison will pass

int main() {
   if (&Func1 == pf)
      Console::WriteLine("&Func1 == pf, comparison succeeds");
   else
      Console::WriteLine("&Func1 != pf, comparison fails");

   // even though comparison fails, stub and function call are correct
   pf();
   Func1();
}
```

Output

```
in Func1
&Func1 != pf, comparison fails
in Func1
in Func1
```

L'exemple suivant montre que vous pouvez définir un pointeur fonction, de sorte que vous indiquez que le pointeur fonction sera uniquement appelé à partir du code managé. Cela permet au compilateur d'appeler directement la fonction managée et d'éviter le point d'entrée natif (double problème de conversion de code).

C++

```
// clrcall3.cpp
// compile with: /clr
void Test() {
   System::Console::WriteLine("in Test");
}

int main() {
   void (*pTest)() = &Test;
   (*pTest)();

   void (__clrcall *pTest2)() = &Test;
   (*pTest2)();
}
```

Voir aussi

Conventions de passage d’arguments et d’affectation de noms
Mots clés

__stdcall

Article • 16/06/2023

La `__stdcall` convention d'appel est utilisée pour appeler des fonctions d'API Win32. L'appelé nettoie la pile, de sorte que le compilateur crée `vararg` des fonctions `__cdecl`. Les fonctions qui utilisent cette convention d'appel requièrent un prototype de fonction. Le `__stdcall` modificateur est spécifique à Microsoft.

Syntaxe

> *type* `__stdcall` de retour *function-name*[`(`*argument-list*`)`]

Notes

La liste suivante illustre l'implémentation de cette convention d'appel.

Élément	Implémentation
Ordre de transmission des arguments	De droite à gauche
Convention de passage d'argument	Par valeur, à moins qu'un type pointeur ou référence soit passé.
Responsabilité de la maintenance de la pile	La fonction appelée enlève ses propres arguments de la pile.
Convention de décoration de nom	Un trait de soulignement (`_`) est précédé du nom. Le nom est suivi du signe au niveau (`@`) suivi du nombre d'octets (en décimal) dans la liste d'arguments. Par conséquent, la fonction déclarée comme `int func( int a, double b )` est décorée comme suit : `_func@12`
Convention de conversion de casse	None

L'option du compilateur /Gz spécifie pour toutes les fonctions qui ne sont pas déclarées `__stdcall` explicitement avec une autre convention d'appel.

Pour la compatibilité avec les versions précédentes, `_stdcall` est un synonyme de `__stdcall` sauf si l'option /Za de compilateur (Désactiver les extensions de langage) est spécifiée.

Les fonctions déclarées à l'aide du `__stdcall` modificateur retournent des valeurs de la même manière que les fonctions déclarées à l'aide de __cdecl.

Sur les processeurs ARM et x64, `__stdcall` est accepté et ignoré par le compilateur ; sur les architectures ARM et x64, par convention, les arguments sont transmis dans les registres lorsque cela est possible, et les arguments suivants sont transmis sur la pile.

Pour les fonctions de classe non statiques, si la fonction est définie hors ligne, il n'est pas nécessaire de spécifier le modificateur de convention d'appel dans la définition hors ligne. En d'autres termes, pour les méthodes membres non statiques de classe, la convention d'appel spécifiée dans le cadre de la déclaration est utilisée par défaut au stade de la définition. Étant donné cette définition de classe,

```
C++
```

```
struct CMyClass {
   void __stdcall mymethod();
};
```

this

```
C++
```

```
void CMyClass::mymethod() { return; }
```

équivaut à ceci

```
C++
```

```
void __stdcall CMyClass::mymethod() { return; }
```

Exemple

Dans l'exemple suivant, utilisez des `__stdcall` résultats dans tous les `WINAPI` types de fonctions gérés comme un appel standard :

```
C++
```

```
// Example of the __stdcall keyword
#define WINAPI __stdcall
// Example of the __stdcall keyword on function pointer
typedef BOOL (__stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD
flags, ...);
```

Voir aussi

Conventions de passage et de nommage des arguments

Mots clés

__fastcall

Article • 16/06/2023

Section spécifique à Microsoft

La `__fastcall` convention d'appel spécifie que les arguments des fonctions doivent être passés dans les registres, lorsque cela est possible. Cette convention d'appel s'applique uniquement à l'architecture x86. La liste suivante illustre l'implémentation de cette convention d'appel.

Élément	Implémentation
Ordre de transmission des arguments	Les deux premiers DWORD ou arguments plus petits qui figurent dans la liste d'arguments de gauche à droite sont transmis dans les registres ECX et EDX ; tous les autres arguments sont transmis sur la pile de droite à gauche.
Responsabilité de la maintenance de la pile	La fonction appelée enlève les arguments de la pile.
Convention de décoration de nom	L'arobase (@) est préfixé aux noms ; un arobase suivi du nombre d'octets (au format décimal) dans la liste de paramètres est suffixé aux noms.
Convention de conversion de casse	Aucune conversion de casse n'est effectuée.

> ⓘ **Notes**
>
> Les futures versions du compilateur peuvent utiliser des registres pour stocker les paramètres.

L'utilisation de l'option de compilateur /Gr entraîne la compilation de chaque fonction dans le module, `__fastcall` sauf si la fonction est déclarée à l'aide d'un attribut en conflit ou si le nom de la fonction est `main`.

La `__fastcall` mot clé est acceptée et ignorée par les compilateurs qui ciblent les architectures ARM et x64 ; sur une puce x64, par convention, les quatre premiers arguments sont passés dans les registres lorsque cela est possible, et des arguments supplémentaires sont transmis sur la pile. Pour plus d'informations, consultez

Convention d'appel x64. Sur un processeur ARM, jusqu'à quatre arguments entiers et huit arguments à virgule flottante peuvent être transmis dans les registres, et des arguments supplémentaires sont transmis sur la pile.

Pour les fonctions de classe non statiques, si la fonction est définie hors ligne, il n'est pas nécessaire de spécifier le modificateur de convention d'appel dans la définition hors ligne. En d'autres termes, pour les méthodes membres non statiques de classe, la convention d'appel spécifiée dans le cadre de la déclaration est utilisée par défaut au stade de la définition. Compte tenu de la définition de classe suivante :

```
C++

struct CMyClass {
   void __fastcall mymethod();
};
```

le code suivant :

```
C++

void CMyClass::mymethod() { return; }
```

équivaut au code :

```
C++

void __fastcall CMyClass::mymethod() { return; }
```

Pour la compatibilité avec les versions précédentes, **_fastcall** est synonyme de `__fastcall` , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Exemple

Dans l'exemple suivant, la fonction `DeleteAggrWrapper` correspond aux arguments transmis dans les registres :

```
C++

// Example of the __fastcall keyword
#define FASTCALL    __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __ fastcall keyword on function pointer
```

```
typedef BOOL (__fastcall *funcname_ptr)(void * arg1, const char * arg2,
DWORD flags, ...);
```

FIN de la section spécifique à Microsoft

Voir aussi

Conventions de passage d'arguments et d'affectation de noms

Mots clés

`__thiscall`

Article • 03/04/2023

La convention **d'appel spécifique à** `__thiscall` Microsoft est utilisée sur les fonctions membres de la classe C++ sur l'architecture x86. Il s'agit de la convention d'appel par défaut utilisée par les fonctions membres qui n'utilisent pas d'arguments variables (`vararg` fonctions).

Sous `__thiscall`, l'appelé nettoie la pile, ce qui est impossible pour les `vararg` fonctions. Les arguments sont poussés sur la pile de droite à gauche. Le `this` pointeur est transmis via le registre ECX et non sur la pile.

Sur les machines ARM, ARM64 et x64, `__thiscall` est accepté et ignoré par le compilateur. Cela est dû au fait qu'ils utilisent une convention d'appel basée sur le registre par défaut.

Une raison d'utilisation `__thiscall` est dans les classes dont les fonctions membres utilisent `__clrcall` par défaut. Dans ce cas, vous pouvez utiliser `__thiscall` pour rendre des fonctions membres individuelles pouvant être appelées à partir de code natif.

Lors de la compilation avec /clr:pure, toutes les fonctions et pointeurs de fonction sont `__clrcall` sauf indication contraire. Les `/clr:pure` options du compilateur et `/clr:safe` sont déconseillées dans Visual Studio 2015 et non prises en charge dans Visual Studio 2017.

`vararg` les fonctions membres utilisent la convention d'appel `__cdecl` . Tous les arguments de fonction sont envoyés sur la pile, le `this` pointeur étant placé en dernier sur la pile.

Étant donné que cette convention d'appel s'applique uniquement à C++, elle n'a pas de schéma de décoration de nom C.

Lorsque vous définissez une fonction membre de classe non statique hors ligne, spécifiez le modificateur de convention d'appel uniquement dans la déclaration. Vous n'avez pas besoin de le spécifier à nouveau sur la définition hors ligne. Le compilateur utilise la convention d'appel spécifiée lors de la déclaration au point de définition.

Voir aussi

Passage des arguments et conventions de nommage

__vectorcall

Article • 16/06/2023

Section spécifique à Microsoft

La `__vectorcall` convention d'appel spécifie que les arguments aux fonctions doivent être transmis dans des registres, lorsque cela est possible. `__vectorcall` utilise plus de registres pour les arguments que __fastcall ou la convention d'appel x64 par défaut. La convention d'appel `__vectorcall` est prise en charge uniquement dans le code natif sur les processeurs x86 et x64 qui incluent les extensions Streaming SIMD 2 (SSE2) et versions ultérieures. Permet `__vectorcall` d'accélérer les fonctions qui passent plusieurs arguments vectoriels SIMD ou à virgule flottante et d'effectuer des opérations qui tirent parti des arguments chargés dans les registres. La liste suivante présente les fonctionnalités communes aux implémentations x86 et x64 de `__vectorcall`. Les différences sont expliquées plus loin dans cet article.

Élément	Implémentation
Convention de décoration de nom C	Deux signes « arobase » (@@) sont ajoutés en suffixe aux noms de fonctions, suivis du nombre d'octets (au format décimal) dans la liste de paramètres.
Convention de conversion de casse	Aucune conversion de casse n'a lieu.

L'utilisation de l'option /Gv du compilateur entraîne la compilation de chaque fonction dans le module, `__vectorcall` sauf si la fonction est une fonction membre, si elle est déclarée avec un attribut de convention d'appel en conflit, utilise une liste d'arguments `vararg` de variable ou a le nom `main`.

Vous pouvez passer trois types d'arguments en vous inscrivant dans `__vectorcall` des fonctions : des valeurs de *type entier* , des valeurs de *type vecteur* et des valeurs *d'agrégation vectorielle homogène* (HVA).

Un type entier répond à deux conditions : il s'adapte à la taille de registre native du processeur (par exemple, 4 octets sur un ordinateur x86 ou 8 octets sur un ordinateur x64) et il est convertible en entier de longueur de registre et de retour sans modifier sa représentation de bits. Par exemple, tout type qui peut être promu `int` sur x86 (sur x64) (`long long` par exemple, un `char` ou `short`) ou qui peut être converti en `int` (`long long` sur x64) et revenir à son type d'origine sans modification est un type entier. Les types entiers incluent les types pointeur, référence et `struct` ou `union` de 4 octets (8

octets sur x64) ou moins. Sur les plateformes x64, les types et `union` les types plus volumineux `struct` sont passés par référence à la mémoire allouée par l'appelant ; sur les plateformes x86, ils sont transmis par valeur sur la pile.

Un type vectoriel est soit un type à virgule flottante, par exemple, un `float` ou `double`, soit un type de vecteur SIMD, par exemple, `__m128` ou `__m256`.

Un type HVA est un type composite comprenant jusqu'à quatre données membres ayant des types vectoriels identiques. Un type HVA doit respecter la même exigence d'alignement que le type vectoriel de ses membres. Il s'agit d'un exemple de définition HVA `struct` qui contient trois types de vecteurs identiques et a un alignement de 32 octets :

C++

```
typedef struct {
   __m256 x;
   __m256 y;
   __m256 z;
} hva3;    // 3 element HVA type on __m256
```

Déclarez explicitement vos fonctions avec le `__vectorcall` mot clé dans les fichiers d'en-tête pour permettre au code compilé séparément de se lier sans erreur. Les fonctions doivent être prototypes pour utiliser `__vectorcall`, et ne peuvent pas utiliser une liste d'arguments `vararg` de longueur variable.

Une fonction membre peut être déclarée à l'aide du `__vectorcall` spécificateur. Le pointeur masqué `this` est passé par registre comme premier argument de type entier.

Sur les machines ARM, `__vectorcall` est accepté et ignoré par le compilateur. Sur ARM64EC, `__vectorcall` n'est pas pris en charge et rejeté par le compilateur.

Pour les fonctions membres de classe non statiques, si la fonction est définie hors ligne, il n'est pas nécessaire de spécifier le modificateur de convention d'appel dans la définition hors ligne. En d'autres termes, pour les membres non statiques d'une classe, la convention d'appel spécifiée pendant la déclaration est prise par défaut au point de définition. Compte tenu de la définition de classe suivante :

C++

```
struct MyClass {
   void __vectorcall mymethod();
};
```

le code suivant :

```
C++

void MyClass::mymethod() { return; }
```

équivaut au code :

```
C++

void __vectorcall MyClass::mymethod() { return; }
```

Le `__vectorcall` modificateur de convention d'appel doit être spécifié lors de la création d'un pointeur vers une `__vectorcall` fonction. L'exemple suivant crée un `typedef` pour un pointeur vers une `__vectorcall` fonction qui prend quatre `double` arguments et retourne une `__m256` valeur :

```
C++

typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

Pour la compatibilité avec les versions précédentes, `_vectorcall` est un synonyme de `__vectorcall` sauf si l'option /Za de compilateur (Désactiver les extensions de langage) est spécifiée.

Convention __vectorcall sur x64

La `__vectorcall` convention d'appel sur x64 étend la convention d'appel x64 standard pour tirer parti des registres supplémentaires. Les arguments de type entier et les arguments de type vectoriel sont mappés aux registres en fonction de leur position dans la liste d'arguments. Les arguments HVA sont alloués aux registres vectoriels inutilisés.

Si certains des quatre premiers arguments, de gauche à droite, sont des arguments de type entier, ils sont passés dans le registre correspondant à cette position : RCX, RDX, R8 ou R9. Un pointeur masqué `this` est traité comme le premier argument de type entier. Lorsqu'un argument HVA dans l'un des quatre premiers arguments ne peut pas être passé dans les registres disponibles, une référence à la mémoire allouée à l'appelant est passée dans le registre de type entier correspondant à la place. Les arguments de type entier après la quatrième position de paramètre sont passés sur la pile.

Si certains des six premiers arguments, de gauche à droite, sont des arguments de type vectoriel, ils sont passés par valeur dans les registres vectoriels SSE 0 à 5, selon leur position. Les types à virgule flottante et `__m128` sont transmis dans les registres XMM, et `__m256` les types sont transmis dans les registres YMM. Cela diffère de la convention d'appel x64 standard, car les types vectoriels sont passés par valeur et non par référence, et des registres supplémentaires sont utilisés. L'espace de pile d'ombres alloué pour les arguments de type vectoriel est fixe à 8 octets et l'option /homeparams ne s'applique pas. Les arguments de type vectoriel à partir de la septième position de paramètre sont passés sur la pile par référence à la mémoire allouée par l'appelant.

Une fois les registres alloués pour les arguments vectoriels, les membres de données des arguments HVA sont alloués, dans l'ordre croissant, aux registres vectoriels non utilisés XMM0 à XMM5 (ou YMM0 à YMM5, pour `__m256` les types), tant qu'il y a suffisamment de registres disponibles pour l'ensemble du HVA. Si un nombre insuffisant de registres sont disponibles, l'argument HVA est passé par référence à la mémoire allouée par l'appelant. L'espace de pile cachée pour un argument HVA est fixé à 8 octets avec un contenu non défini. Les arguments HVA sont assignés aux registres dans l'ordre, de gauche à droite, dans la liste de paramètres, et peuvent être dans n'importe quelle position. Tout argument HVA dans l'une des quatre premières positions d'argument qui n'est pas assigné aux registres vectoriels est passé par référence dans le registre d'entiers correspondant à sa position. Les arguments HVA passés par référence après la quatrième position de paramètre font l'objet d'un push sur la pile.

Les résultats des `__vectorcall` fonctions sont retournés par valeur dans les registres lorsque cela est possible. Les résultats de type entier, y compris les structs et les unions de 8 octets ou moins, sont retournés par valeur dans RAX. Les résultats de type vectoriel sont retournés par valeur dans XMM0 ou YMM0, selon la taille. Les résultats HVA comportent chacun un élément de données retourné par valeur dans les registres XMM0:XMM3 ou YMM0:YMM3, selon la taille de l'élément. Les types de résultats qui ne tiennent pas dans les registres correspondants sont retournés par référence à la mémoire allouée par l'appelant.

La pile est gérée par l'appelant dans l'implémentation x64 de `__vectorcall`. Le code de prologue et d'épilogue de l'appelant alloue et efface la pile pour la fonction appelée. Les arguments font l'objet d'un push sur la pile, de droite à gauche, et l'espace de pile caché est alloué pour les arguments passés dans les registres.

Exemples :

C++

```
// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
```

```
// This example creates an annotated assembly listing in
// crt_vc64.asm.

#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
   __m128 array[2];
} hva2;    // 2 element HVA type on __m128

typedef struct {
   __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
   return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
   return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
// Passes b by element in [XMM0:XMM1];
// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
   return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0,YMM2,YMM4,YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
   return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
```

```
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
   return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
   return b;
}

int __cdecl main( void )
{
   hva4 h4;
   hva2 h2;
   int i;
   float f;
   __m128 a, b, d;
   __m256 c, e;

   a = b = d = _mm_set1_ps(3.0f);
   c = e = _mm256_set1_ps(5.0f);
   h2.array[0] = _mm_set1_ps(6.0f);
   h4.array[0] = _mm256_set1_ps(7.0f);

   b = example1(a, b, c, d, e);
   e = example2(1, b, 3, d, e, 6.0f, 7);
   d = example3(1, h2, 3, 4, 5);
   f = example4(1, 2.0f, h4, d, 5);
   i = example5(1, h2, 3, h4, 5);
   h4 = example6(h2, h4, c, h2);
}
```

Convention __vectorcall sur x86

La `__vectorcall` convention d'appel suit la `__fastcall` convention des arguments de type entier 32 bits et tire parti des registres de vecteurs SSE pour les arguments de type vecteur et HVA.

Les deux premiers arguments de type entier trouvés dans la liste de paramètres, de gauche à droite, sont placés dans ECX et EDX, respectivement. Un pointeur masqué `this` est traité comme le premier argument de type entier et est passé dans ECX. Les six premiers arguments de type vecteur sont passés par valeur via les registres vectoriels SSE 0 à 5, dans les registres XMM ou YMM, selon la taille des arguments.

Les six premiers arguments de type vecteur, dans l'ordre, de gauche à droite, sont passés par valeur dans les registres vectoriels SSE 0 à 5. Les types à virgule flottante et

`__m128` sont transmis dans les registres XMM, et `__m256` les types sont transmis dans les registres YMM. Aucun espace de pile cachée n'est alloué pour les arguments de type vecteur passés par registre. Le septième argument de type vecteur et les suivants sont passés sur la pile par référence à la mémoire allouée par l'appelant. La limitation de l'erreur du compilateur C2719 ne s'applique pas à ces arguments.

Une fois les registres alloués pour les arguments vectoriels, les membres de données des arguments HVA sont alloués par ordre croissant aux registres vectoriels non utilisés XMM0 à XMM5 (ou YMM0 à YMM5, pour `__m256` les types), tant qu'il y a suffisamment de registres disponibles pour l'ensemble du HVA. Si un nombre insuffisant de registres sont disponibles, l'argument HVA est passé sur la pile par référence à la mémoire allouée par l'appelant. Aucun espace de pile cachée n'est alloué pour un argument HVA. Les arguments HVA sont assignés aux registres dans l'ordre, de gauche à droite, dans la liste de paramètres, et peuvent être dans n'importe quelle position.

Les résultats des `__vectorcall` fonctions sont retournés par valeur dans les registres lorsque cela est possible. Les résultats de type entier, y compris les structs et les unions de 4 octets ou moins, sont retournés par valeur dans EAX. Les structs et les unions de type entier de 8 octets ou moins sont retournés par valeur dans EDX:EAX. Les résultats de type vectoriel sont retournés par valeur dans XMM0 ou YMM0, selon la taille. Les résultats HVA comportent chacun un élément de données retourné par valeur dans les registres XMM0:XMM3 ou YMM0:YMM3, selon la taille de l'élément. Les autres types de résultats sont retournés par référence à la mémoire allouée par l'appelant.

L'implémentation x86 de `__vectorcall` suit la convention des arguments envoyés sur la pile de droite à gauche par l'appelant, et la fonction appelée efface la pile juste avant son retour. Seuls les arguments qui ne sont pas placés dans les registres font l'objet d'un push sur la pile.

Exemples :

C++

```
// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmintrin.h>

typedef struct {
   __m128 array[2];
} hva2;    // 2 element HVA type on __m128
```

```
typedef struct {
   __m256 array[4];
} hva4;    // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
   return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
   return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
   return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
   return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
   return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
   return b;
}

int __cdecl main( void )
```

```
{
   hva4 h4;
   hva2 h2;
   int i;
   float f;
   __m128 a, b, d;
   __m256 c, e;

   a = b = d = _mm_set1_ps(3.0f);
   c = e = _mm256_set1_ps(5.0f);
   h2.array[0] = _mm_set1_ps(6.0f);
   h4.array[0] = _mm256_set1_ps(7.0f);

   b = example1(a, b, c, d, e);
   e = example2(1, b, 3, d, e, 6.0f, 7);
   d = example3(1, h2, 3, 4, 5);
   f = example4(1, 2.0f, h4, d, 5);
   i = example5(1, h2, 3, h4, 5);
   h4 = example6(h2, h4, c, h2);
}
```

Mettre fin à Microsoft Spécifique

Voir aussi

Conventions de passage et de nommage des arguments

Mots clés

Exemple d'appel : Prototype et appel de fonction

Article • 16/06/2023

Section spécifique à Microsoft

L'exemple suivant montre les résultats d'un appel de fonction effectué à l'aide de différentes conventions d'appel.

Cet exemple repose sur la structure de fonction suivante. Remplacez `calltype` par la convention d'appel appropriée.

C++

```
void    calltype MyFunc( char c, short s, int i, double f );
.
.
.
void    MyFunc( char c, short s, int i, double f )
    {
    .
    .
    .
    }
.
.
.
MyFunc ('x', 12, 8192, 2.7183);
```

Pour plus d'informations, consultez Résultats de l'exemple d'appel.

FIN de la section spécifique à Microsoft

Voir aussi

Conventions d'appel

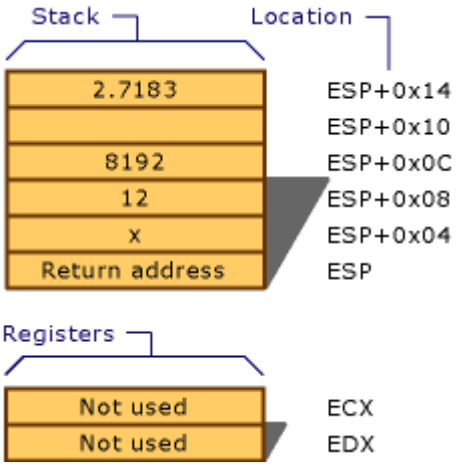
Exemple de résultats de l'appel

Article • 16/06/2023

Section spécifique à Microsoft

`__cdecl`

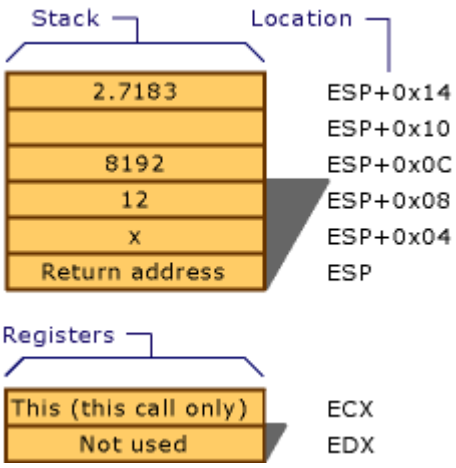
Le nom de la fonction décorée en C est `_MyFunc`.



Convention d'appel `__cdecl`

`__stdcall` et `thiscall`

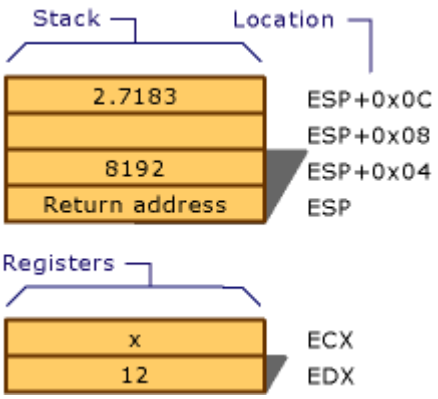
Le nom décoré en C (`__stdcall`) est `_MyFunc@20`. Le nom décoré C++ est spécifique à l'implémentation.



Conventions d'appel __stdcall et thiscall

`__fastcall`

Le nom décoré en C (`__fastcall`) est `@MyFunc@20`. Le nom décoré C++ est spécifique à l'implémentation.



Convention d'appel `__fastcall`

FIN de la section spécifique à Microsoft

Voir aussi

Exemple d'appel : prototype de fonction et appel

Appels de fonction naked

Article • 16/06/2023

Section spécifique à Microsoft

Les fonctions déclarées avec l'attribut `naked` sont émises sans code prolog ou épilogue, ce qui vous permet d'écrire vos propres séquences de prolog/épilogue personnalisées à l'aide de l'assembleur inline. Les fonctions naked sont fournies en tant que fonctionnalité avancée. Elles vous permettent de déclarer une fonction appelée à partir d'un contexte autre que C/C++, et ainsi de faire des hypothèses différentes quant à l'emplacement où se trouvent les paramètres ou sur les registres qui sont conservés. Il s'agit par exemple de routines telles que les gestionnaires d'interruptions. Cette fonctionnalité est particulièrement utile pour les rédacteurs de pilotes de périphériques virtuels (VxDs).

Sur quels éléments souhaitez-vous obtenir des informations supplémentaires ?

- naked
- Règles et limitations pour les fonctions nues
- Considérations relatives à l'écriture de code Prolog/Epilog

FIN de la section spécifique à Microsoft

Voir aussi

Conventions d'appel

Règles et limitations concernant les fonctions naked

Article • 16/06/2023

Section spécifique à Microsoft

Les règles et les limitations suivantes s'appliquent aux fonctions naked :

- L'instruction `return` n'est pas autorisée.
- Les constructions de gestion structurée des exceptions et de gestion des exceptions C++ ne sont pas autorisées, car elles doivent se dérouler sur le frame de pile.
- Pour la même raison, toute forme de `setjmp` est interdite.
- L'utilisation de la fonction `_alloca` est interdite.
- Pour s'assurer qu'aucun code d'initialisation de variables locales n'apparaît avant la séquence de prologue, les variables locales initialisées ne soient pas autorisées dans la portée de la fonction. En particulier, la déclaration d'objets C++ n'est pas autorisée dans la portée de la fonction. En revanche, il peut y avoir des données initialisées dans une portée imbriquée.
- L'optimisation du pointeur de frame (option du compilateur /Oy) n'est pas recommandée, mais elle est supprimée automatiquement pour une fonction naked.
- Vous ne pouvez pas déclarer d'objets de classe C++ dans la portée lexicale de fonction. Vous pouvez toutefois déclarer des objets dans un bloc imbriqué.
- Le `naked` mot clé est ignoré lors de la compilation avec /clr.
- Pour __fastcall fonctions nues, chaque fois qu'il existe une référence dans le code C/C++ à l'un des arguments register, le code prologue doit stocker les valeurs de ce registre dans l'emplacement de pile de cette variable. Par exemple :

C++

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall  power(int i, int j) {
   // calculates i^j, assumes that j >= 0
```

```
    // prolog
    __asm {
       push ebp
       mov ebp, esp
       sub esp, __LOCAL_SIZE
      // store ECX and EDX into stack locations allocated for i and j
      mov i, ecx
      mov j, edx
    }

    {
       int k = 1;   // return value
       while (j-- > 0)
          k *= i;
       __asm {
          mov eax, k
       };
    }

    // epilog
    __asm {
       mov esp, ebp
       pop ebp
       ret
    }
}
```

FIN de la section spécifique à Microsoft

Voir aussi

Appels de fonction naked

Considérations sur l'écriture de code de prologue/épilogue

Article • 16/06/2023

Section spécifique à Microsoft

Avant d'écrire vos propres séquences de code prolog et épilogue, il est important de comprendre comment le cadre de pile est disposé. Il est également utile de savoir comment utiliser le `__LOCAL_SIZE` symbole.

Disposition du frame de pile

Cet exemple montre le code de prologue standard qui peut apparaître dans une fonction 32 bits :

```
push        ebp                 ; Save ebp
mov         ebp, esp            ; Set stack frame pointer
sub         esp, localbytes     ; Allocate space for locals
push        <registers>         ; Save registers
```

La variable `localbytes` représente le nombre d'octets nécessaires sur la pile pour les variables locales. La variable `<registers>` est un espace réservé qui représente la liste de registres à enregistrer sur la pile. Après avoir effectué un push des registres, vous pouvez placer toutes les autres données pertinentes sur la pile. Voici le code d'épilogue correspondant :

```
pop         <registers>   ; Restore registers
mov         esp, ebp      ; Restore stack pointer
pop         ebp           ; Restore ebp
ret                       ; Return from function
```

La pile se réduit toujours (des adresses de mémoire supérieures aux adresses de mémoire inférieures). Le pointeur de base (`ebp`) pointe vers la valeur de `ebp` qui fait l'objet d'un push. La zone Variables locales commence à `ebp-4`. Pour accéder aux variables locales, calculez un décalage à partir de `ebp` en soustrayant la valeur appropriée de `ebp`.

__LOCAL_SIZE

Le compilateur fournit un symbole, `__LOCAL_SIZE`, à utiliser dans le bloc assembleur inline du code prolog de fonction. Ce symbole est utilisé pour allouer de l'espace pour les variables locales sur le frame de pile dans le code de prologue personnalisé.

Le compilateur détermine la valeur de `__LOCAL_SIZE`. Sa valeur représente le nombre total d'octets de toutes les variables locales définies par l'utilisateur et des variables temporaires générées par le compilateur. `__LOCAL_SIZE` ne peut être utilisé que comme opérande immédiat; il ne peut pas être utilisé dans une expression. Vous ne devez pas modifier ou redéfinir la valeur de ce symbole. Par exemple :

```
mov        eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov        eax, [ebp - __LOCAL_SIZE]   ;Error
```

L'exemple suivant d'une fonction nue contenant des séquences de prologue et d'épilogue personnalisées utilise le `__LOCAL_SIZE` symbole dans la séquence de prolog :

C++

```
// the__local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
   int i;
   int j;

   __asm {      /* prolog */
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */
   __asm {   /* epilog */
      mov      esp, ebp
      pop      ebp
      ret
      }
}
```

FIN de la section spécifique à Microsoft

Voir aussi

Appels de fonction naked

Coprocesseur à virgule flottante et conventions d’appel

Article • 16/06/2023

Si vous écrivez des routines d’assembly pour le coprocesseur à virgule flottante, vous devez conserver le mot de contrôle à virgule flottante et propre la pile du coprocesseur, sauf si vous retournez une `float` valeur ou `double` (que votre fonction doit retourner dans ST(0)).

Voir aussi

Conventions d’appel

Conventions d'appel obsolètes

Article • 16/06/2023

Section spécifique à Microsoft

Les **conventions d’appel __pascal**, **__fortran** et **__syscall** ne sont plus prises en charge. Vous pouvez émuler leurs fonctionnalités en utilisant l’une des conventions d’appel prises en charge et les options appropriées de l’Éditeur de liens.

<windows.h> prend désormais en charge la macro WINAPI, qui se traduit par la convention d’appel appropriée pour la cible. Utilisez WINAPI où vous avez précédemment utilisé PASCAL ou **__far __pascal**.

FIN de la section spécifique à Microsoft

Voir aussi

Conventions de passage d’arguments et d’affectation de noms

restrict (C++ AMP)

Article • 03/04/2023

Le spécificateur de restriction peut être appliqué aux déclarations de fonctions et lambda. Il applique des restrictions au code dans la fonction et sur le comportement de cette fonction dans les applications qui utilisent le runtime C++ Accelerated Massive Parallelism (C++ AMP).

> ⓘ Notes
>
> Pour plus d'informations sur les **`restrict`** mot clé qui font partie des attributs de **`__declspec`** classe de stockage, consultez **restreindre**.

La **`restrict`** clause prend les formes suivantes :

Clause	Description
`restrict(cpu)`	La fonction peut utiliser le langage C++ complet. Seules les autres fonctions déclarées en utilisant des fonctions restrict(cpu) peuvent appeler la fonction.
`restrict(amp)`	La fonction peut uniquement utiliser le sous-ensemble du langage C++ que C++ AMP peut accélérer.
Séquence de clauses `restrict(cpu)` et `restrict(amp)`	La fonction doit respecter les limitations de `restrict(cpu)` et `restrict(amp)`. La fonction peut être appelée par des fonctions déclarées à l'aide `restrict(cpu)` de , `restrict(amp)`, `restrict(cpu, amp)` ou `restrict(amp, cpu)`. L'expression `restrict(A) restrict(B)` peut être écrite `restrict(A,B)`.

Notes

Le **`restrict`** mot clé est une mot clé contextuelle. Les spécificateurs de restrictions, `cpu` et `amp`, ne sont pas des mots réservés. La liste des spécificateurs n'est pas extensible. Une fonction qui n'a pas de **`restrict`** clause est identique à une fonction qui a la `restrict(cpu)` clause.

Une fonction qui possède la clause `restrict(amp)` présente les limitations suivantes :

- La fonction peut appeler uniquement des fonctions qui possèdent la clause `restrict(amp)`.
- La fonction doit être en mesure d'être inline.

- La fonction peut déclarer uniquement `int` les variables , `unsigned int`, `float` et `double` les classes et structures qui contiennent uniquement ces types. `bool` Le type est également autorisé, mais il doit être aligné sur 4 octets si vous l'utilisez dans un type composite.
- Les fonctions lambda ne peuvent pas capturer par référence et ne peuvent pas capturer les pointeurs.
- Les références et les pointeurs à une seule indirection sont pris en charge uniquement en tant que variables locales, arguments de fonction ou types de retour.
- Les éléments suivants ne sont pas autorisés :
  - récurrence,
  - Variables déclarées avec le mot clé volatile.
  - fonctions virtuelles,
  - pointeurs désignant des fonctions,
  - pointeurs désignant des fonctions membres,
  - pointeurs dans des structures,
  - pointeurs désignant des pointeurs,
  - Instructions `goto` .
  - instructions étiquetées,
  - `try`, `catch` ou `throw` instructions.
  - Variables globales.
  - variables statiques Utilisez tile_static mot clé à la place.
  - `dynamic_cast` Jette.
  - opérateur `typeid`,
  - déclarations asm,
  - varargs.

Pour plus d'informations sur les limitations de fonction, consultez restrictions de restriction (amp) .

Exemple

L'exemple suivant montre comment utiliser la `restrict(amp)` clause .

C++

```
void functionAmp() restrict(amp) {}
void functionNonAmp() {}

void callFunctions() restrict(amp)
{
    // int is allowed.
    int x;
    // long long int is not allowed in an amp-restricted function. This
generates a compiler error.
    // long long int y;

    // Calling an amp-restricted function is allowed.
    functionAmp();

    // Calling a non-amp-restricted function is not allowed.
    // functionNonAmp();
}
```

Voir aussi

C++ AMP (C++ Accelerated Massive Parallelism)

tile_static, mot clé

Article • 03/04/2023

Le **tile_static** mot clé est utilisé pour déclarer une variable accessible par tous les threads d'une vignette de threads. La durée de vie des variables commence lorsque l'exécution atteint le point de déclaration et se termine lorsque la fonction noyau est retournée. Pour plus d'informations sur l'utilisation des vignettes, consultez Utilisation de vignettes.

Le **mot clé tile_static** présente les limitations suivantes :

- Il peut être utilisé uniquement sur des variables qui sont dans une fonction ayant le modificateur `restrict(amp)`.
- Il ne peut pas être utilisé sur des variables qui sont des types pointeur ou référence.
- Une variable **tile_static** ne peut pas avoir d'initialiseur. Les constructeurs et les destructeurs par défaut ne sont pas appelés automatiquement.
- La valeur d'une variable **de tile_static** non initialisée n'est pas définie.
- Si une variable **tile_static** est déclarée dans un graphique d'appel qui est enraciné par un appel non mosaïque à `parallel_for_each`, un avertissement est généré et le comportement de la variable n'est pas défini.

Exemple

L'exemple suivant montre comment une variable **tile_static** peut être utilisée pour accumuler des données sur plusieurs threads dans une vignette.

C++

```
// Sample data:
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles:
// 2 2    9 7    1 4
// 4 4    8 8    3 4
//
// 1 5    1 2    5 2
// 6 8    3 2    7 2
```

```
// Averages:
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.extent and divide the extent into 2 x 2
tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is
complete, find the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
      }
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
// Sample data.
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles are:
// 2 2    9 7    1 4
// 4 4    8 8    3 4
//
// 1 5    1 2    5 2
```

```
// 6 8    3 2    7 2

// Averages.
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.grid and divide the grid into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is
complete, find the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
      }
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output.
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
```

Voir aussi

Modificateurs spécifiques à Microsoft

C++ AMP Vue d'ensemble

parallel_for_each, fonction (C++ AMP)

Procédure pas à pas : multiplication matricielle

`__declspec`

Article • 16/06/2023

Section spécifique à Microsoft

La syntaxe d'attribut étendue permettant de spécifier des informations de classe de stockage utilise le **`__declspec`** mot clé, qui spécifie qu'un instance d'un type donné doit être stocké avec un attribut de classe de stockage spécifique à Microsoft répertorié ci-dessous. Parmi les exemples d'autres modificateurs de classe de stockage figurent les mots clés **`static`** et **`extern`**. Toutefois, ces mots clés font partie de la spécification ANSI des langages C et C++, et en tant que tels ne sont pas couverts par la syntaxe d'attribut étendue. La syntaxe à attributs étendus simplifie et normalise les extensions spécifiques à Microsoft pour les langages C et C++.

Grammaire

decl-specifier:

`__declspec (` *extended-decl-modifier-seq* **`)`**

extended-decl-modifier-seq:

extended-decl-modifier$_{\text{opt}}$

extended-decl-modifier *extended-decl-modifier-seq*

extended-decl-modifier:

`align(`*number***`)`**

`allocate("`*segname***`")`**

`allocator`

`appdomain`

`code_seg("`*segname***`")`**

`deprecated`

`dllimport`

`dllexport`

`empty_bases`

`jitintrinsic`

`naked`

`noalias`

`noinline`

`noreturn`

`nothrow`

`novtable`

`no_sanitize_address`

`process`

`property(` { `get=`*get-func-name* | `,put=`*put-func-name* } `)`

`restrict`

`safebuffers`

`selectany`

`spectre(nomitigation)`

`thread`

`uuid("`*ComObjectGUID*`")`

Un espace blanc sépare la séquence de modificateur de déclaration. Des exemples sont donnés dans des sections ultérieures.

La grammaire des attributs étendus prend en charge ces attributs de classe de stockage spécifiques à Microsoft : align, empty_basesrestrictappdomainallocatorallocatecode_segdeprecatednoinlinenoaliasnoreturnnakednothrowjitintrinsicnovtableno_sanitize_addressdllimportdllexportsafebufferselectanyprocessspectreet .thread Il prend également en charge les attributs com-object suivants : property et uuid.

Les `code_seg`attributs , `dllexport` `dllimport`, `empty_bases`, `noalias` `no_sanitize_address` `nothrow` `naked` `restrict` `selectany` `property` `thread`et `uuid` storage-class sont des propriétés uniquement de la déclaration de l'objet ou de la fonction à laquelle ils sont appliqués. L'attribut `thread` affecte uniquement les données et les objets. Les `naked` attributs et `spectre` affectent uniquement les fonctions. Les `dllimport` attributs et `dllexport` affectent les fonctions, les données et les objets. Les `property`attributs , `selectany`et `uuid` affectent les objets COM.

Pour la compatibilité avec les versions précédentes, `_declspec` est un synonyme de `__declspec` l'option /Za de compilateur (Désactiver les extensions de langage) spécifiée.

Les `__declspec` mots clés doivent être placés au début d'une déclaration simple. Le compilateur ignore, sans avertissement, les `__declspec` mots clés placés après * ou & et devant l'identificateur de variable dans une déclaration.

Un `__declspec` attribut spécifié au début d'une déclaration de type défini par l'utilisateur s'applique à la variable de ce type. Par exemple :

```
C++
```

```
__declspec(dllimport) class X {} varX;
```

Dans ce cas, l'attribut s'applique à `varX`. Un `__declspec` attribut placé après ou `class` `struct` mot clé s'applique au type défini par l'utilisateur. Par exemple :

```
C++

class __declspec(dllimport) X {};
```

Dans ce cas, l'attribut s'applique à `X`.

Les instructions générales relatives à l'utilisation de l'attribut `__declspec` pour les déclarations simples sont les suivantes :

decl-specifier-seq *init-declarator-list* ;

Le *decl-specifier-seq* doit contenir, entre autres choses, un type de base (par exemple, , `int` `float`, `typedef` ou un nom de classe), une classe de stockage (par exemple, `static`, `extern`) ou l'extension `__declspec` . le *init-declarator-list* doit contenir, entre autres, la partie pointeur des déclarations. Par exemple :

```
C++

__declspec(selectany) int * pi1 = 0;   //Recommended, selectany & int both
part of decl-specifier
int __declspec(selectany) * pi2 = 0;   //OK, selectany & int both part of
decl-specifier
int * __declspec(selectany) pi3 = 0;   //ERROR, selectany is not part of a
declarator
```

Le code suivant déclare une variable locale de thread entière et lui affecte une valeur initiale :

```
C++

// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
Attributs de classe de stockage étendu C

`align` (C++)

Article • 16/06/2023

Dans Visual Studio 2015 et versions ultérieures, utilisez le spécificateur standard **`alignas`** C++11 pour contrôler l'alignement. Pour plus d'informations, consultez Alignement.

Section spécifique à Microsoft

Utilisez `__declspec(align(#))` pour contrôler avec précision l'alignement des données définies par l'utilisateur (par exemple, les allocations statiques ou les données automatiques dans une fonction).

Syntaxe

> **__declspec(align(**#**))***declarator*

Notes

L'écriture d'applications qui utilisent les dernières instructions du processeur introduit de nouvelles contraintes et de nouveaux problèmes. De nombreuses nouvelles instructions nécessitent des données alignées sur des limites de 16 octets. L'alignement des données fréquemment utilisées sur la taille de ligne de cache du processeur améliore les performances du cache. Par exemple, si vous définissez une structure dont la taille est inférieure à 32 octets, vous pouvez souhaiter un alignement sur 32 octets pour vous assurer que les objets de ce type de structure sont mis en cache efficacement.

est la valeur d'alignement. Les entrées valides sont des puissances entières de deux comprises entre 1 et 8192 (octets), telles que 2, 4, 8, 16, 32 ou 64. `declarator` est les données que vous déclarez comme alignées.

Pour plus d'informations sur la façon de retourner une valeur de type `size_t` qui est l'exigence d'alignement du type, consultez alignof. Pour plus d'informations sur la déclaration de pointeurs non alignés lors du ciblage de processeurs 64 bits, consultez __unaligned.

Vous pouvez utiliser `__declspec(align(#))` lorsque vous définissez un **`struct`**, **`union`** ou **`class`**, ou lorsque vous déclarez une variable.

Le compilateur ne garantit pas ou ne tente pas de conserver l'attribut d'alignement des données pendant une opération de copie ou de transformation de données. Par

exemple, memcpy peut copier un struct déclaré avec dans `__declspec(align(#))` n'importe quel emplacement. Les allocateurs ordinaires (par exemple, malloc, C++ operator newet les allocateurs Win32) retournent généralement une mémoire qui n'est pas suffisamment alignée pour `__declspec(align(#))` les structures ou les tableaux de structures. Pour garantir que la destination d'une opération de copie ou de transformation de données est correctement alignée, utilisez _aligned_malloc. Ou écrivez votre propre allocateur.

Vous ne pouvez pas spécifier l'alignement des paramètres de fonction. Lorsque vous transmettez des données qui ont un attribut d'alignement par valeur sur la pile, leur alignement est contrôlé par la convention d'appel. Si l'alignement des données est important dans la fonction appelée, copiez le paramètre dans la mémoire correctement alignée avant de l'utiliser.

Sans `__declspec(align(#))`, le compilateur aligne généralement les données sur les limites naturelles en fonction du processeur cible et de la taille des données, jusqu'à des limites de 4 octets sur les processeurs 32 bits et des limites de 8 octets sur les processeurs 64 bits. Les données des classes ou structures sont alignées dans la classe ou la structure au minimum de son alignement naturel et du paramètre d'empaquetage actuel (à partir de ou de `#pragma pack` l'option du `/Zp` compilateur).

Cet exemple illustre l'utilisation de `__declspec(align(#))` :

C++

```
__declspec(align(32)) struct Str1{
   int a, b, c, d, e;
};
```

Ce type a maintenant un attribut d'alignement de 32 octets. Cela signifie que toutes les instances statiques et automatiques démarrent sur une limite de 32 octets. Les autres types de structure déclarés avec ce type en tant que membre conservent l'attribut d'alignement de ce type. Autrement dit, toute structure avec `Str1` comme élément a un attribut d'alignement d'au moins 32.

Ici, `sizeof(struct Str1)` est égal à 32. Cela implique que si un tableau d'objets `Str1` est créé et que la base du tableau est alignée sur 32 octets, chaque membre du tableau est également aligné sur 32 octets. Pour créer un tableau dont la base est correctement alignée dans la mémoire dynamique, utilisez _aligned_malloc. Ou écrivez votre propre allocateur.

La valeur **`sizeof`** d'une structure quelconque est le décalage du membre final, plus la taille de ce membre, arrondie au multiple le plus proche de la plus grande valeur

membre d'alignement du membre ou de la valeur d'alignement de la structure entière, quelle que soit la valeur la plus grande.

Le compilateur utilise ces règles pour l'alignement de la structure :

- À moins d'une substitution par `__declspec(align(#))`, l'alignement d'un membre de structure scalaire représente le minimum de sa taille et de la compression actuelle.
- À moins d'une substitution par `__declspec(align(#))`, l'alignement d'une structure est la valeur maximale des alignements de ses membres.
- Un membre de structure est placé à un décalage par rapport au début de sa structure parente qui est le plus petit multiple de son alignement supérieur ou égal au décalage de la fin du membre précédent.
- La taille d'une structure est le plus petit multiple de son plus grand alignement supérieur ou égal au décalage de la fin de son dernier membre.

`__declspec(align(#))` peut uniquement augmenter les restrictions d'alignement.

Pour plus d'informations, consultez les pages suivantes :

- align Exemples
- Définition de nouveaux types avec __declspec(align(#))
- Alignement des données dans le stockage local des threads
- Fonctionnement align de l'empaquetage des données
- Exemples d'alignement de structure x64

`align` Exemples

Les exemples suivants montrent comment `__declspec(align(#))` affecte la taille et l'alignement des structures de données. Les exemples supposent les définitions suivantes :

```
C++

#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

Dans cet exemple, la structure `S1` est définie à l'aide de `__declspec(align(32))`. Toutes les utilisations de `S1` pour une définition de variable ou dans d'autres types de déclaration sont alignées sur 32 octets. `sizeof(struct S1)` retourne 32, et `S1` a

16 octets de remplissage après les 16 octets requis pour contenir les quatre entiers. Chaque `int` membre nécessite un alignement sur 4 octets, mais l'alignement de la structure elle-même est déclaré à 32. L'alignement global est alors de 32.

C++

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
   int a, b, c, d;
};
struct S1 s1;   // s1 is 32-byte cache aligned
```

Dans cet exemple, `sizeof(struct S2)` retourne 16, qui est exactement la somme des tailles membres, car il s'agit d'un multiple de la plus grande exigence d'alignement (un multiple de 8).

C++

```
__declspec(align(8)) struct S2 {
   int a, b, c, d;
};
```

Dans l'exemple suivant, `sizeof(struct S3)` retourne 64.

C++

```
struct S3 {
   struct S1 s1;   // S3 inherits cache alignment requirement
                   // from S1 declaration
   int a;          // a is now cache aligned because of s1
                   // 28 bytes of trailing padding
};
```

Dans cet exemple, notez que `a` a l'alignement de son type naturel, dans le cas présent, 4 octets. Toutefois, `S1` doit être aligné sur 32 octets. 28 octets de remplissage suivent `a`, de sorte que `s1` commence au décalage 32. `S4` hérite ensuite de l'exigence d'alignement de `S1`, car il s'agit de la plus grande exigence d'alignement de la structure. `sizeof(struct S4)` retourne 64.

C++

```
struct S4 {
   int a;
   // 28 bytes padding
   struct S1 s1;      // S4 inherits cache alignment requirement of S1
};
```

Les trois déclarations de variable suivantes utilisent également `__declspec(align(#))`. Dans chaque cas, la variable doit être alignée sur 32 octets. Dans le tableau, l'adresse de base du tableau, et non chaque membre du tableau, est alignée sur 32 octets. La **`sizeof`** valeur de chaque membre du tableau n'est pas affectée lorsque vous utilisez `__declspec(align(#))`.

```
C++

CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

Pour aligner chaque membre d'un tableau, écrivez un code tel que le suivant :

```
C++

typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

Dans cet exemple, notez que l'alignement de la structure elle-même et l'alignement du premier élément ont le même effet :

```
C++

CACHE_ALIGN struct S6 {
   int a;
   int b;
};

struct S7 {
   CACHE_ALIGN int a;
               int b;
};
```

`S6` et `S7` ont les mêmes caractéristiques d'alignement, d'allocation et de taille.

Dans cet exemple, l'alignement des adresses de départ de `a`, `b`, `c`et `d` sont respectivement 4, 1, 4 et 1.

```
C++

void fn() {
   int a;
   char b;
   long c;
```

```
    char d[10]
}
```

Quand la mémoire est allouée sur le tas, l'alignement dépend de la fonction d'allocation appelée. Par exemple, si vous utilisez `malloc`, le résultat dépend de la taille d'opérande. Si *arg* >= 8, la mémoire retournée est alignée sur 8 octets. Si *arg* < 8, l'alignement de la mémoire retournée est la première puissance de 2 inférieure à *arg*. Par exemple, si vous utilisez `malloc(7)`, l'alignement est de 4 octets.

Définition de nouveaux types avec `__declspec(align(#))`

Vous pouvez définir un type avec une caractéristique d'alignement.

Par exemple, vous pouvez définir un avec une **`struct`** valeur d'alignement de cette façon :

C++

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

Maintenant, `aType` et `bType` ont la même taille (8 octets), mais les variables de type `bType` sont alignées sur 32 octets.

Alignement des données dans le stockage local de threads

Le stockage local des threads de type statique (TLS) créé avec l'attribut `__declspec(thread)` et placé dans la section TLS de l'image contribue à l'alignement exactement comme les données statiques normales. Pour créer des données TLS, le système d'exploitation alloue de la mémoire de la taille de la section TLS et respecte l'attribut d'alignement de la section TLS.

Cet exemple montre différentes façons de placer des données alignées dans le stockage local des threads.

C++

```
// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;
```

```
// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
   int a;
   int b;
};
// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;
```

Fonctionnement `align` de l'empaquetage des données

L'option `/Zp` du compilateur et le `pack` pragma ont pour effet d'empaqueter les données pour les membres de la structure et de l'union. Cet exemple montre comment `/Zp` et `__declspec(align(#))` fonctionnent ensemble :

C++

```
struct S {
   char a;
   short b;
   double c;
   CACHE_ALIGN double d;
   char e;
   double f;
};
```

Le tableau suivant répertorie le décalage de chaque membre sous des valeurs différentes `/Zp` (ou `#pragma pack`), montrant comment les deux interagissent.

Variable	/Zp1	/Zp2	/Zp4	/Zp8
a	0	0	0	0
b	1	2	2	2
c	3	4	4	8
d	32	32	32	32
e	40	40	40	40
f	41	42	44	48

Variable	/Zp1	/Zp2	/Zp4	/Zp8
sizeof(S)	64	64	64	64

Pour plus d'informations, consultez /Zp (Alignement des membres de struct).

Le décalage d'un objet est basé sur le décalage de l'objet précédent et du paramètre de compactage actif, sauf si l'objet a un attribut `__declspec(align(#))`. Dans ce cas, l'alignement est basé sur le décalage de l'objet précédent et de la valeur `__declspec(align(#))` pour l'objet.

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Vue d'ensemble des conventions ARM ABI
Conventions des logiciels x64

allocate

Article • 16/06/2023

Section spécifique à Microsoft

Le `allocate` spécificateur de déclaration nomme un segment de données dans lequel l'élément de données sera alloué.

Syntaxe

> `__declspec(allocate("`*segname*`"))`*déclarateur*

Notes

Le *nom segname* doit être déclaré à l'aide de l'un des pragmas suivants :

- code_seg
- const_seg
- data_seg
- init_seg
- Section

Exemple

C++

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode"))  int i = 0;

int main() {
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

`allocator`

Article • 16/06/2023

Section spécifique à Microsoft

Le `allocator` spécificateur de déclaration peut être appliqué aux fonctions d'allocation de mémoire personnalisées pour rendre les allocations visibles via le suivi d'événements pour Windows (ETW).

Syntaxe

```
__declspec(allocator)
```

Notes

Le profileur de mémoire natif dans Visual Studio fonctionne en collectant les données d'événement ETW d'allocation émises par pendant l'exécution. Les allocateurs dans le CRT et le Kit de développement logiciel (SDK) Windows ont été annotés au niveau de la source afin que leurs données d'allocation puissent être capturées. Si vous écrivez vos propres allocateurs, toutes les fonctions qui retournent un pointeur vers la mémoire du tas nouvellement allouée peuvent être décorées avec `__declspec(allocator)`, comme indiqué dans cet exemple pour myMalloc :

C++

```
__declspec(allocator) void* myMalloc(size_t size)
```

Pour plus d'informations, consultez Mesurer l'utilisation de la mémoire dans Visual Studio et Événements de tas ETW natifs personnalisés.

FIN de la section spécifique à Microsoft

appdomain

Article • 03/04/2023

Spécifie que chaque domaine d'application de votre application managée doit avoir sa propre copie d'une variable globale ou d'une variable membre static particulière. Pour plus d'informations, consultez Domaines d'application et Visual C++ .

Chaque domaine d'application a sa propre copie d'une variable par domaine d'application. Un constructeur d'une variable de domaine d'application est exécuté lorsqu'un assembly est chargé dans un domaine d'application, et le destructeur est exécuté lorsque le domaine d'application est déchargé.

Si vous voulez que tous les domaines d'application au sein d'un processus dans le Common Langage Runtime partagent une variable globale, utilisez le modificateur `__declspec(process)`. `__declspec(process)` est en vigueur par défaut sous /clr. Les options du compilateur **/clr:pure** et **/clr:safe** sont déconseillées dans Visual Studio 2015 et non prises en charge dans Visual Studio 2017.

`__declspec(appdomain)` n'est valide que lorsque l'une des options du compilateur **/clr** est utilisée. Seules une variable globale, une variable membre static ou une variable locale static peuvent être marquées avec `__declspec(appdomain)`. Il est incorrect d'appliquer `__declspec(appdomain)` aux membres static des types managés, car ils ont toujours ce comportement.

L'utilisation `__declspec(appdomain)` est similaire à l'utilisation du stockage local de thread (TLS). Les threads ont leur propre stockage, tout comme les domaines d'application. L'utilisation de `__declspec(appdomain)` assure que la variable globale possède son propre stockage dans chaque domaine d'application créé pour cette application.

Il existe des limitations quant à la combinaison de l'utilisation de variables par processus et par domaine d'application ; pour plus d'informations, consultez processus .

Par exemple, au lancement du programme, toutes les variables par processus sont initialisées, puis toutes les variables par domaine d'application. Par conséquent, lorsqu'une variable par processus est en cours d'initialisation, elle ne peut pas dépendre de la valeur d'une quelconque variable par domaine d'application. Il n'est pas recommandé de combiner l'utilisation (assignation) de variables par processus et par domaine d'application.

Pour plus d'informations sur l'appel d'une fonction dans un domaine d'application spécifique, consultez call_in_appdomain Function.

Exemple

C++

```
// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
   CGlobal(bool bProcess) {
      Counter = 10;
      m_bProcess = bProcess;
      Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor",
m_bProcess ? (String^)"process" : (String^)"appdomain");
   }

   ~CGlobal() {
      Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor",
m_bProcess ? (String^)"process" : (String^)"appdomain");
   }

   int Counter;

private:
   bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
   static void change() {
      ++appdomain_global.Counter;
   }

   static void display() {
      Console::WriteLine("process_global value in appdomain '{0}': {1}",
         AppDomain::CurrentDomain->FriendlyName,
         process_global.Counter);

      Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
         AppDomain::CurrentDomain->FriendlyName,
         appdomain_global.Counter);
   }
};

int main() {
   AppDomain^ defaultDomain = AppDomain::CurrentDomain;
   AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
   AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
```

```
   CrossAppDomainDelegate^ changeDelegate = gcnew
CrossAppDomainDelegate(&Functions::change);
   CrossAppDomainDelegate^ displayDelegate = gcnew
CrossAppDomainDelegate(&Functions::display);

   // Print the initial values of appdomain_global in all appdomains.
   Console::WriteLine("Initial value");
   defaultDomain->DoCallBack(displayDelegate);
   domain->DoCallBack(displayDelegate);
   domain2->DoCallBack(displayDelegate);

   // Changing the value of appdomain_global in the domain and domain2
   // appdomain_global value in "default" appdomain remain unchanged
   process_global.Counter = 20;
   domain->DoCallBack(changeDelegate);
   domain2->DoCallBack(changeDelegate);
   domain2->DoCallBack(changeDelegate);

   // Print values again
   Console::WriteLine("Changed value");
   defaultDomain->DoCallBack(displayDelegate);
   domain->DoCallBack(displayDelegate);
   domain2->DoCallBack(displayDelegate);

   AppDomain::Unload(domain);
   AppDomain::Unload(domain2);
}
```

Output

```
__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor
```

Voir aussi

__declspec

Mots clés

`__declspec(code_seg)`

Article • 16/06/2023

Section spécifique à Microsoft

L'attribut **`code_seg`** de déclaration nomme un segment de texte exécutable dans le *`.obj`* fichier dans lequel le code objet de la fonction ou des fonctions membres de la classe est stocké.

Syntaxe

> **`__declspec(code_seg("`***`segname`***`"))`** ***`declarator`***

Notes

L'attribut `__declspec(code_seg(...))` permet de placer le code dans des blocs nommés distincts qui peuvent être paginés ou verrouillés individuellement en mémoire. Vous pouvez utiliser cet attribut pour contrôler l'emplacement des modèles instanciés et du code généré par le compilateur.

Un *segment* est un bloc nommé de données dans un *`.obj`* fichier chargé en mémoire en tant qu'unité. Un *segment de texte* est un segment qui contient du code exécutable. Le terme *section* est souvent utilisé indifféremment avec segment.

Le code objet qui est généré lorsque `declarator` est défini est placé dans le segment de texte spécifié par `segname`, qui est un littéral de chaîne étroite. Le nom `segname` n'a pas besoin d'être spécifié dans un pragma de section avant de pouvoir être utilisé dans une déclaration. Par défaut, quand aucun n'est `code_seg` spécifié, le code objet est placé dans un segment nommé `.text`. Un **`code_seg`** attribut remplace toute directive #pragma code_seg existante. Un **`code_seg`** attribut appliqué à une fonction membre remplace tout **`code_seg`** attribut appliqué à la classe englobante.

Si une entité a un **`code_seg`** attribut, toutes les déclarations et définitions de la même entité doivent avoir des attributs identiques **`code_seg`** . Si une classe de base a un **`code_seg`** attribut, les classes dérivées doivent avoir le même attribut.

Lorsqu'un **`code_seg`** attribut est appliqué à une fonction d'étendue d'espace de noms ou à une fonction membre, le code objet de cette fonction est placé dans le segment de texte spécifié. Lorsque cet attribut est appliqué à une classe, toutes les fonctions

membres de la classe et les classes imbriquées, y compris les fonctions membres spéciales générées par le compilateur, sont placées dans le segment spécifié. Les classes définies localement(par exemple, les classes définies dans un corps de fonction membre) n'héritent pas de l'attribut `code_seg` de l'étendue englobante.

Lorsqu'un `code_seg` attribut est appliqué à un modèle de classe ou de fonction, toutes les spécialisations implicites du modèle sont placées dans le segment spécifié. Les spécialisations explicites ou partielles n'héritent pas de l'attribut `code_seg` du modèle principal. Vous pouvez spécifier le même attribut ou un autre `code_seg` attribut sur la spécialisation. Un `code_seg` attribut ne peut pas être appliqué à une instanciation de modèle explicite.

Par défaut, le code généré par le compilateur, tel qu'une fonction membre spéciale, est placé dans le `.text` segment. La `#pragma code_seg` directive ne remplace pas cette valeur par défaut. Utilisez l'attribut `code_seg` sur la classe, le modèle de classe ou le modèle de fonction pour contrôler l'emplacement où le code généré par le compilateur est placé.

Les lambdas héritent `code_seg` des attributs de leur étendue englobante. Pour spécifier un segment pour un lambda, appliquez un `code_seg` attribut après la clause de déclaration de paramètre et avant toute spécification mutable ou d'exception, toute spécification de type de retour de fin et le corps lambda. Pour plus d'informations, consultez Syntaxe d'expression lambda. Cet exemple définit une expression lambda dans un segment nommé PagedMem :

```
C++
```

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t;
};
```

Faites attention lorsque vous placez des fonctions membres dans plusieurs segments, en particulier les fonctions membres virtuelles. Supposons que vous définissiez une fonction virtuelle dans une classe dérivée qui réside dans un segment paginé lorsque la méthode de classe de base réside dans un segment non paginé. D'autres méthodes de classe de base ou le code utilisateur peuvent supposer que l'appel de la méthode virtuelle ne déclenchera pas d'erreur de page.

Exemple

Cet exemple montre comment un `code_seg` attribut contrôle le placement des segments lorsque la spécialisation de modèle implicite et explicite est utilisée :

C++

```
// code_seg.cpp
// Compile: cl /EHsc /W4 code_seg.cpp

// Base template places object code in Segment_1 segment
template<class T>
class __declspec(code_seg("Segment_1")) Example
{
public:
   virtual void VirtualMemberFunction(T /*arg*/) {}
};

// bool specialization places code in default .text segment
template<>
class Example<bool>
{
public:
   virtual void VirtualMemberFunction(bool /*arg*/) {}
};

// int specialization places code in Segment_2 segment
template<>
class __declspec(code_seg("Segment_2")) Example<int>
{
public:
   virtual void VirtualMemberFunction(int /*arg*/) {}
};

// Compiler warns and ignores __declspec(code_seg("Segment_3"))
// in this explicit specialization
__declspec(code_seg("Segment_3")) Example<short>; // C4071

int main()
{
   // implicit double specialization uses base template's
   // __declspec(code_seg("Segment_1")) to place object code
   Example<double> doubleExample{};
   doubleExample.VirtualMemberFunction(3.14L);

   // bool specialization places object code in default .text segment
   Example<bool> boolExample{};
   boolExample.VirtualMemberFunction(true);

   // int specialization uses __declspec(code_seg("Segment_2"))
   // to place object code
   Example<int> intExample{};
   intExample.VirtualMemberFunction(42);
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

deprecated (C++)

Article • 16/06/2023

Cette rubrique concerne la déclaration declspec dépréciée propre à Microsoft. Pour plus d'informations sur l'attribut C++14 `[[deprecated]]` et des conseils sur l'utilisation de cet attribut par rapport au declspec ou à la pragma spécifique à Microsoft, consultez Attributs standard C++.

Avec les exceptions indiquées ci-dessous, la **`deprecated`** déclaration offre les mêmes fonctionnalités que la pragma déconseillée :

- La **`deprecated`** déclaration vous permet de spécifier des formes particulières de surcharges de fonction comme déconseillées, tandis que le formulaire pragma s'applique à toutes les formes surchargées d'un nom de fonction.
- La **`deprecated`** déclaration vous permet de spécifier un message qui s'affichera au moment de la compilation. Le texte du message peut être issu d'une macro.
- Les macros ne peuvent être marquées comme dépréciées qu'avec le **`deprecated`** pragma.

Si le compilateur rencontre l'utilisation d'un identificateur déconseillé ou de l'attribut standard [[deprecated]] , un avertissement C4996 est déclenché.

Exemples

L'exemple suivant montre comment marquer des fonctions comme déconseillés, et comment spécifier un message à afficher au moment de la compilation, lorsque la fonction déconseillée est utilisée.

C++

```
// deprecated.cpp
// compile with: /W3
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void
func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
   func1();
   func1(1);   // C4996
```

```
    func2(1);   // C4996
    func3(1);   // C4996
}
```

L'exemple suivant montre comment marquer des classes comme déconseillés, et comment spécifier un message à afficher au moment de la compilation, lorsque la classe déconseillée est utilisée.

C++

```
// deprecate_class.cpp
// compile with: /W3
struct __declspec(deprecated) X {
   void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
   void f(){}
};

int main() {
   X x;   // C4996
   X2 x2;   // C4996
}
```

Voir aussi

__declspec

Mots clés

`dllexport`, `dllimport`

Article • 03/04/2023

Section spécifique à Microsoft

Les **`dllexport`** attributs de classe de stockage et **`dllimport`** sont des extensions spécifiques à Microsoft pour les langages C et C++. Vous pouvez les utiliser pour exporter et importer des fonctions, des données et des objets vers ou depuis une DLL.

Syntaxe

> **`__declspec( dllimport )`** *`declarator`*
>
> **`__declspec( dllexport )`** *`declarator`*

Notes

Ces attributs définissent explicitement l'interface de la DLL à son client, qui peut être un fichier exécutable ou une autre DLL. La déclaration de fonctions comme **`dllexport`** élimine la nécessité d'un fichier de définition de module (`.def`), du moins en ce qui concerne la spécification des fonctions exportées. L'attribut **`dllexport`** remplace le **`__export`** mot clé.

Si une classe est marquée `__declspec(dllexport)`, toutes les spécialisations des modèles de classe dans la hiérarchie de classes sont implicitement marquées comme `__declspec(dllexport)`. Cela signifie que les modèles de classe sont instanciés explicitement et que les membres de la classe doivent être définis.

`dllexport` d'une fonction expose la fonction avec son nom décoré, parfois appelé « mangling de nom ». Pour les fonctions C++, le nom décoré comprend des caractères supplémentaires qui encodent des informations sur le type et les paramètres. Les fonctions C ou les fonctions déclarées comme `extern "C"` incluent une décoration spécifique à la plateforme basée sur la convention d'appel. Aucune décoration de nom n'est appliquée aux fonctions C exportées ou aux fonctions C++ `extern "C"` qui utilisent la convention d'appel **`__cdecl`** . Pour plus d'informations sur la décoration des noms dans le code C/C++, consultez Noms décorés.

Pour exporter un nom non décoré, vous pouvez créer un lien à l'aide d'un fichier Définition de module (`.def`) qui définit le nom non décoré d'une `EXPORTS` section. Pour plus d'informations, consultez EXPORTS. Une autre façon d'exporter un nom non

optimisé consiste à utiliser une `#pragma comment(linker, "/export:alias=decorated_name")` directive dans le code source.

Lorsque vous déclarez **`dllexport`** ou **`dllimport`**, vous devez utiliser la syntaxe d'attribut étendue et la **`__declspec`** mot clé.

Exemple

C++

```
// Example of the dllimport and dllexport class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

Pour rendre votre code plus lisible, vous pouvez aussi utiliser des définitions de macros :

C++

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Pour plus d'informations, consultez les pages suivantes :

- Définitions et déclarations
- Définition de fonctions C++ inline avec dllexport et dllimport
- Règles générales et limitations
- Utilisation dllimport et dllexport dans les classes C++

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés

Définitions et déclarations (C++)

Article • 16/06/2023

Section spécifique à Microsoft

L'interface DLL fait référence à tous les éléments (fonctions et données) qui sont connus pour être exportés par un programme du système, c'est-à-dire tous les éléments déclarés comme `dllimport` ou `dllexport`. Toutes les déclarations incluses dans l'interface DLL doivent spécifier l'attribut `dllimport` ou `dllexport`. Toutefois, la définition doit spécifier uniquement l'attribut `dllexport` . Par exemple, la définition de fonction suivante génère une erreur de compilation :

```
__declspec( dllimport ) int func() {   // Error; dllimport
                                       // prohibited on definition.
   return 1;
}
```

Le code suivant génère aussi une erreur :

```
__declspec( dllimport ) int i = 10;  // Error; this is a definition.
```

Mais voici la syntaxe correcte :

```
__declspec( dllexport ) int i = 10;  // Okay--export definition
```

L'utilisation de `dllexport` implique une définition tandis que `dllimport` implique une déclaration. Vous devez utiliser le mot clé `extern` avec `dllexport` pour forcer une déclaration. Sinon, une définition est impliquée. Les exemples suivants sont donc corrects :

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
DllImport int j;        // declaration.
```

Les exemples suivants clarifient l'exemple précédent :

```
static __declspec( dllimport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s;  // Error; not declared
                                           // extern.
    __declspec( dllimport ) int m;         // Okay; this is a
                                           // declaration.
    __declspec( dllexport ) int n;         // Error; implies external
                                           // definition in local scope.
    extern __declspec( dllimport ) int i;  // Okay; this is a
                                           // declaration.
    extern __declspec( dllexport ) int k;  // Okay; extern implies
                                           // declaration.
    __declspec( dllexport ) int x = 5;     // Error; implies external
                                           // definition in local scope.
}
```

FIN de la section spécifique à Microsoft

Voir aussi

dllexport, dllimport

Définition de fonctions C++ incorporées avec dllexport et dllimport

Article • 16/06/2023

Section spécifique à Microsoft

Vous pouvez définir comme inline une fonction avec l'attribut `dllexport`. Dans ce cas, la fonction est toujours instanciée et exportée, qu'elle soit référencée ou non par un module du programme. La fonction est présumée être importée par un autre programme.

Vous pouvez également définir comme inline une fonction déclarée avec l'attribut `dllimport`. Dans ce cas, la fonction peut être développée (soumise aux spécifications /Ob) mais jamais instanciée. En particulier, si l'adresse d'une fonction importée inline est reçue, l'adresse de la fonction résidant dans la DLL est retournée. Ce comportement est identique à la réception de l'adresse d'une fonction importée non inline.

Ces règles s'appliquent aux fonctions inline dont les définitions apparaissent dans une définition de classe. De plus, les données locales statiques et les chaînes des fonctions inline conservent les mêmes identités entre la DLL et le client qu'elles le feraient dans un programme unique (autrement dit un fichier exécutable sans interface DLL).

Soyez vigilant en fournissant des fonctions inline importées. Par exemple, si vous mettez à jour la DLL, ne supposez pas que le client utilisera sa version modifiée. Pour vous assurer que vous chargez la version appropriée de la DLL, régénérez aussi le client de la DLL.

FIN de la section spécifique à Microsoft

Voir aussi

dllexport, dllimport

Règles générales et limitations

Article • 16/06/2023

Section spécifique à Microsoft

- Si vous déclarez une fonction ou un objet sans l'attribut `dllimport` ou `dllexport` , la fonction ou l'objet n'est pas considéré comme faisant partie de l'interface DLL. Par conséquent, la définition de la fonction ou de l'objet doit être présente dans ce module ou dans un autre module du même programme. Pour que la fonction ou l'objet fasse partie de l'interface DLL, vous devez déclarer la définition de la fonction ou de l'objet dans l'autre module en tant que `dllexport`. Sinon, une erreur d'éditeur de liens est générée.

  Si vous déclarez une fonction ou un objet avec l'attribut `dllexport` , sa définition doit apparaître dans un module du même programme. Sinon, une erreur d'éditeur de liens est générée.

- Si un seul module de votre programme contient à la fois des `dllimport` déclarations et `dllexport` pour la même fonction ou le même objet, l'attribut `dllexport` est prioritaire sur l'attribut `dllimport` . Toutefois, un avertissement du compilateur est généré. Par exemple :

  ```
  C++
  ```

  ```
  __declspec( dllimport ) int i;
  __declspec( dllexport ) int i;   // Warning; inconsistent;
                                   // dllexport takes precedence.
  ```

- En C++, vous pouvez initialiser un pointeur de données locale globalement déclaré ou statique, ou avec l'adresse d'un objet de données déclaré avec l'attribut `dllimport` , ce qui génère une erreur en C. En outre, vous pouvez initialiser un pointeur de fonction locale statique avec l'adresse d'une fonction déclarée avec l'attribut `dllimport` . En C, ce type d'assignation définit le pointeur sur l'adresse de la conversion de code d'importation de la DLL (stub de code qui transfère un contrôle à la fonction) plutôt que l'adresse de la fonction. En C++, il définit le pointeur sur l'adresse de la fonction. Par exemple :

  ```
  C++
  ```

  ```
  __declspec( dllimport ) void func1( void );
  __declspec( dllimport ) int i;
  ```

```
int *pi = &i;                                  // Error in C
static void ( *pf )( void ) = &func1;          // Address of thunk in C,
                                               // function in C++

void func2()
{
   static int *pi = &i;                       // Error in C
   static void ( *pf )( void ) = &func1;      // Address of thunk in C,
                                              // function in C++
}
```

Toutefois, étant donné qu'un programme qui inclut l'attribut `dllexport` dans la déclaration d'un objet doit fournir la définition de cet objet quelque part dans le programme, vous pouvez initialiser un pointeur de fonction statique global ou local avec l'adresse d'une `dllexport` fonction. De même, vous pouvez initialiser un pointeur de données statiques global ou local avec l'adresse d'un objet de données `dllexport`. Par exemple, le code suivant ne génère pas d'erreur en C ni en C++ :

C++

```
__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                                  // Okay
static void ( *pf )( void ) = &func1;          // Okay

void func2()
{
    static int *pi = &i;                       // Okay
    static void ( *pf )( void ) = &func1;      // Okay
}
```

- Si vous appliquez `dllexport` à une classe régulière qui a une classe de base qui n'est pas marquée comme `dllexport`, le compilateur génère l'erreur C4275.

  Le compilateur génère le même avertissement si la classe de base est une spécialisation d'un modèle de classe. Pour contourner ce problème, marquez la classe de base avec `dllexport`. Le problème avec une spécialisation d'un modèle de classe est l'emplacement où placer le `__declspec(dllexport)` ; vous n'êtes pas autorisé à marquer le modèle de classe. Au lieu de cela, instanciez explicitement le modèle de classe et marquez cette instanciation explicite avec `dllexport`. Par exemple :

  C++

```
template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...
```

Cette solution de contournement échoue si l'argument template est la classe dérivée. Par exemple :

```
C++

class __declspec(dllexport) D : public B<D> {
// ...
```

Étant donné qu'il s'agit d'un modèle courant avec les modèles, le compilateur a modifié la sémantique de `dllexport` lorsqu'il est appliqué à une classe qui a une ou plusieurs classes de base et quand une ou plusieurs classes de base sont une spécialisation d'un modèle de classe. Dans ce cas, le compilateur s'applique `dllexport` implicitement aux spécialisations des modèles de classe. Vous pouvez effectuer les opérations suivantes sans recevoir d'avertissement :

```
C++

class __declspec(dllexport) D : public B<D> {
// ...
```

FIN de la section spécifique à Microsoft

Voir aussi

dllexport, dllimport

Utilisation de dllimport et dllexport dans les classes C++

Article • 03/04/2023

Section spécifique à Microsoft

Vous pouvez déclarer des classes C++ avec l'attribut `dllimport` ou `dllexport` . Ces formes impliquent que toute la classe soit importée ou exportée. Les classes exportées de cette façon sont appelées des classes exportables.

L'exemple suivant définit une classe exportable. Toutes ses fonctions membres et données statiques sont exportées :

C++

```
#define DllExport   __declspec( dllexport )

class DllExport C {
   int i;
   virtual int func( void ) { return 1; }
};
```

Notez que l'utilisation `dllimport` explicite des attributs et `dllexport` sur les membres d'une classe exportable est interdite.

Classes de dllexport

Lorsque vous déclarez une classe `dllexport`, toutes ses fonctions membres et ses membres de données statiques sont exportés. Vous devez fournir les définitions de tous ces membres dans le même programme. Sinon, une erreur d'éditeur de liens est générée. La seule exception à cette règle s'applique aux fonctions virtuelles pures, pour lesquelles vous n'avez pas besoin de fournir de définitions explicites. Toutefois, étant donné qu'un destructeur d'une classe abstraite est toujours appelé par le destructeur de la classe de base, les destructeurs virtuels purs doivent toujours fournir une définition. Notez que ces règles sont identiques pour les classes non exportables.

Si vous exportez des données de classe de type ou des fonctions qui retournent des classes, veillez à exporter la classe.

Classes de dllimport

Lorsque vous déclarez une classe `dllimport`, toutes ses fonctions membres et ses membres de données statiques sont importés. Contrairement au comportement de `dllimport` et `dllexport` sur les types non-classe, les membres de données statiques ne peuvent pas spécifier une définition dans le même programme dans lequel une `dllimport` classe est définie.

Héritage et classes exportables

Toutes les classes de base d'une classe exportable doivent être exportées. Sinon, un avertissement du compilateur est généré. De plus, tous les membres accessibles qui sont également des classes doivent être exportables. Cette règle permet à une `dllexport` classe d'hériter d'une `dllimport` classe et à une `dllimport` classe d'hériter d'une `dllexport` classe (bien que cette dernière ne soit pas recommandée). En règle générale, tout ce qui est accessible au client de la DLL (conformément aux règles d'accès C++) doit faire partie de l'interface exportable. Cela inclut les données membres privées référencées dans les fonctions inline.

Importation/exportation de membres sélectifs

Étant donné que les fonctions membres et les données statiques au sein d'une classe ont implicitement une liaison externe, vous pouvez les déclarer avec l'attribut `dllimport` ou `dllexport` , sauf si la classe entière est exportée. Si la classe entière est importée ou exportée, la déclaration explicite des fonctions membres et des données en tant que `dllimport` ou `dllexport` est interdite. Si vous déclarez un membre de données statique dans une définition de classe en tant que `dllexport`, une définition doit se produire quelque part dans le même programme (comme avec une liaison externe non de classe).

De même, vous pouvez déclarer des fonctions membres avec les `dllimport` attributs ou `dllexport` . Dans ce cas, vous devez fournir une `dllexport` définition quelque part dans le même programme.

Il est intéressant de noter plusieurs aspects importants concernant l'importation et l'exportation de membres sélectifs :

- L'importation/exportation de membres sélectifs permet principalement de fournir une version de l'interface de classe exportée qui est plus restrictive ; autrement dit, une version pour laquelle vous pouvez concevoir une DLL qui expose moins de fonctionnalités publiques et privées que ne le permet le langage. Cela permet également d'affiner l'interface exportable : lorsque vous savez que le client, par

définition, ne peut pas accéder à certaines données privées, vous n'avez pas besoin d'exporter la classe entière.

- Si vous exportez une seule fonction virtuelle d'une classe, vous devez toutes les exporter ou fournir au moins les versions que le client peut utiliser directement.
- Si vous avez une classe dans laquelle vous utilisez l'importation/exportation de membres sélectifs avec des fonctions virtuelles, les fonctions doivent être dans l'interface exportable ou définies inline (visibles par le client).
- Si vous définissez un membre en tant que `dllexport` mais que vous ne l'incluez pas dans la définition de classe, une erreur du compilateur est générée. Vous devez définir le membre dans l'en-tête de classe.
- Bien que la définition des membres de la classe en tant que `dllimport` ou `dllexport` soit autorisée, vous ne pouvez pas remplacer l'interface spécifiée dans la définition de classe.
- Si vous définissez une fonction membre dans un emplacement autre que le corps de la définition de classe dans laquelle vous l'avez déclarée, un avertissement est généré si la fonction est définie comme `dllexport` ou `dllimport` (si cette définition diffère de celle spécifiée dans la déclaration de classe).

FIN de la section spécifique à Microsoft

Voir aussi

dllexport, dllimport

`empty_bases`

Article • 16/06/2023

Section spécifique à Microsoft

La norme C++ exige qu'un objet le plus dérivé ait une taille différente de zéro et qu'il occupe un ou plusieurs octets de stockage. Étant donné que l'exigence s'étend uniquement aux objets les plus dérivés, les sous-objets de classe de base ne sont pas soumis à cette contrainte. L'optimisation de la classe de base vide (EBCO) tire parti de cette liberté. Il en résulte une réduction de la consommation de mémoire, ce qui peut améliorer les performances. Le compilateur Microsoft Visual C++ a toujours eu une prise en charge limitée d'EBCO. Dans Visual Studio 2015 Update 3 et versions ultérieures, nous avons ajouté un nouvel `__declspec(empty_bases)` attribut pour les types de classes qui tire pleinement parti de cette optimisation.

> **ⓘ Important**
>
> L'utilisation de `__declspec(empty_bases)` peut entraîner une modification abi-cassant de la structure et de la disposition des classes où elle est appliquée. Assurez-vous que tout le code client utilise les mêmes définitions pour les structures et les classes que votre code lorsque vous utilisez cet attribut de classe de stockage.

Syntaxe

```
__declspec( empty_bases )
```

Notes

Dans Visual Studio, en l'absence `__declspec(align())` de spécifications ou `alignas()` , une classe vide a une taille de 1 octet :

C++

```
struct Empty1 {};
static_assert(sizeof(Empty1) == 1, "Empty1 should be 1 byte");
```

Une classe avec un seul membre de données non statique de type `char` a également une taille de 1 octet :

C++

```
struct Struct1
{
  char c;
};
static_assert(sizeof(Struct1) == 1, "Struct1 should be 1 byte");
```

La combinaison de ces classes dans une hiérarchie de classes entraîne également une classe d'une taille de 1 octet :

C++

```
struct Derived1 : Empty1
{
  char c;
};
static_assert(sizeof(Derived1) == 1, "Derived1 should be 1 byte");
```

Ce résultat est l'optimisation de la classe de base vide au travail, car sans elle `Derived1` aurait une taille de 2 octets : 1 octet pour `Empty1` et 1 octet pour `Derived1::c`. La disposition des classes est également optimale lorsqu'il existe une chaîne de classes vides :

C++

```
struct Empty2 : Empty1 {};
struct Derived2 : Empty2
{
  char c;
};
static_assert(sizeof(Derived2) == 1, "Derived2 should be 1 byte");
```

Toutefois, la disposition de classe par défaut dans Visual Studio ne tire pas parti d'EBCO dans plusieurs scénarios d'héritage :

C++

```
struct Empty3 {};
struct Derived3 : Empty2, Empty3
{
  char c;
};
static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // Error
```

Bien qu'elle `Derived3` puisse avoir une taille de 1 octet, la disposition de classe par défaut entraîne une taille de 2 octets. L'algorithme de disposition de classe ajoute 1 octet de remplissage entre deux classes de base vides consécutives, ce qui entraîne `Empty2` la consommation d'un octet supplémentaire dans `Derived3`:

```
class Derived3  size(2):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
1  | +--- (base class Empty3)
   | +---
1  | c
   +---
```

Les effets de cette disposition sous-optimale sont aggravés lorsque les exigences d'alignement d'une classe de base ou d'un sous-objet membre ultérieur forcent un remplissage supplémentaire :

C++

```
struct Derived4 : Empty2, Empty3
{
  int i;
};
static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // Error
```

L'alignement naturel d'un objet de type `int` est de 4 octets. Par conséquent, 3 octets de remplissage supplémentaire doivent être ajoutés après `Empty3` pour aligner `Derived4::i`correctement :

```
class Derived4 size(8):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
1  | +--- (base class Empty3)
   | +---
   | <alignment member> (size=3)
4  | i
   +---
```

Un autre problème avec la disposition de classe par défaut est qu'une classe de base vide peut être disposée à un décalage au-delà de la fin de la classe :

C++

```
struct Struct2 : Struct1, Empty1
{
};
static_assert(sizeof(Struct2) == 1, "Struct2 should be 1 byte");
```

```
class Struct2 size(1):
   +---
0  | +--- (base class Struct1)
0  | | c
   | +---
1  | +--- (base class Empty1)
   | +---
   +---
```

Bien que `Struct2` soit la taille optimale, `Empty1` est disposé au décalage 1 dans `Struct2` , mais la taille de `Struct2` n'est pas augmentée pour en tenir compte. Par conséquent, pour un tableau `A` d'objets `Struct2` , l'adresse du `Empty1` sous-objet de `A[0]` sera identique à l'adresse de `A[1]`, ce qui ne devrait pas être le cas. Ce problème ne se produirait pas si `Empty1` étaient disposés au décalage 0 dans `Struct2`, ce qui chevauche le `Struct1` sous-objet.

L'algorithme de disposition par défaut n'a pas été modifié pour répondre à ces limitations et tirer pleinement parti d'EBCO. Une telle modification interromprait la compatibilité binaire. Si la disposition par défaut d'une classe a changé à la suite d'EBCO, chaque fichier objet et bibliothèque contenant la définition de classe doit être recompilé afin qu'ils s'entendent tous sur la disposition de classe. Cette exigence s'étend également aux bibliothèques obtenues à partir de sources externes. Les développeurs de ces bibliothèques doivent fournir des versions indépendantes compilées à la fois avec et sans la disposition EBCO pour prendre en charge les clients qui utilisent différentes versions du compilateur. Bien que nous ne puissions pas modifier la disposition par défaut, nous pouvons fournir un moyen de modifier la disposition par classe avec l'ajout de l'attribut `__declspec(empty_bases)` class. Une classe définie avec cet attribut peut utiliser pleinement EBCO.

C++

```
struct __declspec(empty_bases) Derived3 : Empty2, Empty3
{
  char c;
};
static_assert(sizeof(Derived3) == 1, "Derived3 should be 1 byte"); // No
Error
```

```
class Derived3  size(1):
   +---
0  | +--- (base class Empty2)
0  | | +--- (base class Empty1)
   | | +---
   | +---
0  | +--- (base class Empty3)
   | +---
0  | c
   +---
```

Tous les sous-objets de `Derived3` sont disposés au décalage 0, et sa taille est de 1 octet optimale. Un point important à retenir est que `__declspec(empty_bases)` affecte uniquement la disposition de la classe à laquelle elle est appliquée. Elle n'est pas appliquée de manière récursive aux classes de base :

C++

```
struct __declspec(empty_bases) Derived5 : Derived4
{
};
static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // Error
```

```
class Derived5  size(8):
   +---
0  | +--- (base class Derived4)
0  | | +--- (base class Empty2)
0  | | | +--- (base class Empty1)
   | | | +---
   | | +---
1  | | +--- (base class Empty3)
   | | +---
   | | <alignment member> (size=3)
4  | | i
   | +---
   +---
```

Bien que `__declspec(empty_bases)` soit appliqué à `Derived5`, il n'est pas éligible à EBCO, car il n'a pas de classes de base vides directes, il n'a donc aucun effet. Toutefois, si au lieu de cela elle est appliquée à la `Derived4` classe de base, qui est éligible pour EBCO, et `Derived4` `Derived5` auront une disposition optimale :

C++

```
struct __declspec(empty_bases) Derived4 : Empty2, Empty3
{
  int i;
};
static_assert(sizeof(Derived4) == 4, "Derived4 should be 4 bytes"); // No
Error

struct Derived5 : Derived4
{
};
static_assert(sizeof(Derived5) == 4, "Derived5 should be 4 bytes"); // No
Error
```

```
class Derived5  size(4):
   +---
0  | +--- (base class Derived4)
0  | | +--- (base class Empty2)
0  | | | +--- (base class Empty1)
   | | | +---
   | | +---
0  | | +--- (base class Empty3)
   | | +---
0  | | i
   | +---
   +---
```

En raison de l'exigence que tous les fichiers objet et bibliothèques conviennent de la disposition de classe, `__declspec(empty_bases)` ne peut être appliqué qu'aux classes que vous contrôlez. Il ne peut pas être appliqué aux classes de la bibliothèque standard ou aux classes incluses dans les bibliothèques qui ne sont pas également recompilées avec la disposition EBCO.

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

jitintrinsic

Article • 03/04/2023

Marque la fonction comme significative pour le CLR (Common Langage Runtime) 64 bits. Ceci est utilisé pour certaines fonctions dans les bibliothèques fournies par Microsoft.

Syntaxe

```
__declspec(jitintrinsic)
```

Notes

`jitintrinsic` ajoute un MODOPT (IsJitIntrinsic) à une signature de fonction.

Les utilisateurs sont découragés d'utiliser ce `__declspec` modificateur, car des résultats inattendus peuvent se produire.

Voir aussi

__declspec
Mots clés

naked (C++)

Article • 16/06/2023

Section spécifique à Microsoft

Pour les fonctions déclarées avec l'attribut `naked` , le compilateur génère du code sans code prolog et epilog. Vous pouvez utiliser cette fonctionnalité pour écrire vos propres séquences de code de prologue/épilogue à l'aide de code assembleur inline. Les fonctions naked sont particulièrement utiles pour l'écriture de pilotes de périphériques virtuels. Notez que l'attribut `naked` est valide uniquement sur x86 et ARM, et n'est pas disponible sur x64.

Syntaxe

```
__declspec(naked) declarator
```

Notes

Étant donné que l'attribut `naked` est uniquement pertinent pour la définition d'une fonction et n'est pas un modificateur de type, les fonctions nues doivent utiliser la syntaxe d'attribut étendue et le __declspec mot clé.

Le compilateur ne peut pas générer une fonction inline pour une fonction marquée avec l'attribut naked, même si la fonction est également marquée avec le __forceinline mot clé.

Le compilateur émet une erreur si l'attribut `naked` est appliqué à autre chose que la définition d'une méthode non membre.

Exemples

Ce code définit une fonction avec l'attribut `naked` :

```
__declspec( naked ) int func( formal_parameters ) {}
```

Ou, alternativement :

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

L'attribut `naked` affecte uniquement la nature de la génération de code du compilateur pour les séquences de prologue et d'épilogue de la fonction. Il n'affecte pas le code généré pour appeler de telles fonctions. Ainsi, l'attribut `naked` n'est pas considéré comme faisant partie du type de la fonction, et les pointeurs fonction ne peuvent pas avoir l'attribut `naked`. De plus, l'attribut `naked` ne peut pas être appliqué à une définition de données. Par exemple, cet exemple de code génère une erreur :

```
__declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

L'attribut `naked` se rapporte uniquement à la définition de la fonction et ne peut pas être spécifié dans le prototype de la fonction. Par exemple, cette déclaration génère une erreur du compilateur :

```
__declspec( naked ) int func();  // Error--naked attribute not permitted on
function declarations
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés
Appels de fonction naked

`noalias`

Article • 03/04/2023

Spécifique à Microsoft

`noalias` signifie qu'un appel de fonction ne modifie pas ou ne référence pas l'état global visible et modifie uniquement la mémoire pointée *directement* par les paramètres du pointeur (indirections de premier niveau).

Si une fonction est annotée en tant que `noalias`, l'optimiseur peut supposer que seuls les paramètres eux-mêmes, et seules les indirections de premier niveau des paramètres de pointeur, sont référencés ou modifiés à l'intérieur de la fonction.

L'annotation `noalias` s'applique uniquement dans le corps de la fonction annotée. Le fait de marquer une fonction comme `__declspec(noalias)` n'affecte pas l'aliasing des pointeurs retournés par la fonction.

Pour une autre annotation qui peut avoir un impact sur l'alias, consultez __declspec(restrict).

Exemple

L'exemple suivant illustre l'utilisation de `__declspec(noalias)`.

Lorsque la fonction `multiply` qui accède à la mémoire est annotée `__declspec(noalias)`, elle indique au compilateur que cette fonction ne modifie pas l'état global, sauf par le biais des pointeurs de sa liste de paramètres.

C

```
// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
```

```
    return retval;
}

float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1/k++;
    return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                          a[i * N + k] *
                          b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}
```

Voir aussi

__declspec

Mots clés

__declspec(restrict)

noinline

Article • 16/06/2023

Section spécifique à Microsoft

`__declspec(noinline)` indique au compilateur de ne jamais inliner une fonction membre particulière (fonction dans une classe).

Il peut être préférable de ne pas incorporer une fonction si elle est petite et non essentielle pour les performances de votre code. Autrement dit, s'il s'agit d'une petite fonction et qu'il est probable qu'elle sera rarement appelée, comme par exemple une fonction qui gère une condition d'erreur.

Gardez à l'esprit que si une fonction est marquée `noinline`, la fonction appelante sera plus petite et, par conséquent, elle-même un candidat pour l'inlining du compilateur.

C++

```
class X {
   __declspec(noinline) int mbrfunc() {
      return 0;
   }   // will not inline
};
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés
inline, __inline, __forceinline

noreturn

Article • 16/06/2023

Section spécifique à Microsoft

Cet `__declspec` attribut indique au compilateur qu'une fonction ne retourne pas. Par conséquent, le compilateur sait que le code suivant un appel à une `__declspec(noreturn)` fonction est inaccessible.

Si le compilateur recherche une fonction avec un chemin d'accès au contrôle qui ne retourne pas de valeur, il génère un avertissement (C4715) ou le message d'erreur (C2202). Si le chemin du contrôle n'est pas accessible en raison d'une fonction qui ne retourne jamais, vous pouvez utiliser `__declspec(noreturn)` pour empêcher cet avertissement ou cette erreur.

> ⓘ **Notes**
>
> L'ajout `__declspec(noreturn)` à une fonction qui est censée retourner peut entraîner un comportement non défini.

Exemple

Dans l'exemple suivant, la `else` clause ne contient pas d'instruction de retour. La déclaration `fatal` de comme `__declspec(noreturn)` évite un message d'erreur ou d'avertissement.

C++

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
   if(1)
     return 1;
   else if(0)
     return 0;
   else
     fatal();
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

`no_sanitize_address`

Article • 16/06/2023

Section spécifique à Microsoft

Le `__declspec(no_sanitize_address)` spécificateur indique au compilateur de désactiver l'assainisseur d'adresses sur les fonctions, les variables locales ou les variables globales. Ce spécificateur est utilisé conjointement avec AddressSanitizer.

> ⓘ **Notes**
>
> `__declspec(no_sanitize_address)` désactive le comportement *du compilateur* , et non le comportement *d'exécution* .

Exemple

Pour obtenir des exemples, consultez la référence de build AddressSanitizer .

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés
AddressSanitizer

`nothrow` (C++)

Article • 16/06/2023

Section spécifique à Microsoft

Attribut `__declspec` étendu qui peut être utilisé dans la déclaration de fonctions.

Syntaxe

> *return-type* `__declspec(nothrow)` [*call-convention*] *function-name* ([*argument-list*])

Notes

Nous recommandons que tout nouveau code utilise l'opérateur noexcept plutôt que `__declspec(nothrow)`.

Cet attribut indique au compilateur que la fonction déclarée et les fonctions qu'elle appelle ne lèvent jamais d'exception. Toutefois, elle n'applique pas la directive. En d'autres termes, il n'entraîne std::terminate jamais l'appel, contrairement `noexcept` à ou en `std:c++17` mode (Visual Studio 2017 version 15.5 et ultérieures), `throw()`.

Avec le modèle synchrone de gestion des exceptions, utilisé désormais par défaut, le compilateur peut éliminer les mécanismes de suivi de la durée de vie de certains objets non déroulables dans une telle fonction, et peut réduire considérablement la taille du code. Étant donné la directive de préprocesseur suivante, les trois déclarations de fonction ci-dessous sont équivalentes en `/std:c++14` mode :

C++

```
#define WINAPI __declspec(nothrow) __stdcall

void WINAPI f1();
void __declspec(nothrow) __stdcall f2();
void __stdcall f3() throw();
```

En `/std:c++17` mode , `throw()` n'est pas équivalent aux autres qui utilisent `__declspec(nothrow)` , car il provoque `std::terminate` l'appel si une exception est levée à partir de la fonction .

La `void __stdcall f3() throw();` déclaration utilise la syntaxe définie par la norme C++. Dans C++17, le `throw()` mot clé a été déprécié.

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
noexcept
Mots clés

novtable

Article • 16/06/2023

Section spécifique à Microsoft

Il s'agit d'un `__declspec` attribut étendu.

Cette forme de `__declspec` peut être appliquée à n'importe quelle déclaration de classe, mais ne doit être appliquée qu'aux classes d'interface pures, c'est-à-dire aux classes qui ne seront jamais instanciées seules. le `__declspec` empêche le compilateur de générer du code pour initialiser le vfptr dans le ou les constructeurs et destructeurs de la classe . Dans de nombreux cas, cela supprime les seules références à la vtable qui sont associés à la classe et, par conséquent, l'éditeur de liens supprimera cette vtable. L'utilisation de cette forme de `__declspec` peut entraîner une réduction significative de la taille du code.

Si vous tentez d'instancier une classe marquée avec `novtable` puis d'accéder à un membre de classe, vous recevrez une violation d'accès (AV).

Exemple

C++

```
// novtable.cpp
#include <stdio.h>

struct __declspec(novtable) X {
   virtual void mf();
};

struct Y : public X {
   void mf() {
      printf_s("In Y\n");
   }
};

int main() {
   // X *pX = new X();
   // pX->mf();   // Causes a runtime access violation.

   Y *pY = new Y();
   pY->mf();
}
```

Output

```
In Y
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés

`process`

Article • 03/04/2023

Spécifie que votre processus d'application managé doit comporter une seule copie d'une variable globale particulière, d'une variable membre static ou d'une variable locale static partagée par tous les domaines d'application du processus. Il était principalement destiné à être utilisé lors de la compilation avec `/clr:pure`, qui est déconseillé dans Visual Studio 2015 et non pris en charge dans Visual Studio 2017. Lors de la compilation avec `/clr`, les variables globales et statiques sont par processus par défaut et n'ont pas besoin d'utiliser `__declspec(process)`.

Seule une variable globale, une variable membre statique ou une variable locale statique de type natif peut être marquée avec `__declspec(process)`.

`process` est valide uniquement lors de la compilation avec /clr.

Si vous souhaitez que chaque domaine d'application ait sa propre copie d'une variable globale, utilisez appdomain.

Pour plus d'informations, consultez Domaines d'application et Visual C++ .

Voir aussi

__declspec
Mots clés

property (C++)

Article • 16/06/2023

Section spécifique à Microsoft

Cet attribut peut être appliqué à des « données membres virtuelles » non statiques dans une définition de classe ou de structure. Le compilateur traite ces « données membres virtuelles » comme des données membres en changeant leurs références en appels de fonction.

Syntaxe

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

Notes

Lorsque le compilateur voit un membre de données déclaré avec cet attribut à droite d'un opérateur de sélection de membre (« . » ou « ->«), il convertit l'opération en `get` fonction ou `put` , selon qu'une telle expression est une valeur l ou une valeur r. Dans des contextes plus complexes, tels que « `+=` », une réécriture est effectuée en effectuant à la fois `get` et `put`.

Cet attribut peut également être utilisé dans la déclaration d'un tableau vide dans une définition de classe ou de structure. Par exemple :

C++

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

L'instruction ci-dessus indique que `x[]` peut être utilisé avec un ou plusieurs index de tableau. Dans ce cas, `i=p->x[a][b]` est converti en `i=p->GetX(a, b)` et `p->x[a][b] = i` est converti en `p->PutX(a, b, i);`

FIN de la section spécifique à Microsoft

Exemple

C++

```
// declspec_property.cpp
struct S {
   int i;
   void putprop(int j) {
      i = j;
   }

   int getprop() {
      return i;
   }

   __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
   S s;
   s.the_prop = 5;
   return s.the_prop;
}
```

Voir aussi

__declspec

Mots clés

restrict

Article • 16/06/2023

Section spécifique à Microsoft

Lorsqu'elle est appliquée à une déclaration ou définition de fonction qui retourne un type pointeur, `restrict` indique au compilateur que la fonction retourne un objet qui n'est pas *alias*, c'est-à-dire référencé par d'autres pointeurs. Cela permet au compilateur d'effectuer des optimisations supplémentaires.

Syntaxe

> `__declspec(restrict)` *pointer_return_typefunction*();

Notes

Le compilateur propage `__declspec(restrict)`. Par exemple, la fonction CRT `malloc` a une `__declspec(restrict)` décoration et, par conséquent, le compilateur suppose que les pointeurs initialisés vers des emplacements de mémoire par `malloc` ne sont pas non plus alias par les pointeurs existants.

Le compilateur ne case activée pas que le pointeur retourné n'est pas réellement alias. Il incombe au développeur de s'assurer que le programme n'alias pas un pointeur marqué avec le modificateur **restreindre __declspec** .

Pour une sémantique similaire sur les variables, consultez __restrict.

Pour une autre annotation qui s'applique aux alias dans une fonction, consultez __declspec(noalias).

Pour plus d'informations sur les `restrict` mot clé qui font partie de C++ AMP, consultez restreindre (C++ AMP).

Exemple

L'exemple suivant illustre l'utilisation de `__declspec(restrict)`.

Lorsque `__declspec(restrict)` est appliqué à une fonction qui retourne un pointeur, cela indique au compilateur que la mémoire pointée par la valeur de retour n'est pas aliasée. Dans cet exemple, les pointeurs `mempool` et `memptr` sont globaux, de sorte que le

compilateur ne peut pas être sûr que la mémoire à laquelle ils font référence n'est pas alias. Toutefois, ils sont utilisés dans `ma` et son appelant `init` d'une manière qui retourne de la mémoire qui n'est pas autrement référencée par le programme. Par conséquent , **__decslpec(restreindre)** est utilisé pour aider l'optimiseur. Cela est similaire à la façon dont les en-têtes CRT décorent les fonctions d'allocation, par `malloc` exemple en utilisant `__declspec(restrict)` pour indiquer qu'ils retournent toujours de la mémoire qui ne peut pas être aliasée par des pointeurs existants.

C

```c
// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1f/k++;
    return a;
}

void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                          a[i * N + k] *
```

```
                            b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}
```

FIN de la section spécifique à Microsoft

Voir aussi

Mots clés
__declspec
__declspec(noalias)

safebuffers

Article • 16/06/2023

Section spécifique à Microsoft

Indique au compilateur de ne pas insérer de vérifications de sécurité de dépassement de mémoire tampon pour une fonction.

Syntaxe

```
__declspec( safebuffers )
```

Notes

L'option de compilateur **/GS** permet au compilateur de tester les dépassements de mémoire tampon en insérant des vérifications de sécurité sur la pile. Les types de structures de données éligibles aux vérifications de sécurité sont décrits dans /GS (Buffer Security Check). Pour plus d'informations sur la détection de dépassement de mémoire tampon, consultez Fonctionnalités de sécurité dans MSVC.

Une révision manuelle du code par un expert ou une analyse externe peut déterminer qu'une fonction est protégée contre un dépassement de mémoire tampon. Dans ce cas, vous pouvez supprimer les vérifications de sécurité pour une fonction en appliquant la `__declspec(safebuffers)` mot clé à la déclaration de fonction.

> ⊗ **Attention**
>
> Les vérifications de sécurité de la mémoire tampon assurent une protection importante et ont un impact négligeable sur les performances. Par conséquent, nous vous recommandons de ne pas les supprimer, sauf dans la rare éventualité où les performances d'une fonction est un problème critique et où la fonction est réputée pour être sécurisée.

Fonctions inline

Une *fonction principale* peut utiliser un mot clé inlining pour insérer une copie d'une *fonction secondaire*. Si le `__declspec(safebuffers)` mot clé est appliqué à une fonction, la détection de dépassement de mémoire tampon est supprimée pour cette fonction. Toutefois, l'inlining affecte la `__declspec(safebuffers)` mot clé des manières suivantes.

Supposons que l'option de compilateur **/GS** soit spécifiée pour les deux fonctions, mais que la fonction principale spécifie le `__declspec(safebuffers)` mot clé. Les structures de données présentes dans la fonction secondaire lui permettent de faire l'objet de vérifications de sécurité, et la fonction ne supprime pas ces vérifications. Dans ce cas :

- Spécifiez la __forceinline mot clé sur la fonction secondaire pour forcer le compilateur à l'inliner, quelles que soient les optimisations du compilateur.
- Étant donné que la fonction secondaire est éligible aux vérifications de sécurité, les vérifications de sécurité sont également appliquées à la fonction principale, même si elle spécifie le `__declspec(safebuffers)` mot clé.

Exemple

Le code suivant montre comment utiliser le `__declspec(safebuffers)` mot clé.

C++

```
// compile with: /c /GS
typedef struct {
    int x[20];
} BUFFER;
static int checkBuffers() {
    BUFFER cb;
    // Use the buffer...
    return 0;
};
static __declspec(safebuffers)
    int noCheckBuffers() {
    BUFFER ncb;
    // Use the buffer...
    return 0;
}
int wmain() {
    checkBuffers();
    noCheckBuffers();
    return 0;
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

inline, __inline, __forceinline

strict_gs_check

`selectany`

Article • 16/06/2023

Section spécifique à Microsoft

Indique au compilateur que l'élément de données global déclaré (variable ou objet) est un COMDAT pick-any (une fonction packagée).

Syntaxe

> `__declspec( selectany )` *declarator*

Notes

Au moment d'effectuer le lien, si plusieurs définitions d'un COMDAT s'affichent, l'éditeur de liens en choisit un et ignore le reste. Si l'option /OPT:REF de l'éditeur de liens (Optimisations) est sélectionnée, l'élimination de COMDAT se produit pour supprimer tous les éléments de données non référencés dans la sortie de l'éditeur de liens.

Les constructeurs et l'assignation par fonction globale ou méthodes statiques dans la déclaration ne créent pas de référence et n'empêchent pas la suppression /OPT:REF. Les effets secondaires d'un tel code ne doivent pas créer de dépendances quand il n'existe aucune autre référence à des données.

Pour les objets globaux initialisés dynamiquement, `selectany` ignore également le code d'initialisation d'un objet non référencé.

Un élément de données global ne peut généralement être initialisé qu'une seule fois dans un projet EXE ou DLL. `selectany` peut être utilisé pour initialiser des données globales définies par des en-têtes, lorsque le même en-tête apparaît dans plusieurs fichiers sources. `selectany` est disponible dans les compilateurs C et C++.

> ⓘ **Notes**
>
> `selectany` peut être appliqué à l'initialisation actuelle des éléments de données globaux qui sont visibles.

Exemple : `selectany` attribut

Ce code montre comment utiliser l'attribut `selectany` :

C++

```
//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;};
int i;
};

__declspec(selectany) X x(1);
```

Exemple : utiliser l'attribut `selectany` pour garantir le pliage COMDAT des données

Ce code montre comment utiliser l'attribut pour garantir le `selectany` pliage COMDAT des données lorsque vous utilisez également l'option /OPT:ICF éditeur de liens. Notez que les données doivent être marquées avec `selectany` et placées dans une `const` section (en lecture seule). Vous devez spécifier explicitement la section en lecture seule.

C++

```
// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
   int ij;
```

```
    ij = ix + jx;
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec

Mots clés

spectre

Article • 16/06/2023

Section spécifique à Microsoft

Indique au compilateur de ne pas insérer d'instructions de barrière d'exécution spéculative de la variante 1 spectre pour une fonction.

Syntaxe

> **__declspec(spectre(nomitigation))**

Notes

L'option du compilateur /Qspectre oblige le compilateur à insérer des instructions de barrière d'exécution spéculative. Ils sont insérés à l'emplacement où l'analyse indique qu'une vulnérabilité de sécurité spectre variant 1 existe. Les instructions spécifiques émises dépendent du processeur. Bien que ces instructions aient un impact minimal sur la taille ou les performances du code, il peut arriver que votre code ne soit pas affecté par la vulnérabilité et nécessite des performances maximales.

Une analyse d'expert peut déterminer qu'une fonction est à l'abri d'une limite de spectre de la variante 1 case activée défaut de contournement. Dans ce cas, vous pouvez supprimer la génération de code d'atténuation au sein d'une fonction en l'appliquant `__declspec(spectre(nomitigation))` à la déclaration de fonction.

> ⊗ **Attention**
>
> Les instructions de barrière d'exécution spéculative **/Qspectre** fournissent une protection de sécurité importante et ont un impact négligeable sur les performances. Par conséquent, nous vous recommandons de ne pas les supprimer, sauf dans la rare éventualité où les performances d'une fonction est un problème critique et où la fonction est réputée pour être sécurisée.

Exemple

Le code suivant illustre l'utilisation de `__declspec(spectre(nomitigation))`.

```
C++
```

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés
/Qspectre

thread

Article • 16/06/2023

Section spécifique à Microsoft

Le `thread` modificateur de classe de stockage étendu est utilisé pour déclarer une variable locale de thread. Pour l'équivalent portable en C++11 et versions ultérieures, utilisez le spécificateur de classe de stockage thread_local pour le code portable. Sur Windows `thread_local` est implémenté avec `__declspec(thread)`.

Syntaxe

`__declspec(thread)` *declarator*

Notes

Le stockage local des threads (TLS) est le mécanisme par lequel chaque thread d'un processus multithread alloue de l'espace de stockage pour les données spécifiques aux threads. Dans les programmes multithread standard, les données sont partagées entre tous les threads d'un processus donné, alors que le stockage local des threads est le mécanisme d'allocation des données par thread. Pour une présentation complète des threads, consultez Multithreading.

Les déclarations des variables locales de thread doivent utiliser la syntaxe d'attribut étendue et la `__declspec` mot clé avec le `thread` mot clé. Par exemple, le code suivant déclare une variable locale de thread entière et lui affecte une valeur initiale :

```
C++

__declspec( thread ) int tls_i = 1;
```

Lorsque vous utilisez des variables locales de thread dans des bibliothèques chargées dynamiquement, vous devez connaître les facteurs qui peuvent empêcher l'initialisation correcte d'une variable locale de thread :

1. Si la variable est initialisée avec un appel de fonction (y compris des constructeurs), cette fonction sera appelée uniquement pour le thread qui a provoqué le chargement de la DLL/binaire dans le processus, et pour les threads qui ont démarré après le chargement de la DLL/binaire. Les fonctions d'initialisation ne sont appelées pour aucun autre thread qui était déjà en cours d'exécution lors du

chargement de la DLL. L'initialisation dynamique se produit sur l'appel DllMain pour DLL_THREAD_ATTACH, mais la DLL n'obtient jamais ce message si la DLL n'est pas dans le processus au démarrage du thread.

2. Les variables locales de thread qui sont initialisées statiquement avec des valeurs constantes sont généralement initialisées correctement sur tous les threads. Toutefois, en décembre 2017, il existe un problème de conformité connu dans le compilateur Microsoft C++, dans lequel `constexpr` les variables reçoivent une initialisation dynamique plutôt que statique.

   Remarque : ces deux problèmes devraient être résolus dans les prochaines mises à jour du compilateur.

En outre, vous devez respecter ces instructions lors de la déclaration d'objets et de variables locaux de thread :

- Vous pouvez appliquer l'attribut `thread` uniquement aux déclarations et définitions de classes et de données ; `thread` ne peut pas être utilisé sur les déclarations ou définitions de fonction.

- Vous pouvez spécifier l'attribut `thread` uniquement sur les éléments de données avec une durée de stockage statique. Cela inclut les objets de données globaux (à la fois et `extern`), les `static` objets statiques locaux et les membres de données statiques des classes. Vous ne pouvez pas déclarer d'objets de données automatiques avec l'attribut `thread` .

- Vous devez utiliser l'attribut `thread` pour la déclaration et la définition d'un objet local de thread, que la déclaration et la définition se produisent dans le même fichier ou dans des fichiers distincts.

- Vous ne pouvez pas utiliser l'attribut `thread` comme modificateur de type.

- Étant donné que la déclaration des objets qui utilisent l'attribut `thread` est autorisée, ces deux exemples sont sémantiquement équivalents :

```
C++

// declspec_thread_2.cpp
// compile with: /LD
__declspec( thread ) class B {
public:
   int data;
} BObject;   // BObject declared thread local.

class B2 {
public:
```

```
   int data;
};
__declspec( thread ) B2 BObject2;   // BObject2 declared thread local.
```

- La norme C autorise l'initialisation d'un objet ou d'une variable avec une expression impliquant une référence à lui-même, mais uniquement pour les objets non statiques. Bien que C++ autorise normalement une telle initialisation dynamique d'un objet avec une expression impliquant une référence à lui-même, ce type d'initialisation n'est pas autorisé avec les objets locaux de thread. Par exemple :

  C++

  ```
  // declspec_thread_3.cpp
  // compile with: /LD
  #define Thread __declspec( thread )
  int j = j;   // Okay in C++; C error
  Thread int tls_i = sizeof( tls_i );   // Okay in C and C++
  ```

  Une `sizeof` expression qui inclut l'objet en cours d'initialisation ne constitue pas une référence à lui-même et est autorisée en C et C++.

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés
Stockage local de thread (TLS)

uuid (C++)

Article • 16/06/2023

Section spécifique à Microsoft

Le compilateur attache un GUID à une classe ou une structure déclarée ou définie (définitions d'objet COM complètes uniquement) avec l'attribut `uuid` .

Syntaxe

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Notes

L'attribut `uuid` prend une chaîne comme argument. Cette chaîne nomme un GUID au format de Registre normal avec ou sans les délimiteurs **{ }** . Par exemple :

C++

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

Cet attribut peut être appliqué dans une redéclaration. Cela permet aux en-têtes système de fournir les définitions d'interfaces telles que `IUnknown`, et la redeclaration dans un autre en-tête (comme <comdef.h>) pour fournir le GUID.

Le mot clé __uuidof peut être appliqué pour récupérer le GUID constant attaché à un type défini par l'utilisateur.

FIN de la section spécifique à Microsoft

Voir aussi

__declspec
Mots clés

`__restrict`

Article • 03/04/2023

Comme le **`__declspec` modificateur (restrict),** le `__restrict` mot clé (deux traits de soulignement de début '_') indique qu'un symbole n'est pas alias dans l'étendue actuelle. Le `__restrict` mot clé diffère du `__declspec (restrict)` modificateur des manières suivantes :

- Le `__restrict` mot clé est valide uniquement sur les variables, et `__declspec (restrict)` n'est valide que sur les déclarations et définitions de fonction.
- `__restrict` est similaire à restrict pour C à partir de C99 et disponible en /std:c11 mode ou /std:c17 , mais `__restrict` peut être utilisé dans les programmes C++ et C.
- Quand `__restrict` est utilisé, le compilateur ne propage pas la propriété no-alias d'une variable. Autrement dit, si vous affectez une `__restrict` variable à une autre`__restrict` variable, le compilateur autorise toujours l'alias de la variable non __restrict. Cela diffère du comportement du langage `restrict` C99 mot clé.

En règle générale, si vous souhaitez affecter le comportement d'une fonction entière, utilisez `__declspec (restrict)` au lieu du mot clé.

Pour la compatibilité avec les versions précédentes, `_restrict` est un synonyme de `__restrict` sauf si l'option /Za de compilateur (Désactiver les extensions de langage) est spécifiée.

Dans Visual Studio 2015 et versions ultérieures, `__restrict` peut être utilisé sur les références C++.

> ⓘ **Notes**
>
> En cas d'utilisation sur une variable qui a également le **volatile** mot clé, `volatile` est prioritaire.

Exemple

```
C++
```

```
// __restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
   int i;
   for (i = 0; i < n; i++) {
      a[i] = b[i] + c[i];
      c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
   int * __restrict x;
   double * __restrict y;
};
```

Voir aussi

Mots clés

__sptr, __uptr

Article • 16/06/2023

Section spécifique à Microsoft

Utilisez le `__sptr` modificateur ou `__uptr` sur une déclaration de pointeur 32 bits pour spécifier la façon dont le compilateur convertit un pointeur 32 bits en pointeur 64 bits. Un pointeur 32 bits est converti, par exemple, lorsqu'il est assigné à une variable pointeur 64 bits ou est déréférencé sur une plateforme 64 bits.

La documentation Microsoft pour la prise en charge des plateformes 64 bits fait parfois référence au bit le plus significatif d'un pointeur 32 bits comme bit de signe. Par défaut, le compilateur utilise l'extension de signe pour convertir un pointeur 32 bits en pointeur 64 bits. Autrement dit, les 32 bits les moins significatifs du pointeur 64 bits sont définis avec la valeur du pointeur 32 bits et les 32 bits les plus significatifs sont définis avec la valeur du bit de signe du pointeur 32 bits. Cette conversion génère des résultats corrects si le bit de signe est 0, pas si le bit de signe est 1. Par exemple, l'adresse 32 bits 0x7FFFFFFF retourne l'adresse 64 bits équivalente 0x000000007FFFFFFF, mais l'adresse 32 bits 0x80000000 est modifiée incorrectement en 0xFFFFFFFF80000000.

Le `__sptr` modificateur , ou signé, spécifie qu'une conversion de pointeur définit les bits les plus significatifs d'un pointeur 64 bits vers le bit de signe du pointeur 32 bits. Le `__uptr` modificateur , ou pointeur non signé, spécifie qu'une conversion définit les bits les plus significatifs sur zéro. Les déclarations suivantes montrent les `__sptr` modificateurs et `__uptr` utilisés avec deux pointeurs non qualifiés, deux pointeurs qualifiés avec le type __ptr32 et un paramètre de fonction.

C++

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Utilisez les `__sptr` modificateurs et `__uptr` avec les déclarations de pointeur. Utilisez les modificateurs à la position d'un qualificateur de type pointeur, ce qui signifie que le modificateur doit suivre l'astérisque. Vous ne pouvez pas utiliser les modificateurs avec des pointeurs vers des membres. Les modificateurs n'ont pas d'incidence sur les déclarations non pointeur.

Pour la compatibilité avec les versions précédentes, **_sptr** et **_uptr** sont des synonymes de `__sptr` et `__uptr` , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Exemple

L'exemple suivant déclare des pointeurs 32 bits qui utilisent les `__sptr` modificateurs et `__uptr` , affecte chaque pointeur 32 bits à une variable pointeur 64 bits, puis affiche la valeur hexadécimale de chaque pointeur 64 bits. Il est compilé avec le compilateur 64 bits natif et exécuté sur une plateforme 64 bits.

C++

```
// sptr_uptr.cpp
// processor: x64
#include "stdio.h"

int main()
{
    void *        __ptr64 p64;
    void *        __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

// Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

// The printf() function automatically displays leading zeroes with each 32-
bit pointer. These are unrelated
// to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit
pointer):\n");
    printf("p32d:       %p\n", p32d);
    printf("p32s:       %p\n", p32s);
    printf("p32u:       %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit
pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}
```

Output

```
Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:       0000000087654321
p32s:       0000000087654321
p32u:       0000000087654321

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 0000000087654321
```

FIN de la section spécifique à Microsoft

Voir aussi

Modificateurs spécifiques à Microsoft

`__unaligned`

Article • 16/06/2023

Spécifique à Microsoft. Lorsque vous déclarez un pointeur avec le `__unaligned` modificateur, le compilateur suppose que le pointeur traite les données qui ne sont pas alignées. Ainsi, le code approprié à la plateforme est généré pour gérer les lectures et écritures non alignées via le pointeur.

Notes

Ce modificateur décrit l'alignement des données traitées par le pointeur. Il est supposé que le pointeur lui-même est aligné.

La nécessité de l'mot clé varie selon la `__unaligned` plateforme et l'environnement. Le fait de ne pas marquer les données de manière appropriée peut entraîner des problèmes allant des pénalités de performances aux pannes matérielles. Le `__unaligned` modificateur n'est pas valide pour la plateforme x86.

Pour la compatibilité avec les versions précédentes, `_unaligned` est un synonyme de `__unaligned` l'option /Za de compilateur (Désactiver les extensions de langage) spécifiée.

Pour plus d'informations sur l'alignement, consultez :

- align
- alignof (opérateur)
- pack
- /Zp (Alignement des membres de struct)
- Exemples d'alignement de structure x64

Voir aussi

Mots clés

__w64

Article • 16/06/2023

Cette mot clé spécifique à Microsoft est obsolète. Dans les versions de Visual Studio antérieures à Visual Studio 2013, cela vous permet de marquer des variables, de sorte que lorsque vous compilez avec /Wp64, le compilateur signale tous les avertissements qui seraient signalés si vous compilez avec un compilateur 64 bits.

Syntaxe

> *Type* `__w64` *Identificateur*

Paramètres

type
L'un des trois types susceptibles d'entraîner des problèmes dans le code porté d'un compilateur 32 bits vers un compilateur 64 bits : `int`, `long` ou un pointeur.

identifier
Identificateur de la variable que vous créez.

Notes

> **ⓘ Important**
>
> L'option du compilateur **/Wp64** et `__w64` les mot clé sont dépréciés dans Visual Studio 2010 et Visual Studio 2013 et supprimés à partir de Visual Studio 2013. Si vous utilisez l'option du `/Wp64` compilateur sur la ligne de commande, le compilateur émet des problèmes Command-Line Avertissement D9002. Le `__w64` mot clé est ignoré en mode silencieux. Au lieu d'utiliser cette option et mot clé pour détecter les problèmes de portabilité 64 bits, utilisez un compilateur Microsoft C++ qui cible une plateforme 64 bits. Pour plus d'informations, consultez **Configurer Visual C++ pour les cibles x64 64 bits**.

Tout typedef qui a `__w64` dessus doit être de 32 bits sur x86 et de 64 bits sur x64.

Pour détecter les problèmes de portabilité à l'aide de versions du compilateur Microsoft C++ antérieures à Visual Studio 2010, le `__w64` mot clé doit être spécifié sur les

typesdefs qui changent de taille entre les plateformes 32 bits et 64 bits. Pour un tel type, `__w64` doit apparaître uniquement sur la définition 32 bits du typedef.

Pour la compatibilité avec les versions précédentes, **_w64** est synonyme de `__w64` , sauf si l'option de compilateur /Za (Désactiver les extensions de langage) est spécifiée.

Le `__w64` mot clé est ignoré si la compilation n'utilise `/Wp64`pas .

Pour plus d'informations sur le portage vers 64 bits, consultez les rubriques suivantes :

- Options du compilateur MSVC
- Portage du code 32 bits vers du code 64 bits
- Configurer Visual C++ pour des cibles x64 64 bits

Exemple

C++

```
// __w64.cpp
// compile with: /W3 /Wp64
typedef int Int_32;
#ifdef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
   Int_32 i0 = 5;
   Int_Native i1 = 10;
   i0 = i1;   // C4244 64-bit int assigned to 32-bit int

   // char __w64 c;  error, cannot use __w64 on char
}
```

Voir aussi

Mots clés

__func__

Article • 16/06/2023

(C++11) L'identificateur `__func__` prédéfini est implicitement défini comme une chaîne qui contient le nom non qualifié et non qualifié de la fonction englobante. `__func__` est obligatoire par la norme C++ et n'est pas une extension Microsoft.

Syntaxe

C++

```
__func__
```

Valeur de retour

Retourne un tableau de caractères const avec fin null qui contient le nom de la fonction.

Exemple

C++

```
#include <string>
#include <iostream>

namespace Test
{
    struct Foo
    {
        static void DoSomething(int i, std::string s)
        {
           std::cout << __func__ << std::endl; // Output: DoSomething
        }
    };
}

int main()
{
    Test::Foo::DoSomething(42, "Hello");

    return 0;
}
```

Configuration requise

C++11

Prise en charge COM du compilateur

Article • 16/06/2023

Section spécifique à Microsoft

Le compilateur Microsoft C++ peut lire directement les bibliothèques de types COM (Component Object Model) et traduire le contenu en code source C++ qui peut être inclus dans la compilation. Des extensions de langage sont disponibles pour faciliter la programmation COM côté client pour les applications de bureau.

À l'aide de la directive de préprocesseur #import, le compilateur peut lire une bibliothèque de types et la convertir en un fichier d'en-tête C++ qui décrit les interfaces COM en tant que classes. Un ensemble d'attributs `#import` est disponible pour permettre le contrôle utilisateur du contenu pour les fichiers d'en-tête de la bibliothèque de types obtenus.

Vous pouvez utiliser l'uuidd'attribut étendu __declspec pour affecter un identificateur global unique (GUID) à un objet COM. Le __uuidof mot clé peut être utilisé pour extraire le GUID associé à un objet COM. Un autre `__declspec` attribut, propriété, peut être utilisé pour spécifier les `get` méthodes et `set` pour un membre de données d'un objet COM.

Un ensemble de fonctions et de classes com prises en charge globales est fourni pour prendre en charge les `VARIANT` types et `BSTR` , implémenter des pointeurs intelligents et encapsuler l'objet d'erreur levée par `_com_raise_error`:

- Fonctions globales COM du compilateur
- _bstr_t
- _com_error
- _com_ptr_t
- _variant_t

FIN de la section spécifique à Microsoft

Voir aussi

Classes de prise en charge COM du compilateur
Fonctions globales COM du compilateur

Fonctions globales COM du compilateur

Article • 16/06/2023

Section spécifique à Microsoft

Les routines suivantes sont disponibles :

Fonction	Description
_com_raise_error	Lève une _com_error en réponse à une défaillance.
_set_com_error_handler	Remplace la fonction par défaut utilisée pour la gestion des erreurs COM.
ConvertBSTRToString	Convertit une valeur `BSTR` en `char *`.
ConvertStringToBSTR	Convertit une valeur `char *` en `BSTR`.

FIN de la section spécifique à Microsoft

Voir aussi

Classes de prise en charge COM du compilateur
Prise en charge COM du compilateur

_com_raise_error

Article • 16/06/2023

Section spécifique à Microsoft

Lève une _com_error en réponse à un échec.

Syntaxe

C++

```
void __stdcall _com_raise_error(
   HRESULT hr,
   IErrorInfo* perrinfo = 0
);
```

Paramètres

heure(s)
Informations HRESULT.

perrinfo
Objet `IErrorInfo`.

Notes

_com_raise_error, qui est défini dans <comdef.h>, peut être remplacé par une version écrite par l'utilisateur du même nom et du même prototype. Cette opération peut être effectuée si vous souhaitez utiliser `#import` mais pas la gestion des exceptions C++. Dans ce cas, une version utilisateur de **_com_raise_error** peut décider d'effectuer une opération ou d'afficher `longjmp` une boîte de message et de s'arrêter. Toutefois, cette version utilisateur ne devrait pas être retournée, car le code de prise en charge COM du compilateur n'attend aucun retour de celle-ci.

Vous pouvez également utiliser _set_com_error_handler pour remplacer la fonction de gestion des erreurs par défaut.

Par défaut, **_com_raise_error** est défini comme suit :

C++

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
   throw _com_error(hr, perrinfo);
}
```

FIN de la section spécifique à Microsoft

Spécifications

En-tête:< comdef.h>

Lib: Si **l'option du compilateur wchar_t est Native Type** est activée, utilisez comsuppw.lib ou comsuppwd.lib. Si **wchar_t type natif** est désactivé, utilisez comsupp.lib. Pour plus d'informations, consultez /Zc:wchar_t (wchar_t est de type natif).

Voir aussi

Fonctions globales COM du compilateur
_set_com_error_handler

ConvertStringToBSTR

Article • 16/06/2023

Section spécifique à Microsoft

Convertit une valeur `char *` en `BSTR`.

Syntaxe

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

Paramètres

pSrc
Variable `char *` .

Exemple

C++

```
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
   char* lpszText = "Test";
   printf_s("char * text: %s\n", lpszText);

   BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
   wprintf_s(L"BSTR text: %s\n", bstrText);

   SysFreeString(bstrText);
}
```

Output

```
char * text: Test
BSTR text: Test
```

FIN de la section spécifique à Microsoft

Spécifications

En-tête:< comutil.h>

Lib : comsuppw.lib ou comsuppwd.lib (consultez /Zc:wchar_t (wchar_t type natif) pour plus d'informations)

Voir aussi

Fonctions globales COM du compilateur

ConvertBSTRToString

Article • 16/06/2023

Section spécifique à Microsoft

Convertit une valeur `BSTR` en `char *`.

Syntaxe

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

Paramètres

pSrc
Variable BSTR.

Notes

ConvertBSTRToString alloue une chaîne que vous devez supprimer.

Exemple

C++

```
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
   BSTR bstrText = ::SysAllocString(L"Test");
   wprintf_s(L"BSTR text: %s\n", bstrText);

   char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
   printf_s("char * text: %s\n", lpszText2);

   SysFreeString(bstrText);
   delete[] lpszText2;
}
```

Output

```
BSTR text: Test
char * text: Test
```

FIN de la section spécifique à Microsoft

Configuration requise

En-tête:< comutil.h>

Lib : comsuppw.lib ou comsuppwd.lib (voir /Zc:wchar_t (wchar_t Is Native Type) pour plus d'informations)

Voir aussi

Fonctions globales COM du compilateur

_set_com_error_handler

Article • 16/06/2023

Remplace la fonction par défaut utilisée pour la gestion des erreurs COM. **_set_com_error_handler** est spécifique à Microsoft.

Syntaxe

C++

```
void __stdcall _set_com_error_handler(
   void (__stdcall *pHandler)(
      HRESULT hr,
      IErrorInfo* perrinfo
   )
);
```

Paramètres

pHandler
Pointeur vers la fonction de remplacement.

heure(s)
Informations HRESULT.

perrinfo
Objet `IErrorInfo`.

Notes

Par défaut, _com_raise_error gère toutes les erreurs COM. Vous pouvez modifier ce comportement en utilisant **_set_com_error_handler** pour appeler votre propre fonction de gestion des erreurs.

La fonction de remplacement doit avoir une signature qui est équivalente à celle de `_com_raise_error`.

Exemple

C++

```cpp
// _set_com_error_handler.cpp
// compile with /EHsc
#include <stdio.h>
#include <comdef.h>
#include <comutil.h>

// Importing ado dll to attempt to establish an ado connection.
// Not related to _set_com_error_handler
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace
rename("EOF", "adoEOF")

void __stdcall _My_com_raise_error(HRESULT hr, IErrorInfo* perrinfo)
{
   throw "Unable to establish the connection!";
}

int main()
{
   _set_com_error_handler(_My_com_raise_error);
   _bstr_t bstrEmpty(L"");
   _ConnectionPtr Connection = NULL;
   try
   {
      Connection.CreateInstance(__uuidof(Connection));
      Connection->Open(bstrEmpty, bstrEmpty, bstrEmpty, 0);
   }
   catch(char* errorMessage)
   {
      printf("Exception raised: %s\n", errorMessage);
   }

   return 0;
}
```

Output

```
Exception raised: Unable to establish the connection!
```

Spécifications

En-tête:< comdef.h>

Lib: Si l'option du compilateur **/Zc:wchar_t** est spécifiée (valeur par défaut), utilisez comsuppw.lib ou comsuppwd.lib. Si l'option du compilateur **/Zc:wchar_t-** est spécifiée, utilisez comsupp.lib. Pour plus d'informations, notamment sur la définition de cette option dans l'IDE, consultez /Zc:wchar_t (wchar_t est un type natif).

Voir aussi

Fonctions globales COM du compilateur

Classes du support COM du compilateur

Article • 16/06/2023

Section spécifique à Microsoft

Les classes standard sont utilisées pour prendre en charge certains types COM. Les classes sont définies dans <comdef.h> et les fichiers d'en-tête générés à partir de la bibliothèque de types.

Classe	Objectif
_bstr_t	Encapsule le type `BSTR` pour fournir des opérateurs et des méthodes utiles.
_com_error	Définit l'objet d'erreur levée par _com_raise_error dans la plupart des échecs.
_com_ptr_t	Encapsule les pointeurs d'interface COM et automatise les appels requis à `AddRef`, `Release` et `QueryInterface`.
_variant_t	Encapsule le type `VARIANT` pour fournir des opérateurs et des méthodes utiles.

FIN de la section spécifique à Microsoft

Voir aussi

Prise en charge COM du compilateur
Fonctions globales COM du compilateur
Informations de référence sur le langage C++

Classe `_bstr_t`

Article • 16/06/2023

Section spécifique à Microsoft

Un `_bstr_t` objet encapsule le type de données BSTR. La classe gère l'allocation et la désallocation des ressources par le biais d'appels de fonction à SysAllocString et SysFreeString et d'autres `BSTR` API, le cas échéant. La `_bstr_t` classe utilise le comptage des références pour éviter une surcharge excessive.

Membres

Construction

Constructeur	Description
_bstr_t	Construit un objet `_bstr_t`.

Opérations

Fonction	Description
Assign	Copie un `BSTR` dans le `BSTR` encapsulé par un `_bstr_t`.
Attach	Lie un wrapper `_bstr_t` à un `BSTR`.
copy	Construit une copie du `BSTR` encapsulé.
Detach	Retourne le `BSTR` encapsulé par un `_bstr_t` et détache `BSTR` du `_bstr_t`.
GetAddress	Pointe vers le `BSTR` encapsulé par un `_bstr_t`.
GetBSTR	Pointe sur le début du `BSTR` encapsulé par l'objet `_bstr_t`.
length	Retourne le nombre de caractères du `_bstr_t`.

Opérateurs

Opérateur	Description
operator =	Assigne une nouvelle valeur à un objet `_bstr_t` existant.

Opérateur	Description
operator +=	Ajoute des caractères à la fin de l'objet `_bstr_t`.
operator +	Concatène deux chaînes.
operator !	Vérifie si l'encapsulé `BSTR` est une chaîne NULL.
operator == operator != operator < operator > operator <= operator >=	Compare deux objets `_bstr_t`.
operator wchar_t* operator char*	Extrayez les pointeurs dans l'objet `BSTR` Unicode ou multioctets encapsulé.

FIN de la section spécifique à Microsoft

Configuration requise

En-tête: < comutil.h>

Lib: *`comsuppw.lib`* ou *`comsuppwd.lib`* (Pour plus d'informations, consultez /Zc:wchar_t (wchar_t est de type natif))

Voir aussi

Classes de prise en charge de COM du compilateur

`_bstr_t::_bstr_t`

Article • 16/06/2023

Section spécifique à Microsoft

Construit un objet `_bstr_t`.

Syntaxe

C++

```
_bstr_t( ) throw( );
_bstr_t(
   const _bstr_t& s1
) throw( );
_bstr_t(
   const char* s2
);
_bstr_t(
   const wchar_t* s3
);
_bstr_t(
   const _variant_t& var
);
_bstr_t(
   BSTR bstr,
   bool fCopy
);
```

Paramètres

`s1`

Objet `_bstr_t` à copier.

`s2`

Chaîne multioctet.

`s3`

Chaîne Unicode.

`var`

Objet _variant_t .

bstr
Objet BSTR existant.

fCopy
Si **false** la valeur est , l'argument *bstr* est attaché au nouvel objet sans effectuer de copie en appelant SysAllocString.

Notes

La _bstr_t classe fournit plusieurs constructeurs :

_bstr_t( )
Construit un objet par défaut _bstr_t qui encapsule un objet null BSTR .

_bstr_t(_bstr_t& s1)
Construit un objet _bstr_t comme copie d'un autre. Ce constructeur effectue une copie *superficielle* , qui incrémente le nombre de références de l'objet encapsulé BSTR au lieu d'en créer un nouveau.

_bstr_t(char* s2)
Construit un objet _bstr_t en appelant SysAllocString pour créer un nouvel objet BSTR, puis l'encapsule. Ce constructeur effectue en premier lieu une conversion de chaîne multioctet vers Unicode.

_bstr_t(wchar_t* s3)
Construit un objet _bstr_t en appelant SysAllocString pour créer un nouvel objet BSTR, puis l'encapsule.

_bstr_t(_variant_t& var)
Construit un _bstr_t objet à partir d'un _variant_t objet en récupérant d'abord un BSTR objet à partir de l'objet encapsulé VARIANT .

_bstr_t(BSTR bstr, bool fCopy)
Construit un objet _bstr_t à partir d'un BSTR existant (par opposition à une chaîne wchar_t*). Si *fCopy* a la valeur **false**, le fourni BSTR est attaché au nouvel objet sans effectuer de nouvelle copie à l'aide de SysAllocString. Ce constructeur est utilisé par les fonctions wrapper dans les en-têtes de bibliothèque de types pour encapsuler et prendre possession d'un BSTR qui est retourné par une méthode d'interface.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

_variant_t, classe

`_bstr_t::Assign`

Article • 16/06/2023

Section spécifique à Microsoft

Copie un `BSTR` dans le `BSTR` encapsulé par un `_bstr_t`.

Syntaxe

C++

```
void Assign(
   BSTR s
);
```

Paramètres

`s`
`BSTR` à copier dans le `BSTR` encapsulé par un `_bstr_t`.

Notes

`Assign` effectue une copie binaire de toute la longueur de , `BSTR`quel que soit le contenu.

Exemple

C++

```
// _bstr_t_Assign.cpp

#include <comdef.h>
#include <stdio.h>

int main()
{
    // creates a _bstr_t wrapper
    _bstr_t bstrWrapper;

    // creates BSTR and attaches to it
    bstrWrapper = "some text";
    wprintf_s(L"bstrWrapper = %s\n",
```

```
            static_cast<wchar_t*>(bstrWrapper));

    // bstrWrapper releases its BSTR
    BSTR bstr = bstrWrapper.Detach();
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));
    // "some text"
    wprintf_s(L"bstr = %s\n", bstr);

    bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocedString")));
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // assign a BSTR to our _bstr_t
    bstrWrapper.Assign(bstr);
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));

    // done with BSTR, do manual cleanup
    SysFreeString(bstr);

    // resuse bstr
    bstr= SysAllocString(OLESTR("Yet another string"));
    // two wrappers, one BSTR
    _bstr_t bstrWrapper2 = bstrWrapper;

    *bstrWrapper.GetAddress() = bstr;

    // bstrWrapper and bstrWrapper2 do still point to BSTR
    bstr = 0;
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
              static_cast<wchar_t*>(bstrWrapper2));

    // new value into BSTR
    _snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
                 L"changing BSTR");
    wprintf_s(L"bstrWrapper = %s\n",
              static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
              static_cast<wchar_t*>(bstrWrapper2));
}
```

Output

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
```

```
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::Attach`

Article • 16/06/2023

Section spécifique à Microsoft

Lie un wrapper `_bstr_t` à un `BSTR`.

Syntaxe

```
C++
```

```
void Attach(
   BSTR s
);
```

Paramètres

`s`
`BSTR` à associer ou assigné à la variable `_bstr_t`.

Notes

Si `_bstr_t` a été précédemment attaché à un autre `BSTR`, `_bstr_t` nettoie la ressource `BSTR`, si aucune autre variable `_bstr_t` n'utilise `BSTR`.

Exemple

Consultez _bstr_t::Assign pour obtenir un exemple d'utilisation de **`Attach`**.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t Classe

`_bstr_t::copy`

Article • 16/06/2023

Section spécifique à Microsoft

Construit une copie du `BSTR` encapsulé.

Syntaxe

```
C++

BSTR copy( bool fCopy = true ) const;
```

Paramètres

`fCopy`
Si **`true`**, **`copy`** retourne une copie du contenu , `BSTR` sinon **`copy`** retourne le réel `BSTR`.

Notes

Retourne une copie nouvellement allouée de l'objet encapsulé `BSTR` , ou de l'objet encapsulé lui-même, en fonction du paramètre.

Exemple

```
C++

STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ //  m_bsConStr is
_bstr_t
   *pVal = m_bsConStr.copy();
}
```

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::Detach`

Article • 16/06/2023

Section spécifique à Microsoft

Retourne le `BSTR` encapsulé par un `_bstr_t` et détache `BSTR` du `_bstr_t`.

Syntaxe

```
C++

BSTR Detach( ) throw;
```

Valeur de retour

Retourne le `BSTR` encapsulé par .`_bstr_t`

Exemple

Consultez _bstr_t::Assign pour obtenir un exemple qui utilise **`Detach`**.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::GetAddress`

Article • 16/06/2023

Section spécifique à Microsoft

Libère toute chaîne existante et retourne l'adresse d'une chaîne nouvellement allouée.

Syntaxe

```
C++

BSTR* GetAddress( );
```

Valeur de retour

Pointeur désignant le `BSTR` encapsulé par l'objet `_bstr_t`.

Notes

`GetAddress` affecte tous les `_bstr_t` objets qui partagent un `BSTR`. Plusieurs `_bstr_t` peuvent partager un `BSTR` via l'utilisation du constructeur de copie et **`operator=`**.

Exemple

Consultez _bstr_t::Assign pour obtenir un exemple qui utilise **`GetAddress`**.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::GetBSTR`

Article • 16/06/2023

Section spécifique à Microsoft

Pointe sur le début du `BSTR` encapsulé par l'objet `_bstr_t`.

Syntaxe

```
C++

BSTR& GetBSTR( );
```

Valeur de retour

Début du `BSTR` encapsulé par l'objet `_bstr_t`.

Notes

`GetBSTR` affecte tous les `_bstr_t` objets qui partagent un `BSTR`. `_bstr_t` Plusieurs peuvent partager un `BSTR` via l'utilisation du constructeur de copie et `operator=`.

Exemple

Consultez _bstr_t::Assign pour obtenir un exemple qui utilise **`GetBSTR`**.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::length`

Article • 16/06/2023

Section spécifique à Microsoft

Retourne le nombre de caractères du `_bstr_t`, sans caractère Null de fin, du `BSTR` encapsulé.

Syntaxe

```
C++
unsigned int length ( ) const throw( );
```

Notes

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::operator =`

Article • 16/06/2023

Section spécifique à Microsoft

Assigne une nouvelle valeur à un objet `_bstr_t` existant.

Syntaxe

```
C++
```

```
_bstr_t& operator=(const _bstr_t& s1) throw ( );
_bstr_t& operator=(const char* s2);
_bstr_t& operator=(const wchar_t* s3);
_bstr_t& operator=(const _variant_t& var);
```

Paramètres

`s1`
Objet `_bstr_t` à assigner à un objet `_bstr_t` existant.

`s2`
Chaîne multioctets à assigner à un objet `_bstr_t` existant.

`s3`
Chaîne Unicode à assigner à un objet `_bstr_t` existant.

`var`
Objet `_variant_t` à assigner à un objet `_bstr_t` existant.

FIN de la section spécifique à Microsoft

Exemple

Consultez _bstr_t::Assign pour obtenir un exemple qui utilise **`operator=`**.

Voir aussi

_bstr_t, classe

`_bstr_t::operator +=`, `_bstr_t::operator +`

Article • 16/06/2023

Section spécifique à Microsoft

Ajoute des caractères à la fin de l'objet `_bstr_t` ou concatène deux chaînes.

Syntaxe

```
C++
```

```
_bstr_t& operator+=( const _bstr_t& s1 );
_bstr_t operator+( const _bstr_t& s1 );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1);
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1);
```

Paramètres

`s1`
Objet `_bstr_t`.

`s2`
Chaîne multioctet.

`s3`
Chaîne Unicode.

Notes

Ces opérateurs exécutent une concaténation de chaînes :

- `operator+=( s1 )` Ajoute les caractères dans le encapsulé `BSTR` de *`s1`* à la fin de l'encapsulé `BSTR`de cet objet .
- `operator+( s1 )` Retourne le nouveau `_bstr_t` formé par la concaténation de `BSTR` cet objet et de celui dans *`s1`*.
- `operator+( s2, s1 )` Retourne une nouvelle `_bstr_t` qui est formée par concaténation d'une chaîne *`s2`*multioctet , convertie en Unicode et encapsulée

`BSTR` dans *`s1`*.

- `operator+( s3, s1 )` Retourne un nouveau `_bstr_t` formé en concaténant une chaîne *`s3`* Unicode et le `BSTR` encapsulé dans *`s1`*.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::operator !`

Article • 16/06/2023

Section spécifique à Microsoft

Vérifie si l'encapsulé `BSTR` est une chaîne NULL.

Syntaxe

```
C++
```

```
bool operator!( ) const throw( );
```

Valeur de retour

Elle retourne **`true`** si l'encapsulé `BSTR` est une chaîne NULL, **`false`** si ce n'est pas le cas.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t` opérateurs relationnels

Article • 16/06/2023

Section spécifique à Microsoft

Compare deux objets `_bstr_t`.

Syntaxe

C++

```
bool operator==(const _bstr_t& str) const throw( );
bool operator!=(const _bstr_t& str) const throw( );
bool operator<(const _bstr_t& str) const throw( );
bool operator>(const _bstr_t& str) const throw( );
bool operator<=(const _bstr_t& str) const throw( );
bool operator>=(const _bstr_t& str) const throw( );
```

Notes

Ces opérateurs comparent deux objets `_bstr_t` du point de vue lexicographique. Les opérateurs retournent **`true`** si les comparaisons tiennent; sinon, retournent **`false`**.

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

`_bstr_t::wchar_t*`, `_bstr_t::char*`

Article • 16/06/2023

Section spécifique à Microsoft

Retourne les `BSTR` caractères sous forme d'un tableau de caractères étroit ou large.

Syntaxe

C++

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Notes

Ces opérateurs peuvent être utilisés pour extraire les données de caractères encapsulées par l'objet `BSTR` . L'attribution d'une nouvelle valeur au pointeur retourné ne modifie pas les données d'origine `BSTR` .

FIN de la section spécifique à Microsoft

Voir aussi

_bstr_t, classe

Classe `_com_error`

Article • 16/06/2023

Section spécifique à Microsoft

Un `_com_error` objet représente une condition d'exception détectée par les fonctions wrapper de gestion des erreurs dans les fichiers d'en-tête générés à partir de la bibliothèque de types ou par l'une des classes de prise en charge COM. La `_com_error` classe encapsule le `HRESULT` code d'erreur et tout objet associé `IErrorInfo Interface` .

Construction

Nom	Description
_com_error	Construit un objet `_com_error`.

Opérateurs

Nom	Description
operator =	Assigne un objet `_com_error` existant à un autre.

Fonctions d'extracteur

Nom	Description
Error	Récupère le `HRESULT` passé au constructeur.
ErrorInfo	Récupère l'objet `IErrorInfo` passé au constructeur.
WCode	Récupère le code d'erreur 16 bits mappé dans le encapsulé `HRESULT`.

fonctions `IErrorInfo`

Nom	Description
Description	Appelle `IErrorInfo::GetDescription` la fonction .
HelpContext	Appelle `IErrorInfo::GetHelpContext` la fonction .
HelpFile	Fonction Appels `IErrorInfo::GetHelpFile`

Nom	Description
Source	Appelle `IErrorInfo::GetSource` la fonction .
GUID	Appelle `IErrorInfo::GetGUID` la fonction .

Extracteur de messages de mise en forme

Nom	Description
ErrorMessage	Récupère le message de chaîne pour `HRESULT` stocké dans l'objet `_com_error` .

`ExepInfo.wCode` aux `HRESULT` mappeurs

Nom	Description
HRESULTToWCode	Mappe 32 bits `HRESULT` à 16 bits `wCode`.
WCodeToHRESULT	Mappe 16 bits `wCode` à 32 bits `HRESULT`.

FIN de la section spécifique à Microsoft

Spécifications

En-tête: < comdef.h>

Bibliothèque: *`comsuppw.lib`* ou *`comsuppwd.lib`* (pour plus d'informations, consultez /Zc:wchar_t (wchar_t est de type natif))

Voir aussi

Classes de prise en charge de COM du compilateur
IErrorInfo Interface

`_com_error` fonctions membres

Article • 03/04/2023

Pour plus d'informations sur les `_com_error` fonctions membres, consultez _com_error classe.

Voir aussi

_com_error, classe

`_com_error::_com_error`

Article • 16/06/2023

Section spécifique à Microsoft

Construit un objet `_com_error`.

Syntaxe

```
C++
```

```
_com_error(
   HRESULT hr,
   IErrorInfo* perrinfo = NULL,
   bool fAddRef = false) throw();

_com_error( const _com_error& that ) throw();
```

Paramètres

`hr`
`HRESULT` Informations.

`perrinfo`
Objet `IErrorInfo`.

`fAddRef`
La valeur par défaut empêche le constructeur d'appeler AddRef sur une interface non null `IErrorInfo` . Ce comportement permet de compter correctement les références dans le cas courant où la propriété de l'interface est passée à l'objet `_com_error` , par exemple :

```
C++
```

```
throw _com_error(hr, perrinfo);
```

Si vous ne souhaitez pas que votre code transfère la propriété à l'objet `_com_error` et que le `AddRef` est requis pour décaler dans `Release` le `_com_error` destructeur, construisez l'objet comme suit :

```
C++
```

```
_com_error err(hr, perrinfo, true);
```

that
Objet `_com_error` existant.

Notes

Le premier constructeur crée un objet en fonction d'un `HRESULT` objet et facultatif `IErrorInfo` . La seconde crée une copie d'un objet existant `_com_error` .

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::Description`

Article • 16/06/2023

Section spécifique à Microsoft

Appelle `IErrorInfo::GetDescription` la fonction .

Syntaxe

```
C++
```

```
_bstr_t Description() const;
```

Valeur de retour

Retourne le résultat de `IErrorInfo::GetDescription` pour l'objet `IErrorInfo` enregistré dans l'objet `_com_error` . Le `BSTR` résultant est encapsulé dans un objet `_bstr_t`. Si aucun n'est `IErrorInfo` enregistré, il retourne un vide `_bstr_t`.

Notes

Appelle la `IErrorInfo::GetDescription` fonction et récupère les données enregistrées `IErrorInfo` dans l'objet `_com_error` . Tout échec lors de l'appel de la `IErrorInfo::GetDescription` méthode est ignoré.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::Error`

Article • 16/06/2023

Section spécifique à Microsoft

Récupère le `HRESULT` passé au constructeur.

Syntaxe

```
C++
```

```
HRESULT Error() const throw();
```

Valeur de retour

L'élément `HRESULT` brut est transmis dans le constructeur.

Notes

Récupère l'élément encapsulé `HRESULT` dans un `_com_error` objet .

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

_com_error::ErrorInfo

Article • 16/06/2023

Section spécifique à Microsoft

Récupère l'objet `IErrorInfo` passé au constructeur.

Syntaxe

```
C++
```

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Valeur de retour

L'élément `IErrorInfo` brut est transmis dans le constructeur.

Notes

Récupère l'élément encapsulé `IErrorInfo` dans un `_com_error` objet, ou `NULL` si aucun élément n'est `IErrorInfo` enregistré. L'appelant doit appeler `Release` sur l'objet retourné quand il a terminé de l'utiliser.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::ErrorMessage`

Article • 16/06/2023

Section spécifique à Microsoft

Récupère le message de chaîne pour `HRESULT` stocké dans l'objet `_com_error` .

Syntaxe

```
C++
```

```
const TCHAR * ErrorMessage() const throw();
```

Valeur de retour

Retourne le message de chaîne pour l'enregistré `HRESULT` dans l'objet `_com_error` . Si est `HRESULT` un mappé 16 bits wCode, un message générique « `IDispatch error #<wCode>` » est retourné. Si aucun message n'est trouvé, un message générique « `Unknown error #<hresult>` » est retourné. La chaîne retournée est une chaîne Unicode ou multioctet, selon l'état de la `_UNICODE` macro.

Notes

Récupère le texte de message système approprié pour l'enregistrement `HRESULT` dans l'objet `_com_error` . Le texte du message système est obtenu en appelant la fonction Win32 FormatMessage . La chaîne retournée est allouée par l'API `FormatMessage` , et elle est libérée lorsque l'objet `_com_error` est détruit.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::GUID`

Article • 16/06/2023

Section spécifique à Microsoft

Appelle `IErrorInfo::GetGUID` la fonction.

Syntaxe

```
C++

GUID GUID() const throw();
```

Valeur de retour

Retourne le résultat de `IErrorInfo::GetGUID` pour l'objet `IErrorInfo` enregistré dans l'objet `_com_error` . Si aucun objet n'est `IErrorInfo` enregistré, il retourne `GUID_NULL`.

Notes

Tout échec lors de l'appel de la `IErrorInfo::GetGUID` méthode est ignoré.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::HelpContext`

Article • 16/06/2023

Section spécifique à Microsoft

Appelle `IErrorInfo::GetHelpContext` la fonction.

Syntaxe

```
C++

DWORD HelpContext() const throw();
```

Valeur de retour

Retourne le résultat de `IErrorInfo::GetHelpContext` pour l'objet `IErrorInfo` enregistré dans l'objet `_com_error` . Si aucun objet n'est `IErrorInfo` enregistré, il retourne un zéro.

Notes

Tout échec lors de l'appel de la `IErrorInfo::GetHelpContext` méthode est ignoré.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::HelpFile`

Article • 16/06/2023

Section spécifique à Microsoft

Appelle `IErrorInfo::GetHelpFile` la fonction.

Syntaxe

```
C++
```

```
_bstr_t HelpFile() const;
```

Valeur de retour

Retourne le résultat de `IErrorInfo::GetHelpFile` pour l'objet `IErrorInfo` enregistré dans l'objet `_com_error` . Le BSTR résultant est encapsulé dans un objet `_bstr_t`. Si aucun n'est `IErrorInfo` enregistré, il retourne un vide `_bstr_t`.

Notes

Tout échec lors de l'appel de la `IErrorInfo::GetHelpFile` méthode est ignoré.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::HRESULTToWCode`

Article • 16/06/2023

Section spécifique à Microsoft

Mappe 32 bits `HRESULT` à 16 bits `wCode`.

Syntaxe

```
C++
```

```
static WORD HRESULTToWCode(
   HRESULT hr
) throw();
```

Paramètres

`hr`
32 bits `HRESULT` à mapper à 16 bits `wCode`.

Valeur retournée

16 bits `wCode` mappé à partir du 32 bits `HRESULT`.

Notes

Pour plus d'informations, consultez _com_error::WCode.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error::WCode
_com_error::WCodeToHRESULT
_com_error, classe

`_com_error::Source`

Article • 16/06/2023

Section spécifique à Microsoft

Appelle `IErrorInfo::GetSource` la fonction .

Syntaxe

```
C++
```

```
_bstr_t Source() const;
```

Valeur de retour

Retourne le résultat de `IErrorInfo::GetSource` pour l'objet `IErrorInfo` enregistré dans l'objet `_com_error` . Le `BSTR` résultant est encapsulé dans un objet `_bstr_t`. Si aucun n'est `IErrorInfo` enregistré, il retourne un vide `_bstr_t`.

Notes

Tout échec lors de l'appel de la `IErrorInfo::GetSource` méthode est ignoré.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

`_com_error::WCode`

Article • 16/06/2023

Section spécifique à Microsoft

Récupère le code d'erreur 16 bits mappé dans le encapsulé `HRESULT`.

Syntaxe

```
C++

WORD WCode ( ) const throw();
```

Valeur de retour

Si le `HRESULT` est dans la plage 0x80040200 à 0x8004FFFF, la `WCode` méthode retourne le `HRESULT` moins 0x80040200 ; sinon, elle retourne zéro.

Notes

La `WCode` méthode est utilisée pour annuler un mappage qui se produit dans le code de support COM. Le wrapper d'une propriété ou d'une `dispinterface` méthode appelle une routine de support qui empaquette les arguments et appelle `IDispatch::Invoke`. Lors du retour, si un échec `HRESULT` de `DISP_E_EXCEPTION` est retourné, les informations d'erreur sont récupérées à partir de la `EXCEPINFO` structure passée à `IDispatch::Invoke`. Le code d'erreur peut être une valeur 16 bits stockée dans le `wCode` membre de la `EXCEPINFO` structure ou une valeur 32 bits complète dans le `scode` membre de la `EXCEPINFO` structure. Si un 16 bits `wCode` est retourné, il doit d'abord être mappé à une défaillance `HRESULT`32 bits.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error::HRESULTToWCode
_com_error::WCodeToHRESULT
_com_error, classe

`_com_error::WCodeToHRESULT`

Article • 16/06/2023

Section spécifique à Microsoft

Mappe 16 bits *wCode* à 32 bits `HRESULT`.

Syntaxe

```
C++
```

```
static HRESULT WCodeToHRESULT(
   WORD wCode
) throw();
```

Paramètres

wCode
16 bits *wCode* à mapper à 32 bits `HRESULT`.

Valeur retournée

Mappé 32 bits `HRESULT` à partir du 16 bits *wCode*.

Notes

Consultez la WCode fonction membre.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error::WCode
_com_error::HRESULTToWCode
_com_error, classe

`_com_error`, opérateurs

Article • 19/06/2023

Pour plus d'informations sur les `_com_error` opérateurs, consultez _com_error la classe .

Voir aussi

_com_error, classe

`_com_error::operator=`

Article • 16/06/2023

Section spécifique à Microsoft

Assigne un objet `_com_error` existant à un autre.

Syntaxe

```
C++
```

```
_com_error& operator=(
   const _com_error& that
) throw ();
```

Paramètres

`that`
Objet `_com_error`.

FIN de la section spécifique à Microsoft

Voir aussi

_com_error, classe

_com_ptr_t, classe

Article • 16/06/2023

Section spécifique à Microsoft

Un objet **_com_ptr_t** encapsule un pointeur d'interface COM et est appelé pointeur « intelligent ». Cette classe de modèle gère l'allocation et la désallocation des ressources via des `IUnknown` appels de fonction aux fonctions membres : `QueryInterface`, `AddRef`et `Release`.

Un pointeur intelligent est généralement référencé par la définition de typedef fournie par la macro _COM_SMARTPTR_TYPEDEF. Cette macro prend un nom d'interface et l'IID et déclare une spécialisation de **_com_ptr_t** avec le nom de l'interface plus un suffixe de `Ptr`. Par exemple :

```
C++

_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

déclare la **spécialisation** `IMyInterfacePtr` _com_ptr_t .

Un ensemble de modèles de fonction, qui ne sont pas membres de cette classe de modèle, prend en charge les comparaisons avec un pointeur intelligent à droite de l'opérateur de comparaison.

Construction

Nom	Description
_com_ptr_t	Construit un objet **_com_ptr_t** .

Opérations de bas niveau

Nom	Description
AddRef	Appelle la `AddRef` fonction membre de sur le pointeur d'interface `IUnknown` encapsulé.
Attacher	Encapsule un pointeur d'interface brut du type de ce pointeur intelligent.
CreateInstance	Crée une nouvelle instance d'un objet donné un `CLSID` ou `ProgID`.

Nom	Description
Détacher	Extrait et retourne le pointeur d'interface encapsulé.
GetActiveObject	S'attache à un instance existant d'un objet donné un `CLSID` ou `ProgID`.
GetInterfacePtr	Retourne le pointeur d'interface encapsulé.
QueryInterface	Appelle la `QueryInterface` fonction membre de sur le pointeur d'interface `IUnknown` encapsulé.
Version release	Appelle la `Release` fonction membre de sur le pointeur d'interface `IUnknown` encapsulé.

Opérateurs

Nom	Description
operator =	Affecte une nouvelle valeur à un objet **_com_ptr_t** existant.
operators ==, !=, <, >, <=, >=	Comparent l'objet pointeur intelligent avec un autre pointeur intelligent, un pointeur d'interface brut ou NULL.
Extracteurs	Récupérez le pointeur d'interface COM encapsulé.

FIN de la section spécifique à Microsoft

Spécifications

En-tête:< comip.h>

Lib : comsuppw.lib ou comsuppwd.lib (voir /Zc:wchar_t (wchar_t Is Native Type) pour plus d'informations)

Voir aussi

Classes de prise en charge COM du compilateur

_com_ptr_t, fonctions membres

Article • 16/06/2023

Pour plus d'informations sur les **fonctions membres _com_ptr_t** , consultez _com_ptr_t Classe.

Voir aussi

classe _com_ptr_t

_com_ptr_t::_com_ptr_t

Article • 16/06/2023

Section spécifique à Microsoft

Construit un objet **_com_ptr_t** .

Syntaxe

C++

```
// Default constructor.
// Constructs a NULL smart pointer.
_com_ptr_t() throw();

// Constructs a NULL smart pointer. The NULL argument must be zero.
_com_ptr_t(
   int null
);

// Constructs a smart pointer as a copy of another instance of the
// same smart pointer. AddRef is called to increment the reference
// count for the encapsulated interface pointer.
_com_ptr_t(
   const _com_ptr_t& cp
) throw();

// Move constructor (Visual Studio 2015 Update 3 and later)
_com_ptr_t(_com_ptr_t&& cp) throw();

// Constructs a smart pointer from a raw interface pointer of this
// smart pointer's type. If fAddRef is true, AddRef is called
// to increment the reference count for the encapsulated
// interface pointer. If fAddRef is false, this constructor
// takes ownership of the raw interface pointer without calling AddRef.
_com_ptr_t(
   Interface* pInterface,
   bool fAddRef
) throw();

// Construct pointer for a _variant_t object.
// Constructs a smart pointer from a _variant_t object. The
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or
// it can be converted into one of these two types. If QueryInterface
// fails with an E_NOINTERFACE error, a NULL smart pointer is
// constructed.
_com_ptr_t(
   const _variant_t& varSrc
);
```

```
// Constructs a smart pointer given the CLSID of a coclass. This
// function calls CoCreateInstance, by the member function
//  CreateInstance, to create a new COM object and then queries for
// this smart pointer's interface type. If QueryInterface fails with
// an E_NOINTERFACE error, a NULL smart pointer is constructed.
explicit _com_ptr_t(
   const CLSID& clsid,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Calls CoCreateClass with provided CLSID retrieved from string.
explicit _com_ptr_t(
   LPCWSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
   LPCSTR str,
   IUnknown* pOuter = NULL,
   DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
   Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
   LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
```

```
   const _com_ptr_t<_OtherIID>& p
);

// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
   _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
   _com_ptr_t* p
);
```

Paramètres

pInterface
Pointeur d'interface brut.

fAddRef
Si `true`, `AddRef` est appelé pour incrémenter le nombre de références du pointeur d'interface encapsulé.

cp
Objet **_com_ptr_t** .

P
Pointeur d'interface brut, son type étant différent du type de pointeur intelligent de cet objet **_com_ptr_t** .

varSrc
Objet `_variant_t`.

Clsid
d'une `CLSID` coclasse.

dwClsContext
Contexte d'exécution du code exécutable.

lpcStr
Chaîne multioctet qui contient un `CLSID` (commençant par « { ») ou un `ProgID`.

pOuter
Inconnue externe pour l'agrégation.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_com_ptr_t::AddRef

Article • 16/06/2023

Section spécifique à Microsoft

Appelle la `AddRef` fonction membre de sur le pointeur d'interface `IUnknown` encapsulé.

Syntaxe

```
C++
```

```
void AddRef( );
```

Notes

Appelle `IUnknown::AddRef` sur le pointeur d'interface encapsulé, ce qui déclenche une `E_POINTER` erreur si le pointeur a la valeur NULL.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

_com_ptr_t::Attach

Article • 16/06/2023

Section spécifique à Microsoft

Encapsule un pointeur d'interface brut du type de ce pointeur intelligent.

Syntaxe

```
C++
```

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

Paramètres

pInterface
Pointeur d'interface brut.

fAddRef
S'il s'agit `true` de , est `AddRef` appelé. S'il s'agit `false` de , l'objet `_com_ptr_t` prend possession du pointeur d'interface brut sans appeler `AddRef`.

Notes

- **Attach**(*pInterface*) `AddRef` n'est pas appelé. La propriété de l'interface est passée à cet objet `_com_ptr_t`. `Release` est appelé pour décrémenter le nombre de références pour le pointeur précédemment encapsulé.
- **Attach**(*pInterface,fAddRef*) Si *fAddRef* est `true`, `AddRef` est appelé pour incrémenter le nombre de références pour le pointeur d'interface encapsulé. Si *fAddRef* est `false`, cet `_com_ptr_t` objet prend possession du pointeur d'interface brut sans appeler `AddRef`. `Release` est appelé pour décrémenter le nombre de références pour le pointeur précédemment encapsulé.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_com_ptr_t::CreateInstance

Article • 16/06/2023

Section spécifique à Microsoft

Crée une nouvelle instance d'un objet donné un `CLSID` ou `ProgID`.

Syntaxe

```
HRESULT CreateInstance(
   const CLSID& rclsid,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCWSTR clsidString,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
   LPCSTR clsidStringA,
   IUnknown* pOuter=NULL,
   DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

Paramètres

rclsid
`CLSID` d'un objet .

clsidString
Chaîne Unicode qui contient un `CLSID` (commençant par « { ») ou un `ProgID`.

clsidStringA
Chaîne multioctet, utilisant la page de codes ANSI, qui contient un `CLSID` (commençant par « { ») ou un `ProgID`.

dwClsContext
Contexte d'exécution du code exécutable.

pOuter
Externe inconnu pour l'agrégation.

Notes

Ces fonctions membres appellent `CoCreateInstance` pour créer un objet COM puis des requêtes pour le type d'interface de ce pointeur intelligent. Le pointeur résultant est alors encapsulé dans cet objet `_com_ptr_t`. `Release` est appelé pour décrémenter le nombre de références pour le pointeur précédemment encapsulé. Cette routine retourne le HRESULT pour indiquer la réussite ou l'échec.

- **CreateInstance(***rclsid,dwClsContext***)** Crée une instance en cours d'exécution d'un objet en fonction d'un `CLSID`.
- **CreateInstance(***clsidString,dwClsContext***)** Crée une instance en cours d'exécution d'un objet en fonction d'une chaîne Unicode contenant un `CLSID` (commençant par « { ») ou un `ProgID`.
- **CreateInstance(***clsidStringA,dwClsContext***)** Crée une nouvelle instance d'exécution d'un objet en fonction d'une chaîne de caractères multioctets qui contient un `CLSID` (commençant par « { ») ou un `ProgID`. Appelle MultiByteToWideChar, ce qui suppose que la chaîne se trouve dans la page de codes ANSI plutôt que dans une page de codes OEM.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

_com_ptr_t::Detach

Article • 16/06/2023

Section spécifique à Microsoft

Extrait et retourne le pointeur d'interface encapsulé.

Syntaxe

```
Interface* Detach( ) throw( );
```

Notes

Extrait et retourne le pointeur d'interface encapsulé, puis efface le stockage du pointeur encapsulé en lui affectant la valeur NULL. Cela supprime le pointeur d'interface de l'encapsulation. Il vous appartient d'appeler `Release` sur le pointeur d'interface retourné.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

_com_ptr_t::GetActiveObject

Article • 16/06/2023

Section spécifique à Microsoft

S'attache à un instance existant d'un objet donné un `CLSID` ou `ProgID`.

Syntaxe

```
HRESULT GetActiveObject(
   const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
   LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
   LPCSTR clsidStringA
) throw( );
```

Paramètres

rclsid
`CLSID` d'un objet .

clsidString
Chaîne Unicode qui contient un `CLSID` (commençant par « **{** ») ou un `ProgID`.

clsidStringA
Chaîne multioctet, utilisant la page de codes ANSI, qui contient un `CLSID` (commençant par « **{** ») ou un `ProgID`.

Notes

Ces fonctions membres appellent **GetActiveObject** pour récupérer un pointeur vers un objet en cours d'exécution qui a été inscrit auprès d'OLE, puis interrogent le type d'interface de ce pointeur intelligent. Le pointeur résultant est alors encapsulé dans cet objet `_com_ptr_t`. `Release` est appelé pour décrémenter le nombre de références pour le pointeur précédemment encapsulé. Cette routine retourne le HRESULT pour indiquer la réussite ou l'échec.

- **GetActiveObject(`rclsid`)** S'attache à une instance existante d'un objet donné à `CLSID`.
- **GetActiveObject(`clsidString`)** S'attache à un instance existant d'un objet donné à une chaîne Unicode qui contient un `CLSID` (commençant par « **{** ») ou un `ProgID`.
- **GetActiveObject(`clsidStringA`)** S'attache à une instance existante d'un objet en fonction d'une chaîne de caractères multioctets qui contient un `CLSID` (commençant par « **{** ») ou un `ProgID`. Appelle MultiByteToWideChar, ce qui suppose que la chaîne se trouve dans la page de codes ANSI plutôt que dans une page de codes OEM.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

_com_ptr_t::GetInterfacePtr

Article • 16/06/2023

Section spécifique à Microsoft

Retourne le pointeur d'interface encapsulé.

Syntaxe

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Notes

Retourne le pointeur d'interface encapsulé, qui peut être NULL.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_com_ptr_t::QueryInterface

Article • 16/06/2023

Section spécifique à Microsoft

Appelle la fonction membre **QueryInterface** de sur le pointeur d'interface `IUnknown` encapsulé.

Syntaxe

```
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType*& p
) throw ( );
template<typename _InterfaceType> HRESULT QueryInterface (
   const IID& iid,
   _InterfaceType** p
) throw( );
```

Paramètres

Iid
`IID` d'un pointeur d'interface.

P
Pointeur d'interface brut.

Notes

Appelle `IUnknown::QueryInterface` le pointeur d'interface encapsulé avec le spécifié `IID` et retourne le pointeur d'interface brut résultant en *p*. Cette routine retourne le HRESULT pour indiquer la réussite ou l'échec.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_com_ptr_t::Release

Article • 16/06/2023

Section spécifique à Microsoft

Appelle la fonction **membre Release** de sur le pointeur d'interface `IUnknown` encapsulé.

Syntaxe

```
C++
```

```
void Release( );
```

Notes

Appelle `IUnknown::Release` sur le pointeur d'interface encapsulé, ce qui déclenche une `E_POINTER` erreur si ce pointeur d'interface a la valeur NULL.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

Opérateurs _com_ptr_t

Article • 16/06/2023

Pour plus d'informations sur les `_com_ptr_t` opérateurs, consultez _com_ptr_t Classe.

Voir aussi

classe _com_ptr_t

_com_ptr_t::operator =

Article • 16/06/2023

Section spécifique à Microsoft

Assigne une nouvelle valeur à un objet `_com_ptr_t` existant.

Syntaxe

```
template<typename _OtherIID>
_com_ptr_t& operator=( const _com_ptr_t<_OtherIID>& p );

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=(_InterfaceType* p );

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=( Interface* pInterface ) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=( const _com_ptr_t& cp ) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=( int null );

// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=( const _variant_t& varSrc );
```

Notes

Assigne un pointeur d'interface vers cet objet `_com_ptr_t`.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_com_ptr_t, opérateurs relationnels

Article • 16/06/2023

Section spécifique à Microsoft

Comparent l'objet pointeur intelligent avec un autre pointeur intelligent, un pointeur d'interface brut ou NULL.

Syntaxe

```
template<typename _OtherIID>
bool operator==( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator==( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator==( _InterfaceType* p );

template<>
bool operator==( Interface* p );

template<>
bool operator==( const _com_ptr_t& p ) throw();

template<>
bool operator==( _com_ptr_t& p ) throw();

bool operator==( Int null );

template<typename _OtherIID>
bool operator!=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator!=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator!=( _InterfaceType* p );

bool operator!=( Int null );

template<typename _OtherIID>
bool operator<( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<( _com_ptr_t<_OtherIID>& p );
```

```
template<typename _InterfaceType>
bool operator<( _InterfaceType* p );

template<typename _OtherIID>
bool operator>( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>(_com_ptr_t< _OtherIID>& p );

template<typename _InterfaceType>
bool operator>( _InterfaceType* p );

template<typename _OtherIID>
bool operator<=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<=( _InterfaceType* p );

template<typename _OtherIID>
bool operator>=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>=( _InterfaceType* p );
```

Notes

Compare un objet pointeur intelligent avec un autre pointeur intelligent, un pointeur d'interface brut ou NULL. À l'exception des tests de pointeur NULL, ces opérateurs interrogent d'abord les deux pointeurs pour `IUnknown`, puis comparent les résultats.

FIN de la section spécifique à Microsoft

Voir aussi

_com_ptr_t, classe

`_com_ptr_t` Extracteurs

Article • 16/06/2023

Spécifique à Microsoft

Récupérez le pointeur d'interface COM encapsulé.

Syntaxe

```
c++
```

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Notes

- **`operator Interface*`** Retourne le pointeur d'interface encapsulé, qui peut être NULL.
- **`operator Interface&`** Retourne une référence au pointeur d'interface encapsulé et génère une erreur si le pointeur a la valeur NULL.
- **`operator*`** Permet à un objet pointeur intelligent d'agir comme s'il s'agissait de l'interface encapsulée réelle lorsqu'il est déréférencé.
- **`operator->`** Permet à un objet pointeur intelligent d'agir comme s'il s'agissait de l'interface encapsulée réelle lorsqu'il est déréférencé.
- **`operator&`** Libère tout pointeur d'interface encapsulé, en le remplaçant par NULL, et retourne l'adresse du pointeur encapsulé. Cet opérateur vous permet de passer le pointeur intelligent par adresse à une fonction qui a un paramètre *out* via lequel il retourne un pointeur d'interface.
- **`operator bool`** Autorise l'utilisation d'un objet pointeur intelligent dans une expression conditionnelle. Cet opérateur retourne **`true`** si le pointeur n'est pas NULL.

> **Notes**
>
> Étant donné que `operator bool` n'est pas déclaré comme `explicit`, `_com_ptr_t` est implicitement convertible en `bool`, qui est convertible en tout type scalaire. Cela peut avoir des conséquences inattendues dans votre code. Activez **l'avertissement du compilateur (niveau 4) C4800** pour éviter toute utilisation involontaire de cette conversion.

Voir aussi

_com_ptr_t (classe)

Modèles de fonction relationnelle

Article • 16/06/2023

Section spécifique à Microsoft

Syntaxe

```
template<typename _InterfaceType> bool operator==(
   int NULL,
   _com_ptr_t<_InterfaceType>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator==(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator!=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator!=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator<(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator<=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
```

```
   typename _InterfacePtr> bool operator<=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
template<typename _Interface> bool operator>=(
   int NULL,
   _com_ptr_t<_Interface>& p
);
template<typename _Interface,
   typename _InterfacePtr> bool operator>=(
   _Interface* i,
   _com_ptr_t<_InterfacePtr>& p
);
```

Paramètres

i

Pointeur d'interface brut.

P

Un pointeur intelligent.

Notes

Ces modèles de fonction permettent la comparaison avec un pointeur intelligent à droite de l'opérateur de comparaison. Ce ne sont pas des fonctions membres de `_com_ptr_t`.

FIN de la section spécifique à Microsoft

Voir aussi

classe _com_ptr_t

_variant_t (classe)

Article • 16/06/2023

Section spécifique à Microsoft

Un objet **_variant_t** encapsule le type de `VARIANT` données. La classe gère l'allocation et la désallocation des ressources et effectue des appels de fonction à `VariantInit` et `VariantClear` le cas échéant.

Construction

Nom	Description
_variant_t	Construit un objet **_variant_t** .

Opérations

Nom	Description
Attacher	Attache un `VARIANT` objet à **l'objet _variant_t** .
Clear	Efface l'objet encapsulé `VARIANT` .
ChangeType	Remplace le type de l'objet **_variant_t** par le indiqué `VARTYPE`.
Détacher	Détache l'objet encapsulé `VARIANT` de cet objet **_variant_t** .
SetString	Affecte une chaîne à cet objet **_variant_t** .

Opérateurs

Nom	Description
Opérateur =	Affecte une nouvelle valeur à un objet **_variant_t** existant.
operator ==, !=	Comparez deux **objets _variant_t** pour l'égalité ou l'inégalité.
Extracteurs	Extrayez des données de l'objet encapsulé `VARIANT` .

FIN de la section spécifique à Microsoft

Configuration requise

En-tête:< comutil.h>

Lib : comsuppw.lib ou comsuppwd.lib (consultez /Zc:wchar_t (wchar_t type natif) pour plus d'informations)

Voir aussi

Classes de prise en charge COM du compilateur

_variant_t, fonctions membres

Article • 03/04/2023

Pour plus d'informations sur les **fonctions membres _variant_t** , consultez _variant_t Classe.

Voir aussi

classe _variant_t

`_variant_t::_variant_t`

Article • 03/04/2023

Section spécifique à Microsoft

Construit un objet `_variant_t`.

Syntaxe

C++

```
_variant_t( ) throw( );

_variant_t(
   const VARIANT& varSrc
);

_variant_t(
   const VARIANT* pVarSrc
);

_variant_t(
   const _variant_t& var_t_Src
);

_variant_t(
   VARIANT& varSrc,
   bool fCopy
);

_variant_t(
   short sSrc,
   VARTYPE vtSrc = VT_I2
);

_variant_t(
   long lSrc,
   VARTYPE vtSrc = VT_I4
);

_variant_t(
   float fltSrc
) throw( );

_variant_t(
   double dblSrc,
   VARTYPE vtSrc = VT_R8
);

_variant_t(
```

```
   const CY& cySrc
) throw( );

_variant_t(
   const _bstr_t& bstrSrc
);

_variant_t(
   const wchar_t *wstrSrc
);

_variant_t(
   const char* strSrc
);

_variant_t(
   IDispatch* pDispSrc,
   bool fAddRef = true
) throw( );

_variant_t(
   bool bSrc
) throw( );

_variant_t(
   IUnknown* pIUknownSrc,
   bool fAddRef = true
) throw( );

_variant_t(
   const DECIMAL& decSrc
) throw( );

_variant_t(
   BYTE bSrc
) throw( );

variant_t(
   char cSrc
) throw();

_variant_t(
   unsigned short usSrc
) throw();

_variant_t(
   unsigned long ulSrc
) throw();

_variant_t(
   int iSrc
) throw();

_variant_t(
   unsigned int uiSrc
```

```
) throw();

_variant_t(
   __int64 i8Src
) throw();

_variant_t(
   unsigned __int64 ui8Src
) throw();
```

Paramètres

varSrc

Objet `VARIANT` à copier dans le nouvel objet `_variant_t`.

pVarSrc

Pointeur vers un `VARIANT` objet à copier dans le nouvel `_variant_t` objet.

var_t_Src

Objet `_variant_t` à copier dans le nouvel objet `_variant_t`.

fCopy

Si **`false`** la valeur est , l'objet fourni `VARIANT` est attaché au nouvel `_variant_t` objet sans effectuer de nouvelle copie par `VariantCopy`.

lSrc, *sSrc*

Valeur entière à copier dans le nouvel objet `_variant_t`.

vtSrc

`VARTYPE` pour le nouvel `_variant_t` objet .

fltSrc, *dblSrc*

Valeur numérique à copier dans le nouvel objet `_variant_t`.

cySrc

Objet `CY` à copier dans le nouvel objet `_variant_t`.

bstrSrc

Objet `_bstr_t` à copier dans le nouvel objet `_variant_t`.

strSrc, *wstrSrc*

Chaîne à copier dans le nouvel objet `_variant_t`.

bSrc

Valeur **bool** à copier dans le nouvel objet _variant_t.

pIUknownSrc

Pointeur d'interface COM vers un objet VT_UNKNOWN à encapsuler dans le nouvel _variant_t objet.

pDispSrc

Pointeur d'interface COM vers un objet VT_DISPATCH à encapsuler dans le nouvel _variant_t objet.

decSrc

Valeur DECIMAL à copier dans le nouvel objet _variant_t.

bSrc

Valeur BYTE à copier dans le nouvel objet _variant_t.

cSrc

Valeur **char** à copier dans le nouvel objet _variant_t.

usSrc

Valeur **unsigned short** à copier dans le nouvel objet _variant_t.

ulSrc

Valeur **unsigned long** à copier dans le nouvel objet _variant_t.

iSrc

Valeur **int** à copier dans le nouvel _variant_t objet.

uiSrc

Valeur **unsigned int** à copier dans le nouvel _variant_t objet.

i8Src

Valeur **__int64** à copier dans le nouvel _variant_t objet.

ui8Src

Valeur **unsigned __int64** à copier dans le nouvel _variant_t objet.

Notes

- **_variant_t()** Construit un objet vide _variant_t , VT_EMPTY.

- **`_variant_t( VARIANT& varSrc )`** Construit un `_variant_t` objet à partir d'une copie de l'objet `VARIANT` . Le type variant est conservé.
- **`_variant_t( VARIANT* pVarSrc )`** Construit un `_variant_t` objet à partir d'une copie de l'objet `VARIANT` . Le type variant est conservé.
- **`_variant_t( _variant_t& var_t_Src )`** Construit un `_variant_t` objet à partir d'un autre `_variant_t` objet. Le type variant est conservé.
- **`_variant_t( VARIANT& varSrc, bool fCopy )`** Construit un `_variant_t` objet à partir d'un objet existant `VARIANT` . Si *`fCopy`* a la valeur **`false`**, l'objet **`VARIANT`** est attaché au nouvel objet sans effectuer de copie.
- **`_variant_t( short sSrc, VARTYPE vtSrc = VT_I2 )`** Construit un `_variant_t` objet de type `VT_I2` ou `VT_BOOL` à partir d'une **`short`** valeur entière. Toute autre `VARTYPE` erreur se `E_INVALIDARG` produit.
- **`_variant_t( long lSrc, VARTYPE vtSrc = VT_I4 )`** Construit un `_variant_t` objet de type `VT_I4`, `VT_BOOL`ou `VT_ERROR` à partir d'une **`long`** valeur entière. Toute autre `VARTYPE` erreur se `E_INVALIDARG` produit.
- **`_variant_t( float fltSrc )`** Construit un `_variant_t` objet de type `VT_R4` à partir d'une **`float`** valeur numérique.
- **`_variant_t( double dblSrc, VARTYPE vtSrc = VT_R8 )`** Construit un `_variant_t` objet de type `VT_R8` ou `VT_DATE` à partir d'une **`double`** valeur numérique. Toute autre `VARTYPE` erreur se `E_INVALIDARG` produit.
- **`_variant_t( CY& cySrc )`** Construit un `_variant_t` objet de type `VT_CY` à partir d'un `CY` objet .
- *`*_variant_t( _bstr_t& bstrSrc )`** Construit un `_variant_t` objet de type `VT_BSTR` à partir d'un `_bstr_t` objet . Un nouvel objet `BSTR` est alloué.
- **`_variant_t( wchar_t* wstrSrc )`** Construit un `_variant_t` objet de type `VT_BSTR` à partir d'une chaîne Unicode. Un nouvel objet `BSTR` est alloué.
- **`_variant_t( char* strSrc )`** Construit un `_variant_t` objet de type `VT_BSTR` à partir d'une chaîne. Un nouvel objet `BSTR` est alloué.
- **`_variant_t( bool bSrc )`** Construit un `_variant_t` objet de type `VT_BOOL` à partir d'une **`bool`** valeur.

- **`_variant_t( IUnknown* pIUknownSrc, bool fAddRef = true )`** Construit un `_variant_t` objet de type `VT_UNKNOWN` à partir d'un pointeur d'interface COM. Si `fAddRef` a la valeur **`true`**, `AddRef` est appelé sur le pointeur d'interface fourni pour correspondre à l'appel à `Release` qui se produit lorsque l'objet `_variant_t` est détruit. Il vous appartient d'appeler `Release` sur le pointeur d'interface fourni. Si `fAddRef` a la valeur **`false`**, ce constructeur prend possession du pointeur d'interface fourni ; n'appelez `Release` pas sur le pointeur d'interface fourni.

- **`_variant_t( IDispatch* pDispSrc, bool fAddRef = true )`** Construit un `_variant_t` objet de type `VT_DISPATCH` à partir d'un pointeur d'interface COM. Si `fAddRef` a la valeur **`true`**, `AddRef` est appelé sur le pointeur d'interface fourni pour correspondre à l'appel à `Release` qui se produit lorsque l'objet `_variant_t` est détruit. C'est à vous d'appeler `Release` sur le pointeur d'interface fourni. Si `fAddRef` a la valeur **`false`**, ce constructeur prend possession du pointeur d'interface fourni ; n'appelez `Release` pas sur le pointeur d'interface fourni.

- **`_variant_t( DECIMAL& decSrc )`** Construit un `_variant_t` objet de type `VT_DECIMAL` à partir d'une `DECIMAL` valeur.

- **`_variant_t( BYTE bSrc )`** Construit un `_variant_t` objet de type `VT_UI1` à partir d'une `BYTE` valeur.

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t Classe

_variant_t::Attach

Article • 16/06/2023

Section spécifique à Microsoft

Attache un `VARIANT` objet à **l'objet _variant_t** .

Syntaxe

```
C++

void Attach(VARIANT& varSrc);
```

Paramètres

varSrc
Objet `VARIANT` à attacher à cet **objet _variant_t** .

Notes

Prend possession du encapsulant `VARIANT` . Cette fonction membre libère tout encapsulé `VARIANT` existant, puis copie le fourni `VARIANT` et définit son `VARTYPE` sur VT_EMPTY pour s'assurer que ses ressources ne peuvent être libérées que par le destructeur **_variant_t** .

FIN de la section spécifique à Microsoft

Voir aussi

classe _variant_t

_variant_t::Clear

Article • 16/06/2023

Section spécifique à Microsoft

Efface l'objet encapsulé `VARIANT` .

Syntaxe

```
C++
```

```
void Clear( );
```

Notes

Appelle `VariantClear` l'objet encapsulé `VARIANT` .

FIN de la section spécifique à Microsoft

Voir aussi

classe _variant_t

_variant_t::ChangeType

Article • 16/06/2023

Section spécifique à Microsoft

Remplace le type de l'objet `_variant_t` par le indiqué `VARTYPE`.

Syntaxe

```
C++
```

```
void ChangeType(
   VARTYPE vartype,
   const _variant_t* pSrc = NULL
);
```

Paramètres

vartype
`VARTYPE` pour cet `_variant_t` objet .

pSrc
Pointeur vers l'objet `_variant_t` à convertir. Si cette valeur est NULL, la conversion s'effectue sur place.

Notes

Cette fonction membre convertit un `_variant_t` objet en .`VARTYPE` Si *pSrc* a la valeur NULL, la conversion est effectuée sur place. Sinon, cet `_variant_t` objet est copié à partir de *pSrc* , puis converti.

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t, classe

_variant_t::Detach

Article • 16/06/2023

Section spécifique à Microsoft

Détache l'objet encapsulé `VARIANT` de cet `_variant_t` objet.

Syntaxe

```
VARIANT Detach( );
```

Valeur de retour

encapsulé `VARIANT`.

Notes

Extrait et retourne le encapsulé `VARIANT`, puis efface cet `_variant_t` objet sans le détruire. Cette fonction membre supprime le de l'encapsulation `VARIANT` et définit le `VARTYPE` de cet `_variant_t` objet sur VT_EMPTY. Il vous appartient de libérer le retourné `VARIANT` en appelant la fonction VariantClear .

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t, classe

_variant_t::SetString

Article • 16/06/2023

Section spécifique à Microsoft

Assigne une chaîne à cet objet `_variant_t`.

Syntaxe

```
C++
```

```
void SetString(const char* pSrc);
```

Paramètres

pSrc
Pointeur vers la chaîne de caractères.

Notes

Convertit une chaîne de caractères ANSI en chaîne Unicode `BSTR` et l'assigne à cet objet `_variant_t`.

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t, classe

_variant_t, opérateurs

Article • 03/04/2023

Pour plus d'informations sur les opérateurs **de _variant_t** , consultez classe _variant_t.

Voir aussi

classe _variant_t

`_variant_t::operator=`

Article • 16/06/2023

Affecte une nouvelle valeur à un `_variant_t` instance.

La `_variant_t` classe et son `operator=` membre sont **spécifiques à Microsoft**.

Syntaxe

C++

```
_variant_t& operator=( const VARIANT& varSrc );
_variant_t& operator=( const VARIANT* pVarSrc );
_variant_t& operator=( const _variant_t& var_t_Src );
_variant_t& operator=( short sSrc );
_variant_t& operator=( long lSrc );
_variant_t& operator=( float fltSrc );
_variant_t& operator=( double dblSrc );
_variant_t& operator=( const CY& cySrc );
_variant_t& operator=( const _bstr_t& bstrSrc );
_variant_t& operator=( const wchar_t* wstrSrc );
_variant_t& operator=( const char* strSrc );
_variant_t& operator=( IDispatch* pDispSrc );
_variant_t& operator=( bool bSrc );
_variant_t& operator=( IUnknown* pSrc );
_variant_t& operator=( const DECIMAL& decSrc );
_variant_t& operator=( BYTE byteSrc );
_variant_t& operator=( char cSrc );
_variant_t& operator=( unsigned short usSrc );
_variant_t& operator=( unsigned long ulSrc );
_variant_t& operator=( int iSrc );
_variant_t& operator=( unsigned int uiSrc );
_variant_t& operator=( __int64 i8Src );
_variant_t& operator=( unsigned __int64 ui8Src );
```

Paramètres

`varSrc`

Référence à un `VARIANT` à partir duquel copier le contenu et `VT_*` le type.

`pVarSrc`

Pointeur vers un `VARIANT` à partir duquel copier le contenu et `VT_*` le type.

`var_t_Src`

Référence à un `_variant_t` à partir duquel copier le contenu et `VT_*` le type.

sSrc

Valeur **short** entière à copier. VT_BOOL Type donné si *this est de type VT_BOOL. Sinon, elle est de type VT_I2.

lSrc

Valeur **long** entière à copier. VT_BOOL Type donné si *this est de type VT_BOOL. VT_ERROR Type donné si *this est de type VT_ERROR. Dans le cas contraire, donné le type VT_I4.

fltSrc

Valeur **float** numérique à copier. VT_R4 Type donné .

dblSrc

Valeur **double** numérique à copier. VT_DATE Type donné si this est de type VT_DATE. Dans le cas contraire, donné le type VT_R8.

cySrc

Objet CY à copier. VT_CY Type donné .

bstrSrc

Objet BSTR à copier. VT_BSTR Type donné .

wstrSrc

Chaîne Unicode à copier, stockée en tant que BSTR type VT_BSTR et donné .

strSrc

Chaîne multioctet à copier, stockée en tant que BSTR type VT_BSTR et donné .

pDispSrc

Pointeur IDispatch pour copier avec un appel à AddRef. VT_DISPATCH Type donné .

bSrc

Valeur **bool** à copier. VT_BOOL Type donné .

pSrc

Pointeur IUnknown pour copier avec un appel à AddRef. VT_UNKNOWN Type donné .

decSrc

Objet DECIMAL à copier. VT_DECIMAL Type donné .

byteSrc

Valeur BYTE à copier. VT_UI1 Type donné .

cSrc

Valeur **char** à copier. VT_I1Type donné .

usSrc

Valeur **unsigned short** à copier. VT_UI2Type donné .

ulSrc

Valeur **unsigned long** à copier. VT_UI4Type donné .

iSrc

Valeur **int** à copier. VT_INTType donné .

uiSrc

Valeur **unsigned int** à copier. VT_UINTType donné .

i8Src

Valeur **__int64** ou **long long** à copier. VT_I8Type donné .

ui8Src

Valeur **unsigned __int64** ou **unsigned long long** à copier. VT_UI8Type donné .

Notes

L'opérateur operator= d'assignation efface toute valeur existante, ce qui supprime les types d'objets, ou appelle Release des IDispatch* types et IUnknown* . Ensuite, il copie une nouvelle valeur dans l'objet _variant_t . Il modifie le _variant_t type pour qu'il corresponde à la valeur affectée, sauf comme indiqué pour **short**les arguments , **long**et **double** . Les types valeur sont copiés directement. Un VARIANT pointeur ou _variant_t un argument de référence copie le contenu et le type de l'objet affecté. D'autres arguments de type pointeur ou référence créent une copie de l'objet affecté. L'opérateur d'assignation appelle AddRef des IDispatch* arguments et IUnknown* .

operator=_com_raise_error appelle si une erreur se produit.

operator= retourne une référence à l'objet mis à jour _variant_t .

Voir aussi

_variant_t, classe

_variant_t, opérateurs relationnels

Article • 16/06/2023

Section spécifique à Microsoft

Comparez deux objets `_variant_t` pour déterminer leur égalité ou inégalité.

Syntaxe

```
bool operator==(
   const VARIANT& varSrc) const;
bool operator==(
   const VARIANT* pSrc) const;
bool operator!=(
   const VARIANT& varSrc) const;
bool operator!=(
   const VARIANT* pSrc) const;
```

Paramètres

varSrc
`VARIANT` à comparer à l'objet `_variant_t` .

pSrc
Pointeur vers le `VARIANT` à comparer à l'objet `_variant_t` .

Valeur renvoyée

Retourne **`true`** si la comparaison tient, **`false`** si ce n'est pas le cas.

Notes

Compare un `_variant_t` objet avec un `VARIANT`, testant l'égalité ou l'inégalité.

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t, classe

Extracteurs _variant_t

Article • 16/06/2023

Section spécifique à Microsoft

Extrayez les données de l'objet encapsulé `VARIANT` .

Syntaxe

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

Notes

Extrait les données brutes d'un encapsulé `VARIANT`. Si le `VARIANT` n'est pas déjà le type approprié, `VariantChangeType` est utilisé pour tenter une conversion et une erreur est générée en cas d'échec :

- **operator short( )** Extrait une `short` valeur entière.
- **operator long( )** Extrait une `long` valeur entière.
- **operator float( )** Extrait une `float` valeur numérique.
- **operator double( )** Extrait une `double` valeur entière.

- **operator CY( )** Extrait un `CY` objet .
- **operator bool( )** Extrait une `bool` valeur.
- **operator DECIMAL( )** Extrait une `DECIMAL` valeur.
- **operator BYTE( )** Extrait une `BYTE` valeur.
- **operator _bstr_t( )** Extrait une chaîne, qui est encapsulée dans un `_bstr_t` objet .
- **operator IDispatch*( )** Extrait un pointeur de dispinterface d'un encapsulé `VARIANT`. `AddRef` est appelé sur le pointeur résultant, il vous appartient donc d'appeler `Release` pour le libérer.
- **operator IUnknown*( )** Extrait un pointeur d'interface COM à partir d'un encapsulé `VARIANT`. `AddRef` est appelé sur le pointeur résultant, il vous appartient donc d'appeler `Release` pour le libérer.

FIN de la section spécifique à Microsoft

Voir aussi

_variant_t, classe

Extensions Microsoft

Article • 16/06/2023

asm-statement:

__asm *assembly-instruction* **;** opt

__asm { *assembly-instruction-list* **} ;** opt

assembly-instruction-list:

assembly-instruction **;** opt

assembly-instruction **;** *assembly-instruction-list* **;** opt

ms-modifier-list:

ms-modifier *ms-modifier-list* opt

ms-modifier:

__cdecl

__fastcall

__stdcall

__syscall (réservé aux implémentations ultérieures)

__oldcall (réservé aux implémentations ultérieures)

__unaligned (réservé aux implémentations ultérieures)

based-modifier

based-modifier:

__based (*based-type* **)**

based-type:

name

Comportement non standard

Article • 03/04/2023

Les sections suivantes répertorient certains des endroits où l'implémentation Microsoft de C++ n'est pas conforme à la norme C++. Les numéros de section ci-dessous correspondent aux numéros de section de la norme C++11 (ISO/IEC 14882:2011(E)).

La liste des limites du compilateur qui diffèrent de celles définies dans la norme C++ est indiquée dans Limites du compilateur.

Types de retour covariant

Les classes de base virtuelle ne sont pas prises en charge en tant que types de retour covariants lorsque la fonction virtuelle a un nombre d'arguments variable. Cela n'est pas conforme à la section 10.3, paragraphe 7 de la spécification ISO C++ 11. L'exemple suivant ne se compile pas ; il génère l'erreur du compilateur C2688 :

C++

```
// CovariantReturn.cpp
class A
{
   virtual A* f(int c, ...);   // remove ...
};

class B : virtual A
{
   B* f(int c, ...);   // C2688 remove ...
};
```

Liaison de noms non dépendants dans des modèles

Le compilateur Microsoft C++ ne prend actuellement pas en charge la liaison de noms non indépendants lors de l'analyse initiale d'un modèle. Cela n'est pas conforme à la section 14.6.3 de la spécification ISO C++ 11. Cela peut provoquer la déclaration de surcharges après le modèle à afficher (mais avant que le modèle soit instancié).

C++

```
#include <iostream>
using namespace std;
```

```
namespace N {
   void f(int) { cout << "f(int)" << endl;}
}

template <class T> void g(T) {
    N::f('a');   // calls f(char), should call f(int)
}

namespace N {
    void f(char) { cout << "f(char)" << endl;}
}

int main() {
    g('c');
}
// Output: f(char)
```

Spécificateurs d'exceptions de fonctions

Les spécificateurs d'exceptions de fonction autres que `throw()` sont analysés, mais pas utilisés. Cela n'est pas conforme à la section 15.4 de la spécification ISO C++ 11. Par exemple :

C++

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

Pour plus d'informations sur les spécifications d'exception, consultez Spécifications des exceptions.

char_traits::eof()

La norme C++ indique que char_traits::eof ne doit pas correspondre à une valeur valide `char_type` . Le compilateur Microsoft C++ applique cette contrainte pour le type **`char`**, mais pas pour le type **`wchar_t`**. Cela n'est pas conforme à l'exigence du tableau 62 de la section 12.1.1 de la spécification ISO C++ 11. L'exemple ci-dessous illustre ce comportement.

C++

```
#include <iostream>

int main()
```

```
{
    using namespace std;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

Emplacement de stockage des objets

La norme C++ (section 1.8, paragraphe 6) exige que les objets C++ complets aient des emplacements de stockage uniques. Toutefois, avec Microsoft C++, il existe des cas où des types sans membres de données partagent un emplacement de stockage avec d'autres types pendant la durée de vie de l'objet.

Limites du compilateur

Article • 08/06/2023

La norme C++ recommande certaines limites pour diverses constructions de langage. Voici une liste de cas dans lesquels le compilateur Microsoft C++ n'implémente pas les limites recommandées. Le premier nombre est la limite établie dans la norme ISO C++ 11 (INCITS/ISO/IEC 14882-2011[2012], Annexe B) et le deuxième nombre est la limite implémentée par le compilateur Microsoft C++ :

- Imbrication des niveaux d'instructions composées, de structures de contrôle d'itération et de structures de contrôle de sélection - Norme C++ : 256, compilateur Microsoft C++ : dépend de la combinaison d'instructions imbriquées, mais généralement comprise entre 100 et 110.

- Paramètres dans une définition de macro : C++ standard : 256, compilateur Microsoft C++ utilisant `/Zc:preprocessor-`:127 ou utilisant `/Zc:preprocessor`:32767.

- Arguments dans un appel de macro - C++ standard : 256, compilateur Microsoft C++ utilisant `/Zc:preprocessor-`:127 ou utilisant `/Zc:preprocessor`:32767.

- Caractères d'un littéral de chaîne de caractères ou d'un littéral de chaîne large (après concaténation) - C++ standard : 65536, compilateur Microsoft C++ : 65535 caractères codés sur un octet, y compris la marque de fin NULL, et 32 767 caractères codés sur deux octets, y compris la marque de fin NULL.

- Niveaux de définitions de classe, de structure ou d'union imbriquées dans une seule `struct-declaration-list` norme C++ : 256, compilateur Microsoft C++ : 16.

- Initialiseurs membres dans une définition de constructeur - standard C++ : 6144, compilateur Microsoft C++ : au moins 6144.

- Qualifications d'étendue d'un identificateur - C++ standard : 256, compilateur Microsoft C++ : 127.

- Spécifications imbriquées **`extern`** : standard C++ : 1024, compilateur Microsoft C++ : 9 (sans compter la spécification implicite **`extern`** dans l'étendue globale, ou 10, si vous comptez la spécification implicite **`extern`** dans l'étendue globale.).

- Arguments de modèle dans une déclaration de modèle - C++ standard : 1024, compilateur Microsoft C++ : 2046.

Voir aussi

Comportement non standard

Informations de référence sur le préprocesseur C/C++

Article • 16/06/2023

La *référence du préprocesseur C/C++ explique le préprocesseur* tel qu'il est implémenté dans Microsoft C/C++. Le préprocesseur exécute des opérations préliminaires sur les fichiers C et C++ avant qu'ils soient passées au compilateur. Vous pouvez utiliser le préprocesseur pour compiler du code de façon conditionnelle, insérer des fichiers, spécifier des messages d'erreur au moment de la compilation et appliquer des règles propres à l'ordinateur à des sections de code.

Dans Visual Studio 2019, l'option du compilateur /Zc:préprocesseur fournit un préprocesseur entièrement conforme C11 et C17. Il s'agit de la valeur par défaut lorsque vous utilisez l'indicateur `/std:c11` du compilateur ou `/std:c17`.

Contenu de cette section

Préprocesseur
Fournit une vue d'ensemble des préprocesseurs traditionnels et nouveaux conformes.

Directives de préprocesseur
Décrit les directives généralement utilisées pour rendre les programmes sources faciles à modifier et à compiler dans différents environnements d'exécution.

Opérateurs de préprocesseur
Présente les quatre opérateurs propres au préprocesseur utilisés dans le contexte de la directive `#define`.

Macros prédéfinies
Décrit les macros prédéfinies telles que spécifiées par les normes C et C++ et par Microsoft C++.

Pragmas
Présente les pragmas, qui permettent à chaque compilateur d'offrir des fonctionnalités propres aux ordinateurs et aux systèmes d'exploitation tout en conservant une compatibilité globale avec les langages C et C++.

Sections connexes

Informations de référence sur le langage C++

Fournit des documents de référence pour l'implémentation Microsoft du langage C++.

Informations de référence sur le langage C

Fournit des documents de référence pour l'implémentation Microsoft du langage C.

Informations de référence sur les builds C/C++

Fournit des liens vers des rubriques décrivant les options du compilateur et de l'éditeur de liens.

Projets Visual Studio – C++

Décrit l'interface utilisateur de Visual Studio qui vous permet de spécifier les répertoires dans lesquels le système de projet effectuera ses recherches pour trouver les fichiers de votre projet C++.

Référence de bibliothèque standard C++ (STL)

Article • 16/06/2023

Un programme C++ peut appeler un grand nombre de fonctions à partir de cette implémentation conforme de la bibliothèque C++ Standard. Ces fonctions exécutent des services tels que l'entrée et la sortie et fournissent des implémentations efficaces d'opérations fréquemment utilisées.

Pour plus d'informations sur la liaison avec le fichier runtime `.lib` Visual C++ approprié, consultez Fichiers de runtime C (CRT) et de bibliothèque C++ Standard (STL). .lib

> ⓘ **Notes**
>
> L'implémentation par Microsoft de la bibliothèque standard C++ est souvent appelée *bibliothèque stl* ou *bibliothèque de modèles standard*. Bien que *C++ Standard Library* soit le nom officiel de la bibliothèque tel que défini dans ISO 14882, en raison de l'utilisation populaire de « STL » et de « Bibliothèque de modèles standard » dans les moteurs de recherche, nous utilisons parfois ces noms pour faciliter la recherche de notre documentation.

D'un point de vue historique, « STL » faisait à l'origine référence à la bibliothèque de modèles standard écrite par Alexander Stepanov. Certaines parties de cette bibliothèque ont été standardisées dans la bibliothèque standard C++, ainsi que la bibliothèque runtime ISO C, des parties de la bibliothèque Boost et d'autres fonctionnalités. Parfois, « STL » est utilisé pour faire référence aux composants de conteneurs et d'algorithmes de la bibliothèque standard C++ adaptée à partir du STL de Stepanov. Dans cette documentation, la bibliothèque de modèles standard (STL) fait référence à la bibliothèque standard C++ dans son ensemble.

Contenu de cette section

Vue d'ensemble de la bibliothèque C++ Standard Fournit une vue d'ensemble de l'implémentation Microsoft de la bibliothèque standard C++.

iostream programmation Fournit une vue d'ensemble de la `iostream` programmation.

Référence des fichiers d'en-tête Fournit des liens vers des rubriques de référence sur les fichiers d'en-tête de la bibliothèque C++ Standard, avec des exemples de code.