Un livre simple, clair et drôle pour découvrir Le C++ !

C++ POUR LES NULS

2e édition

La première collection informatique dans le monde

A mettre entre toutes les mains !

Stephen Randy Davis

C++ POUR LES NULS

2e édition

C++ pour LES NULS

2e édition

Stephen Randy Davis

FIRST Interactive

C++ pour les Nuls 2e édition
Titre de l'édition originale : C++ For Dummies 5th Edition
Publié par Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774

Edition française publiée en accord avec Wiley Publishing, Inc.
© 2004 Éditions First Interactive
27, rue Cassette
75006 Paris - France
Tél. 01 45 49 60 00
Fax 01 45 49 60 01
E-mail : firstinfo@efirst.com
Web : www.efirst.com

ISBN : 2-84427-642-3
Dépôt légal : 3e trimestre 2004

Collection dirigée par Jean-Pierre Cano
Edition : Pierre Chauvot
Traduction : Daniel Rougé
Couverture : Antoine Paolucci
Production : Emmanuelle Clément

Imprimé en Italie

Sommaire

Introduction

Bienvenue dans la deuxième édition de *C++ pour les Nuls*. Considérez ce livre comme une sorte de pense-bête qui répondra à toutes les questions que vous vous posez au sujet de tous ces trucs fort ennuyeux.

Qu'est-ce qu'il y a dans ce livre ?

C++ pour les Nuls est une introduction au langage C++. *C++ pour les Nuls* démarre au ras des pâquerettes, en s'intéressant aux concepts de base avant d'attaquer des techniques un peu plus sophistiquées. Il ne part pas du principe que vous avez déjà des connaissances antérieures, du moins dans le domaine de la programmation.

Contrairement à d'autres ouvrages sur la programmation, *C++ pour les Nuls* considère que le "pourquoi" est aussi important que le "comment". Les fonctionnalités du langage C++ sont comme un puzzle. Et, plutôt que de présenter bêtement des fonctions, il nous a paru plus important de faire en sorte que vous compreniez comment celles-ci s'agencent entre elles.

Si vous ne comprenez pas pourquoi telle ou telle fonction fait partie du langage, vous avez peu de chance de comprendre comment elle fonctionne. À la fin de ce livre, vous serez en mesure d'écrire un programme en C++ qui tient la route et, ce qui est tout aussi important, vous saurez comment et pourquoi il fonctionne.

C++ pour les Nuls peut aussi servir de référence. Si vous voulez par exemple comprendre ce qu'est un modèle, rendez-vous directement au Chapitre 27. Et si vous n'êtes pas passé par la case Départ, vous trouverez des renvois vers d'autres chapitres lorsque ce sera nécessaire.

C++ pour les Nuls n'aborde pas la programmation sous tel ou tel système d'exploitation. Que vous soyez sous Windows, UNIX ou Linux, c'est pareil. Foin de XP ou de .NET ! De toute façon, il faut bien maîtriser le C++ avant d'aborder ce genre de sujet.

Où se trouvent les programmes du livre ?

Le code source des exemples développés dans ce livre sont disponibles en téléchargement sur le site de l'éditeur (`www.efirst.com`). Vous ouvrez le menu `First Interactive` en haut de la fenêtre de navigation, vous cliquez sur le lien `Téléchargement`, puis sur le titre du livre, et c'est parti !

Votre ordinateur ne peut pas exécuter directement des programmes C++. Il vous faut pour cela un *environnement de développement*. Mais ne regardez pas votre portefeuille d'un air triste : nous verrons dans le Chapitre 1 comment charger et installer tout ce dont vous avez besoin gratuitement. Elle est pas belle la vie ?

Les programmes de *C++ pour les Nuls* sont compatibles avec n'importe quel environnement C++ standard. Y compris donc Dev-C++, celui que je vais vous proposer d'installer. Bien sûr, vous pourrez y étudier et tester les programmes proposés dans le livre, mais aussi toutes les applications que vous écrirez vous-même.

Vous possédez déjà Microsoft Visual Studio .NET ? Formidable... Mais ne vous inquiétez pas : les programmes du livre sont aussi compatibles avec Visual Studio .NET. Je le sais : j'ai essayé. Mais vous trouverez ici de quoi entrer dans l'aventure avant d'aborder le côté sombre de la force...

C'est quoi, le C++ ?

Le C++ est un langage de programmation de bas niveau, orienté objet, aux normes ANSI et ISO.

Orienté objet signifie que le C++ supporte des styles de programmation qui simplifient l'élaboration d'applications évolutives à une grande échelle.

Parce qu'il est de bas niveau, le langage C++ produit des programmes très efficaces et très rapides.

Les certifications ANSI (*American National Standard Institute*) et ISO (*International Organization for Standardization*) garantissent la portabilité du C++. Les programmes C++ sont compatibles avec quasiment tous les environnements de développement modernes. Il existe des compilateurs C++ pour les principaux systèmes d'exploitation, et tous supportent le même langage (du moins l'ensemble du noyau formant le C++, certains systèmes y ajoutant des extensions qui leur sont propres).

Conventions utilisées dans ce livre

Une information ou un message affiché à l'écran apparaît sous la forme suivante :

```
Salut vous autres !
```

Quant aux listings de code, c'est ainsi qu'ils se présentent :

```
// Un programme
void ()
{
   ...
}
```

Si vous saisissez un programme à la main, vous devez le taper exactement comme il se présente, à une exception près : le nombre d'espaces n'est pas important. Pas de problème donc si vous en tapez trop ou pas assez.

Les mots du C++ qui ne sont pas du langage (anglais, *of course*) courant, mais purement informatiques, apparaissent `sous cette forme`. Les noms de fonctions sont toujours suivis de parenthèses ouverte et fermée comme dans `maFonctionPréférée()`. Les arguments des fonctions sont laissés de côté, sauf si cela facilite la lecture. Il est bien plus simple de dire : "voici `maFonctionPréférée()`" que "voici `maFonctionPréférée(int, double)`".

Il vous sera parfois demandé d'entrer des commandes au clavier. Par exemple, si le texte demande d'appuyer sur Ctrl+C, vous devez maintenir la touche Ctrl enfoncée et appuyer sur la touche C, puis relâcher les deux en même temps. Ne tapez jamais le signe "+".

Dans le même ordre d'idée, si vous voyez écrit Fichier/Ouvrir ou encore Fichier->Ouvrir, cela signifie simplement que vous devez utiliser le clavier ou la souris pour ouvrir le menu Fichier puis choisir la commande Ouvrir.

Enfin, vous savez que de nombreuses applications offrent des raccourcis pour accéder aux principales commandes. C'est à cela que servent les touches de fonction et leurs multiples combinaisons avec Ctrl, Alt ou Maj. Comme Dev-C++ et Visual Studio .NET ne se servent pas des mêmes raccourcis, je n'insisterai pas de trop sur ce point.

Ce que vous n'avez pas à lire

Le C++ est un gros morceau. Il y a des parties faciles à avaler, d'autres qui le sont moins. Pour ne pas vous noyer sous des informations qui ne sont pas utiles pour le moment, tout ce qui est technique est signalé par une icône spéciale (reportez-vous à la section "Pictogrammes utilisés dans ce livre").

De plus, certaines informations de fond ont été placées dans des encadrés. Si vous êtes vraiment saturé, vous pouvez sauter cette littérature en première lecture. Mais pensez quand même à y revenir à un moment ou à un autre. Car en C++, ce que vous ignorez vous retombera dessus. Enfin, parfois...

Quelques hypothèses risquées

C++ pour les Nuls ne se pose aucune question quant au niveau du lecteur en programmation, ni même s'il a déjà programmé. Avoir déjà touché à un ordinateur vous aidera bien sûr, mais ce n'est pas vraiment nécessaire.

Cette édition de *C++ pour les Nuls* commence par les concepts les plus élémentaires de la programmation. La progression s'effectue de la syntaxe de base, en veillant à ce qu'elle soit bien acquise, jusqu'à la programmation orientée objet. Le lecteur qui aura assimilé le contenu de ce livre ne devrait rencontrer aucun problème pour impressionner amis et collègues.

Comment ce livre est organisé

Chacune des fonctionnalités commence par ces trois questions :

- *Qu'est* cette nouvelle fonction ?
- *Pourquoi* a-t-elle été introduite dans ce langage ?
- *Comment* agit-elle ?

Des petits bouts de programme sont généreusement distribués tout au long des chapitres. Chacun illustre une nouvelle fonction ou met en évidence quelques brillants points ou éléments de mon cru. Ces fragments sont loin d'être complets et, en eux-mêmes, ils ne font pas grand chose. En revanche, tout concept important est illustré par au moins un programme fonctionnel.

Plus encore pour les nuls...

Je pense qu'il est important de voir les fonctionnalités du C++ œuvrer ensemble dans un programme complet, et non au travers d'exemples épars qui ne donnent pas une vue d'ensemble et dispersent l'attention. J'ai passé plus de temps à me demander comment illustrer ce que fait chaque programme que de comprendre les fonctionnalités qu'il recèle. Et en plus, il n'est pas évident de les comparer car ils ne font pas la même chose.

C'est pourquoi je m'en suis tenu à un seul programme élaboré appelé BUDGET. Il n'est d'abord qu'un simple programme opérationnel, sous la forme d'une variante dénommée BUDGET1, mettant en application les notions étudiées dans les deux premières parties. Ce programme se contente de gérer des comptes bancaires. Il s'étoffe ensuite pour intégrer, sous le nom BUDGET2, les concepts de programmation orientée objet qui sont abordés dans la troisième partie. Et ainsi de suite jusqu'à atteindre le point culminant : BUDGET5.

Vous cherchez déjà le programme BUDGET dans le livre ? Que nenni ! Ce "bonus" (livré brut de forme, parce qu'il faut bien vous faire travailler) fait partie des programmes disponibles en téléchargement sur le site `www.efirst.com`. En moins de temps qu'il ne faut pour le dire, il sera à vous !

Première partie : Introduction à la programmation en C++

La première partie est le début du périple. Vous commencez par découvrir ce qu'écrire un programme signifie. Puis, vous avancerez pas à pas à travers les arcanes de la syntaxe du langage, autrement dit, la signification des commandes C++.

Deuxième partie : Devenir un programmeur opérationnel

Dans cette partie, vous mettrez en pratique votre nouveau savoir sur les commandes C++ en les réunissant pour créer des modules réutilisables dans des programmes.

C'est dans cette partie qu'est présenté le plus redouté de tous les sujets : les pointeurs C++. Si vous ne savez pas ce que cela signifie, vous l'apprendrez bien assez tôt.

Troisième partie : Programmer avec classe

C'est le gros morceau du livre. La troisième partie démarre sur un exposé de la programmation orientée objet. Cette dernière est véritablement la raison d'être du langage C++. Retirez toute la partie OO du C++, et il ne vous reste plus que l'antique langage C. J'aborde aussi quelques subtilités, comme les classes, les constructeurs, les destructeurs et les nachos. Ne vous en faites pas si tous ces concepts vous échappent. Sauf pour les nachos : si vous ne savez pas ce que c'est, vous êtes plutôt mal parti.

Quatrième partie : Les héritages de classe

C'est vraiment au niveau des héritages que la programmation orientée objet donne le meilleur d'elle-même. La compréhension de cet important concept est au cœur de toute programmation efficace en C++, et c'est le but de cette quatrième partie. C'est le point de non-retour : une fois que vous aurez terminé cette étude, vous pourrez vous bombarder Programmeur Orienté Objet de Première Classe.

Cinquième partie : Facultatif, mais si important

La lecture de la cinquième partie vous permettra d'acquérir tout ce dont vous avez besoin pour programmer efficacement en C++. C'est là que j'aborde les autres caractéristiques du langage : entrées/sorties dans les fichiers, gestion des erreurs et modèles.

Sixième partie : Les dix commandements

Que serait *Le C++ pour les Nuls* sans cette partie ? Dans le premier chapitre de cette dernière partie, vous découvrirez comment éviter l'introduction de bogues dans vos programmes.

Vous avez remarqué le nombre d'options de compilation, ces jours-ci ? Nous terminerons par les outils et les options les plus importants dans Dev-C++. Cet environnement ne fait pas partie du langage C++, mais il va devenir le vôtre à force de travailler avec lui. Comprendre ces options facilitera votre vie de programmeur.

Pictogrammes utilisés dans ce livre

C'est de la littérature technique que vous pouvez sauter en première lecture.

Ce pictogramme indique une astuce qui vous fera gagner du temps et vous épargnera bien des efforts.

Rappelez-vous bien de ce qui figure ici car c'est important.

Rappelez-vous aussi de ça ! Cette icône apparaît généralement là où on l'attend le moins et signale un de ces bogues pas faciles à trouver et à éradiquer.

Et maintenant ?

Apprendre un langage de programmation n'est pas une mince affaire. Je vais m'efforcer de vous faciliter la tâche au maximum, mais vous devrez quand même allumer le PC et vous mettre sérieusement au travail. Dégourdissez vos doigts, posez le livre bien à plat à côté du clavier, le dos bien cassé (celui du livre, évidemment) pour qu'il reste grand ouvert (comme ça, vous ne serez pas tenté de le rendre au libraire) et au boulot !

Première partie

Introduction à la programmation en C++

"Nous n'avons pas réussi à sortir notre logiciel Halloween pour le 31 octobre, mais nous développons une version plus stable qui devrait être prête vers l'Ascension."

Dans cette partie...

Le simulateur de vol le plus perfectionné et le programme de comptabilité le plus simple – ce qui ne l'empêche pas d'être puissant – utilisent tous deux les mêmes blocs de base. Dans cette partie, vous découvrirez les fonctions élémentaires qui vous permettront de créer votre application vedette.

Chapitre 1

Votre premier programme en C++

Dans ce chapitre :

- Découvrir ce qu'est le C++.
- Installer le logiciel Dev-C++.
- Écrire votre premier programme en C++.
- Exécuter le programme.

Eh bien nous y voilà ! Et personne d'autre dans les parages que vous et moi. Rien d'autre à faire que d'aller de l'avant. Commençons par le commencement.

Un ordinateur, c'est une machine formidable, mais d'une remarquable bêtise. Il peut réaliser tout ce que vous lui demandez (dans les limites du raisonnable) mais il ne fait strictement que ce qu'on lui demande. Rien de plus et rien de moins.

Malheureusement pour nous, un ordinateur ne comprend rien au langage humain. Il ne parle même pas français. Bon, je sais ce que vous allez dire : "J'ai déjà vu un ordinateur qui comprend de français". En fait, ce que vous avez vu, c'est un ordinateur qui exécute un *logiciel* qui comprend effectivement le français (je suis un peu vague au sujet de ce concept de compréhension d'un langage par un ordinateur, mais comme je n'arrive pas même à expliquer tout cela à mon fils, je n'insisterai pas).

Un ordinateur comprend une des langues appelées *langage informatique* ou *langage machine*. Il est possible mais très difficile pour un humain de parler en langage machine. C'est pourquoi les ordinateurs et les humains ont conclu une sorte d'accord, à savoir l'utilisation de langages intermédiaires comme le C++.

Les humains peuvent parler en C++ (enfin, plus ou moins) et le C++ est traduit en langage machine compréhensible par l'ordinateur.

Saisir les concepts du C++

Au début des années soixante-dix, un consortium de gens vraiment intelligents a travaillé sur un système nommé Multix. Multix était sensé donner à tous les foyers un accès informatique aux graphismes, au courrier électronique, aux données boursières et à la pornographie (bon, d'accord, ne parlons plus de la pornographie). C'était bien sûr une idée complètement folle et le concept tout entier échoua lamentablement.

Une petite équipe d'ingénieurs qui travaillait pour Bell Labs décida de sauver ce qui pouvait l'être de Multix. C'est ainsi qu'elle récupéra un petit système d'exploitation plutôt compact qu'elle surnomma UNIX (UN-IX, Mult-ix, vous saisissez ?).

Malheureusement pour ces ingénieurs, ils ne disposaient pas d'un gros ordinateur mais seulement de petites machines provenant de différents fabricants. En ce temps-là, tout développement informatique était dépendant de telle ou telle machine. Chaque programme devait expressément être réécrit pour chaque type d'ordinateur. Par opposition à ce principe, ces ingénieurs inventèrent un petit langage très puissant, le C.

Le C se répandit comme une traînée de poudre. De nouvelles techniques de programmation virent le jour – notamment la programmation orientée objet –, laissant le langage C à la traîne. Pour qu'il ne soit pas totalement évincé, la communauté des ingénieurs ajouta des fonctionnalités nouvelles au C, qui fut rebaptisé C++.

Le langage C++ est constitué des éléments suivants :

- **La sémantique :** C'est un vocabulaire fait de commandes compréhensibles par le commun des mortels et qui peut être converti assez facilement en langage machine.
- **La syntaxe :** Il s'agit d'un langage structuré (ou *grammaire*) qui permet aux humains de combiner les commandes C++ en un programme qui fait véritablement quelque chose (enfin, qui essaie de faire quelque chose...).

Pour faire simple, la sémantique sert à construire les briques d'un programme C++, tandis que la syntaxe permet de les assembler correctement.

Qu'est ce qu'un programme ?

Un programme en C++ est un fichier de texte contenant une succession de commandes C++ conformes aux règles de la grammaire C++. Ce texte est appelé *fichier source* (sans doute parce qu'il est la source de toutes les frustrations). Il comporte une extension .CPP (N.D.T. : initiales de "C Plus Plus", le lecteur l'aura deviné), tout comme un fichier Microsoft Word se termine par .DOC ou un fichier de commandes MS-DOS se termine par .BAT. Le concept d'extension .CPP n'est jamais qu'une convention, d'ailleurs presque exclusivement utilisée dans l'univers PC.

Le but d'une programmation, c'est d'écrire une succession de commandes C++ susceptible d'être convertie en un programme en langage machine qui effectue la tâche prévue. Ce programme machine exécutable est doté d'une extension .EXE. La création d'un programme exécutable à partir d'un programme C++ est appelée *compilation* (ou *construction* ; il y a une subtile différence dont nous reparlerons dans le Chapitre 22).

Tout ça n'a pas l'air bien compliqué. Alors, où est-ce que ça coince ? Pas de panique, nous y viendrons.

Comment programmer ?

Pour écrire un programme, il vous faut deux choses : un *éditeur* pour élaborer le fichier source *.CPP* et un logiciel qui convertit le fichier source en fichier exécutable *.EXE*, capable d'exploiter les commandes (ouvrir un feuille de calcul, faire de la musique, détruire un astéroïde, enfin tout ce que vous voulez). L'outil qui effectue cette conversion s'appelle un *compilateur*.

De nos jours, les outils dont nous disposons (ou *environnement de développement*) réunissent à la fois l'éditeur et le compilateur. Après avoir tapé votre programme, il suffit de cliquer sur un bouton pour obtenir le fichier exécutable.

Le Visual C++ .NET de Microsoft est l'environnement C++ le plus populaire (n'oubliez pas de prononcer *point NET*). Tous les programmes de ce livre peuvent être compilés et exécutés avec lui. La plupart d'entre vous ne possèdent hélas pas un exemplaire du Visual C++ .NET ou n'ont pas les moyens de se l'offrir.

Il existe fort heureusement des environnements C++ dans le domaine public, comme le GNU C++. Vous trouverez des informations pour télécharger ce progiciel sur le site officiel GNU `www.gnu.org` ainsi que sur le site en version française `www.gnu.org/home.fr.html`.

GNU sont les initiales autoréférentielles de "*GNU is Not UNIX*" (GNU n'est pas UNIX). Le jeu de mots renvoie aux origines du C++. Le GNU est un ensemble d'outils élaboré par la Free Software Foundation.

Dans ce livre, nous allons utiliser un autre environnement de développement, lui aussi totalement gratuit. Il s'appelle Dev-C++, et vous devrez le télécharger sur le site `www.bloodshed.net`. Rassurez-vous : chargement et installation s'effectuent sans douleur aucune. Nous allons y revenir dans un instant.

Bien d'autres programmes du domaine public sont disponibles sur Internet. Certains ne sont pas gratuits ; il vous est demandé de payer une (généralement modeste) redevance. Dev-C++, lui, est totalement gratuit (mais personne ne vous interdit de donner votre obole pour soutenir ce projet).

Dev-C++ n'est pas du tout une édition plus ou moins vérolée, boguée et limitée d'un compilateur C++, programmé à la va-comme-je-te-pousse. Il s'agit d'un logiciel à part entière, un environnement C++ doté de toutes les fonctionnalités nécessaires. Dev-C++ supporte l'intégralité du langage C++ et exécute tous les programmes de ce livre (et ceux des autres livres sur le C++).

Dev-C++ produit des programmes 32 bits compatibles avec Windows, mais il n'est pas capable (*a priori*) de créer des applications ayant le *look* de Windows. Ce n'est pas un progiciel destiné à l'environnement Windows. Si c'est cela que vous recherchez, vous devrez mettre la main au portefeuille et acquérir un logiciel commercial tel que Visual C++ .NET. Cela étant posé, je vous conseille vivement d'étudier les exemples proposés dans ce livre afin d'apprendre C++ *avant* de vous attaquer à l'environnement Windows. Ce sont deux choses différentes. Ne l'oubliez jamais !

Les sections qui suivent expliquent comment installer Dev-C++ et écrire votre premier programme C++. Ce programme est un convertisseur de température Celsius/Fahrenheit.

Les programmes proposés dans ce livre sont compatibles avec Visual C++ .NET comme avec la partie C++ de Visual Studio .NET (en fait, c'est pour l'essentiel la même chose). Reportez-vous à la documentation de Visual C++ .NET pour procéder à son installation. Il est vrai que les messages d'erreur délivrés par Visual C++ .NET ne sont pas les mêmes (tout en restant aussi difficiles à décoder). Mais le territoire devrait sembler étrangement familier. C'est comme en musique : l'orchestration peut changer, mais la chanson est toujours la même !

Installer Dev-C++

Tous les exemples de programmes de ce livre sont compatibles avec les compilateurs C++, y compris Dev-C++. Si vous avez choisi de le télécharger à partir du site `www.bloodshed.net`, voici comment vous devez l'installer :

1. **Dans la fenêtre principale de Bloodshed, cliquez sur lien** `Download`. **Dans la fenêtre suivante, cliquez en haut sur le lien** `Dev-C++`.

2. **Ne vous inquiétez pas de voir la mention *currently beta*. Tout va très bien se passer. Descendez un peu pour trouver le lien** `Go to Download Page`. **Cliquez dessus.**

3. **La version proposée à la date où ce livre est rédigé porte le numéro 4.9.8.0. C'est elle que nous allons charger (voir la Figure 1.1).**

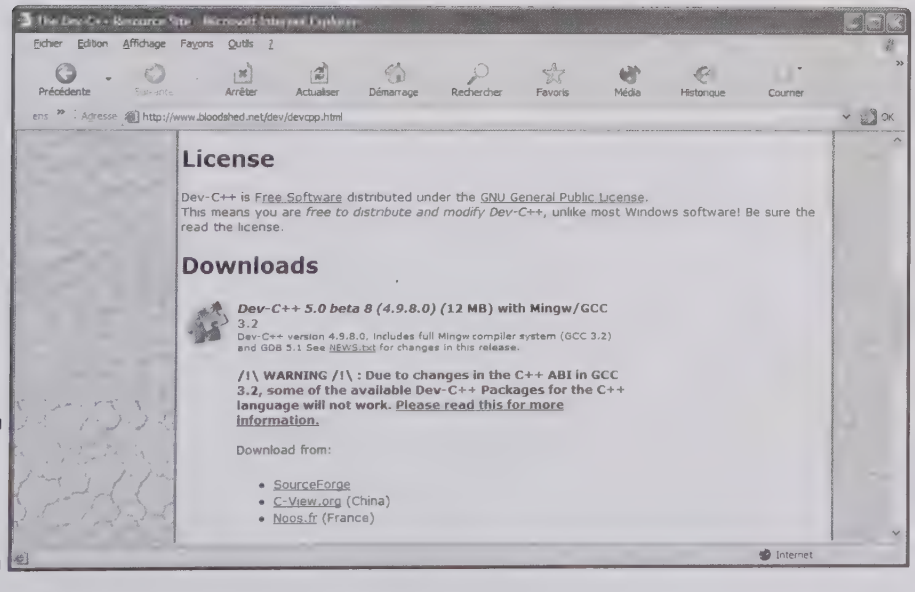


Figure 1.1
Le lien de chargement de Dev-C++.

4. **Choisissez vote site (par exemple, celui de Noos.fr). Définissez le dossier de destination sur votre disque dur. Patientez pendant le chargement du fichier d'installation (il s'appelle** `DEVCPP4980.EXE`**). Vous avez juste le temps de vous préparer un bon café.**

5. **Ouvrez le dossier dans lequel le programme vient d'être enregistré, puis faites un double clic sur son nom. L'installation va débuter.**

6. **Celle-ci commence par un avertissement clair : si une ancienne version est déjà présente sur votre système, il faut la retirer tout de suite (voir la Figure 1.2). Ce n'est pas le cas ? Passez directement à l'Étape 9.**

Figure 1.2
N'oubliez pas de désinstaller d'abord votre ancienne version de Dev-C++.

Vous n'aviez jamais entendu parlé de Dev-C++ auparavant ? Ne tenez pas compte du message : ce n'est qu'un rappel.

7. **Validez, puis quittez le programme d'installation (bouton Cancel). Ouvrez le dossier dans lequel se trouve l'ancienne version (il s'agit probablement de `Dev-Cpp`). Faites un double clic sur le fichier `UPDATE.EXE`. Laissez-le faire son travail.**

8. **Vous pouvez maintenant supprimer définitivement le dossier de l'ancienne version, puis revenir à l'Étape 5 pour reprendre la procédure d'installation.**

9. **Lisez le contrat de licence. Étant donné le prix que vous avez payé, vous pouvez cliquer sans aucun remord sur le bouton `I agree`.**

10. **Dev-C++ ouvre maintenant la fenêtre illustrée sur la Figure 1.3. Les options par défaut conviennent parfaitement. Vous devez cependant veiller à deux points. Tout d'abord, la ligne Mingw compiler system doit absolument rester cochée. D'autre part, l'option Associate C and C++ files to Dev-C++ devrait être désactivée si vous ne voulez pas ouvrir automatiquement les fichiers .CPP dans Dev-C++ (ce qui peut être le cas si vous envisagez d'installer aussi Visual C++ .NET). Ces remarques étant faites, vous pouvez cliquer sur le bouton `Next`.**

N'associez pas les fichiers .CPP à Dev-C++ si Visual Studio .NET est également installé. Tous deux peuvent coexister pacifiquement, mais cela ne signifie pas que l'on puisse impunément chatouiller Visual Studio .NET.

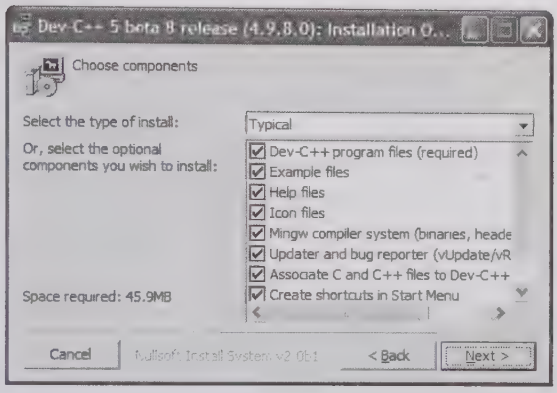


Figure 1.3
Les options d'installation par défaut devraient convenir à la plupart des utilisateurs.

Vous aurez toujours la possibilité de choisir votre éditeur en cliquant avec le bouton droit de la souris sur un fichier .CPP puis en sélectionnant la commande Ouvrir avec. J'avoue que c'est ma méthode préférée. D'ailleurs, Dev-C++ se lance nettement plus vite que Visual Studio .NET.

11. **Le programme d'installation vous demande maintenant dans quel dossier vous voulez copier Dev-C++ (voir la Figure 1.4). La bonne idée, c'est d'accepter la proposition par défaut (C:\Dev-Cpp).**

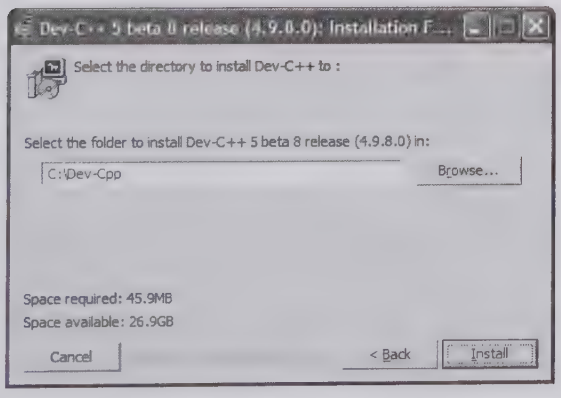


Figure 1.4
L'emplacement par défaut pour installer l'environnement Dev-C++.

N'installez pas Dev-C++ dans le dossier habituel \Program Files. En effet, les noms de dossiers comportant des espaces peuvent poser des problèmes (n'oubliez pas que les programmes que vous compilerez seront de type MS-DOS). Limitez-vous à un intitulé simple. Le nom par défaut convient parfaitement.

12. **Avant de poursuivre, il serait peut-être judicieux de vérifier l'espace libre dont vous disposez. Certes, Dev-C++ ne consomme qu'un peu plus de 45 Mo, mais on ne sait jamais.**

13. **Vous pouvez maintenant cliquer sur `Install`. Le programme d'installation va copier tous les fichiers dans le dossier Dev-Cpp (et rien dans celui de Windows). Patientez en vérifiant du coin de l'œil l'état d'avancement de l'opération (voir la Figure 1.5).**

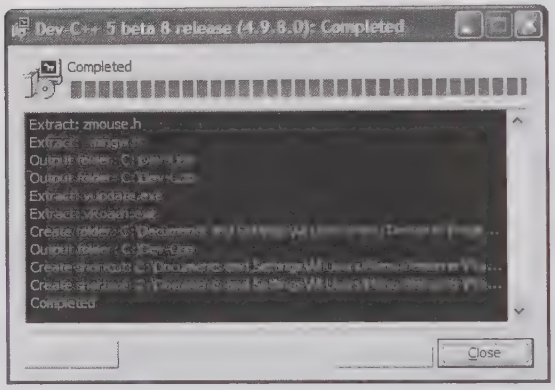


Figure 1.5
L'installateur décompresse un grand nombre de fichiers.

14. **Dev-C++ va vous demander si l'installation doit être faite pour tous les utilisateurs (*all users*). Autrement dit : si une autre personne que vous se sert de votre ordinateur, aura-t-elle le droit d'exécuter Dev-C++ ? Pour mon compte, la réponse est *oui*.**

15. **Il ne vous reste plus qu'à cliquer sur le bouton `Close` pour terminer l'installation.**

Normalement, Dev-C++ devrait se lancer automatiquement. Comme il s'agit d'une première utilisation, il devrait vous demander de choisir un style d'interface (question de goût) et surtout votre langue préférée. Si vous craignez d'avoir oublié votre vocabulaire suédois de base, n'hésitez pas un instant : passez tout de suite au français !

Configurer les options

Si vous avez l'habitude d'installer des logiciels, vous savez depuis longtemps que c'est un travail souvent plus délicat que ce qu'en disent les documentations (sauf dans le cas de Dev-C++ !). Et vous savez aussi que configurer un

programme est une procédure en soi. Pour ce qui nous concerne ici, deux options de Dev-C++ doivent être correctement définies avant de pouvoir l'utiliser. Voyons cela d'un peu plus près.

1. **À partir de l'interface de Dev-C++, choisissez la commande Options du compilateur dans le menu Outils. Vous pourriez modifier ces réglages plus tard, mais il vaut mieux le faire tout de suite.**

2. **Cliquez sur l'onglet Options. Dans le menu de gauche, choisissez la ligne Génération du code.**

3. **Dans la liste de droite, cliquez à la fin de la ligne Gestion des exceptions. Servez-vous de la petite flèche de droite pour sélectionner le mot Yes (voir la Figure 1.6).**

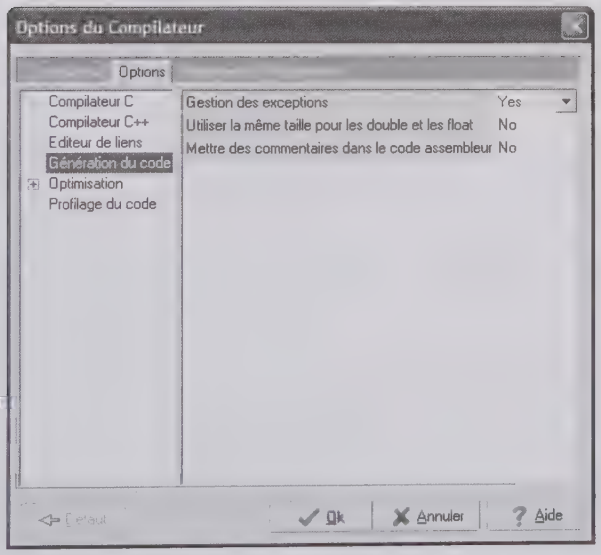


Figure 1.6
La gestion des exceptions doit être activée.

4. **Cliquez dans le menu de gauche sur la ligne Éditeur de liens. Activez de la même façon que ci-dessus l'option intitulée Générer des informations de débogage (voir la Figure 1.7).**

5. **Cliquez sur OK. Vos options sont automatiquement sauvegardées et l'installation est terminée !**

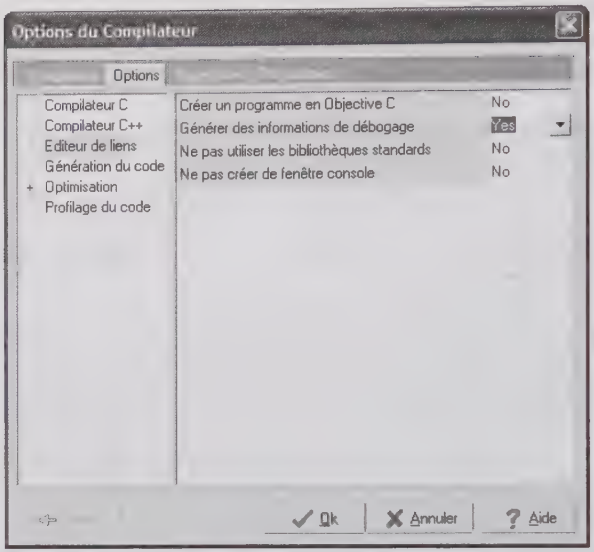


Figure 1.7
La génération des informations de débogage doit aussi être activée.

Créer votre premier programme C++

Dans cette section, vous allez créer votre premier programme C++. Vous taperez d'abord le *code* (ou si vous préférez le *programme*) dans un fichier nommé `Convert.cpp`, puis vous le convertirez en un programme exécutable.

Taper le code C++

La première phase de toute écriture d'un programme en C++ consiste à taper les instructions dans un éditeur de texte. Voilà qui tombe bien : l'interface utilisateur de Dev-C++ est justement bâtie autour d'un éditeur spécifiquement conçu pour créer des programmes C++.

1. **Dans le menu Démarrer, choisissez Tous les programmes, puis ouvrez le groupe Bloodshed Dev-C++ et cliquez enfin sur la ligne Dev-C++. Consultez si vous le souhaitez le conseil du jour.**

   L'interface utilisateur de Dev-C++ ressemble fondamentalement à n'importe quelle autre application Windows (il y a en a de plus belles, mais c'est bien du Windows pur et dur).

   Trop de clics d'un coup vous fatigue ? Créez un raccourci vers Dev-C++ sur votre bureau, vous gagnerez du temps. Il vous suffit pour cela de

cliquer sur le fond du bureau avec le bouton droit de la souris. Choisissez dans le menu l'option Nouveau, puis Raccourci. Localisez dans le dossier \Dev-cpp l'exécutable devcpp.exe. Donnez au raccourci un nom à votre convenance. À partir de maintenant, un double clic suffira à lancer Dev-C++.

2. **Dans le menu Fichier, cliquez sur Nouveau, puis choisissez Fichier Source.**

Le raccourci Ctrl+N plaira aux lecteurs pressés.

Dev-C++ ouvre une fenêtre (appelée par défaut *SansNom1*) dans laquelle vous allez saisir votre nouveau code. Vous vous demandez ce que vous pourriez bien entrer ? Justement, c'est pour le savoir que vous avez acheté ce livre !

3. **Entrez le programme exactement comme il est écrit ci-dessous.**

Si vous êtes malin, vous n'aurez même pas à faire cet effort. Il vous suffit de charger les exemples du livre sur le site de First Interactive. Décompressez, ouvrez, c'est déjà prêt !

Ne vous souciez pas trop des indentations et espacements. Le fait qu'une ligne soit en retrait de deux ou trois espaces, ou que deux ou trois espaces séparent des mots n'a aucune importance. Retenez cependant que le langage C++ fait la différence entre les majuscules et les minuscules. Assurez-vous de tout taper en minuscules.

```
//
//  Conversion de températures de degrés Celsius
//  en degrés Fahrenheit :
//  Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{

  // saisie de la température en Celsius
  int celsius;
  cout << "Entrez la température en Celsius : ";
  cin >> celsius;
```

```
    // calcul du facteur de conversion de Celsius
    // en Fahrenheit
    int factor;
    factor = 212 - 32;

    // applique le facteur de conversion pour le passage
    // de degrés Celsius en degrés Fahrenheit
    int fahrenheit;
    fahrenheit = factor * celsius/100 + 32;

    // affiche les résultats (suivis d'un saut de ligne)
    cout << "Valeur en degrés Fahrenheit : ";
    cout << fahrenheit << endl;

    // avant de terminer le programme, attend le signal de
    // l'utilisateur pour lui permettre de voir le résultat
    system("PAUSE");
    return 0;
}
```

Après avoir tapé ce code, cliquez sur Fichier/Sauvegarder sous en nommant le fichier `Conversion.cpp`.

Je sais bien que tout ça n'a pas l'air très enthousiasmant, mais il se trouve que vous venez bel et bien d'écrire votre premier programme en C++ !

Pour les besoins du livre, j'ai créé dans le dossier de Dev-C++ un sous-répertoire appelé `Nuls`. Pour enregistrer ce premier exemple, j'ai ajouté un niveau de plus que j'ai appelé `Chap01`. C'est dans ce dernier dossier que j'ai sauvegardé `Conversion.cpp`. Notez bien que si les noms possédant plus de huit caractères ne font pas peur à Dev-C++, il n'en va pas forcément de même avec ceux qui contiennent des espaces.

Construire le programme

Après avoir enregistré le programme `Conversion.cpp` sur le disque dur, nous pouvons envisager de générer les instructions exécutables par l'ordinateur.

Pour construire le programme `Conversion.cpp`, sélectionnez l'option Compiler dans le menu Exécuter ou appuyez simplement sur la combinaison de touches Ctrl+F9. Vous pouvez aussi cliquer sur le petit bouton qui se trouve à gauche de la seconde barre d'outils (laissez le pointeur de la souris quelques instants au-dessus des boutons pour repérer leur rôle).

Dev-C++ va ouvrir une fenêtre de compilation. Il semble que rien ne se passe (en fait, le compilateur réfléchit). Au bout de quelques secondes, vous devriez voir apparaître le message illustré sur la Figure 1.8. Il indique que tout s'est bien passé.

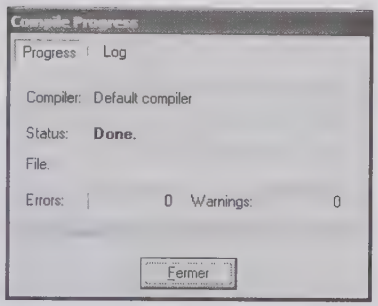


Figure 1.8
Le message Done indique que la compilation s'est effectuée sans erreur.

Dev-C++ génère un message d'erreur s'il trouve une faute quelconque dans votre programme. Et les erreurs de programmation sont aussi fréquentes que la neige sur le Mont-Blanc... Vous aurez sans aucun doute droit à bon nombre de messages de mises en garde et d'erreurs, peut-être même dans un programme aussi élémentaire que `Conversion.cpp`. Pour vous donner une idée de ce qu'est une erreur, remplacez, dans la Ligne 17, `cin >> celsius;` par `cin >>> celsius;`.

Cette erreur semble bien vénielle, pardonnable par vous ou par moi, mais certainement pas par le C++. Dev-C++ va ouvrir l'onglet Compilateur (en bas de la fenêtre principale) et afficher le message illustré sur la Figure 1.9. Le texte est peut-être un peu laconique, mais il indique bien qu'une erreur a été rencontrée en abordant la série de caractères ">" sur la ligne 17. Supprimez le signe en trop et recommencez la compilation.

Le terme *parse* (analyse) signifie qu'il faut convertir les commandes C++ en quelque chose de compréhensible par le générateur du code machine.

L'exécution du programme

Le moment est venu d'exécuter votre nouvelle création. C'est-à-dire de lancer le fichier `Conversion.exe` et de taper quelque chose pour voir s'il fonctionne bien.

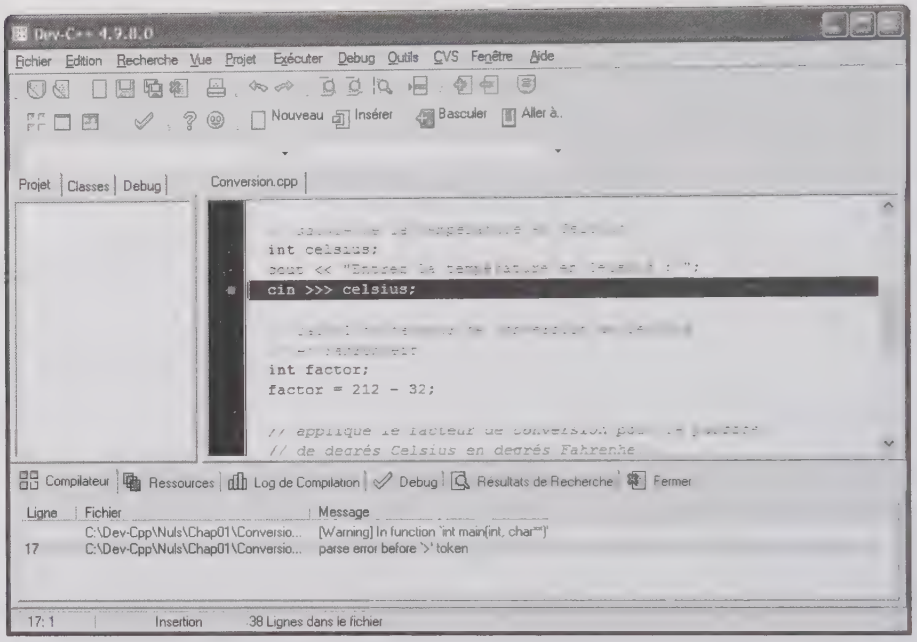


Figure 1.9
Les programmes mal nés génèrent des messages d'erreur dans la fenêtre Compilateur.

Pourquoi le C++ est-il si pinailleur ?

Le langage C++ n'a pas été long à repérer l'erreur que j'avais subrepticement glissée dans l'exemple précédent. Mais si Dev-C++ est si malin, comment se fait-il qu'il n'ait pas pu corriger le problème tout seul ?

La réponse est simple et profonde. Dev-C++ a pensé que j'avais fait une faute de frappe en tapant le symbole ">>", mais il n'en est pas sûr. Ce qui semblait être une faute de saisie aurait pu être une erreur en fait complètement différente. Si le compilateur s'était contenté de corriger l'erreur, Dev-C++ aurait masqué le véritable problème.

Localiser une erreur enfouie dans un programme compilé sans avertissements explicites serait terriblement long et compliqué, et cela ferait perdre un temps fou. Il vaut mieux que le compilateur détecte tout de suite ce qui ne va pas. Autrement, ce serait une sacrée perte de temps (de *notre* temps). Il vaut mieux que le compilateur travaille quelques secondes de plus pour nous éviter parfois des heures de recherche.

Pour exécuter le programme `Conversion`, cliquez sur l'option Exécuter du menu portant le même nom, ou appuyez sur la combinaison de touches Ctrl+F10 (je ne sais pas pourquoi ils ont choisi cette combinaison : je pensais naïvement qu'une action aussi courante que l'exécution d'un programme serait associée à une seule touche de fonction ; mais il y a sans doute des raisons qui me dépassent).

Une fenêtre s'ouvre aussitôt et vous demande d'entrer une température en degrés Celsius, 100 par exemple. Après avoir appuyé sur la touche Entrée, le programme retourne l'équivalent en degrés Fahrenheit, soit 212° F. Ce qui nous donne le dialogue ci-dessous :

```
Entrez la température en Celsius : 100
Valeur en degrés Fahrenheit : 212
Appuyez sur une touche pour continuer...
```

La dernière ligne vous permet de lire le contenu de la fenêtre avant qu'elle ne se referme. Appuyez sur une touche quelconque et c'est parti de l'écran... Félicitations ! Vous venez d'écrire, compiler et exécuter votre premier programme en C++.

Dev-C++ est Windowsphobe

Notez que Dev-C++ n'a pas été développé pour programmer des applications Windows. C'est théoriquement possible (si l'on en juge par les exemples fournis avec Dev-C++), mais ce n'est pas facile. Il faut reconnaître que Visual Studio .NET est nettement plus compétent dans ce domaine.

Les applications Windows sont dotées d'une interface graphique et s'affichent dans des fenêtres sur votre bureau. Or, `Conversion.exe` est un programme 32 bits qui s'exécute *sous* Windows, mais ce n'est pas un programme Windows au sens visuel du terme.

Ne vous en faites pas si vous ne savez pas ce qu'est un programme 32 bits. Comme je l'avais signalé précédemment, ce livre n'aborde pas l'écriture de programmes Windows. Les programmes en C++ que vous créerez avec ce livre affichent une ligne de commande qui s'exécute dans une "fenêtre MS-DOS" (ce que Windows XP appelle *l'invite de commandes*).

Que les programmeurs Windows ne se désespèrent pas, car ils n'auront perdu ni leur temps ni leur argent. L'apprentissage du C++ est un préliminaire à l'écriture de programmes Windows.

L'aide de Dev-C++

Dev-C++ est doté d'une aide (en anglais) accessible via l'habituel menu. Cliquez sur Aide, puis choisissez l'option Aide sur Dev-C++. On peut regretter que les informations fournies concernent essentiellement l'environnement de développement lui-même. Ne vous attendez pas à y découvrir autre chose qu'une présentation limitée du langage C++.

Relire le programme annoté

Lire un programme écrit par quelqu'un d'autre n'est pas très folichon. Vous arriverez tout de même à reconnaître dès à présent quelques fonctions de `Conversion.cpp` et nous pouvons le parcourir à la recherche d'éléments communs à tous les programmes.

Examen du cadre de référence de tous les programmes C++

Tous les programmes en C++ que vous écrirez avec ce livre utilisent le même cadre (ou modèle) de référence :

```
//
//  Template - fournit un cadre de référence à utiliser
//             comme point de départ
//
// Les fichiers inclus ici définissent la majorité des
// fonctions dont peuvent avoir besoin presque tous
// les programmes que vous pourrez écrire
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // votre code C++ débute ici

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
```

```
    system("PAUSE");
    return 0;
}
```

Sans entrer dans les détails ennuyeux, disons que l'exécution commence avec le code contenu entre les accolades ouverte et fermée, juste après la ligne `main()`.

Le code ci-dessus est disponible sous le nom `Template.cpp` dans le fichier d'accompagnement que vous pouvez télécharger sur le site de l'éditeur.

Clarifier le code source par des commentaires

Les premières lignes de `Conversion.cpp` semblent être de la prose lisible par un être humain, et peu importe qu'elles ne signifient pas grand chose pour Dev-C++. Ces six premières lignes sont en effet des *commentaires,* c'est-à-dire des explications rédigées par le programmeur et qui servent à expliciter ce qu'il ou elle fait ou pense en écrivant un morceau de code particulier. Le compilateur les ignore superbement.

Un commentaire en C++ commence par une double barre (//) et se termine par un caractère de nouvelle ligne. N'importe quel caractère y est autorisé. Une ligne de commentaire peut être aussi longue que vous le désirez, mais il est de règle de la limiter à quatre-vingts caractères afin qu'elle tienne sur une largeur l'écran, ou encore sur une feuille de papier. C'est juste une question pratique et un problème de lisibilité.

À la bonne vieille époque de la machine à écrire, où la saisie s'appelait encore la frappe, le caractère de nouvelle ligne était appelé "retour chariot". Un *caractère de nouvelle ligne* est un caractère qui termine une ligne de commande.

Le langage C++ autorise une autre forme de commentaire : tout texte placé entre les signes /* et */ est ignoré. Mais cette syntaxe n'est normalement plus utilisée en C++ (sauf dans un cas que nous verrons plus tard).

Il peut sembler bizarre de disposer d'une commande du C++ (ou de tout autre langage informatique) qui est tout spécialement ignorée par l'ordinateur. En fait, tous les langages de programmation possèdent une forme de commentaire. Il est en effet important que le programmeur puisse expliquer ce qui lui passe par la tête lorsqu'il écrit son code. Car rien ne garantit qu'un autre développeur comprendra d'emblée le programme qu'il est sensé utiliser ou modifier. Après plusieurs mois, le créateur lui-même peut finir par oublier ce qu'il a si savamment concocté.

Programmes de base et instructions C++

Tous les programmes (du moins ceux qui nous intéressent dans ce livre) sont basés sur ce qu'il est convenu d'appeler des *instructions* C++. Cette section expose les instructions qui forment le cadre de travail du programme `Conversion.cpp`.

Une *instruction* est un ensemble simple de commandes. Toutes les instructions autres que les commentaires se terminent par un point-virgule (la raison pour laquelle les commentaires ne se terminent pas par un point-virgule est obscure ; à mon avis, ça aurait dû être le cas, ne serait-ce que par cohérence. Mais justement, personne ne me demande mon avis.)

L'exécution du programme commence par la première instruction C++ venant après l'accolade ouverte et se poursuit tout au long du listing, à raison d'une instruction à la fois.

En parcourant le programme, vous rencontrez des espaces, des tabulations et des nouvelles lignes. En ce qui me concerne, j'ai fait un retour à la ligne après chacune des instructions d'un programme. Ces caractères sont connus sous le nom générique d'*espaces blancs* car ils n'affichent rien sur le moniteur.

Vous pouvez ajouter des espaces blancs n'importe où dans votre programme afin d'améliorer sa lisibilité, hormis au milieu d'un mot.

```
Vous voyez ce qu
e je veux dire ?
```

Le C++ ignore les espaces blancs, mais pas la *casse*. C'est même une obsession chez lui. La variable `vitessemaxi` et la variable `VitesseMaxi` n'ont rien de commun. De même, alors que la commande `int` est parfaitement comprise par le C++, il n'aura aucune idée de ce que peut bien signifier `INT`.

Écrire des déclarations

La ligne `int nCelsius;` est une instruction de déclaration. Une *déclaration* est une instruction qui définit une variable. Une *variable* est un "conteneur" qui renferme une valeur d'un certain type (comme un nombre ou une chaîne de caractères).

Le terme variable est issu de la terminologie de l'algèbre, comme dans :

```
x = 10
y = 3 * x
```

Dans la deuxième expression, y représente trois fois x, mais qu'est x ? La variable x est le conteneur d'une valeur. Dans ce cas, la valeur de x est 10, mais elle aurait aussi bien pu être 20, 30 ou -1. La deuxième formule tient la route, quelle que soit la valeur de x.

En algèbre, vous avez le droit de commencer par une instruction comme x = 10. Mais en C++, le programmeur doit d'abord définir la variable x avant de pouvoir l'utiliser.

En C++, une variable a un type et un nom. La variable définie à la ligne 11 est appelée *celsius* ; `celsius` a été déclaré de manière à contenir une valeur entière. Car en anglais, `int` est l'abréviation de *integer,* entier.

Le nom d'une variable n'a pas de signification particulière. Il faut néanmoins qu'elle commence par une lettre de A à Z, ou bien de a à z. Tous les caractères qui suivent doivent être une lettre, un chiffre de 0 à 9, ou un caractère de soulignement (_). La longueur du nom d'une variable n'est pas limitée.

Il est de règle que le nom d'une variable commence par une minuscule, et que chaque nouvel élément de nom commence par une majuscule, comme dans `maVariable`. J'expliquerai dans le Chapitre 5 la raison d'être du `n` qui précède `Celsius`.

Efforcez-vous de donner à vos variables des noms brefs et évocateurs. Évitez un nom comme x car il ne signifie rien. Un nom de variable comme `longueurDuSegmentDeLigne` est beaucoup plus descriptif.

Générer les sorties

Les lignes qui commencent par `cout` et `cin` sont des instructions dites d'entrée/sortie ; elles sont souvent abrégées sous le nom d'instructions E/S (à l'instar des technocrates, les programmeurs affectionnent les abréviations).

La première instruction E/S demande de sortir la phrase `Entrez la température en Celsius` dans `cout` (les anglophones prononcent ce mot "*see out*", voir en sortie) ; `cout` est le nom générique de tout périphérique de sortie standard C++. Il s'agit en l'occurrence du moniteur.

La ligne suivante est exactement le contraire. Elle demande d'extraire une valeur du périphérique d'entrée C++ et la stocke dans la variable entière `celsius`. Le périphérique d'entrée C++ est normalement le clavier. Cette expression est analogue à la formule algébrique x = 10 mentionnée précédemment. Pour la suite du programme, la valeur de `celsius` est celle entrée ici par l'utilisateur.

Calculer des expressions

Tous les programmes, même les plus simples, effectuent des calculs d'un type ou d'un autre. En C++, une *expression* est une instruction qui effectue un calcul. Ou, en d'autres termes, une expression est une instruction qui contient (ou possède) une valeur. Un *opérateur* est une commande qui génère une valeur.

Regardons le programme `Conversion`. Dans les deux lignes de calcul d'expression, le programme déclare une variable `factor` et lui assigne (on dit aussi *affecte*) la valeur résultant d'un calcul. Cette commande calcule la différence entre 212 et 32. Dans cet exemple, l'opérateur est le signe moins, "-", tandis que l'expression est "212-32".

Stocker les résultats d'une expression

Le langage parlé peut être très ambigu. Le terme *égal* est l'une de ces ambiguïtés, car il signifie que deux choses ont une même valeur, comme dans "une brique est égale à dix mille francs". Le terme "égal" peut aussi être une affectation, comme dans la formule mathématique "y est égal à trois fois x".

Pour éviter toute ambiguïté, les programmeurs C++ parlent d'*opérateur d'affectation*. Celui-ci stocke dans la variable située à gauche du signe "=" le résultat d'une expression située à droite du signe "=". Pour un programmeur, la valeur 212-32 a été *affectée* à "factor".

Examen de la suite de Conversion.cpp

La deuxième expression de `Conversion.cpp` est un peu plus compliquée que la première. Elle utilise des symboles mathématiques, comme "*" pour la multiplication, "/" pour la division et "+" pour l'addition. Mais dans ce cas précis, les calculs sont effectués sur des variables et non uniquement sur des constantes.

La valeur contenue dans la variable `factor` (qui est d'ailleurs calculée immédiatement avant) est multipliée par la valeur contenue dans `nCelsius` (qui a été

entrée au clavier). Le résultat est divisé par 100 puis 32 est ajouté. Le résultat de la totalité de l'expression est affecté à la variable entière `fahrenheit`.

Les deux dernières commandes sortent à l'écran la chaîne "Valeur en degrés Fahrenheit :", suivie de la valeur de `fahrenheit`.

Comme vous pouvez le constater en testant ce programme, Dev-C++ a quelques problèmes avec les caractères accentués. Ce n'est d'ailleurs pas de sa faute. Les coupables sont surtout C++ lui-même et le fait que les programmes sont conçus pour être exécutés en mode ligne de commandes (c'est-à-dire sous MS-DOS). Désolé, mais il n'y a pas de solution simple et évidente à ce problème !

Chapitre 2

Déclarer des variables en permanence

Dans ce chapitre :

- Déclarer les variables.
- Déclarer différents types de variables.
- Utiliser des variables en virgule flottante.
- Déclarer et utiliser d'autres types de variables.

De tous les concepts du C++, le plus important est celui de *variable*. Une variable est comparable à une petite boîte dans laquelle il est possible de stocker des informations, notamment des nombres, que vous utiliserez ultérieurement. La notion de variable est empruntée à la mathématique. Dans une déclaration comme :

```
x = 1
```

la valeur 1 est stockée dans la variable x. Partant de là, la variable x pourra être utilisée à la place de la constante 1, jusqu'au moment où une valeur est affectée à *x*.

C'est ce principe qui régit les variables du C++. Vous écrirez donc l'affectation sous cette forme :

```
x = 1;
```

À partir de ce moment précis, et à moins que la valeur ait varié, toute référence à x est l'équivalent d'une référence à 1. Nous disons que la valeur de x est 1.

Malheureusement, en C++, les variables sont un peu plus compliquées que d'un point de vue strictement mathématique. Ce chapitre est consacré aux différents aspects des variables C++.

Déclarer des variables

Le C++ stocke les valeurs numériques dans des petites boîtes de stockage appelées *variables*. Un mathématicien ne se prive pas de les utiliser à tout bout de champ. Il écrira par exemple :

```
(x + 2) = y / 2
x + 4 = y
solution pour x et y
```

Le lecteur se rend compte que le mathématicien a introduit les variables *x* et *y* sans avoir à préciser préalablement que ce sont des variables. Le C++ n'est pas aussi malin (je vous avais bien dit que les ordinateurs sont rapides mais bêtes).

Vous devez déclarer chaque variable à C++ avant de pouvoir les utiliser. Vous serez donc obligé d'écrire :

```
int x;
x = 10;

int y;
y = 5;
```

Vous déclarez ainsi qu'il existe deux variables (x et y) de type `int` (entier) ; nous étudierons les types de variables dans la prochaine section. Vous pouvez déclarer des variables partout (ou presque) dans le programme, du moment qu'elles le sont *avant* d'être utilisées.

Déclarer différents types de variables

Vous vous imaginez sans doute qu'en maths, les variables sont de vagues boîtes capables de retenir tout ce que l'on y fourre. Ce qui vous conduirait à écrire :

```
x = 1;
x = 2.3
x = "Ceci est une phrase"
x = Ile de France
```

Le C++ n'est pas aussi souple. Mais il sait faire des choses auxquelles vous ne sauriez prétendre, comme additionner un million de chiffres en une seconde. En fait, il existe autant de types de variables qu'il y a de sortes de boîtes de stockage. Certaines sont si petites qu'elles ne peuvent contenir qu'un seul nombre. Celles qui stockent des phrases doivent être un peu plus grandes. Et bien sûr, aucune n'est suffisamment vaste pour contenir l'Ile-de-France (peut-être Paris, mais rien n'est moins sûr...)

Vous devez indiquer à C++ la taille de la boîte qu'il doit réserver avant de pouvoir utiliser une variable C++. De plus, les propriétés des variables changent d'un type à un autre. Pour le moment, nous n'avons rencontré que la variable de type *int* :

```
int x;
x = 1;
```

Le type de variable `int` est l'équivalent d'un nombre entier. Un *entier* est un nombre qui ne possède pas de partie décimale. Les nombres entiers sont aussi appelés "nombres naturels".

Les entiers sont parfaits pour de nombreux calculs. Vous pourriez faire presque toute votre scolarité en n'utilisant qu'eux. C'est le cas dans les petites classes, jusqu'au moment où les choses se gâtent à cause des fractions. Bref, il en va de même en C++, où plus de 90 % des variables sont des entiers, c'est-à-dire des variables de type `int`.

Les variables `int` ne conviennent malheureusement pas toujours aux calculs à effectuer. Si vous avez essayé le programme de conversion de température du Chapitre 1, vous aurez remarqué – mais ce n'est pas évident – qu'il a un problème : il ne gère que les températures exprimées par un entier. Cette limitation aux entiers n'est pas rédhibitoire pour un usage domestique, car il n'est ni très important – sauf pour un météorologue – ni très amusant d'entrer une température sous une forme fractionnelle comme 10,5°. Pire encore, le programme de conversion ignore totalement et sans rechigner la partie décimale de la température qu'il calcule (puisque pour lui, ce n'est que valeur entière). Du coup, il peut en résulter qu'une région de France batte un record de froid alors qu'une autre aura atteint une température plus basse d'un demi-

degré, mais qui n'aura pas été prise en compte à cause de la troncature des valeurs.

N.D.T. : du fait que, dans les langages informatiques, le point décimal est utilisé comme séparateur dans les nombres réels, cette notation est évidemment respectée pour les programmes et les exemples en C++. En revanche, nous conserverons la virgule décimale dans les exemples et les démonstrations purement arithmétiques.

Les entiers en C++ et leurs limites

En C++, la variable `int` est un nombre entier. Elle souffre des mêmes limitations que son équivalent mathématique.

Arrondi

Prenons comme exemple le calcul de la moyenne de trois nombres. Avec trois variables `int` nommées `nValeur1`, `nValeur2` et `nValeur3`, le calcul de la moyenne s'effectue selon la formule :

```
(nValeur1 + nValeur2 + nValeur3) / 3
```

Du fait que ces trois valeurs sont des entiers, le résultat est sensé être lui aussi un entier. La somme de trois valeurs 1, 2 et 2 est 5. Divisée par trois, cette somme donne 1,666. Du fait que les trois variables `nValeur1`, `nValeur2` et `nValeur3` sont des entiers, la somme est aussi supposée être un entier. Contrairement aux humains (qui sont des gens raisonnables), les ordinateurs (qui ne le sont pas) produisent un quotient entier en forçant 1,666 en 1.

Le fait de ne pas tenir compte de la partie décimale d'un nombre est appelé *troncature* ou arrondi. Pour de nombreuses applications, la troncature n'est pas une affaire. Certains pourraient s'en accommoder mais évidemment pas les matheux et les boursicoteurs. La troncature peut même s'avérer pire que cela ! Considérons la formule suivante :

```
nValeur1/3 + nValeur2/3 + nValeur3/3
```

Avec les mêmes valeurs respectives 1, 2 et 2, vous obtenez 0 ! Pour comprendre comment c'est possible, il suffit de savoir que la troncature de 1/3 donne 0, et qu'il en va de même pour la troncature de 2/3. La somme de trois fois 0 donne 0 (sauf pour Raymond Devos, pour qui "trois fois rien, c'est pas

rien"). Ce qui démontre bien que la troncature peut s'avérer totalement inacceptable.

Limites de plage

L'autre problème du type de variable `int`, c'est sa plage de valeurs limitée. Une variable `int` normale peut stocker une valeur maximale de 2 147 483 647 et une valeur minimale de -2 147 483 648, soit plus ou moins deux milliards et des poussières, pour une plage totale d'un peu plus de quatre milliards.

Deux milliards, c'est déjà beaucoup. C'est largement suffisant pour la plupart des applications. Mais pas toujours. Par exemple, la fréquence dhorloge de votre ordinateur peut être supérieure à 2 GHz (ou alors il commence à dater). Le préfixe *giga* signifie précisément milliard. Sachez aussi qu'une fibre optique est capable de véhiculer plus de deux milliards de bits par seconde. Impressionnant, non ?

C++ peut (plus ou moins) vous aider à dépasser les bornes en indiquant qu'un entier n'est pas signé (autrement qu'il est forcément positif). Il faut alors le déclarer de type `unsigned int` et la plage de valeurs autorisée s'étend alors de 0 à 4 294 967 295. Voilà qui est beaucoup mieux encore que le Loto !

La déclaration `unsigned` est suffisante. Dans ce cas, `int` est implicitement sous-entendu.

Résoudre les problèmes de troncature

Les limitations des variables `int` sont inacceptables dans certaines applications. Fort heureusement, le C++ reconnaît les nombres décimaux. Un nombre décimal peut comporter une valeur fractionnelle non nulle (les mathématiciens appellent ces chiffres des *nombres réels*). La plupart des limitations des nombres entiers ne s'appliquent pas aux nombres décimaux. Remarquez qu'un nombre décimal "peut" avoir une partie fractionnelle non nulle. Ce n'est donc pas une obligation : en C++, 1.0 est un nombre décimal au même titre que 1.5 ; l'équivalent en nombre entier est simplement exprimé sous la forme 1. Les nombres décimaux négatifs sont également autorisés, comme dans -2.3.

En C++, les nombres décimaux sont des "nombres en virgule flottante" (quoi que *point flottant* serait plus approprié), ou tout simplement des *flottants*. Le terme "point flottant" provient du point utilisé, dans certains pays, pour séparer la partie entière d'un chiffre de sa partie fractionnaire. Ces variables sont déclarées de la même manière que les entiers :

```
double fValeur1;
```

Il existe un autre type comparable appelé `float`. Comme il est (relativement) moins précis que le type `double` et que votre ordinateur est certainement assez puissant pour faire des calculs avec les doubles, autant s'en tenir là.

La variable `fValeur1` est ici déclarée comme `double` (nombre décimal exprimé sur 64 bits). Une fois déclarée, une variable ne peut plus changer de type ; `fValeur1` est désormais une variable `double`, et ceci jusqu'à la fin du programme, comme dans toutes les instructions où elle est utilisée. Pour savoir comment un nombre à point flottant vient à bout du problème de troncature inhérent aux entiers, nous allons convertir les variables `int` en variables `double` :

```
double dValeur;
dValeur = 1/3 + 2/3 + 2/3
```

est l'équivalent de :

```
dValeur = 0,333... + 0,666... + 0,666...
```

soit :

```
dValeur = 1.666...;
```

Vous remarquerez les points qui suivent le chiffre 6. Ils indiquent que la séquence de chiffres est en théorie illimitée. Bien sûr, votre ordinateur n'est pas capable d'aller jusque là et il existe une limite à la précision du type `double`. Mais vous n'êtes pas prêt d'y arriver.

Limites de la virgule flottante

Bien que les variables de type `double` (à virgule ou point flottant) puissent résoudre bon nombre de problèmes tels que la troncature, elles n'en possèdent pas moins des limites. Ces inconvénients sont en quelque sorte à l'opposé de ceux des entiers. Ces variables ne peuvent pas être utilisées comme compteurs. Elles sont plus difficiles à gérer par l'ordinateur. Elles sont sujettes aussi à des erreurs d'arrondi (mais dans une mesure moindre que les variables `int`).

Compteurs

Vous ne pouvez pas utiliser de variables à point flottant dans des applications où le comptage est important. C'est notamment le cas des constructions C++ qui exigent des possibilités de dénombrement. Le langage C++ est en effet incapable de vérifier à quel nombre naturel correspond un nombre flottant.

Il est par exemple évident que 1,0 est égal à 1. Mais qu'en est-il de 0,9 ou de 1,1 ? Doivent-ils être considérés comme valant 1 ? C++ élude tout simplement ce problème en insistant sur le recours à des variables `int` lorsqu'il s'agit de compter.

Vitesse de calcul

Depuis toujours, un processeur traite les nombres entiers beaucoup plus vite que les nombres à virgule ou à point flottant. Alors qu'il additionnera un millier de nombres entiers dans une période de temps donnée, il n'effectuera dans le même temps que deux cents opérations à virgule ou point flottant.

Du fait des performances sans cesse croissantes des processeurs, la vitesse de calcul est devenue un problème secondaire. Les processeurs les plus récents sont équipés de circuits spéciaux consacrés uniquement aux calculs en virgule flottante, qui sont désormais presque aussi rapides que les calculs des entiers.

Perte d'exactitude

Les variables à point flottant ne résolvent pas tous les problèmes de calcul. Leur précision est limitée à six chiffres (une version particulière de ces variables, dite "à double précision", peut gérer jusqu'à quinze chiffres significatifs).

Pour vous rendre compte du problème, exprimons la fraction 1/3 sous la forme 0,333... à l'infini. Ce concept de série infinie se conçoit certes en mathématique, mais pas sur un ordinateur, car l'exactitude de ce dernier est finie : faites la moyenne de 1, 2 et 2, et vous obtiendrez 1,666667.

Le C++ est capable de corriger de nombreuses formes d'erreurs d'arrondi. Par exemple, en sortie, il saura déterminer si, plutôt que 0.999999, l'utilisateur n'attend pas 1. Mais dans d'autres cas, le C++ ne pourra rien faire.

Une plage pas si limitée

La plage numérique du type de données `double` est, elle aussi, limitée, bien sûr moins que celle des entiers. Alors que la valeur maximale d'une variable `int` est un peu au-dessus de deux milliards, celle d'une variable `double` est en gros de 10^{38}, soit 1 suivi de 38 zéros (imaginez en comparaison le déficit des finances publiques !).

Seuls les treize premiers chiffres sont significatifs car les autres subissent l'erreur d'arrondi due au point flottant.

Déclarer les types de variables

Nous avons vu que des variables doivent être déclarées et que leur type doit être précisé. Le langage C++ gère différents types de variables ; elles figurent dans le Tableau 2.1, avec leurs avantages et leurs inconvénients.

Tableau 2.1 : Les variables C++.

Variable	Exemple	Utilisation
`int`	1	Simple nombre naturel, dit "de comptage", positif ou négatif.
`unsigned int`	1U	Comme ci-dessus, mais l'entier est uniquement positif.
`long`	10L	Version potentiellement plus grande que `int`. Sous Dev-C++ et Visual C++ .NET, `long` et `int` sont équivalents.
`unsigned long`	10UL	Entier long positif.
`float`	1.0F	Nombre réel en simple précision. C'est un peu comme `double`, mais en moins précis et avec une plage de valeurs plus réduite.
`double`	1.0	Variable flottante standard. Ce type exige plus de mémoire que `float`, mais il est plus précis et sa plage numérique est plus vaste.
`char`	'C'	Une variable `char` stocke un seul caractère alphabétique ou numérique. Inutilisable en arithmétique.
`string`	"Ceci est une chaîne"	Chaîne de caractères formant un texte.
`bool`	true	Peut prendre comme valeur vrai (true) ou faux (false). Pas un faux nez ou un faux espoir. Juste une valeur logique totalement vraie ou totalement fausse.

Que choisir entre `float` et `double` ? N'oubliez pas que les ordinateurs modernes sont très rapides et puissants, et que la mémoire n'est plus un problème depuis des années. Le plus est donc le mieux. Oubliez `float`, soyez `double` !

L'instruction suivante déclare une variable `lVariable` comme type `long` et la met à 1, tandis que `dVariable` est une variable en double précision prenant comme valeur 1.0.

```
// Déclaration de la variable et mise à la valeur 1
long lVariable;
lVariable = 1;

// Déclaration de la variable de type double et mise à 1.0
double dVariable;
dVariable = 1.0;
```

Une variable peut être déclarée et initialisée dans une même instruction :

```
int nVariable = 1; // Déclaration de la variable et
                   // initialisation à 1
```

Le seul avantage d'initialiser une variable dans sa déclaration, c'est que l'on économise de la frappe. De telles déclarations sont courantes.

Une variable `char` ne peut contenir qu'un seul caractère alors qu'une chaîne en contient toute une succession. De ce fait, `a` est le caractère "a", mais `a` est aussi une chaîne de caractères ne contenant que la lettre "a" (une chaîne n'est pas véritablement un type de variable, mais la plupart du temps, vous pourrez la considérer comme telle). Le Chapitre 9 décrit les chaînes en détail.

Le caractère `a` et la chaîne `a` ne sont pas la même chose. Si une application exige une chaîne, vous ne devez pas déclarer un caractère, même si cette chaîne n'en contient qu'un seul.

Types de constantes

Une constante est un nombre explicite ou un caractère (comme 1, 0.5 ou "c") qui n'est pas destiné à changer. À l'instar des variables, les constantes ont un type. Dans une expression comme `n = 1;`, la constante `1` est de type `int`. Pour faire de `1` un entier `long`, l'instruction doit être écrite sous la forme `n = 1L`. L'analogie est la suivante : 1 représente une seule citrouille placée sur le

plateau d'une camionnette tandis que 1L est une citrouille chargée dans un semi-remorque. La cucurbitacée est la même, mais pas la taille du conteneur.

Poursuivons la comparaison entre *int* et *long*. 1.0 représente la valeur 1, mais dans un conteneur à point flottant. Remarquez cependant que, par défaut, une constante flottante est de type `double`. C'est pourquoi 1.0 est un nombre à `double` précision, et non de type `float`.

Ne confondez pas `true` et "true". La première est une constante de type `bool`. La seconde est une chaîne de caractères. Vous sentez la différence ? N'oubliez pas aussi que C++ différencie la casse des caractères : `true` a un sens, tandis que `TRUE` n'en a pas.

Caractères spéciaux

Quasiment n'importe quel caractère imprimable peut être stocké dans une variable `char` ou `string`. Il est aussi possible de stocker un ensemble de caractères non imprimables qui seront utilisés comme constantes de caractère. Ils sont décrits dans le Tableau 2.2.

Tableau 2.2 : Les caractères spéciaux.

Constante de caractère	Action
`\n`	Nouvelle ligne
`\t`	Tabulation
`\0`	Nul
`\\`	Barre inversée

Vous avez déjà rencontré le caractère de nouvelle ligne à la fin de chaînes. Il rompt une chaîne de caractères en lignes séparées. Une nouvelle ligne peut toutefois apparaître n'importe où dans une chaîne. Exemple :

```
"Ceci est la ligne 1\nCeci est la ligne 2"
```

apparaît en sortie sous la forme :

```
Ceci est la ligne 1
Ceci est la ligne 2
```

De même, le caractère de tabulation \t déplace la sortie à la prochaine tabulation. La position peut varier selon le type d'ordinateur. Du fait que le caractère "barre inversée" sert à signaler un caractère spécial, il en faut deux pour signaler la barre inversée elle-même : le caractère doublé \\ représente donc la barre inversée.

Collision de nom de fichiers

Windows (comme MS-DOS) utilise le caractère "barre inversée" pour séparer les noms des différents dossiers dans le chemin qui mène à un fichier. De ce fait, `Racine\DossierA\Fichier` indique que le fichier se trouve dans le `DossierA` qui est un sous-répertoire de `Racine`.

Malheureusement, la barre inversée du système d'exploitation entre en conflit avec celle qui, en C++, signale le caractère d'échappement. Le chemin `Racine\DossierA\Fichier` est représenté en C++ sous la forme `Racine\\DossierA\\Fichier`.

Les calculs sont-ils tous logiques ?

C++ fournit un type de variable appelé `bool`. Ce nom est dérivé de *booléen*, lui-même issu de Boole, l'inventeur du calcul logique. Une variable `bool` peut prendre deux valeurs : `true` ou `false`.

Certains calculs peuvent donner un résultat logique, c'est-à-dire booléen. Par exemple "x est égal à y" est soit vrai, soit faux.

Les modes d'expression mixtes

Le C++ permet de mêler les types de variable dans une seule expression. Autrement dit, il vous autorise à utiliser un entier avec une variable double précision. L'expression qui suit, où `nValeur1` est un entier `int`, est autorisée :

```
// Dans l'expression qui suit, la valeur de nValeur1
// est convertie en double précision avant d'exécuter
// l'instruction.
int nValeur1 = 1;
nValeur1 + 1.0;
```

Une expression dans laquelle les deux opérandes ne sont pas du même type est appelée *expression en mode mixte*. Les expressions en mode mixte génèrent une valeur dont le type est égal au plus approprié des deux opérandes. Dans ce cas de figure, `nValeur1` est converti en `double` avant le début du calcul. De même, une expression d'un type peut recevoir une variable d'un type différent, comme dans l'instruction qui suit :

```
// Dans l'instruction suivante, la partie entière
// du nombre contenu dans fVariable est stockée
// dans nVariable
double dVariable = 1.0;
int nVariable;
nVariable = dVariable;
```

Vous risquez de perdre de la précision ou de limiter la plage des valeurs si la variable située à gauche du signe d'affectation est inférieure à celle qui figure à droite. Dans l'exemple ci dessus, C++ tronquera la valeur de `fVariable` avant de la stocker dans `nVariable`.

La conversion d'une valeur plus grande vers un type où elle devient plus petite est parfois appelée *démotion*. L'inverse est la *promotion*. Un développeur dira que la variable `int` appelée `nVariable` est promue dans le type `double`. Par exemple :

```
int nVariable1 = 1;
double dVariable = nVariable1;
```

Les expressions en mode mixte sont à déconseiller. Évitez de demander à C++ de faire des conversions à votre place.

Les conventions de nom

Vous avez sans doute remarqué que le nom de chaque variable commence par un caractère spécial qui lui semble étranger. Leur signification figure dans le tableau ci-dessous. Grâce à eux, il est par exemple facile de reconnaître immédiatement que `dVariable` est du type `double` précision.

Caractère	Type
n	int
l	long
f	float
d	double
c	char
sz	string

Ces préfixes aident le programmeur à savoir de quel type est telle ou telle variable. Il reconnaîtra sans peine que, dans le mode d'affectation mixte qui suit, une variable est de type `int` et l'autre de type `long` :

```
nVariable = lVariable
```

Bien que, dans ce livre, le nom de certaines variables comporte des caractères spéciaux, ceux-ci n'ont aucune signification en C++. Si vous le préférez, rien ne vous empêche d'utiliser la lettre "e" pour un entier. Mais la convention qui figure dans le tableau est celle adoptée par de nombreux développeurs. Cela ne peut qu'aider à mieux se comprendre.

Chapitre 3

Effectuer des opérations mathématiques

Dans ce chapitre :

- Définir les opérateurs mathématiques en C++.
- Utiliser les opérateurs mathématiques C++.
- Identifier les expressions.
- Améliorer la clarté grâce à des opérateurs spéciaux.

Un mathématicien utilise bien plus de variables que celles décrites dans le Chapitre 2. Il doit en faire quelque chose, les additionner, les soustraire, les multiplier, etc.

Le C++ est lui aussi doté d'opérations de base. Les programmes écrits en C++ sont capables de multiplier, d'ajouter, de diviser, etc. Ils effectuent ces opérations dans un ordre donné. Car, à quoi pourrait bien servir un logiciel d'assurance s'il est incapable de calculer la prime que vous devez (sur)payer ?

Les opérations C++ ressemblent à celles que vous feriez avec un crayon et du papier, sauf qu'en l'occurrence, les variables doivent être déclarées avant de pouvoir être utilisées, comme nous le disions dans le Chapitre 2 :

```
int var1;
int var2 = 1;
var1 = 2 * var2;
```

Deux variables, `var1` et `var2`, sont déclarées ; `var2` est initialisé à 1 et `var1` reçoit la valeur résultant de l'opération "deux fois la valeur de `var2`".

Ce chapitre décrit tous les opérateurs mathématiques du C++.

Notions d'arithmétique binaire

Un opérateur binaire a deux arguments. Si vous dites `var1 op var2`, "op" doit être un opérateur binaire. Les opérateurs binaires les plus courants sont les opérations élémentaires que vous avez apprises à l'école primaire. Ils sont recensés dans le Tableau 3.1.

Tableau 3.1 : Les opérateurs mathématiques classés par ordre de priorité.

Priorité	Opérateur	Action
1	+ (unaire)	Aucune action
1	- (unaire)	Retourne l'opposé de son argument
2	++ (unaire)	Incrémentation
2	-- (unaire)	Décrémentation
3	* (binaire)	Multiplication
3	/ (binaire)	Division
3	% (binaire)	Modulo
4	+ (binaire)	Addition
4	- (binaire)	Soustraction
5	= *= %= += -= (spécial)	Types d'affectation

La multiplication, la division, le modulo, l'addition et la soustraction sont des opérateurs utilisés pour effectuer des opérations mathématiques. En pratique, ils agissent exactement comme leurs équivalents arithmétiques :

```
double var = 133 / 12;
```

Chacun de ces opérateurs binaires a la même signification que ceux que vous aviez étudiés en classe à une exception près : vous n'avez peut-être pas rencontré le modulo lors de vos études.

L'opérateur modulo (%) correspond au reste d'une division. Par exemple, on peut placer 3 fois la valeur 4 unités dans 15, et il reste 3 unités. En langage C++, nous disons que 15 modulo 4 donne 3.

```
int var = 15 % 4; // var est initialisé à 3
```

Comme les développeurs adorent en jeter plein la vue aux non-programmeurs, même avec les choses les plus simples, ils définissent le modulo comme suit :

```
IntValeur % IntDiviseur
```

Ce qui est l'équivalent de :

```
IntValeur - (IntValeur / IntDiviseur) * IntDiviseur
```

Essayez cet exemple :

```
15 % 4 est égal à 15 - (15/4) * 4
                  15 - 3 * 4
                  15   12
                  3
```

Le modulo n'est pas défini pour les variables flottantes du fait de l'erreur d'arrondi inhérente aux entiers (les erreurs d'arrondi ont été expliquées dans le Chapitre 2).

Décomposer des expressions

L'expression est le type d'instruction le plus courant en C++. Une *expression* est une instruction C++ contenant une valeur. Toutes les expressions ont un type tel que `int`, `double`, `char`, etc. Une instruction impliquant un quelconque opérateur mathématique est une expression car tous ces opérateurs retournent une valeur. Par exemple, 1 + 2 est une expression dont la valeur est 3 et le type `int` (rappelez-vous que les constantes dépourvues de point décimal sont des entiers).

Les expressions peuvent être complexes ou extrêmement simples. En fait, l'instruction 1 est une expression car elle a une valeur (1) et un type (int). L'instruction qui suit contient cinq expressions :

```
z = x * y + w;
```

Les expressions sont :

```
x * y + w
x * y
x
y
w
```

La particularité du C++, c'est qu'une expression est une instruction à part entière. De ce fait, il est admis d'écrire l'instruction C++ suivante :

```
1;
```

Toutes les expressions ont un type. Le type de l'expression `1` est `int`.

Déterminer l'ordre des opérations

Tous les opérateurs effectuent une fonction bien précise. Qui plus est, les opérateurs ont une *priorité*. La priorité des opérations détermine l'ordre dans lequel elles sont évaluées. Regardons de près l'instruction suivante :

```
int var = 2 * 3 + 1;
```

Si l'addition est effectuée avant la multiplication, la valeur de l'expression est 2 fois 4, soit 8. Si la multiplication est effectuée d'abord, la valeur est 6 plus 1, soit 7.

La priorité des opérateurs détermine l'ordre des calculs. Dans le Tableau 3.1, la multiplication est prioritaire sur l'addition, de sorte que le résultat est 7 (le concept de priorité existe d'ailleurs en arithmétique ; le C++ adhère complètement à cette vision des choses).

Bien. Mais que se passe-t-il lorsque nous utilisons deux opérateurs de même priorité dans une même expression ?

```
int var = 8 / 4 / 2;
```

Dans cette expression, 8 est-il divisé par 2 ou par 4, ou 2 est-il divisé par 2 ou par 1 ? Quand des opérateurs de même priorité apparaissent dans une expression donnée, ils sont évalués de gauche à droite, comme c'est d'ailleurs la règle en arithmétique. De ce fait, 8 est divisé par 4, qui est divisé par 2, et le résultat donne 1.

L'expression :

```
x / 100 + 32
```

divise x par 100 avant d'ajouter 32. Mais comment fera le programmeur s'il veut diviser x par la somme de 100 et de 32 ? Il peut mettre les opérations entre parenthèses comme dans :

```
x/(100 + 32)
```

Le résultat est le même qu'une division de x par 132.

L'expression originale :

```
x / 100 + 32
```

est équivalente à :

```
(x/100) + 32
```

Pourquoi le C++ a-t-il regroupé les expressions de la sorte ? Dans une expression donnée, le C++ effectue la multiplication et la division avant l'addition ou la soustraction. La priorité de la multiplication et de la division est en effet supérieure à celle de l'addition et de la soustraction.

En résumé : la priorité définit l'ordre dans lequel les opérateurs sont évalués. L'opérateur dont la priorité est la plus élevée est exécuté en premier. Il est possible d'outrepasser la priorité d'un opérateur en utilisant des parenthèses.

Exécuter des opérations unaires

Les opérateurs arithmétiques binaires, qui utilisent deux arguments, vous sont familiers. Vous avez sans doute effectué des opérations binaires dès le cours préparatoire. Les opérateurs *unaires*, eux, n'utilisent qu'un seul argument, `-a` par exemple. Nombre de ces opérations ne sont pas si connues.

Les opérateurs mathématiques unaires sont `+`, `-`, `++` et `--`. Par exemple :

```
int var1 = 10;
int var2 = -var1;
```

Cette dernière expression utilise l'opérateur unaire - pour calculer l'opposé de 10.

L'opérateur "moins" modifie le signe de son argument. Les nombres positifs deviennent négatifs et inversement. L'opérateur "plus" ne modifie pas le signe de son argument. À vrai dire, l'opérateur "plus" n'a aucun effet.

Les opérateurs `++` et `--` sont peut-être nouveaux pour vous : ils incrémentent et décrémentent respectivement leur argument d'une unité entière. Ces opérations ne concernent que les variables possédant un type entier. Après exécution, la valeur de `var`, dans l'expression qui suit, est 11.

```
int var = 10; // initialisation de var
var++;        // incrémentation
              // la valeur de var est maintenant 11
```

Les opérateurs d'incrémentation et de décrémentation sont un peu bizarres dans la mesure où ils existent sous deux formes : une version avec préfixe et une version avec suffixe. Attardons-nous sur l'opérateur d'incrémentation (la décrémentation fonctionne selon le même principe).

Supposons que la valeur de la variable `n` est 5 ; `++n` et `n++` incrémentent tous les deux `n` qui devient 6. La différence entre les deux, c'est que dans une expression, la valeur de `++n` est 6 alors que celle de `n++` est 5, ce que démontre l'exemple suivant :

```
// Déclaration de trois variables entières
int n1, n2, n3;

// La valeur de n1 et n2 est 6
```

```
n1 = 5;
n2 = ++n1;

// La valeur de n1 est 6, mais celle de n3 est 5
n1 = 5;
n3 = n1++;
```

De ce fait, `n2` reçoit la valeur de `n1` *après* que `n1` a été incrémenté à l'aide de l'opérateur de préincrémentation, alors que `n3` reçoit la valeur de `n1` *avant* que ce dernier n'ait été modifié à l'aide de l'opérateur de postincrémentation.

Pourquoi définir un opérateur d'incrémentation distinct ?

Les auteurs du C++ ont remarqué que les programmeurs ajoutent souvent 1, plus souvent que toute autre constante. Par convenance, une instruction spécialement chargée d'ajouter 1 a été intégrée au langage.

Qui plus est, la plupart des processeurs sont dotés d'une instruction d'incrémentation qui est bien plus rapide que l'instruction logicielle. Quand le C++ a été créé, les ordinateurs étant ce qu'ils étaient, économiser quelques instructions était crucial.

Utiliser les opérateurs d'affectation

Les opérateurs d'affectation sont des opérateurs binaires qui modifient la valeur de l'argument placé à gauche. Le simple opérateur d'affectation = est incontournable dans n'importe quel langage de programmation. Il transfère la valeur qui se trouve à droite dans l'argument situé à gauche. Les autres opérateurs d'affectation sont un peu étranges. Caprice de mathématicien ?

Les auteurs du langage C++ ont remarqué que les affectations prennent souvent la forme :

```
variable = variable # constante
```

où # est un opérateur binaire quelconque. De ce fait, pour incrémenter un opérateur entier de deux, un programmeur écrira :

```
nVariable = nVariable + 2;
```

Il indique que "2 doit être ajouté à `nVariable` et le résultat de nouveau stocké dans `nVariable`".

Il est très courant de voir une même variable figurer de part et d'autre d'une affectation.

Comme la même variable apparaît de part et d'autre du signe =, les auteurs du C++ ont décidé de créer une version de l'affectation dans laquelle un opérateur binaire est attaché au signe d'égalité. Ce qui revient à demander directement d'effectuer un certain calcul sur une variable, puis d'affecter le résultat à cette même variable.

Tous les opérateurs binaires possèdent une telle forme d'affectation. Par conséquent, l'affectation ci-dessus aurait pu être écrite :

```
nVariable += 2;
```

Là encore, "2 est ajouté à la valeur de `nVariable`". La seconde variante est moins habituelle, mais reconnaissez qu'elle fait gagner du temps.

Outre l'affectation elle-même, ces styles d'opérateurs ne sont pas très souvent utilisés. Mais dans certains cas, ils facilitent la lisibilité du programme.

Chapitre 4

Effectuer des opérations logiques

Dans ce chapitre :

- Quelques opérateurs logiques parfois illogiques.
- Définir les variables logiques.
- Utiliser des opérateurs de bits.

L'expression est l'instruction C++ la plus courante. La plupart impliquent des opérateurs arithmétiques comme l'addition (+), la soustraction () et la multiplication (*). Le chapitre précédent décrit ces types d'expressions.

Il existe une toute autre classe, celle des *opérateurs logiques*. Ils sont nettement moins connus des foules que les opérateurs arithmétiques. Ce n'est pas faute de les utiliser car en fait, les gens pensent constamment sous la forme ET et OU. Par exemple, je bois mon café avec du sucre ET du lait (beaucoup de lait). Et j'aime bien parfois une petite vodka SI elle est à l'herbe de bison ET que quelqu'un d'autre paie la tournée. Les gens utilisent tout le temps des opérateurs logiques, mais sans en avoir vraiment conscience.

Il existe deux types d'opérateurs logiques. ET et OU, par exemple, sont ce que l'on pourrait appeler des opérateurs logiques simples. L'autre type, que l'on peut dire "de bits", est propre à l'informatique. Son rôle est d'examiner chacun des bits qui, dans la mémoire de l'ordinateur, servent à représenter un nombre.

Pourquoi utiliser des opérations logiques ?

Si j'ai réussi à vivre jusqu'à ce jour sans me soucier des opérateurs logiques, pourquoi devrais-je m'y mettre maintenant ? Le C++ a des décisions à prendre. Un programme qui ne saurait en prendre n'irait pas très loin. Le programme `Conversion` (voir Chapitre 1) est l'un des plus complexes que l'on puisse élaborer sans y intégrer de prises de décision du type : "Fais ceci si la variable entrée est négative, mais fais cela si la variable entrée est positive". C'est exactement le genre de choses qui fait croire que les ordinateurs sont intelligents (ou à l'inverse particulièrement stupides si le programme prend la mauvaise décision).

En résumé, prendre des décisions exige des opérateurs logiques.

Les opérateurs logiques simples

Les opérateurs logiques simples, énumérés dans le Tableau 4.1, évaluent un résultat sous la forme Vrai (`true`) ou Faux (`false`).

En programmation, les opérateurs logiques reposent sur un vocabulaire anglais dont voici la traduction :

`TRUE`	Vrai	`AND`	Et	`NOT`	Pas (ou Non)
`FALSE`	Faux	`OR`	Ou	`XOR`	Ou exclusif

Tableau 4.1 : Les opérateurs simples de la logique quotidienne.

Opérateur	Signification
`==`	Égalité. Vrai si la valeur de l'argument placé à gauche est la même que celle de l'argument de droite.
`!=`	Inégalité. Contraire de l'égalité.
`>` `<`	Plus grand que ; plus petit que. Vrai si la valeur de l'argument placé à gauche est plus grande ou plus petite que celle de l'argument de droite.
`>=` `<=`	Plus grand ou égal à ; plus petit ou égal à. Vrai si > ou == est vrai (ou si < ou == est vrai).
`&&`	Et logique (AND). Vrai si les arguments sont tous deux vrais.

Tableau 4.1 : Les opérateurs simples de la logique quotidienne. (*suite*)

Opérateur	Signification
||	Ou (OR). Vrai si l'un des deux arguments est vrai.
!	Pas (NOT). Vrai si son argument est faux.

Les six premières entrées du Tableau 4.1 sont des opérateurs de comparaison. L'opérateur d'égalité sert à comparer deux nombres. Par exemple, ce qui suit est vrai si la valeur de n est 0, et faux dans le cas contraire :

```
n == 0;
```

Ne confondez pas l'opérateur d'égalité == avec l'opérateur d'affectation =. C'est non seulement une erreur commune mais aussi, et surtout, une erreur que le compilateur C++ ne saura pas détecter.

```
n = 0; // Le programmeur aurait dû écrire n == 0
```

Les opérateurs plus grand que (>) et plus petit que (<) sont eux aussi très courants dans la vie quotidienne. La comparaison logique ci-dessous est vraie :

```
int n1 = 1;
int n2 = 2;
n1 < n2
```

Il est facile d'oublier ce qui est plus grand et ce qui est plus petit. Rappelez-vous simplement que l'opérateur est vrai lorsque la flèche pointe vers le plus petit des deux termes.

Vous pourriez vous demander si n1 est plus grand ou plus petit que n2 en omettant la possibilité que n1 et n2 puissent être égaux. Les opérateurs plus grand ou égal à (<=) et plus petit ou égal à (>=) sont équivalents aux opérateurs plus grand que, et plus petit que, sauf que, contrairement à ces derniers, ils tiennent compte de l'égalité.

Les opérateurs & (et, AND) et || (ou, OR) sont également très courants. Ils sont généralement associés à d'autres opérateurs logiques comme dans :

```
// Vrai si n2 est plus grand que n1 mais plus petit que n3
// et donc si n2 appartient à l'intervalle ]n1;n3[
(n1 < n2)& & (n2 < n3);
```

Soit dit en passant, vous pouvez aussi définir l'opérateur plus grand ou égal de la manière suivante :

```
n1 <= n2 est identique à (n1 < n2) || (n1 == n2)
```

Mémoriser des valeurs logiques

Le résultat d'une opération logique peut être affecté à une variable de type `bool` :

```
int n1 = 1;
int n2 = 2;
bool b;
b = (n1 == n2);
```

Cet exemple illustre bien la différence entre affectation (=) et comparaison (==). La dernière ligne se lit : "Comparer les variables `n1` et `n2`. Enregistrer le résultat de cette comparaison dans la variable `b`)".

Dans l'arbre des priorités, les opérateurs d'affectation se situent en bas, du côté des racines. La comparaison logique est évaluée avant l'affectation, ce qui signifie que les parenthèses ne sont pas nécessaires. Elles sont cependant conseillées pour éviter toute confusion, comme dans :

```
b = n1 == n2; // compare n1 avec n2; retourne true si
              // n1 a la même valeur que n2, false sinon
              // place le résultat, true or false, dans b
```

Le programme suivant (`Booltest.cpp`) illustre l'usage d'une variable booléenne.

```
// BoolTest - compare les variables entrées au
//            clavier et place le résultat dans
//            une variable logique.
#include <cstdio>
```

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

int main(int nNumberofArgs, char* pszArgs[])
{
    // définit le format de sortie pour les
    // variables logiques sous la forme true/false
    // au lieu de 1 et 0
    cout.setf(cout.boolalpha);

    // initialise deux arguments
    int nArg1;
    cout << "Entrez la valeur 1 : ";
    cin >> nArg1;

    int nArg2;
    cout << "Entrez la valeur 2 : ";
    cin >> nArg2;

    bool b;
    b = nArg1 == nArg2;

    cout << "L'instruction, " << nArg1
         << " = "             << nArg2
         << " vaut "          << b
         << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La première ligne (`cout.setf()`) s'assure que notre variable booléenne sera affichée sous la forme `"true"` ou `"false"`. Nous allons voir dans la section suivante pourquoi c'est nécessaire.

Le programme demande deux valeurs saisies au clavier, puis il affiche le résultat de leur comparaison :

```
Entrez la valeur 1 : 32
Entrez la valeur 2 : 32
L'instruction, 32 = 32 vaut true
Appuyez sur une touche pour continuer...
```

La valeur spéciale `endl` insère un changement de ligne. La différence avec le caractère réservé `\n` décrit au Chapitre 2 est assez subtile. Nous y reviendrons dans le Chapitre 24.

Utiliser des variables logiques entières

Le langage C++ n'a pas toujours offert un type de variable logique (`bool`). Dans sa préhistoire, il se servait d'entiers pour stocker ces valeurs. Un zéro voulait dire `false` et tout autre nombre représentait `true`. Plus précisément, un opérateur logique générait 0 pour dire `false` et 1 pour `true`. Conséquence "logique" : l'expression "0 est faux" était... vraie.

Pour rester compatible avec tous les anciens programmes, C++ a conservé ce lien étroit entre les types `bool` et `int`. Si vous supprimez la ligne :

```
cout.setf(cout.boolalpha);
```

dans le programme `Booltest`, la réponse sera tout à fait différente :

```
Entrez la valeur 1 : 32
Entrez la valeur 2 : 32
L'instruction, 32 = 32 vaut 1
Appuyez sur une touche pour continuer...
```

Les variables entières et booléennes peuvent parfaitement être mélangées dans des expressions. Par exemple, Dev-C++ ne bronchera pas s'il rencontre l'étrange séquence qui suit :

```
int n;
n = nArg1 == nArg2;
```

Cette scorie que les compilateurs C++ modernes ont héritée de leurs ancêtres est une source de confusion. Il vaut mieux s'en tenir au type `bool`. D'autres compilateurs pourraient être moins conciliants.

Attention aux opérations logiques sur les variables flottantes

Les nombres réels se caractérisent par une partie décimale. C'est pourquoi ils ne peuvent pas servir à effectuer des comptages. Vous pouvez parler de premier (1er), de deuxième (2e), de troisième, quatrième, etc. car les relations entre 1, 2 et 3 sont connues. Mais il est illogique de parler de 4,5e dans une suite (on imagine aisément qu'il se place entre le quatrième et le cinquième, mais c'est quand même aberrant).

Cette remarque vaut pour les types flottants du C++. La situation est même pire car, contrairement aux nombres réels, les valeurs flottantes comportent une quantité limitée de chiffres après le point décimal. C'est pourquoi vous devez être très prudent lorsque vous utilisez des opérateurs de comparaison avec des nombres en virgule flottante. Prenons le cas de figure suivant :

```
float f1 = 10.0;
float f2 = f1 / 3;
f1 == (f2 * 3.0);  // Ces deux-là sont-ils égaux ?
```

Dans cet exemple, la comparaison n'est pas *nécessairement* vraie. Une variable à point flottant ne peut pas avoir un nombre illimité de chiffres significatifs. De ce fait, `f2` n'est pas égal à 3 et un tiers, mais à 3.33333. Contrairement au concept mathématique, la quantité de trois après le point décimal est finie. Après avoir multiplié 3.3333 par 3, vous obtiendrez plutôt 9.9999 que 10. Cette petite différence peut passer inaperçue aux yeux de quelqu'un, mais pas pour l'ordinateur. En informatique, l'égalité implique une pure égalité, parfaitement exacte.

Une variable de type `float` supporte jusqu'à six chiffres significatifs (environ). Le type `double` porte cette précision à treize chiffres significatifs. Je précise *environ*, car les aléas des calculs en virgule flottante peuvent produire des nombres du genre 3.3333347.

Les processeurs modernes, très sophistiqués, se tirent fort bien de ces calculs. Ils s'accommodent des erreurs d'arrondi, mais il est impossible de prévoir comment ils procéderont à partir des instructions écrites en C++.

Des problèmes peuvent surgir même dans un calcul aussi simple que celui-ci :

```
float f1 = 10.0;
float f2 = 100 % 30;
f1 == f2;          // Ces deux-là sont-ils égaux ?
```

Théoriquement, `f1` et `f2` devraient être égaux (reportez-vous au Chapitre 3 si vous avez oublié ce qu'est un modulo). Aucun problème d'arrondi ne semble montrer le bout de son nez. Mais vous ne pouvez pas en être certain car vous n'avez aucune idée de la manière dont l'ordinateur se représente les nombres flottants en interne. Affirmer platement que 100 % 30 n'entraîne aucune erreur d'arrondi, c'est se faire des idées sur ce qui se passe dans l'unité centrale de la machine.

Voici une comparaison plus sûre :

```
float f1 = 10.0;
float f2 = f1 / 3;
float f3 = f2 * 3.0;
(f1 - f3) < 0.0001 && (f3 - f1) < 0.0001;
```

Cette comparaison est vraie si `f1` et `f3` ne diffèrent que d'un *delta*, ce qui devrait être vrai même en tenant compte d'une légère erreur d'arrondi.

Courts-circuits et C++

Les opérateurs `&&` et `||` effectuent ce que l'on appelle une *évaluation en court-circuit*. Étudions cette instruction :

```
condition1 && condition2
```

Si `condition1` n'est pas vrai, le résultat ne l'est pas non plus quelle que soit la valeur de `condition2` (donc `condition2` pourrait être vrai ou faux sans que le résultat change). Examinons de même l'instruction suivante :

```
condition1 || condition2
```

Si `condition1` est vrai, le résultat l'est aussi quelle que soit la valeur de `condition2`.

Pour gagner du temps, C++ (dans sa grande sagesse) évalue d'abord `condition1`. Il n'évalue pas `condition2` si `condition1` est faux avec `&&`, ou si `condition1` est vrai avec `||`.

L'expression des nombres binaires

Les variables C++ sont stockées en interne sous la forme de nombres dit "binaires". Les nombres binaires sont des successions de 0 et de 1 appelés "bit". La plupart du temps, vous n'avez pas à vous soucier des bits qui servent à représenter les nombres ; mais dans certains cas, il est commode de savoir les manipuler. Le C++ propose à cette fin tout un ensemble d'opérateurs.

Comme vous n'aurez pas souvent l'occasion de travailler au niveau des bits avec les variables C++, considérez que tout ce qui suit est de la littérature technique sur laquelle vous pourrez revenir ultérieurement.

Les opérateurs de bits agissent sur chacun des bits d'un argument. Pour bien comprendre cela, commençons par étudier comment un ordinateur stocke les variables.

Le système de numération décimal

Les nombres qui nous sont familiers sont des *nombres décimaux* car ils sont basés sur le chiffre 10. Les programmeurs expriment généralement les variables C++ en nombres décimaux. C'est pourquoi vous direz par exemple que la valeur de `var` est 123. Mais cela mérite d'être approfondi.

Un nombre comme 123 découle de l'expression `1 * 100 + 2 * 10 + 3 * 1`. Chacun de ces nombres de base – 100, 10 et 1 – sont des puissances de 10.

```
123 = 1*100 + 2*10 + 3*1
```

Exprimé d'une façon légèrement différente, nous obtenons :

$$123 = 1*10^2 + 2*10^1 + 3*10^0$$

Rappelez-vous que *tout* nombre mis à la puissance 0 donne 1.

Les autres systèmes de numération

L'adoption de la base 10 pour notre manière de compter provient sans doute du fait que la main de l'être humain a dix doigts, et qu'elle fut le premier outil pour compter (mais la discussion continue). Une alternative aurait été la base 20 (N.D.T. : c'est elle qui prévaut en fait dans le nom de certains nombres en

français, comme quatre-vingts – *quatre fois vingt* –. Cette façon d'exprimer les nombres remonterait à l'époque gauloise, où nos ancêtres comptaient effectivement sur tous les doigts, même ceux des pieds sans doute).

Si notre système de numération avait été inventé par les chiens, nous compterions en base 8, en tenant compte du fait que l'un des doigts d'une patte, placé en arrière, est hors de vue. Ce système octal fonctionne parfaitement :

$$123_{10} = 1*8^2 + 7*8^1 + 3*8^0 = 173_8$$

Les indices 10 et 8 indiquent la base de numération, 10 pour le système décimal, 8 pour le système octal. Un système de numération peut reposer sur n'importe quelle base positive.

Le système de numération binaire

Les ordinateurs comptent sur deux doigts (voilà pourquoi ils sont si bêtes). Autrement dit, ils calculent en base 2. Le nombre 123 sera donc exprimé sous la forme :

$$123_{10} = 0*128 + 1*64 + 1*32 + 1*16 + 1*8 + 0*4 + 1*2 + 1*1 = 01111011_2$$

C'est toujours par convention que les chiffres binaires sont exprimés par paquets de 4, 8, 32 ou 64 nombres binaires, même si les premiers chiffres sont des zéros. Cette convention découle de la manière dont les ordinateurs manipulent ces chiffres en interne.

Le terme *bit* est la contraction de l'anglais *binary digit*, ou chiffre binaire. Huit bits forment un *octet*. Un *mot* est formé de deux ou quatre octets selon qu'il est court ou long.

Avec une base aussi restreinte, il faut beaucoup de chiffres pour exprimer un nombre. Il peut paraître excessif de recourir à une expression comme 01111011, en base 2, au lieu d'un banal 123 en base 10. Quant aux programmeurs, ils préfèrent exprimer les nombres par octets, c'est-à-dire par ensemble de huit bits.

Le système de numération hexadécimal

Les informaticiens utilisent un système de numération basé sur le chiffre 16 : c'est le système *hexadécimal*. Il utilise les chiffres de 0 à 9, puis les six premières lettres de l'alphabet : A pour 10, B pour 11, etc. Un chiffre consomme donc 4 bits. Il en découle que 123 devient 7B en hexadécimal :

$123_{10} = 7*16^1 + B(\text{soit } 11)*16^0 = 7B_{16}$

Comme les programmeurs préfèrent exprimer les nombres sur 4, 8, 32 ou 64 bits, ils aiment aussi écrire les nombres hexadécimaux sur 1, 2, 4 ou 8 chiffres hexadécimaux, même s'ils sont précédés de zéros.

Il n'est finalement pas commode d'exprimer un nombre hexadécimal comme $7B_{16}$ en le faisant suivre d'un indice indiquant la base utilisée, car les terminaux ne supportent pas cette expression. Même dans un traitement de texte, mettre des indices à tout bout de champ n'est guère pratique. C'est pourquoi les informaticiens utilisent une autre convention pour signaler la base d'un nombre. Ils font précéder un nombre hexadécimal par 0x (cet étrange préfixe provient tout droit de l'époque héroïque du langage C).

De ce fait, `7B` devient `0x7B`. Selon cette convention, 0x7B est égal à 123 alors que 0x123 est égal à 291. Quand vous aurez bien compris cela, vous ferez fureur dans les soirées branchées.

Tous les opérateurs mathématiques utilisés avec les valeurs décimales sont applicables de la même manière aux nombres hexadécimaux. L'impossibilité de multiplier mentalement 0xC par 0xE réside dans nos neurones, qui sont plutôt accoutumés aux tables de multiplication apprises à l'école. L'ordinateur, lui, s'en tire fort bien.

Les opérations logiques au niveau du bit

Tous les nombres du C++ peuvent être exprimés sous forme binaire. Un nombre binaire n'est représenté que par des 0 et des 1. Le Tableau 4.2 présente les opérateurs qui n'agissent sur les nombres que bit par bit, d'où le terme "opérateur de bits".

Les opérations de bits peuvent potentiellement stocker une grande quantité d'informations dans peu de mémoire. Beaucoup d'éléments, dans le monde, se caractérisent par deux états ou n'offrent que deux possibilités : c'est comme

Les chiffres romains

Il est intéressant de s'attarder sur des systèmes de numération qui ont ralenti les progrès du calcul. C'est le cas des chiffres romains qui ont considérablement retardé le développement des mathématiques.

L'addition de deux chiffres romains n'est pas trop difficile :

XIX + XXVI = XLV

L'opération s'effectue ainsi :

a) Dans IX + VI, le I après le V annule le I avant le X, ce qui donne V et X comme retenue.

b) Dans X + XXX, la retenue X donne XXXX, ce qui se traduit par XL.

La soustraction est légèrement plus difficile.

En revanche, pour multiplier des chiffres romains, il faut vraiment être très doué en maths. Vous vous retrouvez avec des règles qui stipulent que X décale le chiffre d'une lettre vers la droite, de sorte que XIV devient XL. Pour la division, il faut sortir de Polytechnique ; quant aux opérations plus sophistiquées, comme l'intégration, elles sont complètement impossibles.

Tableau 4.2 : Les opérateurs de bits.

Opérateurs	Fonction
~	NOT : met chaque bit 0 à 1 et chaque bit 1 à 0.
&	AND : chaque bit de l'argument de gauche est évalué avec celui de droite.
|	OR : selon le même principe que AND.
^	XOR : (ou exclusif) selon le même principe que AND.

ceci ou c'est comme cela. Vous êtes homme ou femme (du moins d'après l'état civil), vous êtes marié ou vous ne l'êtes pas (même si vous êtes divorcé, vous n'êtes actuellement plus marié). En C++, chacune de ces caractéristiques peut être stockée dans un seul bit (que ferait un honnête homme de plus de bits ?). Vous pouvez stocker trente-deux propriétés distinctes dans un seul entier de type `int`.

Qui plus est, les opérations sur les bits sont extrêmement rapides. Pas de retard, pas de pénalité, pas de souci.

Même si la mémoire est actuellement bon marché, elle n'est tout de même pas illimitée. Lorsque vous avez à traiter de grandes quantités de données, pouvoir placer tout un ensemble de propriétés dans un seul mot reste un avantage important.

Les opérateurs à un bit

Les opérateurs de bits `AND` (`&`), `OR` (`|`) et `NOT` (`~`) effectuent des opérations logiques sur un seul bit. Si 0 est faux et si 1 est vrai (ce n'est pas forcément le cas, mais cette convention est acceptée par tous les informaticiens), vous pouvez admettre ce qui suit pour l'opérateur `NOT` :

```
NOT 1 (vrai) est 0 (faux)
NOT 0 (faux) est 1 (vrai)
```

Dans la même logique, l'opérateur `AND` est défini comme suit :

```
1 (vrai) AND 1 (vrai) est 1 (vrai)
1 (vrai) AND 0 (faux) est 0 (faux)
```

C'est pareil pour l'opérateur `OR` :

```
1 (vrai) OR 0 (faux) est 1 (vrai)
0 (faux) OR 0 (faux) est 0 (faux)
```

La définition de la table de vérité de l'opérateur `AND` apparaît dans le Tableau 4.3 ci-dessous.

Tableau 4.3 : La table de vérité de l'opérateur AND.

AND	1	0
1	1	0
0	0	0

La valeur d'un argument correspond à une colonne et celle de l'autre argument est proposée sur une ligne. Prenez par exemple la colonne "1" et la ligne "0". L'expression `1 & 0` retourne donc 0 (vrai *et* faux donne faux). La seule combinaison renvoyant autre chose que 0 est `1 & 1`. C'est cela que l'on appelle *table de vérité*.

La table de vérité de l'opérateur `OR` est donnée dans le Tableau 4.4.

Tableau 4.4 : La table de vérité de l'opérateur OR.

OR	1	0
1	1	1
0	1	0

Il existe une autre opération logique, peu usitée dans la vie courante : l'opérateur "ou exclusif", communément appelé `XOR`. Il retourne vrai si l'un *et seulement un* des deux arguments est vrai. Le Tableau 4.5 montre la table de vérité de l'opérateur `XOR`.

Tableau 4.5 : La table de vérité de l'opérateur XOR.

XOR	1	0
1	0	1
0	1	0

Armés de ces opérateurs sur un bit, nous pourrons revenir aux opérations de bits du C++.

Utiliser les opérateurs de bits

Les opérateurs de bits agissent séparément sur chaque bit. Ils sont utilisés à peu près comme les autres opérateurs arithmétiques. L'opérateur `NOT` est le plus facile à comprendre. L'appliquer à un nombre équivaut à mettre chaque bit de ce nombre sur `NOT` (autrement dit, à l'inverser) :

```
~0110₂ (0x6)
 1001₂ (0x9)
```

En procédant ainsi, nous disons que `~0x6` est égal à `0x9`.

Le calcul ci-dessous montre l'effet de l'opérateur `&` :

```
  0110₂
&
  0011₂
  0010₂
```

En commençant par le bit le plus significatif (celui de gauche), `0 AND 0` donne 0. Sur le bit suivant, `1 AND 0` donne 0. Pour le troisième bit, `1 AND 1` donne 1. Avec le bit le moins significatif (celui de droite), `0 AND 1` donne 0.

Les mêmes calculs peuvent être réalisés en hexadécimal, en convertissant d'abord le nombre au format binaire, en effectuant l'opération, puis en convertissant le résultat en sens inverse :

```
 0x6          0110₂
&       --->  &
 0x3          0011₂
              0010₂  ---> 0x2
```

En bref, nous pouvons dire que `0x6 & 0x3` est égal à `0x2`.

Faites ce test : que donne `0x6 | 0x3` ? Résolvez ce problème et vous serez au septième ciel. Plantez-vous et vous serez sur la première marche qui mène aux Enfers. Il m'a fallu à peine moins de sept minutes pour venir à bout de cette colle.

Un petit test

Le programme qui suit est un exemple d'opérations sur les bits. Il initialise deux variables et sort le résultat selon le résultat fourni par les opérateurs `AND`, `OR` et `XOR`.

```
// BitTest - initialisation de deux variables et sortie des
//           résultats obtenus avec les opérateurs ~,& , | et ^
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
int main(int nNumberofArgs, char* pszArgs[])
{
    // met le format de sortie en hexadécimal
    cout.setf(cout.hex);

    // initialisation des deux arguments
    int nArg1;
    nArg1 = 0x1234;

    int nArg2;
    nArg2 = 0x00ff;

    // chaque opération est effectuée tour à tour
    // en commençant par l'opérateur unaire NOT
    cout << "Arg1          = 0x" << nArg1 << "\n";
    cout << "Arg2          = 0x" << nArg2 << "\n";
    cout << "~nArg1        = 0x" << ~nArg1 << "\n";
    cout << "~nArg2        = 0x" << ~nArg2 << "\n";

    // au tour des opérateurs binaires
    cout << "nArg1 & nArg2 = 0x"
         << (nArg1 & nArg2)
         << "\n";
    cout << "nArg1 | nArg2 = 0x"
         << (nArg1 | nArg2)
         << "\n";
    cout << "nArg1 ^ nArg2 = 0x"
         << (nArg1 ^ nArg2)
         << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La première instruction du programme, `cout.setf(cout.hex);`, remplace le format de sortie par défaut, qui est décimal, par une sortie en hexadécimal.

Cela ne marche pas ? Question de configuration, vraisemblablement. Dans ce cas, remplacez la ligne ci-dessus par `cout << hex;`. Voilà qui devrait donner satisfaction. Pour revenir à un affichage décimal, il vous suffit d'insérer l'instruction `cout << dec;` au bon endroit.

Le restant du programme n'est pas très compliqué. Il lit les arguments `nArg1` et `nArg2` entrés au clavier puis sort toutes les combinaisons possibles de calculs au niveau des bits.

L'exécution du programme sur les valeurs `0x1234` et `00xff` produit la sortie suivante :

```
Arg1          = 0x1234
Arg2          = 0xff
~nArg1        = 0xffffedcb
~nArg2        = 0xffffff00
nArg1 & nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
Appuyez sur une touche pour continuer...
```

Pourquoi définir des opérateurs ?

La raison d'être de la plupart des opérateurs est claire. Personne ne remettrait en cause l'existence des opérateurs "plus" ou "moins". L'utilisation des opérateurs "<" et ">" tombe sous le sens. Mais pour un débutant, savoir quand et comment utiliser les opérateurs de bits n'est pas si évident.

L'opérateur `AND` est souvent utilisé pour masquer une information. Supposons par exemple que vous vouliez extraire le chiffre hexadécimal le moins significatif d'un nombre à quatre chiffres :

```
 0x1234            0001  0010  0011  0100
&          --->&
 0x000F             0000  0000  0000  1111
                   0000  0000  0000  0100  --->  0x0004
```

L'opérateur `AND` sert aussi à définir et extraire tel ou tel bit.

Supposons que vous utilisiez un seul octet pour stocker, dans une base de données que vous êtes en train de construire, des informations concernant une personne. Le bit le plus significatif sera par exemple mis à 1 si cette personne est de sexe masculin, le suivant sera mis à 1 si sa profession est "programmeur", le suivant à 1 si la personne a une voiture, et un autre sera à 1 si elle possède un chien. Le Tableau 4.6 récapitule ces informations.

Tableau 4.6 : Affectation de bits.

Bit	Signification
0	1 = masculin
1	1 = programmeur
2	1 = possède une voiture
3	1 = propriétaire d'un chien

Ce quartet est encodé pour chaque élément de la base de données puis stocké avec le nom, le numéro de Sécurité sociale et toutes sortes d'autres informations plus ou moins légales.

Un homme – un vrai, programmeur de surcroît – qui n'a pas de voiture, mais un doux clébard, sera codé 1101 (en binaire). Si vous voulez tester tous les enregistrements à la recherche d'un programmeur, quel que soit son sexe, qu'il ou elle ait ou non un chien, vous utiliserez la comparaison suivante :

```
(valeurBaseDeDonnées & 0x0110) == 0x0100
                        *^^*       ^  ->   0 = pas de voiture
                                   ^ ->    1 = programmeur
                        * -> sans intérêt
                        ^ -> intéressant
```

Dans ce cas, la valeur 0110 est appelée "masque" car elle masque, ou filtre, les propriétés de bits sans intérêt.

Esprit logique, calculs logiques...

S'exercer à effectuer des calculs logiques sur des nombres ou des bits n'est pas si méchant (du moins pour certains d'entre nous). Mais utiliser ces valeurs d'une façon pratique, concrète et efficace dans un programme est un tout autre problème. Passons donc à des choses plus sérieuses : l'emploi d'opérateurs logiques pour contrôler le déroulement d'un programme.

Chapitre 5

Contrôler le déroulement d'un programme

Dans ce chapitre :

- Contrôler le cours du programme.
- Exécuter un groupe d'instructions répétitivement.
- Éviter les boucles infinies.

Les programmes élémentaires qui apparaissent dans les quatre premiers chapitres traitent un nombre défini d'entrées, sortent le résultat du calcul et se terminent. Il leur manque un moyen de contrôler le cours de l'exécution. Ils ne peuvent notamment procéder à aucun test. Quasiment tous les programmes informatiques doivent prendre des décisions : quand l'utilisateur appuie sur une touche, il faut bien que l'ordinateur réponde à la commande.

Par exemple, si l'utilisateur appuie sur les touches Ctrl+C, l'ordinateur copie la zone sélectionnée dans le Presse-papiers. Si l'utilisateur déplace la souris, le pointeur se déplace corrélativement à l'écran. S'il appuie sur le bouton droit de la souris en maintenant la touche Windows enfoncée, l'ordinateur se plante. Et ainsi de suite... Un programme qui ne prend aucune décision est pour le moins ennuyeux.

Les commandes de contrôle permettent au programme de décider de l'action à entreprendre selon les résultats d'opérations logiques exécutées en C++ (voir à ce sujet le Chapitre 4). Il existe fondamentalement trois types d'instructions de contrôle du cours d'un programme : les branchements, les boucles et les commutateurs.

Contrôler le cours du programme par des commandes de branchement

Le *branchement* est la forme de contrôle la plus simple. Cette instruction autorise le programme à décider lequel des deux chemins il doit prendre, en fonction du résultat produit par une expression logique (reportez-vous au Chapitre 4 pour en savoir plus sur les expressions logiques).

En C++, une instruction de branchement est implémentée par l'instruction `if` :

```
if (m > n)
{
   // Chemin 1
   // ... instruction à exécuter si
   // m est plus grand que n
}
else
{
   // Chemin 2
   // ... instruction à exécuter si non
}
```

L'expression logique `m > n` est d'abord évaluée. Si le résultat de l'expression est `true` (vrai), le contrôle est donné, dans ce petit programme, au `Chemin 1`. Si l'expression n'est pas vraie, le contrôle prend le `Chemin 2`. La clause `else` (autrement) est facultative. Si elle n'est pas présente, le C++ agit comme si elle existait, mais avec un contenu vide.

En fait, les accolades sont facultatives s'il n'y a qu'une instruction à exécuter pour `if`. Mais si vous les omettez, il est très facile de se tromper sans que le compilateur C++ s'en aperçoive. C'est pourquoi il est toujours plus sûr d'utiliser les accolades. Si vos amis ou collègues vous incitent à vous en passer, dites-leur simplement : "Non !".

Le programme ci-dessous démontre l'utilisation de l'instruction `if` (remarquez toutes ces belles accolades) :

```
// BranchDemo - Entrez deux nombres. Le programme choisit un chemin
//              si le premier argument est plus grand que le
//              deuxième, sinon il prend l'autre chemin
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // entrez le premier argument...
    int arg1;
    cout << "Entrez arg1 : ";
    cin  >> arg1;

    // ... puis le deuxième
    int arg2;
    cout << "Entrez arg2 : ";
    cin  >> arg2;

    // prise de décision :
    if (arg1 > arg2)
    {
        cout << "L'argument 1 est plus grand que l'argument 2"
             << endl;
    }
    else
    {
        cout << "L'argument 1 n'est pas plus grand que l'argument 2"
             << endl;
    }

  // attend pour terminer le programme que l'utilisateur
  // lise le contenu de la fenêtre puis appuie sur une touche
  system("PAUSE");
  return 0;
}
```

Ici, le programme lit deux entiers entrés au clavier puis il les compare. Si l'expression "l'argument 1 est plus grand que l'argument 2" est vraie, le cours du programme est dans ce cas dérouté vers l'instruction `cout << "L'argument 1 est plus grand que l'argument 2 "`. Si `arg1` est inférieur ou égal à `arg2`, le contrôle applique la clause `else` qui déroute l'exécution du programme vers `cout << "L'argument 1 n'est pas plus grand que l'argument 2"`. Exemple :

```
Entrez arg1 : 4
Entrez arg2 : 6
L'argument 1 n'est pas plus grand que l'argument 2
Appuyez sur une touche pour continuer...
```

Exécuter des boucles

Les instructions de branchement permettent de contrôler le cours du programme à travers différents chémins possibles. C'est certes une amélioration notable, mais pas encore suffisante.

Prenons le cas d'une mise à jour de l'affichage. Sur un PC, plusieurs centaines de milliers de pixels sont recalculés lors du rafraîchissement de l'image. Un programme qui n'a pas été doté de la capacité d'exécuter répétitivement un même code devrait comporter plusieurs centaines de milliers de fois la même instruction.

Pour que l'ordinateur puisse exécuter un grand nombre de fois une même (brève) suite d'instructions, il doit faire appel à une *boucle*.

Exécuter une boucle lorsqu'une condition est vraie

L'instruction de boucle la plus simple s'appelle `while`. Elle se présente sous la forme suivante :

```
while(condition)
{
   // ...code répété aussi longtemps que la
   // condition est Vraie
}
```

La `condition` est testée. Cette condition pourrait être de type `if var > 10` ou `if var1 == var2`, ou n'importe quelle autre instruction qui vous vient à l'esprit. Si elle est vraie, les instructions entre les accolades sont exécutées. Lorsqu'il rencontre une accolade fermée, le programme reprend à l'accolade ouverte correspondante, et le processus est répété aussi longtemps que la condition est vraie (un peu comme on fait le tour du pâté de maisons avec son chien jusqu'à ce qu'il ait déposé sa..., son..., enfin bref...)

Si la condition est vraie dès la première passe, comment pourra-t-elle être fausse par la suite ? Étudions le programme que voici :

```
// WhileDemo - Entrée dans une boucle. Exécute une boucle
//             un certain nombre de fois.
#include <cstdio>
```

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

int main(int nNumberofArgs, char* pszArgs[])
{
    // compteur de boucles
    int loopCount;
    cout << "Indiquez le nombre de boucles : ";
    cin  >> loopCount;

    // exécution du nombre de boucles
    while (loopCount > 0)
    {
        loopCount = loopCount - 1;
        cout << "Il reste encore " << loopCount << " boucle(s)\n";
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le programme `WhileDemo` commence par récupérer le nombre de boucles à effectuer tel qu'il est spécifié par l'utilisateur. Il stocke cette valeur dans un compteur (la variable `loopCount`). Puis il exécute une boucle `while` qui teste d'abord `loopCount`. Si la valeur de ce compteur est supérieure à zéro, le programme entre dans le corps de la boucle (le corps est la partie de code qui se trouve entre les accolades). Il y décrémente `loopCount` de 1, puis il affiche le contenu du compteur. Le programme retourne ensuite au début de la boucle afin de vérifier si `loopCount` est encore supérieur à zéro.

Lorsqu'il est exécuté, le programme `loopCount` affiche les résultats visibles ci-dessous. Vous voyez dans cet exemple le résultat d'une boucle exécutée cinq fois avec, sur chaque ligne, la valeur courante du compteur :

```
Indiquez le nombre de boucles : 5
Il reste encore 4 boucle(s)
Il reste encore 3 boucle(s)
Il reste encore 2 boucle(s)
Il reste encore 1 boucle(s)
Il reste encore 0 boucle(s)
Appuyez sur une touche pour continuer...
```

Si l'utilisateur entre une valeur négative, le programme saute entièrement la boucle. En effet, comme la condition n'est jamais vraie, il saute directement à l'instruction qui suit l'accolade fermante. Si l'utilisateur entre une valeur très élevée, la boucle peut durer un certain temps (voire un temps certain).

La commande `do...while` est une autre version de la boucle `while`, moins usitée. Elle est identique, sauf que la condition est testée *à la fin* de la boucle :

```
do
{
   // ...contenu de la boucle
} while (condition);
```

Du fait que la condition n'est pas testée avant la fin de la boucle, le corps de `do...while` est exécuté au moins une fois.

La condition n'est contrôlée qu'au début de la boucle `while` ou à la fin de la boucle `do...while`. Même si la condition cesse d'être vraie à un certain moment lors de l'exécution de la boucle, on ne sort de celle-ci qu'au moment où la condition est de nouveau testée.

La fonction d'incrémentation/décrémentation automatique

Les programmeurs utilisent souvent les opérateurs d'auto-incrémentation `++` et d'autodécrémentation `--` dans les boucles de type "compteur". Remarquez comment, dans l'extrait de code du programme `WhileDemo`, la décrémentation de la boucle est obtenue par des instructions d'affectation et de soustraction :

```
// Exécution du nombre de boucles
while (loopCount > 0)
{
    loopCount = loopCount - 1;
    cout << "Il reste encore " << loopCount << " boucle(s)\n";
}
```

La fonction d'autodécrémentation permet d'être plus concis :

```
while (loopCount > 0)
{
```

```
    loopCount --;
    cout << "Il reste encore " << loopCount << " boucle(s)\n";
}
```

La logique est la même que dans le programme original, sauf que `loopCount` est automatiquement décrémenté.

Comme l'autodécrémentation décrémente un argument tout en retournant sa valeur, l'opération de décrémentation peut en fait être associée à la boucle `while` pour produire une programmation encore plus concise :

```
while (loopCount-- > 0)
{
    cout << "Il reste encore " << loopCount << " boucle(s)\n";
}
```

Croyez-le ou pas, `loopCount-- > 0` est la syntaxe adoptée par la plupart des programmeurs C++. Ce n'est pas que les programmeurs soient particulièrement malins (si, quand même), mais il se trouve que les fonctions d'auto-incrémentation et d'autodécrémentation facilitent considérablement la lecture d'un programme.

Les expressions `loopCount --` et `--loopCount` décrémentent toutes deux la variable `loopCount`. Mais la première retourne la valeur de `loopCount` *avant* la décrémentation, l'autre *après* la décrémentation.

Combien de fois la version de décrémentation de `WhileDemo` devra-t-elle être exécutée lorsque l'utilisateur entre une valeur de boucle de 1 ? Si vous utilisez la version avec décrémentation en préfixe, la valeur de `--loopCount` est 0 et le programme n'entre jamais dans le corps de la boucle. Avec la décrémentation en suffixe, la valeur initiale de `loopCount--` est 1 et le programme exécute une fois la boucle.

N'allez pas croire que la version du programme comportant la commande de décrémentation est exécutée plus rapidement parce qu'elle contient moins d'instructions. Il n'en est vraisemblablement rien car les compilateurs modernes savent réduire le nombre d'instructions en langage machine au strict minimum et cela, indépendamment de l'instruction de décrémentation que vous avez choisi d'utiliser.

La boucle for

L'instruction `for` est une autre forme de boucle, certainement la plus populaire. Elle est préférée à la forme élémentaire `while` car elle est généralement plus facile à lire. Elle n'offre d'ailleurs aucun autre avantage.

Le format de la boucle `for` est le suivant :

```
for (initialisation; condition; incrémentation)
{
   // ...corps de la boucle
}
```

La *clause d'initialisation* est ainsi nommée car c'est généralement à ce moment que la variable de compteur est initialisée. L'initialisation n'est exécutée qu'une seule fois, à l'instant où la boucle `for` est rencontrée.

L'exécution se poursuit par une *clause conditionnelle*. À l'instar de la boucle `while`, la boucle `for` continue de s'exécuter aussi longtemps que la clause conditionnelle est vraie.

Après avoir exécuté le code du corps de la boucle, le programme passe à la *clause d'incrémentation* avant de retourner et vérifier la condition, en vue d'une répétition du traitement. La clause d'incrémentation héberge normalement les instructions d'auto-incrémentation ou d'autodécrémentation qui mettent à jour la variable de compteur.

L'équivalent `while` de la boucle `for` est :

```
initialisation;
while(condition)
{
   {
      // ...corps de la boucle
   }
   incrémentation;
}
```

Ces trois clauses sont facultatives. Si les clauses d'initialisation ou d'incrémentation ont été omises, C++ les ignore. Si la clause conditionnelle est absente, C++ exécute la boucle `for` à l'infini (ou jusqu'à ce que quelque chose fasse quitter la boucle).

Un exemple éclairera mieux ce qu'est une boucle `for`. Le programme `ForDemo` qui suit n'est rien d'autre que le programme `WhileDemo` adapté à l'utilisation d'une boucle `for`.

```
// ForDemo1 - Entrée dans une boucle. Exécute une boucle
//            un certain nombre de fois.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // compteur de boucles
    int loopCount;
    cout << "Indiquez le nombre de boucles : ";
    cin  >> loopCount;

    // exécution du nombre de boucles
    for (; loopCount > 0;)
    {
        loopCount = loopCount - 1;
        cout << "Il reste encore " << loopCount << " boucle(s)\n";
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le programme stocke la valeur entrée au clavier dans la variable `loopCount`. La boucle `for` débute en comparant cette variable à zéro. Si ce test est vérifié, on entre dans le corps de la boucle. Le compteur est décrémenté et on affiche un message. Après cela, le programme revient au point de départ de la boucle `for` et ce, jusqu'à ce que `loopCount` devienne nul.

Les trois sections d'une boucle `for` peuvent être vides. Une clause d'initialisation ou d'incrémentation absente ne fait rien. Une clause conditionnelle vide est traitée comme s'il s'agissait d'une valeur booléenne `true`.

Cette boucle `for` a deux petits problèmes. Tout d'abord, elle est destructive. Pas dans le sens : "Le chien a encore détruit ma pantoufle gauche". Simplement parce qu'elle change la valeur de `loopCount` et donc "détruit" sa valeur d'origine. En second lieu, elle fonctionne à reculons, allant de valeurs plus

grandes à des valeurs plus petites. Tout cela peut être contourné en associant à la boucle `for` une variable de comptage dédiée. D'où cette nouvelle version :

```
// ForDemo2 - Entrée dans une boucle. Exécute une boucle
//            un certain nombre de fois.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // compteur de boucles
    int loopCount;
    cout << "Indiquez le nombre de boucles : ";
    cin  >> loopCount;

    // exécution du nombre de boucles
    for (int i = 1; i <= loopCount; i++)
    {
        cout << "Nous avons exécuté " << i << " boucle(s)\n";
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Cette version modifiée de `WhileDemo` exécute les mêmes boucles que précédemment sauf que, au lieu de faire varier la valeur de `loopCount`, elle utilise une variable de compteur.

Le programme commence par déclarer une variable (`i`) et l'initialise à la valeur contenue dans `loopCount`. Il vérifie ensuite la variable `i` pour s'assurer qu'elle est supérieure à 0. Si c'est le cas, le programme exécute l'instruction de sortie, décrémente `i` et recommence.

Lorsqu'elle est déclarée au moment de l'initialisation de la boucle `for`, la variable d'indexation n'est cependant reconnue qu'à l'intérieur de la boucle `for` elle-même. C'est pourquoi, les allumés du C++ disent que la portée de la variable est limitée à la boucle `for`. Dans l'exemple ci-dessus, la variable `i` n'est plus accessible à partir de l'instruction de retour (`return`) car cette instruction ne se trouve pas dans la boucle. Tous les compilateurs ne s'en tiennent toutefois pas à cette règle. Ainsi Dev-C++ (par exemple) émet un message d'avertis-

sement si vous utilisez `i` en dehors de la boucle `for`, mais il accepte tout de même de travailler avec cette variable.

Éviter la redoutable boucle infinie

Une *boucle infinie* est une exécution qui se poursuit éternellement (en théorie, du moins). Elle se produit lorsque la condition qui aurait pu mettre fin à la boucle n'est jamais vérifiée, généralement à cause d'une erreur de programmation.

Étudions cette variation mineure de la boucle que nous avons vue précédemment :

```
while (loopCount > 0)
   {
        cout << "Il reste encore " << loopCount << " boucle(s)\n";
   }
```

Le programmeur a oublié de décrémenter la variable `loopCount`. Par conséquent, le compteur ne change jamais, de sorte que la condition testée sera soit toujours fausse, soit toujours vraie. Le programme entre alors dans une boucle infinie.

Je sais bien que rien n'est infini : le courant peut être coupé, l'ordinateur peut se planter, Microsoft courir à la faillite et les chiens s'entendre avec les chats... À un moment ou à un autre, la boucle finira bien par s'arrêter.

Il existe bien d'autres façons de créer une boucle infinie que celle montrée ici. Et beaucoup sont autrement plus difficiles à repérer que celle-ci.

Appliquer des contrôles de boucle spéciaux

Le langage C++ est doté de deux commandes de contrôle du cours du programme nommées `break` et `continue`. Il peut arriver que la condition de fin de boucle ne se produise ni au début de la boucle ni à la fin, mais au milieu. Imaginons un programme qui accumule les nombres entrés par l'utilisateur ; la boucle se termine lorsqu'un nombre négatif est entré.

Ce qui est épineux avec ce problème, c'est que le programme ne peut pas sortir de la boucle tant que l'utilisateur n'a pas entré une valeur. En revanche, il doit pouvoir en sortir avant que la valeur ait été ajoutée à la somme.

Dans un tel cas, le C++ recourt à la commande `break` ; lorsqu'il la rencontre, le programme quitte aussitôt la boucle. Il passe directement de la commande `break` à l'instruction qui se trouve immédiatement après l'accolade fermante.

Voici le format de la commande `break` :

```
while (condition) // un break fonctionne aussi avec
                  // une boucle for
{
   if (une autre condition)
   {
      break;      // sortie de la boucle
   }
}                 // le programme se poursuit ici
                  // après l'instruction break
```

Fort de cette nouvelle commande `break`, voici la solution au problème d'accumulation des nombres :

```
// BreakDemo - Entrée d'une série de nombres.
//             Accumulation de ces nombres
//             jusqu'à ce que l'utilisateur
//             entre un nombre négatif.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // entrée dans la boucle de comptage
    int accumulator = 0;
    cout << "Ce programme additionne les valeurs "
         << "saisies par l'utilisateur\n";
    cout << "Sortez de la boucle en entrant "
         << "un nombre négatif\n";

    // boucle "infinie"
    for(;;)
    {
        // saisie d'un autre nombre
        int value = 0;
        cout << "Entrez un autre nombre : ";
        cin  >> value;
```

```
        // s'il est négatif...
        if (value < 0)
        {
            // ...sortir de la boucle.
            break;
        }

        // ...autrement, ajouter le nombre à
        // l'accumulateur
        accumulator = accumulator + value;
    }

    // maintenant que nous avons quitté la boucle,
    // on affiche la valeur obtenue par accumulation
    cout << "\nLe total est "
         << accumulator
         << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Après avoir expliqué la règle à l'utilisateur (saisir un nombre négatif pour sortir de la boucle), le programme entre dans ce qui semble être une boucle `for` infinie. Une fois dedans, `BreakDemo` récupère le nombre entré au clavier et le teste pour voir s'il répond au critère de sortie de la boucle. Si le nombre est négatif, le programme passe le contrôle à `break`, ce qui entraîne la sortie de la boucle. Dans le cas contraire, le programme ignore la commande `break` et ajoute ce nombre à celui qui se trouve dans l'accumulateur. Après avoir quitté la boucle, le programme affiche la valeur obtenue par accumulation puis s'arrête.

Lorsqu'une opération est continuellement appliquée à une variable, assurez-vous que celle-ci a été convenablement initialisée *avant* d'entrer dans la boucle. Dans notre exemple, le programme met la variable `accumulator` à zéro avant d'entrer dans la boucle où le contenu de `value` lui est ajouté.

À titre indicatif, vous pouvez tester le programme avec la séquence suivante :

```
Ce programme additionne les valeurs saisies par l'utilisateur
Sortez de la boucle en entrant un nombre négatif
Entrez un autre nombre : 1
Entrez un autre nombre : 2
```

```
Entrez un autre nombre : 3
Entrez un autre nombre : -1

Le total est 6
Appuyez sur une touche pour continuer...
```

La commande `continue` est plus rarement utilisée. Lorsque le programme la rencontre, il retourne immédiatement au début de la boucle. Le restant des instructions de la boucle est ignoré pour l'itération courante.

L'extrait de code qui suit ignore les nombres négatifs qui pourraient être entrés. La sortie se fait uniquement lorsque l'utilisateur tape un zéro :

```
while (true // a le même effet que for(;;)
{
   // Entrée d'une valeur
   cout <<"Entrez une valeur :";
   cin >> imputVal;

   // Si la valeur est négative...
   if (inputVal < 0)
   {

      // ... sortie d'un message d'erreur...
      cout << "Les nombres négatifs ne sont pas admis\n";

      // ... puis retour au début de la boucle
      continue;
   }

   // ... entrée comme d'habitude
}
```

Imbriquer des commandes de contrôle

Revenons à notre problème de réaffichage de l'écran du PC. Il est certain qu'une quelconque structure est utilisée pour tracer une ligne de pixels de gauche à droite (les moniteurs hébreux et arabes affichent-ils le balayage de droite à gauche ?). Mais comment programmer ensuite un parcours de haut en bas (les moniteurs des aborigènes australiens affichent-ils les lignes de bas en haut ?). Pour cela, nous devrons placer un balayage de ligne gauche/droite à l'intérieur d'un balayage haut/bas.

Une boucle placée à l'intérieur d'une autre boucle est une *boucle imbriquée*. Pour en avoir un exemple, vous pouvez modifier `BreakDemo` pour écrire un programme qui accumule n'importe quelle quantité de séquences numériques. Dans le programme `NestedDemo`, la boucle interne effectue la somme des nombres tapés au clavier jusqu'à ce qu'un nombre négatif ait été entré. La boucle externe continue d'accumuler des chiffres jusqu'à ce que la somme soit égale à 0. Voyons cela de plus près :

```
// NestedDemo - Entrée d'une série de nombres.
//              L'accumulation de la somme
//              de ces nombres continue jusqu'à
//              ce que l'utilisateur entre un nombre négatif.
//              Répétition du processus jusqu'à
//              ce que la somme soit égale à 0.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // boucle extérieure
    cout << "Ce programme additionne plusieurs séries\n"
         << "de nombres. Terminez chaque séquence\n"
         << "en entrant un nombre négatif.\n"
         << "Terminez les séries en entrant deux\n"
         << "nombres négatifs à la suite.\n";

    // continuation de l'accumulation des séquences
    int accumulator;
    do
    {
        // début de l'entrée de la prochaine séquence
        // de nombres.
        accumulator = 0;
        cout << "\nTapez la prochaine suite\n";

        // boucle infinie
        for(;;)
        {
            // saisie d'un autre nombre
            int value = 0;
            cout << "Entrez un autre nombre : ";
            cin  >> value;

            // s'il est négatif...
            if (value < 0)
```

```
            {
                // ... sortir de la boucle
                break;
            }

            // ... sinon, ajouter le nombre
            // à l'accumulateur
            accumulator = accumulator + value;
        }

        // afficher le contenu de l'accumulateur...
        cout << "\nLe total courant est "
            << accumulator
            << endl << endl;

        // ... et redémarrage avec une nouvelle suite
        // si la séquence accumulée n'est pas égale à 0.
    } while (accumulator != 0);

    // on va quitter le programme
    cout << "Fin du programme\n" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

On passe à autre chose ?

La dernière instruction de contrôle n'est utile que dans quelques rares cas de figure. L'instruction `switch` ressemble à un `if` évolué, dont les possibilités sont plus nombreuses et non limitées à un seul test.

```
switch(expression)
{
   cas c1:
      // Se brancher là si l'expression = c1
      break;
   cas c2:
      // Se brancher là si l'expression = c2
      break;
```

```
    else
       // Se brancher là s'il n'y a pas de correspondance
}
```

La valeur de l'expression doit être un entier (`int`, `long` ou `char`). Les valeurs de cas `c1`, `c2` et `c3` doivent être des constantes. Quand l'instruction `switch` est rencontrée, l'expression est évaluée et comparée avec les diverses constantes. Le branchement s'effectue sur le cas qui correspond. Si aucune correspondance n'est trouvée dans les cas, le programme passe à la clause `else`.

Examinons ce fragment de code :

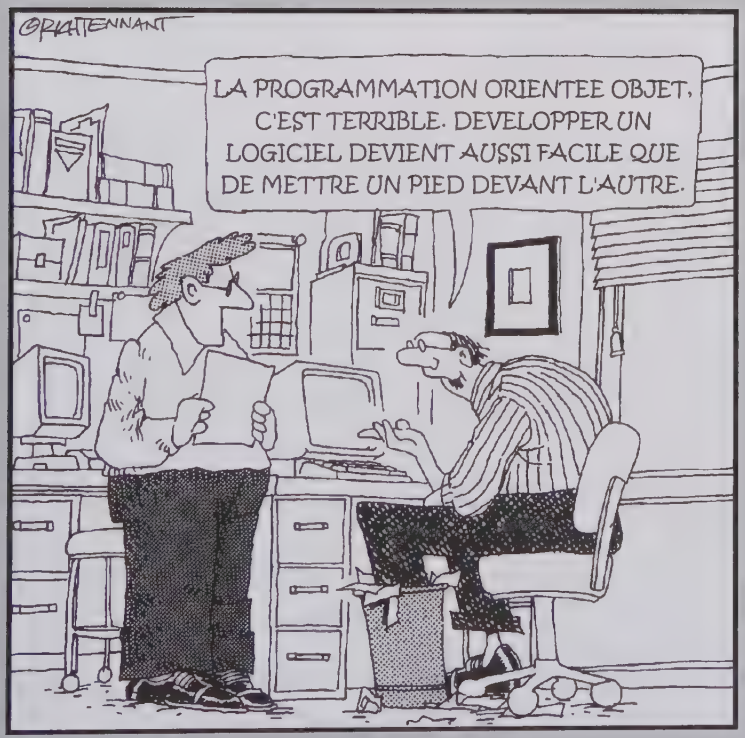
```
cout << "Entrez 1, 2 ou 3 :";
cin >> choice

switch(choice)
{
cas 1 :
   // Traitement du "1"
   break;

cas 2 :
   // Traitement du "2"
   break;

cas 3 :
   // Traitement du "3"
   break;

default;
   cout << "Vous n'avez tapé ni 1, ni 2, ni 3\n";
}
```

Là aussi, l'instruction `switch` a un équivalent : l'instruction `if`. Mais, s'il faut prendre en compte plus de deux ou trois cas, la structure `switch` sera plus limpide.

Les instructions `break` sont indispensables pour quitter la commande `switch`. Sinon, le programme se poursuit d'un cas à un autre.

Deuxième partie

Devenir un programmeur opérationnel



Dans cette partie...

C'est une chose d'effectuer des opérations comme l'addition et la multiplication, même avec de la logique (`AND`, `OR`, etc.). C'en est une autre d'écrire un programme digne de ce nom. Cette section vous présente les fonctions qui vous feront entrer de plain-pied dans l'univers impitoyable de la programmation.

Chapitre 6

Créer des fonctions

Dans ce chapitre :

- Écrire des fonctions.
- Transmettre des données à des fonctions.
- Nommer les fonctions avec différents arguments.
- Créer des prototypes de fonctions.
- Les variables et leur portée.
- Travailler avec les fichiers inclus.

Les développeurs ont souvent besoin de pouvoir diviser un programme en parties plus petites, plus faciles à manier. Les programmes développés dans les chapitres précédents étaient suffisamment modestes pour qu'aucune subdivision ne s'impose. Mais dans la "réalité", un programme peut compter des milliers, voire des millions de lignes de code. Sans la possibilité de le diviser en plusieurs parties, le développement de logiciels ambitieux s'avérerait rapidement impossible.

Le C++ permet aux programmeurs de diviser leur travail en blocs appelés "fonctions". Une fonction dotée d'une description simple et d'une interface bien définie peut être écrite et déboguée sans se soucier du code qui l'environne.

L'approche "diviser pour régner" réduit considérablement les difficultés inhérentes au développement d'un programme d'assez grande taille. Il s'agit d'une forme d'encapsulation simple (reportez-vous au Chapitre 15 pour en savoir plus sur ce procédé).

Écrire et utiliser une fonction

C'est par l'exemple que l'on comprend le mieux ce que sont les fonctions. C'est pourquoi cette section débute avec le programme `FonctionDemo` qui simplifie nettement le programme `NestedDemo` du Chapitre 5 en définissant une fonction contenant une partie du traitement. Nous verrons avec cet exemple comment une fonction est définie et comment elle est appelée.

`NestedDemo` repose sur une boucle interne accumulant une succession de chiffres ; elle est entourée par une boucle externe qui répète le processus jusqu'à ce que l'utilisateur arrête le jeu. La séparation des deux boucles simplifie le programme en permettant au lecteur de se concentrer sur chacune d'entre elles.

Le programme `FonctionDemo` qui suit montre comment `NestedDemo` peut être simplifié à l'aide d'une fonction `sumSequence()`.

Les noms de fonction sont normalement suivis par une parenthèse ouverte et une parenthèse fermée.

```
// FonctionDemo - Démontre l'utilisation des fonctions
//                en divisant la boucle interne du
//                programme NestedDemo pour en faire
//                une fonction.

#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// sumSequence - Additionne une séquence de chiffres entrée
//               au clavier jusqu'à ce que l'utilisateur
//               entre un nombre négatif.
//               Retourne la somme des nombres saisis.
int sumSequence(void)
{
    // boucle infinie
    int accumulator = 0;
    for(;;)
    {
        // entrée d'un autre nombre
        int value = 0;
        cout << "Entrez un autre nombre : ";
        cin  >> value;
```

```
        // s'il est négatif...
        if (value < 0)
        {
            // ...sortir de la boucle
            break;
        }

        // ...sinon, ajouter le nombre à l'accumulateur
        accumulator= accumulator + value;
    }

    // retourne la valeur accumulée
    return accumulator;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Ce programme additionne plusieurs séries\n"
         << "de nombres. Terminez chaque séquence\n"
         << "en entrant un nombre négatif.\n"
         << "Terminez les séries en entrant deux\n"
         << "nombres négatifs à la suite.\n"
         << endl;

    // accumulation des suites de nombres...
    int accumulatedValue;
    for(;;)
    {
        // effectue la somme de la suite de nombres
        // saisie au clavier
        cout << "Entrez la prochaine suite" << endl;
        accumulatedValue = sumSequence();

        // sortie de la boucle si sumSequence()
        // retourne un zéro
        if (accumulatedValue == 0)
        {
            break;
        }

        // affichage du résultat de l'accumulation
        cout << "Le total est "
             << accumulatedValue
             << "\n"
             << endl;
    }
```

```
    cout << "Merci beaucoup" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Définir la fonction sumSequence ()

La définition de la fonction `sumSequence()` commence à l'instruction `int sumSequence(void)`. Le bloc de code entre les accolades est le *corps de la fonction*. Le corps de la fonction `sumSequence()` est identique à celui de la boucle interne de `NestedDemo`.

Appeler la fonction sumSequence ()

Commencez par examiner attentivement la partie principale du programme qui se trouve entre les accolades qui suivent la fonction `main()`. Cette partie de la programmation ressemble beaucoup à celle de `NestedDemo`.

La principale différence réside dans l'expression `accumulateValue = sumSequence()` ; elle se trouve à peu près au milieu de la partie `main()`. Là, `sumSequence()` appelle la fonction qui porte ce même nom. La valeur retournée par la fonction est stockée dans la variable `accumulatedValue` ; cette valeur est alors affichée. Le programme principal continue de boucler jusqu'à ce que la somme retournée par la fonction interne soit zéro, ce qui indique que l'utilisateur a fini de calculer la somme.

Appeler une fonction signifie que le code contenu dans la fonction doit être exécuté. Ceci fait, le programme reprend à l'instruction qui suit immédiatement l'appel de fonction.

Diviser pour régner

Le programme `FonctionDemo` a séparé les deux boucles (extérieure et intérieure) pour faire de la seconde une fonction autonome appelée `sumSequence()`. Cette division n'a rien d'arbitraire. La fonction joue en effet un rôle spécifique, tout à fait indépendant des contrôles opérés dans `FonctionDemo`. En d'autres termes, il suffirait dans le corps de la fonction de changer la ligne :

```
accumulator= accumulator + value;
```

pour effectuer un type de calcul tout à fait différent sans rien changer au reste du programme.

Une bonne fonction est facile à décrire. On doit pouvoir en faire le "pitch" en une seule phrase contenant un minimum de mots du style *et*, *sinon*, *ou*, *mais*. Exemple : "La fonction `sumSequence` totalise une suite de valeurs entières entrées par l'utilisateur". C'est clair, concis, sans verbiage inutile.

Essayez maintenant d'exécuter `FonctionDemo`. L'affichage devrait ressembler de très près à celui de `NestedDemo`. Par exemple :

```
Ce programme additionne plusieurs séries
de nombres. Terminez chaque séquence
en entrant un nombre négatif.
Terminez les séries en entrant deux
nombres négatifs à la suite.

Entrez la prochaine suite
Entrez un autre nombre : 1
Entrez un autre nombre : 2
Entrez un autre nombre : 3
Entrez un autre nombre : -1
Le total est 6

Entrez la prochaine suite
Entrez un autre nombre : 1
Entrez un autre nombre : 2
Entrez un autre nombre : -1
Le total est 3

Entrez la prochaine suite
Entrez un autre nombre : -1
Merci beaucoup
Appuyez sur une touche pour continuer...
```

Comprendre le détail des fonctions

Les fonctions jouent un rôle si fondamental dans la création de programmes en C++ qu'il est important de bien assimiler tous les détails concernant leurs définitions, leur création et leurs tests. Ayant à l'esprit notre programme `FonctionDemo`, tentons de définir ce qu'est une fonction.

Une *fonction* est un bloc de code C++ organisé logiquement et dont la forme se présente ainsi :

```
<type de retour> nom <arguments de la fonction>
{
   // ...
   return <expression>
}
```

Les *arguments* d'une fonction sont les valeurs que la fonction peut utiliser en entrée. La *valeur de retour* est la valeur renvoyée par la fonction. Par exemple, dans l'appel de la fonction `square(10)`, 10 est l'argument de la fonction `square()`. La valeur retournée est le carré de l'argument, soit 100.

Les arguments et les valeurs de retour sont tous deux optionnels. Si les deux sont absents, le mot-clé *void* est utilisé à la place. Autrement dit, si la liste des arguments de la fonction est vide, la fonction ne prend aucun argument lorsqu'elle est appelée (ce qui fut le cas dans le programme `FunctionDemo`). Si le type de retour est vide, la fonction ne retourne aucune valeur au code appelant.

Dans le programme `FonctionDemo`, le nom de la fonction est `sumSequence()`, le type de retour `int` et il n'existe aucun argument.

Le type d'argument par défaut d'une fonction est `void`, ce qui signifie qu'aucun argument n'est pris. Une fonction `int fn(void)` peut aussi être déclarée sous la forme `int fn()`.

La construction de la fonction m'a permis d'écrire deux parties distinctes du programme `FunctionDemo`. Je me suis simplement attaché à obtenir, avec `sumSequence()`, la somme d'une suite de nombres sans me préoccuper du restant du code susceptible d'appeler cette fonction.

De même, quand j'ai écrit `main()`, je ne me suis occupé que de la somme retournée par `sumSequence()`, en me consacrant entièrement à ce que fait cette fonction, sans avoir à me demander comment elle fonctionne.

Comprendre les fonctions simples

La simple fonction `sumSequence()` retourne une valeur entière qu'elle vient de calculer. Une fonction peut retourner n'importe quel type de variable, comme `double` ou `char` par exemple (`int`, `double` et `char` sont quelques-uns des types étudiés dans le Chapitre 2).

Si une fonction ne retourne aucune valeur, le type de retour de la fonction est déclaré "vide" (*void*).

Une fonction peut être étiquetée selon son type de retour. De ce fait, une fonction qui retourne un entier est souvent appelée *fonction entière*. Une fonction qui ne retourne aucune valeur est une *fonction vide*.

Par exemple, la fonction vide ci-dessous effectue une opération mais ne retourne aucune valeur :

```
void echoSquare()
{
   cout << "Entrez une valeur :";
   cin >> value;
   cout << "n\Le carré est : " << value * value << "\n";
   return;
}
```

Le contrôle du programme s'effectue à partir de l'accolade ouvrante et se poursuit jusqu'au-delà de l'instruction de retour. L'instruction de retour d'une fonction vide n'est pas suivie d'une valeur.

L'instruction de retour d'une fonction vide est facultative. Si elle n'est pas présente, l'exécution reprend à la ligne de code appelante dès que le programme rencontre l'accolade fermante.

Comprendre les fonctions avec arguments

Les fonctions simples sont assez limitées dans leur usage car leur communication est unidirectionnelle : elles retournent une valeur. Les fonctions ayant des arguments établissent une communication bidirectionnelle.

Les fonctions avec arguments

Un *argument* est une variable dont la valeur est transmise à la fonction au moment où elle est appelée. L'exemple qui suit définit et utilise une fonction `square()` qui retourne le carré d'un nombre flottant en double précision qui lui a été passé :

```
// SquareDemo - Démonstration d'une fonction
//              qui traite des arguments
```

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

//         square - retourne le carré de l'argument
//      doubleVar - valeur à mettre au carré
//       retourne - carré de doubleVar
double square(double doubleVar)
{
    return doubleVar * doubleVar;
}

// sumSequence - additionne les carrés des nombres tapés
//               au clavier jusqu'à ce que l'utilisateur
//               entre un nombre négatif.
double sumSequence(void)
{
    // boucle infinie
    double accumulator= 0.0;
    for(;;)
    {
        // entrée d'un autre nombre
        double dValue = 0;
        cout << "Entrez un autre nombre : ";
        cin  >> dValue;

        // s'il est négatif...
        if (dValue < 0)
        {
            // ...sortir de la boucle
            break;
        }

        // ...sinon, mettre au carré
        double value = square(dValue);

        // ajouter le carré à l'accumulateur
        accumulator= accumulator + value;
    }

    // retourne la valeur accumulée
    return accumulator;
}

int main(int nNumberofArgs, char* pszArgs[])
{
```

```
    cout << "Ce programme fait la somme de suites de\n"
         << "nombres mis au carré. Terminez chaque suite\n"
         << "en entrant un nombre négatif.\n"
         << "Terminez les suites en entrant deux\n"
         << "nombres négatifs successifs\n"
         << endl;

    // continuation de l'accumulation des nombres...
    double accumulatedValue;
    for(;;)
    {
        // effectue la somme de la suite des nombres
        // saisis au clavier
        cout << "Entrez la prochaine suite" << endl;
        accumulatedValue = sumSequence();

        // termine si la somme est inférieure ou égale à 0
        if (accumulatedValue <= 0.0)
        {
            break;
        }

        // affichage du résultat de l'accumulation
        cout << "\nLa somme des carré vaut "
             << accumulatedValue
             << endl;
    }

    cout << "Merci beaucoup" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

C'est toujours le programme `FunctionDemo`, sauf que `SquareDemo` met les valeurs entrées au carré et les additionne. La fonction `square()` retourne la valeur de son unique argument multiplié par lui-même. L'adaptation de la fonction `sumSequence()` est simple : au lieu d'accumuler les valeurs entrées, la fonction accumule désormais les résultats retournés par `square()`.

Fonctions avec plusieurs arguments

Les fonctions peuvent avoir plusieurs arguments séparés par des virgules. C'est ainsi que la fonction ci-dessous retourne le produit de ses deux arguments :

```
int product(int arg1, int arg2)
{
   return arg1 * arg2;
}
```

Un mot sur main()

Malgré ses arguments bizarroïdes, le "mot-clé" `main()`, dans notre programme d'exemple, n'est rien de plus qu'une fonction.

Lors de la construction d'un programme, C++ ajoute du code réutilisable qui s'exécute avant même le démarrage de votre propre programme. Il définit l'environnement dans lequel le programme opérera. Il ouvre par exemple les canaux d'entrée et de sortie par défaut.

Après avoir établi l'environnement, le code réutilisable du C++ appelle la fonction `main()`, ce qui lance l'exécution de votre code. Lorsque votre programme s'achève, il quitte la fonction `main()`. Le code réutilisable peut donc faire le ménage avant de rendre la main au système d'exploitation.

La surcharge des noms de fonction

Le C++ permet au programmeur d'affecter un même nom à plusieurs fonctions. Cette utilisation multiple d'un nom est appelée *fonction de surcharge* ou simplement *surcharge*.

En règle générale, deux fonctions ne peuvent pas avoir le même nom dans un programme donné. Si tel était le cas, C++ ne saurait pas les différencier.

Le nom d'une fonction comprend cependant le nombre et le type de ses arguments (mais pas l'argument de retour). C'est pourquoi les définitions qui suivent correspondent à des fonctions différentes :

```
void uneFonction(void)
{
   // Exécute une fonction
```

```
}
void uneFonction(int n)
{
   // Exécute une autre fonction
}
void uneFonction(double d)
{
   // Exécute une fonction très différente
}
void uneFonction(int n1, int n2)
{
   // Celle-là fait encore autre chose
}
```

C++ sait bien que les fonctions `uneFonction(void)`, `uneFonction(int)`, `uneFonction(double)` et `uneFonction(int, int)` ne sont pas les mêmes. Comme souvent en informatique, on peut recourir à une analogie humaine.

En tant que type d'argument, `void` est optionnel ; `sumFunction(void)` et `sumFunction()` sont une seule et même fonction. Une fonction a un diminutif, comme `uneFonction()` par exemple. Comme mon diminutif est Stephen (en fait, mon surnom est Randy, qui me va très bien) et qu'il n'y a aucun autre Stephen à la ronde, on sait de qui il s'agit quand les gens parlent de moi. Mais s'il y a d'autres Stephen dans les parages, et quelle que soit leur apparence, les gens devront utiliser leur nom complet pour ne pas les confondre ; dans mon cas, c'est Stephen Davis. Tant que le nom complet est utilisé, aucun quiproquo n'est possible, et cela quel que soit le nombre de Stephen qui grouillent dans le coin. Le nom complet de `uneFonction()` est `uneFonction(int)`. Tant que ce nom complet reste unique, il ne se produira aucune confusion.

L'analogie entre l'informatique et le genre humain est rarement surprenante puisque ce sont les humains qui ont inventé les ordinateurs. Je me demande ce qui se passerait si les chiens construisaient des ordinateurs... On compterait peut-être en tas d'os au lieu de bits.

Voyons un cas typique d'utilisation des surcharges :

```
int intVariable1, intVariable 2;    // équivalent à :
                                    // int Variable 1;
                                    // int Variable 2;
double doubleVariable;

// Les fonctions sont différenciées grâce au type
// de l'argument qui est passé.
```

```
uneFonction();                              // appelle uneFonction(void)
uneFonction(intVariable1);                  // appelle uneFonction(int)
uneFonction(doubleVariable);                // appelle uneFonction(double)
uneFonction(intVariable1, intVariable2); // appelle uneFonction(int, int)

// Ce principe fonctionne aussi avec les constantes :
uneFonction(1);                             // appelle uneFonction(int)
uneFonction(1.0);                           // appelle uneFonction(double)
uneFonction(1, 2);                          // appelle uneFonction(int, int)
```

Dans chaque cas, le type de l'argument correspond au nom complet des trois fonctions.

Le type de la valeur retournée ne fait pas partie du nom étendu de la fonction (aussi appelé "signature de la fonction"). Les deux fonctions qui suivent ont le même nom et ne peuvent donc pas coexister dans un même programme :

```
int uneFonction(int n);    // le nom complet de la fonction
                           // est uneFonction(int)
double uneFonction(int n)  // Même nom
```

Ce qui suit est admis parce que le type de la variable source (en l'occurrence un entier) est plus restreint que celui de la variable cible (un double dans ce cas) :

```
int uneFonction(int n)
double d = uneFonction(10); // Promeut la valeur retournée
```

L'entier `int` retourné par `uneFonction()` est promu en `double`. En revanche, le code ci-dessous prête à confusion :

```
int uneFonction(int n);
double uneFonction(int n);
double d = uneFonction(10) // Promotion du int retourné ?
                           // ou utilisation tel quel du
                           // double retourné ?
```

C++ ne saura pas s'il doit utiliser la valeur retournée par la version `double` de `uneFonction` ou promouvoir la valeur retournée par la version `int`.

Définir des prototypes de fonction

Le programmeur peut attribuer au restant d'un fichier source C++, ou module, un nom étendu (le nom et les fonctions) pendant la définition de la fonction.

Les fonctions cibles `sumSequence()` et `square()`, rencontrées précédemment dans ce chapitre, étaient toutes deux définies dans du code qui apparaissait avant l'appel lui-même. Mais ce n'est pas une obligation. Une fonction peut en effet être définie n'importe où dans le module (un *module* est un autre nom donné à un fichier source C++).

Il faut cependant que `main()` soit informé du nom complet d'une fonction avant qu'elle puisse être appelée. Voyons le petit bout de code qui suit :

```
int main(int argc, char* pArgs[])
{
   uneFonc(1, 2);
}
int uneFonc(double, arg1, int arg2)
{
   // ...fait quelque chose
}
```

`main()` ne connaît pas le nom complet de la fonction lorsqu'il est appelé. Il pourrait déduire d'après les arguments que le nom est `uneFonc(int, int)` et que son type de retour est `void` ; cependant, comme vous pouvez le constater, cette conclusion est erronée.

Je sais, je sais... Le C++ pourrait être un peu moins cossard et se donner la peine de déterminer le nom complet de `uneFonc()` par lui-même en regardant un peu plus loin. C'est comme pour ma vieille voiture : j'ai dû apprendre à vivre avec.

Ce qu'il faudrait, c'est un moyen d'indiquer à `main()` le nom complet de `uneFonc()` avant qu'elle soit utilisée, c'est-à-dire une déclaration d'utilisation préliminaire, ce que l'on appelle un *prototype* (à ne pas confondre avec ma voiture).

Une déclaration de prototype ressemble à une fonction sans corps :

```
int uneFonc(double, int);
int main(int, argc, char* pArgs[])
{
```

```
   uneFonc(1, 2);
}
int uneFonc(double, arg1, int arg2)
{
   // ...fait quelque chose
}
```

La déclaration de prototype clame haut et fort que le nom de `uneFonc()` est `uneFonc(double, int)`. La fonction `main()` saura qu'il faut explicitement convertir 1 en `double` avant de procéder à l'appel. De plus, `main()` est prévenu que la valeur retournée par `uneFonc()` est un entier `int`.

La portée des variables

Les variables de fonction sont stockées selon trois niveaux différents. Celles qui sont déclarées à l'intérieur d'une fonction sont dites *locales*. Dans l'exemple qui suit, la variable `localVariable` est locale à la fonction `fn()` :

```
int globalVariable;
void fn()
{
   int localVariable;
   static int staticVariable;
}
```

La variable `localVariable` n'existe pas avant que la fonction `fn()` soit appelée ; elle cesse d'exister lorsque la fonction se termine. Lors du retour, toute valeur stockée dans `localVariable` est perdue. De plus, seul `fn()` peut accéder à `localVariable` ; les autres fonctions ne peuvent pas lire ce qu'elle contient.

La variable `globalVariable`, quant à elle, existe aussi longtemps que le programme est en cours d'exécution. Toutes les fonctions peuvent y accéder à tout moment.

La variable *statique* `staticVariable` est un hybride entre une variable locale et une variable globale. Elle est créée lorsque le programme atteint la déclaration (en gros, quand la fonction `fn()` est appelée pour la première fois). De plus, `staticVariable` n'est accessible qu'à l'intérieur de `fn()`. Cependant, contrairement à `localVariable`, `staticVariable` continue d'exister après que l'on quitte `fn()`. Si `fn()` affecte une certaine valeur à `staticVariable`, elle la retrouvera lors du prochain appel.

Au cas où quelqu'un vous le ferait remarquer, il existe en effet un quatrième type, `auto`. Mais, comme ce type de variable est identique à `local`, nous pouvons l'ignorer. C'est un peu comme la fumée bleue qui sort du pot d'échappement de ma voiture : je l'ignore.

Inclure des fichiers inclus

Il est courant de placer les prototypes de fonctions dans un fichier à part (on dit un *fichier inclus*) que le programmeur C++ peut ensuite incorporer dans son code source. Cela revient à demander au préprocesseur C++ (qui est exécuté avant de passer le relais au compilateur proprement dit) d'insérer le texte de `monfichier` (par exemple) là où il rencontre une instruction du genre `#include "monfichier"`.

Une définition typique pour un fichier de calcul d'expressions mathématiques pourrait alors se présenter ainsi :

```
// fichier inclus pour des calculs mathématiques:
// fournit des prototypes pour des fonctions utiles
// dans de nombreux programmes.

// abs - retourne la valeur absolue de l'argument
double abs(double d);

// square - retourne le carré de l'argument
double square(double d);
```

Un programme pourrait alors se servir de ces définitions en déclarant :

```
// MonProgramme -
#include "math"

using namespace std;
// mon code continue ici
```

La seconde ligne se lit ainsi : "Remplacer cette directive par le contenu du fichier `math`".

La *directive* `#include` n'a pas le même format que les instructions C++ car elle est traitée par un interpréteur spécifique, qui s'exécute *avant* que le compilateur ne débute son travail. En particulier, le caractère # doit se trouver en début de ligne (ou encore en colonne 1) et la directive doit être suivie d'un

retour chariot. Le nom du fichier inclus peut être placé entre crochets ou entre guillemets. Comme les crochets servent aux fonctions déclarées dans les librairies de C++ lui-même, il vaut mieux utiliser uniquement des guillemets pour vos propres définitions. Ceci évitera toute confusion inutile.

L'environnement C++ fournit de multiples fichiers inclus, tels que `cstdio` ou encore `iostream`. Nous avons par exemple rencontré dans le Chapitre 4 une fonction appelée `setf()`. Son prototype se trouve précisément dans `iostream`.

Autrefois, les développeurs avaient l'habitude d'associer l'extension `.h` aux fichiers inclus. Ces dernières années, le standard C++ ISO a supprimé cette obligation (par exemple, `cstdio` s'appelait auparavant `stdio.h`). Mais de nombreux programmeurs continuent obstinément à ajouter l'extension `.h`. (Eh oui ! Même le monde du high-tech a ses traditions.)

Chapitre 7

Stocker des éléments dans des tableaux

Dans ce chapitre :

- Pourquoi les tableaux sont indispensables.
- Présentation des types de données des tableaux.
- Utiliser un tableau.
- Utiliser le type de tableau le plus courant : la chaîne de caractères.

Un *tableau* est une suite de données (ou de variables) qui partagent le même nom et qui sont référencées par un index. Les tableaux sont de bien belles petites choses qui permettent de stocker un grand nombre de valeurs ayant quelques liens ou relations entre elles. Par exemple, les résultats scolaires de chacun des élèves d'une classe sont de parfaits candidats pour être mis en tableau. Un tableau peut être multidimensionnel, ce qui permet de stocker les notes par mois.

Dans ce chapitre, vous découvrirez comment initialiser et utiliser des tableaux. Vous découvrirez aussi une forme fort utile de tableau, la *chaîne*, qui, en C++, n'est jamais qu'un tableau de type `char`.

Pourquoi les tableaux sont indispensables

Considérons le problème suivant : il vous faut un programme capable de lire une suite de nombres entrés au clavier. Nous adopterons la règle qui veut qu'un nombre négatif mette fin à la saisie. Une fois que les nombres ont été lus, le programme devra les afficher sur un périphérique de sortie standard avant de donner leur moyenne.

Vous pourriez essayer de stocker les nombres dans un ensemble de variables indépendantes comme dans :

```
cin >> valeur1;
if (valeur1 > 0)
{
   cin >> valeur2;
   if (valeur2 >= 0)
   {
      ...
```

Vous vous rendez compte que cette approche est incapable de gérer efficacement des suites comportant un grand nombre d'éléments. Qui plus est, elle est dépourvue de toute élégance. Ce qu'il vous faut, c'est une structure dont le nom serait celui d'une variable mais qui contiendrait de nombreuses valeurs. C'est là qu'intervient le tableau.

Un tableau résout élégamment le problème des suites. Par exemple, le morceau de code ci-dessous déclare un tableau nommé `valueArray` capable de stocker jusqu'à 128 valeurs entières. Puis il remplit ce tableau avec des nombres tapés au clavier.

```
int value;

// Déclaration d'un tableau capable de
// contenir jusqu'à 128 entiers.
int valueArray[128];

// Définition de l'index utilisé pour
// accéder aux membres suivants du
// tableau, dans la limite des 128 entiers.
for (int i = 0; i < 128; i++)
{
   cin >> value;

   // Sort de la boucle lorsque l'utilisateur
   // entre un nombre négatif.
   if (value < 0)
   {
      break;
   }
   valueArray[i] = value;
}
```

La deuxième ligne de code de ce petit programme déclare le tableau `valueArray`. Une déclaration de tableau commence par le type des membres du tableau (`int` en l'occurrence) suivi du nom du tableau. Le dernier élément de la déclaration, entre crochets, indique le nombre maximum d'éléments que le tableau peut contenir. Dans notre exemple, le tableau `valueArray` accepte jusqu'à 128 nombres entiers.

Le programme récupère un nombre entré au clavier et le stocke dans un membre du tableau `valueArray`. L'accès à un élément d'un tableau s'effectue en donnant le nom du tableau suivi par un numéro d'index entre crochets. Le premier entier du tableau est donc `valueArray[0]`, le deuxième `valueArray[1]`, et ainsi de suite.

En pratique, `valueArray[i]` représente le $i^{ème}$ élément du tableau. La variable d'indexation `i` doit être un compteur, autrement dit posséder le type `int`, `char` ou `long`. Si `valueArray` est un tableau d'entiers, `valueArray[i]` est par conséquent un entier.

Utiliser un tableau

Le programme qui suit entre une suite d'entiers, saisis au clavier, jusqu'à ce que l'utilisateur entre un nombre négatif. Il affiche ensuite les entrées et leur somme.

```
// ArrayDemo - Démontre l'utilisation des tableaux
//             au travers de la lecture d'une suite d'entiers
//             et les affiche ensuite dans l'ordre de la saisie
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// déclaration des prototypes
int sumArray(int integerArray[], int sizeOfloatArray);
void displayArray(int integerArray[], int sizeOfloatArray);

int main(int nNumberofArgs, char* pszArgs[])
{
    // entrée dans la boucle de comptage
    int nAccumulator = 0;
    cout << "Ce programme additionne les valeurs saisies "
         << "par l'utilisateur\n";
    cout << "Quittez la boucle en tapant "
         << "un nombre négatif.\n";
```

```
    cout << endl;

    // stockage des nombres dans le tableau
    int inputValues[128];

    int numberOfValues;
    for(numberOfValues = 0;
        numberOfValues < 128;
        numberOfValues++)
    {
        // entrée d'un autre nombre
        int integerValue;
        cout << "Tapez le prochain nombre : ";
        cin  >> integerValue;

        // s'il est négatif...
        if (integerValue < 0)
        {
            // ...sortie de la boucle
            break;
        }

        // ...sinon, enregistrement du nombre
        // dans le tableau
        inputValues[numberOfValues] = integerValue;
    }

    // affichage des valeurs et somme des valeurs
    displayArray(inputValues, numberOfValues);
    cout << "La somme vaut "
         << sumArray(inputValues, numberOfValues)
         << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

// displayArray : affiche les membres du tableau
//                selon la longueur de sizeOfloatArray
void displayArray(int integerArray[], int sizeOfArray)
{
    cout << "Valeur des membres du tableau :" << endl;
    for (int i = 0; i < sizeOfArray; i++)
    {
        cout.width(3);
```

```
        cout << i << " : " << integerArray[i] << endl;
    }
    cout << endl;
}

// sumArray : retourne de la somme des membres d'un
//            tableau d'entiers
int sumArray(int integerArray[], int sizeOfArray)
{
    int accumulator = 0;
    for (int i = 0; i < sizeOfArray; i++)
    {
        accumulator += integerArray[i];
    }
    return accumulator;
}
```

Le programme `ArrayDemo` commence par une déclaration de prototype des fonctions `sumArray()` et `displayArray()` qu'il utilisera par la suite. Le corps principal contient une boucle pour les entrées. Mais cette fois, les valeurs entrées sont stockées dans le tableau `inputValues`.

Une entrée se produit dans la boucle initiale `for`. Elle est d'abord stockée en externe dans la variable locale `integerValue` ; si elle s'avère négative, un contrôle `break` fait quitter la boucle. Sinon, `integerValue` est copié dans le tableau. La variable entière `numberOfValues` est la variable d'indexation du tableau.

La variable `numberOfValues` a été initialisée à 0 au début de la boucle `for` ; l'index est incrémenté à chaque itération de la boucle. Le test de la boucle `for` évite au programme d'acquérir plus de 128 entrées et, par conséquent, de dépasser la taille du tableau. Lorsque ce chiffre est atteint, le programme passe à la partie réservée à la sortie, que l'utilisateur ait entré un nombre négatif ou non.

Le tableau `inputValues` est déclaré en tant que "128 entiers longs". Si vous vous imaginez que vous pouvez vous en tenir à ça, détrompez-vous. L'écriture de données excédentaires dans un tableau entraîne en effet un fonctionnement erratique du programme qui finit souvent par se planter. Quelle que soit la taille d'un tableau, vous devez toujours effectuer un contrôle pour vous assurer que la limite spécifiée ne risque pas d'être dépassée.

La fonction principale se termine par l'affichage du contenu du tableau et de sa somme (fonction `displayArray()`).

L'environnement de Dev-C++ vous aide à mieux gérer vos fonctions. La Figure 7.1 montre le contenu de l'onglet Classes. Vous y trouvez le nom et le prototype de chaque fonction. Un double clic sur une ligne donne directement accès à la fonction correspondante dans le fichier source .CPP.

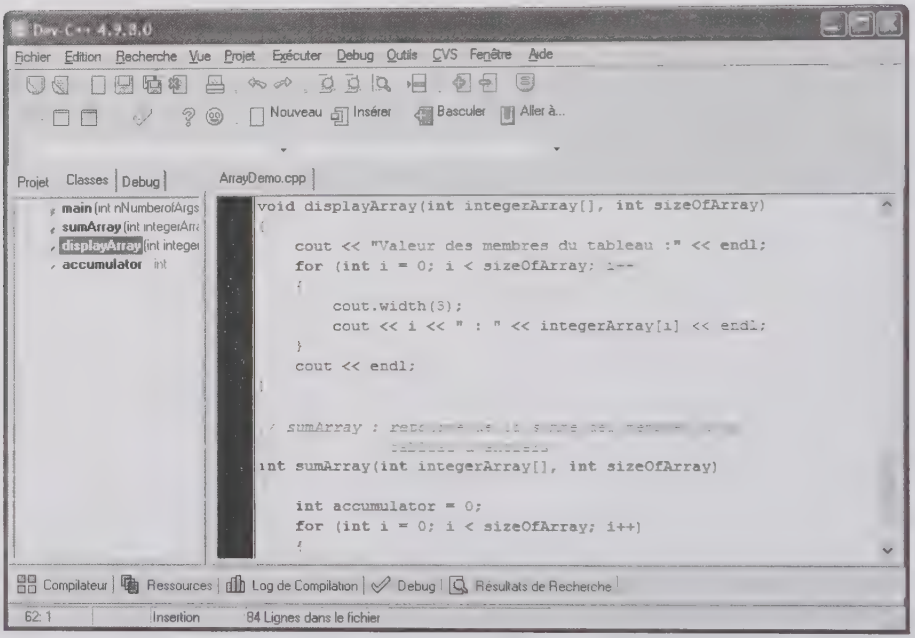


Figure 7.1
L'onglet Classes affiche des informations sur les fonctions du programme.

La fonction `displayArray()` contient la boucle `for` classiquement utilisée pour parcourir le tableau. Chacune des entrées du tableau est ajoutée à la variable `accumulator`. La variable `sizeOfArray` passée à la fonction indique le nombre de valeurs contenues dans le tableau.

Remarquez dès à présent que l'indice est initialisé à 0 et non à 1. Remarquez aussi comment la boucle `for` se termine avant que `i` soit égal à `sizeOfArray` (donc à la taille du tableau). Vous n'avez pas besoin d'ajouter tous les 128 éléments de `integerArray` dans `accumulator` car aucun des éléments situés après `sizeOfArray` ne contient de donnée valide.

Voici un exemple d'affichage produit pas notre programme :

```
Ce programme additionne les valeurs saisies par l'utilisateur
Quittez la boucle en tapant un nombre négatif.

Tapez le prochain nombre : 1
Tapez le prochain nombre : 2
Tapez le prochain nombre : 3
Tapez le prochain nombre : -1
Valeur des membres du tableau :
  0 : 1
  1 : 2
  2 : 3

La somme vaut 6
Appuyez sur une touche pour continuer...
```

Signalons à l'intention des non-programmeurs que le terme *itération* signifie que le programme parcourt un ensemble d'objets, comme c'est le cas dans le tableau. Pour les programmeurs, la fonction `sumArray()` effectue une itération dans le tableau. Dans le même esprit, la fonction `displayArray()` effectue une itération dans `integerArray` en affichant chacun de ses éléments.

Initialiser un tableau

Une variable locale ne commence pas à exister seulement au moment où une valeur valide, y compris zéro, lui est affectée. Ou, en d'autres termes, une variable locale contient des informations incohérentes jusqu'au moment où vous y stockez véritablement des données. Il en va de même des tableaux déclarés en local : chaque élément contient des informations erratiques jusqu'au moment où vous lui affectez des données. C'est pourquoi vous devriez initialiser les variables locales au moment où vous les déclarez. Cette règle est encore plus vraie pour les tableaux. Car il est vraiment trop facile d'accéder à des éléments de tableau non initialisés en croyant naïvement que ce sont des valeurs valides.

Fort heureusement, un tableau peut être initialisé au moment de sa déclaration. Le petit bout de code que voici montre comment :

```
float floatArray[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Il initialise `floatArray[0]` à 0, `floatArray[1]` à 1, `floatArray[2]` à 2 et ainsi de suite.

Le nombre de constantes d'initialisation peut déterminer la taille du tableau. Vous devriez par exemple savoir que `floatArray` contient cinq éléments rien qu'en comptant les valeurs entre les accolades. C++ peut aussi le faire (voilà au moins une chose qu'il sait faire par lui-même).

La déclaration qui suit est identique à la précédente :

```
float floatArray[] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Vous pouvez initialiser tous les éléments d'un tableau à une valeur commune en ne faisant figurer que cette donnée. La ligne suivante, par exemple, initialise vingt-cinq emplacements de `floatArray` à 1.0 :

```
float floatArray[25] = {1.0};
```

Trop loin, hors du tableau

Les mathématiciens comptent dans un tableau à partir de 1. Par conséquent, le premier membre d'un tableau mathématique x est `x(1)`. Nombre de langages de programmation commencent eux aussi à 1. Mais le C++, lui, commence à 0. C'est pourquoi le premier membre d'un tableau C++ est `valeurTableau[0]`.

Je me demande parfois si C++ ne signifie pas Contrariant++ ; quoi qu'il en soit, avec une indexation de tableau qui commence à 0, le dernier élément d'un tableau de 128 entiers est forcément numéroté `tableauEntiers[127]` et non `tableauEntiers[128]`.

Le langage C++ ne teste hélas pas l'index que vous fournissez pour vérifier s'il se trouve à l'intérieur du tableau. Il est parfaitement capable d'accepter sans vergogne un indice `tableauEntiers[200]`, voire même `tableauEntiers[-15]`.

Par analogie, écrire `tableau[20]` dans un tableau à dix éléments retourne une valeur plus ou moins aléatoire ; les résultats sont imprévisibles. Il peut ne rien se produire, ou encore en résulter un comportement erratique voire un plantage pur et simple du programme.

L'erreur la plus courante consiste à écrire `tableauEntiers[128]`. Comme cet élément se trouve juste après la fin du tableau spécifié, lire ou écrire à cet emplacement est aussi dangereux que n'importe quelle autre adresse erronée.

Utiliser des tableaux

De prime abord, le programme `ArrayDemo` ne fait rien de plus que le programme précédent, qui n'avait pas recours à un tableau. Il est vrai qu'il est capable de restituer les entrées en les affichant avant d'en faire la somme, mais on ne peut pas dire que cette fonctionnalité bouleverse l'histoire du monde.

La possibilité de réafficher des valeurs est certes un avantage indéniable des tableaux. Ils permettent à un programme de traiter des séries de nombres à plusieurs reprises. La partie principale du programme est ainsi susceptible de passer le tableau des valeurs entrées à `displayArray()` afin de les afficher, puis de les repasser à `sumArray()` pour les additionner.

Définir et utiliser des tableaux de tableaux

Les tableaux se prêtent bien au stockage de suites de nombres. Or, certaines applications exigent des suites de suites. Le tableur est un exemple classique d'une telle configuration *matricielle*. Disposée comme dans les cases d'un échiquier, chaque cellule d'un tableau est définie par sa position en `x` et en `y`.

Le langage C++ implémente une matrice de la façon suivante :

```
int intMatrix[10][5];
```

Cette matrice a dix éléments dans une dimension et cinq dans une autre, soit un total de cinquante éléments. Autrement dit, `intMatrix` est un tableau à dix éléments dont chaque élément est un tableau à cinq éléments entiers. Comme vous l'avez deviné, un coin de cette matrice se trouve à `intMatrix[0][0]` tandis que l'autre coin correspond à l'indice `intMatrix[9][4]`.

Considérer que `intMatrix` a dix éléments dans la dimension `x` ou dans la dimension `y` n'est qu'une question de choix personnel. Une matrice peut être initialisée de la même manière qu'un tableau :

```
int intMatrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Cette ligne initialise le tableau à trois éléments `intMatrix[0]` à 1, 2 et 3, et le tableau à trois éléments `intMatrix[1]` à 4, 5 et 6.

Les tableaux de caractères

Les éléments d'un tableau peuvent être de n'importe quel type ; il n'est pas rare de trouver des tableaux de flottants, de doubles et de longs. Les tableaux de caractères sont cependant un peu particuliers.

Créer un tableau de caractères

Les mots et les phrases peuvent être exprimés au travers d'un tableau de caractères. Celui contenant mon prénom se présenterait ainsi :

```
char sMonPrenom[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'}
```

Ce petit programme affiche mon nom :

```
// CharDisplay - affiche un tableau de caractères sur la
//               sortie standard, c'est-à-dire dans
//               une fenêtre MS-DOS (invite de commandes).
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// déclaration de prototype
void displayCharArray(char stringArray[],
                      int sizeOfloatArray);

int main(int nArg, char* pszArgs[])
{
    char charMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
    displayCharArray(charMyName, 7);
    cout << "\n";
    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

// displayCharArray - affiche un tableau de caractères
//                    en ne sortant qu'un seul
//                    caractère à la fois.
void displayCharArray(char stringArray[],
                      int sizeOfloatArray)
```

```
{
    for(int i = 0; i< sizeOfloatArray; i++)
    {
        cout << stringArray[i];
    }
}
```

Le programme déclare un tableau de caractères fixe, `charMyName`, qui contient – qui l'eut cru ? – mon nom. Ce tableau est passé à la fonction `displayCharArray()`, en même temps que sa longueur. Cette fonction est identique à la fonction `displayArray()` de l'exemple précédent, sauf que celle-ci affiche des caractères (`char`) au lieu d'afficher des entiers (`int`).

Ce programme fonctionne parfaitement ; il n'est cependant pas pratique de passer la longueur du tableau en même temps que le tableau lui-même. Faute de syntaxe nous permettant de nous dispenser de passer la longueur du tableau, nous pourrons savoir que le tableau est plein lorsque nous rencontrerons un caractère de programmation spécial.

Créer une chaîne de caractères

Avec C++, fixons comme règle que le caractère "spécial" 0 marque la fin d'un tableau de caractères.

Un caractère dont la valeur (ou le code) est 0 n'est pas la même chose que 0. La valeur d'un 0 est 0x10 ; mais un caractère dont la valeur est 0 est souvent écrit sous la forme `\0`, dont la valeur est 0x0 (juste pour bien indiquer qu'il s'agit d'un caractère et non d'un chiffre).

Le caractère \y est un caractère dont la valeur numérique est y. Le caractère \0 est aussi appelé "caractère nul". Pour clarifier la situation, et selon cette règle, reprenons le petit programme que nous venons de taper :

```
// DisplayString - affiche un tableau de caractères sur la
//                 sortie standard, c'est-à-dire dans
//                 une fenêtre MS-DOS (invite de commandes).
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;
```

```
// déclaration de prototype
void displayString(char stringArray[]);

int main(int nNumberofArgs, char* pszArgs[])
{
    char charMyName[] =
              {'S', 't', 'e', 'p', 'h', 'e', 'n', 0};
    displayString(charMyName);
    cout << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

// displayString - affiche une chaîne de caractères
//                 en ne sortant qu'un seul
//                 caractère à la fois.
void displayString(char stringArray[])
{
    for(int i = 0; stringArray[i] != '\0'; i++)
    {
        cout << stringArray[i];
    }
}
```

La déclaration `charMyName` déclare le tableau de caractères avec un caractère nul \0 supplémentaire à la fin. La partie du programme `displayString` effectue des itérations à travers le tableau de caractères jusqu'à ce que ce caractère nul ait été rencontré.

La fonction `displayString()` est plus simple à utiliser que sa précédente `displayCharArray()` car il n'est plus nécessaire de passer la longueur du tableau de caractères. De plus, `displayString()` fonctionne même lorsque la taille de la chaîne de caractères n'est pas connue au moment de la compilation. Ce cas se produit plus souvent que vous pourriez le croire.

Cette façon de terminer un tableau de caractères par un caractère nul est si commode qu'elle est utilisée à tout bout de champ dans le langage C++. Le C++ a même donné un nom spécial à ce genre de tableau.

Une *chaîne de caractères* est un tableau de caractères terminé par un caractère nul. Pour ajouter à la confusion, on parle communément de *chaîne*, bien que C++ définisse pour cela un type spécifique : `string`.

Le choix de '\0' comme caractère de fin ne relève pas du hasard. N'oubliez pas que zéro est la seule valeur numérique qui peut être logiquement évaluée comme étant `false`. Toutes les autres se traduisent par `true`. Ceci signifie que la boucle `for` pourrait (et même devrait) s'écrire de la façon suivante :

```
for(int i = 0; stringArray[i]; i++)
```

Tout cela imprègne tellement la culture des développeurs que C++ propose même une méthode d'initialisation de chaîne, bien plus commode, basée sur l'usage de guillemets au lieu de l'apostrophe employée avec les caractères. Les deux déclarations qui suivent sont donc équivalentes :

```
char szMyName[] = "Stephen";
char charMyName[] =
            {'S', 't', 'e', 'p', 'h', 'e', 'n', 0};
```

La syntaxe utilisée ici est une convention de nom dont C++ n'a que faire. Le préfixe `sz` est une autre convention pour désigner une *chaîne terminée par un zéro*.

La chaîne de caractères `Stephen` contient huit caractères et non sept, car le caractère nul après le `n` est pris en compte. De ce fait, la chaîne `""` contenant ce même caractère a une longueur égale à 1.

Manipuler des chaînes de caractères

Un programmeur en C++ est souvent amené à manipuler des chaînes. Le C++ est doté d'un certain nombre de fonctions de manipulation de chaînes qui lui facilitent la tâche. Le Tableau 7.1 en décrit quelques unes.

Tableau 7.1 : Fonctions de manipulation de chaînes.

Nom	Opération
`int strlen(chaîne)`	Retourne le nombre de caractères dans une chaîne.
`void strcpy(cible, source, n)`	Copie la chaîne source dans un tableau cible.

Tableau 7.1 : Fonctions de manipulation de chaînes. (*suite*)

Nom	Opération
`void strcat(cible, source, n)`	Ajoute la chaîne source à la fin de la chaîne cible.
`void strncpy(cible, source, n)`	Copie jusqu'à *n* caractères de la chaîne source dans un tableau cible.
`void strncat(cible, source, n)`	Ajoute au plus *n* caractères de la chaîne source à la fin de la chaîne cible.
`int strnstr(chaîne, motif, n)`	Trouve la première occurrence d'une chaîne servant de motif dans une autre.
`int strncmp(source1, source2, n)`	Compare les *n* premiers caractères de deux chaînes. Retourne 0 si les deux chaînes sont identiques.
`int strnicmp(source1, source2)`	Compare deux chaînes sans tenir compte de la casse.

Vous devez ajouter la directive `#include <strings.h>` au début de tout programme qui utilise une fonction dont le nom débute par *str*.

Le standard ANSI C++ actuel suggère d'éviter le recours aux fonctions *str*. Pour lui, ces fonctions sont périmées. Elles sont donc conservées pour l'instant, mais elles pourraient disparaître un de ces jours. C'est pourquoi la directive ci-dessus utilise l'ancienne forme d'extension `.h`. Le standard ANSI conseille l'usage du type `string` (voir ci-dessous). Mais cela n'empêche pas un grand nombre de programmes de continuer à appeler ces fonctions.

Écriture d'une fonction de concaténation

Il est temps de passer à la pratique. Le programme qui suit *concatène* (autrement dit, met bout à bout) deux chaînes entrées au clavier pour en construire une nouvelle.

```
// Concatenate - concaténation de deux chaînes
//               séparées par un " - ".
#include <cstdio>
#include <cstdlib>
```

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

// Le fichier include suivant est obsolète mais
// nécessaire pour les fonctions de type str.
#include <strings.h>

int main(int nNumberofArgs, char* pszArgs[])
{
    // lecture de la première chaîne...
    char szString1[256];
    cout << "Entrez la chaine 1 : ";
    cin >> szString1;
    // une alternative plus sûre serait
    // cin.getline(szString1, 128);

    // ...puis de la deuxième chaîne...
    char szString2[128];
    cout << "Entrez la chaine 2 : ";
    cin >> szString2;
    // une alternative plus sûre serait
    // cin.getline(szString1, 128);

    // on définit un tampon pour y placer les deux chaînes
    char szString[260];

    // on copie la première chaîne dans le tampon...
    strncpy(szString, szString1, 128);

    // ...on lui ajoute le caractère " - "...
    strncat(szString, " - ", 4);

    // ...puis la seconde chaîne...
    strncat(szString, szString2, 128);

    // ...et on affiche le résultat
    cout << "\n" << szString << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Ce programme lit deux chaînes de caractères et il les colle en insérant un "- " au milieu.

Attention à l'ordre des arguments dans les fonctions `str` : pour n'importe qui de censé, elles sont placées à l'envers. Lisez-les de droite à gauche. Ainsi, `strncat(cible, source, n)` place la seconde chaîne, `source`, à la fin de la première, `cible`.

À titre indicatif, voici ce que ce programme peut afficher :

```
Entrez la chaine 1 : Chien
Entrez la chaine 2 : Medor

Chien - Medor
Appuyez sur une touche pour continuer...
```

Le programme commence par lire une chaîne tapée au clavier. L'instruction `cin >> szString1` prend fin dès qu'elle rencontre un type quelconque d'espace blanc. Les caractères situés avant le premier espace sont placés dans la première chaîne, les suivants étant mis en attente dans le tampon du clavier, puis affectés à l'instruction `cin >>` qui suit. Autrement dit, si vous entrez "Le chien", la première chaîne aura comme contenu `Le` et la seconde recevra directement `chien`, pour afficher en sortie `Le - chien`.

L'extracteur `cin >>` ne sait rien de la longueur de la chaîne. Il est parfaitement capable de lire des milliers de caractères à la suite et de les envoyer dans `szString1`. C'est donc une instruction potentiellement dangereuse, du genre qu'affectionnent les pirates pour placer un virus dans votre programme.

C++ fournit diverses solutions à la plupart des problèmes de débordement de chaînes de caractères. Par exemple, la fonction `getline()` demande la saisie d'une ligne de texte tout en acceptant en argument la longueur de la chaîne (en ignorant pour l'instant la syntaxe un peu bizarre `cin.`) :

```
cin.getline(chaîne, longueurDeLaChaîne);
```

De même, les fonctions `strncpy()` et `strncat()` peuvent recevoir en argument la longueur du tampon de la cible. L'appel `strncpy(szString, szString1, 128)` peut se lire : "Copier les caractères de `szString1` dans `szString2` jusqu'à ce que le caractère nul soit rencontré ou sinon que l'on ait copié 128 caractères". Cette limite ne sera donc atteinte que si `szString1` comporte au moins 128 caractères.

Les fonctions `str...()` comportent deux versions : avec ou sans argument de longueur maximale. La seconde forme est intéressante si vous ne connaissez

pas la taille du tampon, mais elle parfaitement capable d'écrire au-delà de la fin de la chaîne cible.

Variables de type String

Le standard ANSI C++ comprend un type `string` qui est conçu pour faciliter la manipulation des chaînes de caractères.

Le terme *chaîne* fait référence à un tableau de caractères terminé par \0, tandis que *type chaîne* indique une variable de type `string`. Vous sentez la différence ? Le type `string` est associé à des fonctions spécifiques pour copier, concaténer, capitaliser et effectuer diverses manipulations presque magiques. Il évite les problèmes de débordement inhérents aux chaînes. Ces fonctions sont définies dans le fichier inclus `<string>`.

Le listing ci-dessous propose une variante du programme de concaténation précédent basée sur le type `string` :

```
// StringConcatenate - concaténation de deux chaînes
//                     séparées par un " - ".
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // lecture de la première chaîne...
    string string1;
    cout << "Entrez la chaine 1 : ";
    cin >> string1;

    // ...puis de la deuxième chaîne...
    string string2;
    cout << "Entrez la chaine 2 : ";
    cin >> string2;

    // accumule les deux chaînes dans un tampon
    string buffer;
    string divider = " - ";
    buffer = string1 + divider + string2;

    // ...et on affiche le résultat
```

```
    cout << "\n" << buffer << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Cette fonction de concaténation définit deux variables, `string1` et `string2` (bien vu). Une variable de type `string` n'a pas de longueur fixe. Elle peut être réduite ou étendue selon le nombre de caractères qu'elle contient (dans la limite de la mémoire disponible ou jusqu'à ce que les poules aient des dents, selon l'événement qui se produit en premier). Non seulement vous n'avez pas à vous soucier de la taille du tableau de caractères cible mais de surcroît, vous n'avez pas à craindre qu'un utilisateur plante votre programme en entrant trop de caractères. Le programme `StringConcatenate` manipule donc les variables de type `string` sans se poser de questions existentielles.

Remarquez que certaines opérations n'ont pas tout à fait le même sens que leur équivalent arithmétique. Par exemple, ajouter des variables de type `string` signifie qu'elles sont concaténées. De plus, C++ peut convertir une chaîne terminée par le caractère nul en une variable de type `string` sans instruction explicite.

À la différence de `int` ou `float`, le type `string` n'est pas intrinsèque à C++. Cela signifie que les opérations portant sur ce type ne font pas partie de la syntaxe du langage, mais sont définies dans un fichier inclus : `string`. Nous retrouverons la *classe* `string` dans le Chapitre 27. La présentation effectuée ici a pour but de vous montrer que le type `string` est souvent plus simple à utiliser que les manipulations de tableaux de caractères terminés par un nul (non, non, je ne parlais pas de vous).

Chapitre 8

Premier coup d'œil sur les pointeurs C++

Dans ce chapitre :

- Adresser des variables dans la mémoire.
- Déclarer et utiliser des variables de pointeur.
- Reconnaître les risques inhérents aux pointeurs.
- Passer des pointeurs à des fonctions.
- Heap, hop, le tas !

Comparé à d'autres langages de programmation, le C++ est relativement conventionnel. Certains langages informatiques manquent d'opérateurs logiques (voir Chapitre 4). Le C++ a une syntaxe unique qui lui est propre ; il fait vraiment bande à part, notamment à cause de l'utilisation des variables de pointeur. Les *pointeurs* sont des variables qui "pointent vers" d'autres variables. Autrement dit, les pointeurs contiennent l'adresse d'un emplacement en mémoire.

Ce chapitre présente le type de variable pointeur. Il commence par présenter le concept général, puis décrit la syntaxe d'un pointeur et propose quelques bonnes raisons d'adopter les pointeurs.

Taille des variables

La taille d'une variable n'a rien à voir avec celle d'un être humain. On ne la mesure d'ailleurs pas en mètres et centimètres, mais en octets ou en bits. Le programme qui suit vous permet d'afficher la taille de différents types de variables, autrement dit l'espace mémoire dont ils ont besoin :

```
// VariableSize - affiche la taille de chaque
//                type de variable
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    bool   b;
    char   c;
    int    n;
    long   l;
    float  f;
    double d;

    cout << "taille d'un bool   = "  << sizeof b << endl;
    cout << "taille d'un char   = " << sizeof c << endl;
    cout << "taille d'un int    = " << sizeof n << endl;
    cout << "taille d'un long   = " << sizeof l << endl;
    cout << "taille d'un float  = " << sizeof f << endl;
    cout << "taille d'un double = " << sizeof d << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La fonction `sizeof()` est une construction C++ qui retourne la taille de son argument en octets. Le programme ci-dessus génère l'affichage suivant :

```
taille d'un bool   = 1
taille d'un char   = 1
taille d'un int    = 4
taille d'un long   = 4
taille d'un float  = 4
taille d'un double = 8
Appuyez sur une touche pour continuer...
```

Ne vous inquiétez pas si vous utilisez un autre compilateur que Dev-C++ et si les résultats que vous obtenez sont différents. Le programme peut par exemple vous dire que le type `int` est plus petit que le type `float`. C++ n'indique pas quelle *doit* être la taille exacte d'un certain type de variable. Il vous dit simplement qu'un `long` occupe le même espace qu'un `int` ou qu'il est plus grand.

Même chose pour un `double` par rapport à un `float`. Les renseignements renvoyés par le programme précédent sont caractéristiques d'un processeur 32 bits (ce qui est le cas de la famille des processeurs Pentium).

Qu'est-ce qu'une adresse ?

"Il faut bien que quelqu'un soit quelque part" dit-on. C'est pourquoi chaque variable C++ est stockée quelque part dans la mémoire de l'ordinateur. La mémoire vive est constituée d'octets qui possèdent chacun leur propre adresse numérotée 0, 1, 2, et ainsi de suite.

Une variable `intToto` peut se trouver à l'adresse 0x100 tandis que la variable `floatLecteur` pourrait par exemple être à l'emplacement 0x180 (par convention, les adresses en mémoire sont exprimées en hexadécimal). Ce n'est bien sûr qu'une image et seul l'ordinateur connaît exactement cette information, et encore n'est-elle valable qu'à l'instant où le programme est exécuté.

Opérateurs d'adressage

Deux opérateurs concernent les pointeurs. Ils sont décrits dans le Tableau 8.1. L'opérateur `&` signifie en gros "Dis-moi ton adresse", tandis que `*` peut se traduire par "Ton adresse est".

Tableau 8.1 : Opérateurs pour les pointeurs.

Opérateur	Signification
& (unaire)	L'adresse de
* (unaire)	Dans une expression : la chose pointée par
* (unaire)	Dans une déclaration : pointeur vers

Le programme ci-dessous montre comment utiliser l'opérateur `&` pour afficher la disposition des variables en mémoire :

```
// Layout - Ce programme essaie de donner une idée
//          de l'utilisation de la mémoire locale
//          pour un certain compilateur.
#include <cstdio>
#include <cstdlib>
```

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

int main(int nNumberofArgs, char* pszArgs[])
{
    int    end;
    int    n;
    long   l;
    float  f;
    double d;

    // affichage en mode hexadécimal
    cout.setf(ios::hex);
    cout.unsetf(ios::dec);

    // affiche l'adresse de chaque variable
    // pour avoir une idée de la façon dont
    // les variables sont mémorisées
    cout << "--- = 0x" << &end << "\n";
    cout << "&n  = 0x" << &n  << "\n";
    cout << "&l  = 0x" << &l  << "\n";
    cout << "&f  = 0x" << &f  << "\n";
    cout << "&d  = 0x" << &d  << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le programme déclare une série de variables. Il applique ensuite l'opérateur `&` pour déterminer où se trouve chacune de ces variables en mémoire. Voici à quoi peut ressembler l'affichage résultant :

```
--- = 0x0x22ff6c
&n  = 0x0x22ff68
&l  = 0x0x22ff64
&f  = 0x0x22ff60
&d  = 0x0x22ff58
Appuyez sur une touche pour continuer...
```

Vous obtiendrez forcément un résultat différent. L'adresse réelle des variables dépend d'une multitude de facteurs. Elle change même d'une exécution à une autre.

Remarquez que la variable `n` se trouve exactement quatre octets plus loin que la première variable déclarée (`end`). La suivante, `l`, est aussi à quatre octets de distance. Quant au `double` appelé `d`, il est décalé de huit octets par rapport à son voisin immédiat, `f`. Chaque variable se voit allouer l'espace exact nécessité par son type.

Aucune règle n'impose au compilateur C++ de "chaîner" les variables en mémoire sans aucun espacement entre elles. Dev-C++ aurait parfaitement pu les répartir autrement.

Utiliser les variables de pointeur

Une *variable de pointeur* est une variable qui contient une adresse, généralement celle d'une autre variable. Prenons un exemple : si je dis à ma femme que je serai au bureau jusqu'à 20 heures, elle devient une variable de pointeur. Il suffit que quelqu'un lui demande vers 19 heures où je suis, elle pourra répondre sans hésitation. L'histoire ne dit pas si je lui ai menti...

Les deux lignes ci-dessous montrent comment utiliser les opérateurs décrits dans le Tableau 8.1 pour représenter l'exemple précédent :

```
maFemme = &monBureau;  // Dit à maFemme quel est mon horaire
auTravail = *maFemme;      // "Mon mari est au bureau jusqu'à"
```

Plus sérieusement :

```
void fn()
{
   int  intVar;
   int* pintVar;

   pintVar  = &intVar;  // pintVar pointe maintenant vers intVar
   *pintVar = 10;       // stocke 10 à l'emplacement int
                        // vers lequel pointe pintVar.
}
```

Ici, la fonction `fn()` commence par la déclaration de `intVar`. L'instruction suivante déclare la variable `pintVar` comme variable de pointeur de type `int` (à propos, `pintVar` se prononce "p", "int", "Var" et non "pinte", "Var").

Les variables de pointeur sont déclarées comme les variables normales, sauf en ce qui concerne l'ajout du caractère unaire `*`. Ce dernier peut apparaître

n'importe où entre le nom de type de base – `int` en l'occurrence – et le nom de la variable ; il est cependant devenu de plus en plus courant d'ajouter l'astérisque à la fin du type de variable.

De nombreux programmeurs adoptent une convention dans laquelle le premier caractère du nom d'une variable indique son type, comme `n` pour `int`, `d` pour `double`, et ainsi de suite. En vertu de ce principe, ils placent un `p` au début d'un nom de variable de pointeur.

Dans une expression, l'opérateur unaire `&` signifie "adresse de". C'est pourquoi, dans la première instruction, l'adresse de `intVar` est stockée dans `pintVar`.

Pour rendre cela plus concret, supposons que la mémoire de la fonction `fn()` commence à l'emplacement 0x102 et que `pintVar` se trouve à 0x106. La disposition est ici plus simple que le résultat produit par le programme `Layout`, mais le concept est le même.

La première affectation stocke la valeur de `& intVar` (0x102) dans la variable de pointeur `pintVar`. La deuxième stocke 10 à l'emplacement pointé par `pintVar`. Cette valeur 10 est placée à l'adresse contenue dans `pintVar`, soit 0x102 (l'adresse de `intVar`).

Adresse informatique et adresse postale

Un pointeur est comparable à une adresse postale. À l'instar de votre domicile dont l'adresse est unique, chacun des octets de la mémoire a une adresse qui lui est propre. Une adresse postale est faite de numéros et de mots. La mienne est 123 Rue Principale (c'est bien sûr une adresse fictive, je n'ai pas envie de voir des rôdeurs autour de ma maison, sauf si ce sont des femmes). Une adresse mémoire est uniquement faite de chiffres, comme 123456. Pour des raisons de commodité, elle est généralement exprimée en hexadécimal, mais l'emplacement réel est tout à fait immatériel.

Vous pouvez stocker un canapé au 123 Rue Principale, tout comme vous pouvez stocker un nombre à l'adresse 0x123456. De même, vous pouvez inscrire une adresse, disons le 123 Rue Principale, sur un petit bout de papier que vous collerez sur le canapé. En fait, c'est ainsi que travaillent les livreurs : ils apportent l'objet à l'adresse qui leur a été indiquée sur le bon de commande, qu'il s'agisse du 123 Rue Principale ou de toute autre adresse.

En C++, cette livraison s'exprimerait ainsi :

```
Domicile monDomicile
Domicile* adresseDomicile;
adresseDomicile = &monDomicile;
*adresseDomicile = canapé;
```

En langage humain, vous diriez que monDomicile est le lieu où j'habite, que adresseDomicile est l'adresse postale de ce lieu. Vous affectez l'adresse de monDomicile au pointeur de Domicile, c'est-à-dire adresseDomicile. Puis vous stockez un canapé dans le domicile situé à l'adresse figurant dans adresseDomicile.

Ceci dit, voyons la version int et int* de notre petit morceau de code précédent :

```
int monInt;
int* intAdresse;
intAdresse = &monInt;
*intAdresse = 10;
```

Dans ce listing, monInt est un entier int et intAdresse un pointeur vers une valeur de type int. L'adresse de monInt est affectée au pointeur intAdresse. Et finalement, 10 est affecté au int sur lequel pointe intAdresse.

Utiliser différents types de pointeurs

Chaque expression possède un type et contient une valeur. Le type d'expression de intVar est un pointeur vers un entier, écrit sous la forme int*. La comparaison avec la déclaration de pintVar révèle que les types concordent parfaitement :

```
int* pintVar = &intVar;   // des deux côtés, l'affectation
                          // est de type int*
```

De même, puisque que pintVar est de type int*, le type de *pintVar est int :

```
*pintVar = 10;   // des deux côtés, l'affectation est
                 // de type int
```

L'objet pointé par `pintVar` est de type `int`. Cela revient à dire que si `adresseDomicile` est l'adresse d'une maison, l'objet pointé par `adresseDomicile` doit être une maison. Incroyable mais vrai !

Les pointeurs vers d'autres types de variables sont exprimés de la même façon :

```
double doubleVar;
double* pdoubleVar = &doubleVar;
*pdoubleVar = 10.0;
```

Le pointeur d'un ordinateur de type Pentium utilise quatre octets, quel que soit l'objet vers lequel il pointe. Autrement dit, sur un Pentium, la longueur de l'adresse est de quatre octets. Point.

Il est extrêmement important de faire correspondre les types des pointeurs. Voyez ce que donnerait ce listing si la syntaxe ci-dessous était admise :

```
int   n1;
int*  pintVar;
pintVar  = &n1;
*pintVar = 100.0;
```

La deuxième affectation tente de stocker la valeur double 100.0, codée sur huit octets, dans l'espace de quatre octets alloué à `n1`. En fait, ce n'est pas aussi tragique qu'il y paraît car le C++ est suffisamment tolérant pour rétrograder la constante 100.0 en un entier de type `int` avant d'exécuter l'instruction.

Il est possible de convertir explicitement un type de variable en un autre :

```
int iVar;
double dVar = 10.0;
iVar = (int)dVar;
```

De même, il est possible de convertir explicitement un type de pointeur en un autre :

```
int* piVar;
double dVar = 10.0;
double* pdVar;
piVar = (int)pdVar;
```

Imaginez cependant la catastrophe qui pourrait survenir si ce type de conversion explicite de pointeur n'était pas rigoureux. L'enregistrement d'une variable dans une zone de taille inadéquate entraînerait la disparition de quasiment toutes les variables environnantes. Ce phénomène est démontré graphiquement par le programme `LayoutError` que voici :

```
// LayoutError - Démontre les résultats d'un
//               usage inadéquat d'un pointeur
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    int    upper = 0;
    int    n     = 0;
    int    lower = 0;

    // affiche les valeurs initiales des trois variables...
    cout << "Les valeurs initiales sont :" << endl;
    cout << "upper = " << upper << endl;
    cout << "n     = " << n     << endl;
    cout << "lower = " << lower << endl;

    // stockage d'un double dans un espace
    // alloué à un entier
    cout << "\nEnregistrement de 13.0 dans &n" << endl;
    double* pD = (double*)&n;
    *pD = 13.0;

    // affiche les résultats
    cout << "\nLes valeurs finales sont :" << endl;
    cout << "upper = " << upper << endl;
    cout << "n     = " << n     << endl;
    cout << "lower = " << lower << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Les trois premières lignes de `main()` déclarent trois entiers de la manière habituelle. Ces trois variables sont supposées se suivre en mémoire.

Les trois lignes exécutables qui suivent affichent les valeurs de ces variables qui, ce n'est pas surprenant, sont égales à 0. L'affectation `*pD = 13.0;` stocke la valeur en double précision 13.0 dans la variable entière `n`. Nous affichons alors les mêmes variables après cette affectation.

Après avoir affecté la valeur double 13.0 dans la variable entière `n`, `n` lui-même n'est en rien modifié. La variable `upper` reçoit cependant des informations incohérentes. Ce qui n'est pas bien du tout :

```
Les valeurs initiales sont :
upper = 0
n     = 0
lower = 0

Enregistrement de 13.0 dans &n

Les valeurs finales sont :
upper = 1076494336
n     = 0
lower = 0
Appuyez sur une touche pour continuer...
```

Un équivalent "voyage d'affaires" pourrait s'écrire ainsi :

```
Domicile* adresseDomicile = &"123, Rue Principale";
Hotel* adresseHotel;
adresseHotel = (Hotel*)adresseDomicile;
*adresseHotel = LeRitz;
```

La variable `adresseDomicile` est initialisée pour pointer vers mon domicile. La variable `adresseHotel` pointe vers un hôtel. Maintenant, l'adresse du domicile est explicitement convertie en adresse d'hôtel et enregistrée comme telle. Enfin, `LeRitz` est lâché sur mon domicile. Comme cet hôtel est plus grand que mon habitation (disons, un peu plus grand), il n'est pas étonnant qu'il écrase aussi les maisons de mes voisins.

Le type de pointeur évite au programmeur de placer un objet dans un espace trop grand ou trop petit. L'affectation `*pintVar = 100.0;` ne cause en fait aucun problème car le C++ sait que `pintVar` pointe vers un entier `int` ; il sait qu'il doit rétrograder 100.0 en un entier `int` avant de procéder à l'affectation.

Passer des pointeurs à des fonctions

Les variables de pointeur servent aussi à passer des arguments à des fonctions. Pour en comprendre l'importance, il faut comprendre comment les arguments sont transmis à une fonction.

Passage par valeur

Vous avez peut-être remarqué qu'il n'est normalement pas possible de modifier la valeur d'une variable passée à une fonction à l'intérieur de celle-ci. Examinons ce fragment de code :

```
void fn(int intArg)
{
   intArg = 10;
   // Ici, la valeur de intArg est 10.
}

void parent(void)
{
   int n1 = 0;
   fn(n1);
   // Ici, la valeur de n1 est 0.
}
```

Dans ce listing, la fonction `parent()` initialise la variable entière `n1` à 0. La valeur de `n1` est ensuite passée à `fn()`. Quand il entre dans cette fonction, `intArg` est égal à 0, la valeur passée. Puis `fn()` change la valeur de `intArg` avant de retourner à `parent()`. C'est peut-être étonnant, mais en retournant à `parent()`, la valeur de `n1` est encore à 0.

Cela provient du fait que le C++ ne passe pas la variable elle-même, mais la valeur qu'elle contient au moment de l'appel. L'expression est donc évaluée (même s'il ne s'agit que d'un nom de variable) *puis* le résultat obtenu est passé à la fonction.

Même quand on dit "passer la variable `x` à la fonction `fn()`", il faut comprendre que c'est la *valeur* de l'expression `x` qui est passée.

Passage des valeurs de pointeur

Comme tout autre type intrinsèque, un pointeur peut être passé à une fonction en tant qu'argument :

```
void fn(int* pintArg)
{
   *pintArg = 10;
}

void parent(void)
{
   int n = 0;

   fn(&n);   // Passe l'adresse de n
             // La valeur de n est maintenant 10
}
```

Dans ce cas, l'adresse de `n` est passée à la fonction `fn()` à la place de la valeur de `n`. La signification de cette différence saute aux yeux si vous regardez l'affectation que contient `fn()`.

Supposons que `n` se trouve à l'adresse 0x102. Au lieu de la valeur 0, l'appel `fn(&n)` passe la valeur 0x102. À l'intérieur de `fn()`, l'affectation `*pintArg = 10` stocke le nombre 10 dans la variable `int` située à l'emplacement 0x102, remplaçant ainsi la valeur précédente, 0. Quand on revient à `parent()`, la valeur de `n` est 10 parce que `&n` est tout simplement un autre nom pour 0x102.

Passage par référence

Le C++ propose un raccourci pour ce qui précède ; il évite d'avoir à utiliser les pointeurs eux-mêmes. Dans l'exemple qui suit, la variable `n` est passée par référence.

Lors d'un passage par référence, la fonction parent passe une référence à une variable au lieu d'une valeur. Le mot *référence* est un synonyme de "adresse".

```
void fn(int& intArg)
{
   intArg = 10;
}
```

```
void parent(void)
{
   int n = 0;
   fn(n)
           // ici, la valeur de n est 10.
}
```

Ici, c'est une référence à `n`, et non sa valeur, qui est reçue en argument par la fonction `fn()`. Celle-ci stocke la valeur 10 à l'emplacement `int` référencé par `intArg`.

Remarquez qu'une référence n'est pas un vrai type. C'est pourquoi le nom complet de la fonction est `fn(int)` et non `fn(int&)`.

Utiliser un bloc de mémoire appelé le heap

Le *heap* (ou *tas*) est un bloc de mémoire non structuré auquel le programme peut avoir accès si nécessaire. Cette section explique ce qu'ils sont, pourquoi ils existent et comment les utiliser.

Visual C++ .NET permet au programmeur d'écrire du code dans un mode où le compilateur gère directement l'allocation et la désallocation de la mémoire. Il est le seul actuellement à supporter ce mode. Les questions évoquées dans cette section ne concernent donc pas Visual C++ .NET. Mais c'est là un sujet qui dépasse notre propos.

De même qu'il est possible de passer un pointeur à une fonction, il est aussi possible pour une fonction de retourner un pointeur. Une fonction qui retourne l'adresse d'un `double` est déclarée ainsi :

```
double* fn(void);
```

Il faut cependant être très prudent lors du retour d'un pointeur. Pour comprendre les risques, vous devez savoir ce qu'est la *portée* d'une variable (non, il ne s'agit pas d'une portée de musique, encore moins d'une portée de chiens ou de chats).

Limiter la portée

En plus de leur valeur et de leur type, les variables C++ ont une propriété appelée portée. La *portée* est la plage dans laquelle une variable est définie.

Étudions cet extrait de code :

```
// Scope - Pour comprendre la portée
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// La variable qui suit est accessible à toutes
// les fonctions. Elle est définie aussi
// longtemps que le programme est en cours
// d'exécution (portée globale).
int intGlobal;

// La variable intChild n'est accessible qu'à
// la fonction child(). Elle n'est définie que pendant
// le temps où le C++ exécute child() ou une
// fonction dans laquelle child() effectue un
// appel (portée de fonction).
void child(void)
{
   int intChild;
}

// La variable suivante intParent a une
// portée de fonction.
void parent(void)
{
   int intParent = 0;
   child();

   int intPlusTard = 0;
   intParent = intPlusTard;
}

int main(int nArgs, char* pArgs[])
{
   parent();

   // attend pour terminer le programme que l'utilisateur
   // lise le contenu de la fenêtre puis appuie sur une touche
```

```
    system("PAUSE");
    return 0;
}
```

L'exécution commence avec `main()`. La fonction `main()` appelle aussitôt `parent()`. La déclaration de `intParent` est la première chose que le processeur détecte dans cette fonction. À ce point, `intParent` arrive à portée, c'est-à-dire qu'il est défini et disponible pour le restant de l'utilisation de la fonction `parent()`.

La deuxième instruction de `parent()` est un appel à `child()`. Là encore, la fonction `child()` déclare une variable locale, `intChild` en l'occurrence. La variable `intChild` est alors à portée de `child()`. Techniquement, `intParent` ne se trouve pas à portée de `child()` parce que ce dernier n'a pas accès à `intParent` ; la variable `intParent` continue néanmoins d'exister.

Quand `child()` se termine, la variable `intChild` se met hors de portée ; la variable `intChild` est non seulement inaccessible, mais elle cesse d'exister (car la mémoire occupée par `intChild` est rendue au pot commun afin d'être utilisée pour autre chose).

L'exécution de `parent()` se poursuit. La variable `intPlusTard` arrive à portée au moment de sa déclaration. Lorsque `parent()` se termine à son tour et que l'exécution du programme retourne à `main()`, `intParent` et `intPlusTard` cessent d'être à portée. On peut déclarer une variable hors de toute fonction ; ce type de variable, dite *globale*, reste disponible durant toute l'exécution du programme.

Comme `intGlobal` est déclarée globalement dans cet exemple, elle est utilisable par les trois fonctions et le reste pendant toute la durée de vie du programme.

Un problème de portée

Le segment de code ci-dessous est compilé sans qu'aucune erreur n'ait été détectée, mais il ne fonctionne pour autant pas :

```
double* child(void)
{
    double dLocalVariable;
    return &dLocalVariable;
}
```

```
void parent(void)
{
   double* pdLocal;
   pdLocal = child();
   *pdLocal = 1.0;
}
```

Le problème de cette fonction réside dans le fait que `dLocalVariable` n'est défini que dans la portée de la fonction `child()`. Par conséquent, au moment où l'adresse en mémoire de `dLocalVariable` est retournée par `child()`, la fonction se réfère à une variable qui n'existe plus. La partie de mémoire que `dLocalVariable` occupait auparavant est probablement utilisée à d'autres fins.

Il s'agit là d'une erreur très fréquente qui peut surgir d'un tas de façons différentes. Elle n'entraîne malheureusement pas un arrêt immédiat du programme. Pire, celui-ci peut fonctionner parfaitement la plupart du temps, en fait aussi longtemps que la mémoire occupée auparavant par `dLocalVariable` n'est pas réutilisée. Ces problèmes intermittents sont les plus difficiles à résoudre.

La solution proposée par le heap

Le problème de portée provient du fait que le C++ a retourné la mémoire définie localement avant que le programmeur soit prêt. Ce qu'il faudrait donc, c'est un bloc de mémoire contrôlé par le programmeur, susceptible d'allouer de la mémoire et de la reprendre lorsqu'il le juge utile, et non parce que le C++ en a ainsi décidé. Un tel bloc de mémoire est appelé le *heap*.

La mémoire heap est allouée grâce à la commande `new` suivie par le type de l'objet à allouer. Par exemple, le listing ci-dessous alloue une variable `double` sur le heap :

```
double* child(void)
{
   double* pdLocalVariable = new double;
   return pdLocalVariable;
}
```

Bien que la variable `pdLocalVariable` cesse d'être à portée lorsque la fonction `child()` se termine, il n'en va pas de même pour la mémoire à laquelle `pdLocalVariable` se réfère. Un emplacement de mémoire retourné par `new` ne se trouve

pas hors de portée s'il n'est pas explicitement rendu au heap à l'aide de la commande de suppression `delete` :

```
void parent(void)
{
   // child() retourne l'adresse d'un bloc.
   // de mémoire heap
   double * pdMyDouble = child();

   // stockage d'une valeur.
   *pdMyDouble = 1.1;

   //...

   // retourne maintenant la mémoire au heap.
   delete pdMyDouble;
   pdMyDouble = 0;

   //...
}
```

Ici, le pointeur retourné par `child()` est utilisé pour stocker une valeur `double`. Une fois que la fonction en a fini avec l'emplacement en mémoire, celui-ci est retourné au heap. La fonction `parent()` définit le pointeur à 0 une fois que la mémoire heap a été rendue ; ce n'est pas une obligation, mais c'est vivement recommandé. Car si le programmeur tentait malencontreusement de stocker quoi que soit dans `*pdMyDouble` après la commande `delete`, le programme planterait aussitôt.

Un programme qui plante immédiatement après avoir rencontré une erreur est plus facile à corriger qu'un autre programme dont le comportement est imprévisible.

Chapitre 9

Deuxième coup d'œil sur les pointeurs C++

Dans ce chapitre :

- Effectuer des opérations mathématiques sur des pointeurs de caractères.
- Examen de la relation entre pointeurs et tableaux.
- Application de cette relation pour augmenter les performances du programme.
- Extension des opérations de pointeur à différents types de pointeur.
- Comprendre les arguments de `main()` dans notre modèle de programme C++.

Le C++ permet au programmeur d'agir sur les variables de pointeur à peu près comme s'il s'agissait de simples types de variables (le concept de variable de pointeur a été présenté dans le Chapitre 8). C'est un sujet délicat, avec de multiples implications. Et c'est l'objet de ce chapitre.

Définir des opérations sur des pointeurs de variable

Quelques-uns des opérateurs étudiés dans le Chapitre 3 sont applicables aux pointeurs. Cette section s'intéresse aux conséquences de cette application aussi bien sur les pointeurs que sur les tableaux (ces derniers ont été présentés dans le Chapitre 7). Le Tableau 9.1 énumère les trois opérations fondamentales qui peuvent être définies sur des pointeurs.

Ici, `offset` est de type `long` (bien qu'absents du Tableau 9.1, les opérateurs d'addition et de soustraction comme `++` et `+=` sont eux aussi supportés).

Tableau 9.1 : Les trois opérations définissables pour les types de pointeur.

Opération	Résultat	Signification
`pointeur + offset` (décalage)	pointeur	Calcule l'adresse des entrées entières d'un objet `depuis le pointeur`.
`pointeur - offset` (décalage)	pointeur	Contraire de l'addition.
`pointeur1 - pointeur2`	décalage	Calcule le nombre d'entrées entre `pointeur1` et `pointeur2`.

Le parallèle avec l'immobilier, utilisé si efficacement (du moins, je pense) dans le Chapitre 8 est commode pour expliquer le fonctionnement des opérateurs mathématiques sur les pointeurs. Imaginons un pâté de maisons dans lequel tous les immeubles sont numérotés (on suppose bien sûr qu'ils se suivent un par un, et non par séries de numéros pairs ou impairs). La maison à côté du 123 Rue Principale sera le 124 Rue Principale (ou le 122, si vous marchez dans l'autre sens comme les gauchers et les Britanniques).

Il en découle bien évidemment que la quatrième maison à partir du 123 Rue Principale est au 127. Nous pourrions donc écrire `123 Principale + 4 = 127 Principale`. De même, si je veux savoir combien il y a de maisons du 123 Rue Principale au 127 Rue Principale, la réponse sera quatre : `127 Principale - 123 Principale = 4`. Soit dit en passant, une maison est à zéro maison d'elle-même puisque `123 Principale - 123 Principale = 0`.

Élargissons cette réflexion. Ajouter 123 Rue Principale et 127 Rue Principale n'aurait aucun sens. De la même manière, vous ne pouvez pas additionner deux adresses en C++. Pas plus d'ailleurs que les multiplier, les diviser, les mettre au carré, en extraire la racine carrée, et ainsi de suite.

Revoir les tableaux à la lumière des variables de pointeur

Revenons à l'étrange et mystérieux univers des tableaux. Une fois de plus, je pense à mon voisinage. Un tableau ressemble à un pâté de maisons. Chacun de ses éléments correspond à une habitation du bloc. Mais là, les éléments du tableau sont comparés au nombre de maisons à partir du coin de la rue. Supposons que la numérotation de ce bloc débute à 123. La maison suivante sera le

124, et ainsi de suite. Dans la langue des tableaux, `maisonRue[0]` sera donc égal à 0, `maisonRue[1]` sera égal à 124, etc.

Projetons-nous maintenant dans la mémoire de l'ordinateur. Prenons le cas d'un tableau de trente-deux caractères codés sur 1 octet, nommé `charArray`. Si le premier octet de ce tableau est stocké à l'adresse 0x110, ce tableau s'étendrait sur une plage allant de 0x110 à 0x12f. Comme `charArray[0]` se trouve à l'adresse 0x110, `charArray[1]` est à 0x111, `charArray[2]` à 0x112, et ainsi de suite.

Appliquons ce même modèle aux variables de type pointeur. Après exécution de l'expression :

```
ptr = &charArray[0];
```

le pointeur `ptr` contient l'adresse 0x110. L'ajout d'un décalage entier à un pointeur est défini de manière à ce que la relation illustrée dans le Tableau 9.2 soit vraie. Ce tableau montre aussi pourquoi l'ajout d'un décalage `n` à `ptr` calcule l'adresse du $n^{ième}$ élément de `charArray`.

Tableau 9.2 : Ajout de décalages (offset).

Décalage	Résultat	Correspond à
+0	0x110	`charArray[0]`
+1	0x111	`charArray[1]`
+2	0x1102	`charArray[2]`
...	...	...
+n	0x110+n	`charArray[n]`

L'ajout d'un décalage à un pointeur équivaut à utiliser un indice dans un tableau.

Si nous écrivons :

```
char* ptr = &charArray[0];
```

alors :

```
*(ptr + n) <- correspond à -> charArray[n];
```

Du fait que l'opérateur * est prioritaire sur l'addition, `*ptr + n` ajoute `n` au caractère vers lequel pointe `ptr`. Les parenthèses sont indispensables pour que l'addition soit faite avant l'indirection. L'expression `*(ptr + n)` récupère le caractère pointé par `ptr` plus le décalage `n`.

En fait, la correspondance entre les deux formes d'expression est si forte que, pour le C++, `array[n]` n'est rien de plus qu'une version simplifiée de `*(ptr + n)` dans laquelle `ptr` pointe vers le premier élément du tableau `array`.

```
array[n] -- est interprété par C++ comme -> *(&array[0] + n)
```

Pour terminer cette association, signalons que le C++ propose un autre raccourci. Si :

```
char charArray[20];
```

alors :

```
charArray est défini comme &charArray[20];
```

Autrement dit, le nom d'un tableau sans aucun indice est l'adresse du tableau lui-même. Nous pouvons donc simplifier l'association en :

```
array[n] --> interprété par le C++ en --> *(array + n)
```

Appliquer des opérateurs à l'adresse d'un tableau

La correspondance entre indices de tableau et arithmétique des pointeurs est un concept très utile (en aurais-je parlé si ce n'était pas le cas ?)

Par exemple, une fonction `displayArray()` servant à afficher le contenu d'un tableau d'entiers pourrait s'écrire de la manière suivante :

```
// displayArray - affiche les membres d'un tableau
//                de longueur nSize.
void displayArray (int intArray[], int nSize)
{
   cout << "Les valeurs du tableau sont :\n";

   for(int n=0; n < nSize; n++)
   {
      cout << n << " : " << intArray[n] << "\n";
   }
   cout << "\n";
}
```

Cette version utilise des opérations de tableau qui vous sont familières. La version "pointeur" de ce listing apparaît sous cette forme :

```
// displayArray - affiche les membres d'un tableau
//                de longueur nSize.
void displayArray (int intArray[], int nSize)
{
   cout << "Les valeurs du tableau sont :\n";

   int* pArray = intArray;
   for(int n=0; n < nSize; n++, pArray++)
   {
      cout << n << " : " << *pArray << "\n";
   }
   cout << "\n";
}
```

La nouvelle fonction `displayArray()` commence par créer un pointeur vers un entier (`pArray`) qui contient l'adresse du premier élément de `intArray`.

Le préfixe `p` du nom d'une variable indique que cette variable est un pointeur. Il s'agit uniquement d'une convention usuelle et non pas d'une obligation imposée par le langage C++.

La fonction parcourt ensuite chacun des éléments du tableau. À chaque boucle, `displayArray()` affiche l'entier courant, c'est-à-dire celui qui est pointé par `pArray` avant d'incrémenter ce dernier pour lire la prochaine entrée dans `intArray`. Cette fonction peut être testée grâce à la version de `main()` proposée ci-dessous :

```
int main(int nNumberofArgs, char* pszArgs[])
{
    int array[] = {4, 3, 2, 1};
    displayArray(array, 4);

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le listing complet est proposé en chargement sous le nom `DisplayArrayPnt`.

Voici ce qu'affiche ce programme :

```
Les valeurs du tableau sont :
0 : 4
1 : 3
2 : 2
3 : 1

Appuyez sur une touche pour continuer...
```

Une telle conversion peut vous paraître toute bête, mais il se trouve que la version "pointeur" de `displayArray` est plus courante que la version "tableau". Pour d'obscures raisons, les programmeurs ne semblent pas aimer les tableaux.

L'utilisation de pointeurs pour accéder à des éléments de tableaux n'est nulle part aussi courante que dans les tableaux de caractères.

Étendre les opérations de pointeur à une chaîne

Une chaîne est simplement un tableau de caractères dont le dernier caractère est un nul. Le C++ utilise ce caractère nul comme terminateur. Ce style de tableau est quasiment un type de variable en soi (reportez-vous au Chapitre 7 pour en apprendre plus sur les tableaux de chaîne). Les programmeurs en C++ utilisent souvent les pointeurs de caractères pour manipuler de telles chaînes. Les exemples de code qui suivent ont pour but de comparer cette technique à celle évoquée précédemment, basée sur les indices des tableaux.

Les pointeurs de caractères bénéficient des mêmes relations avec un tableau de caractères que celles entretenues par tout autre couple pointeur/tableau. Cependant, le fait que les chaînes s'achèvent par un terminateur nul les rend particulièrement aptes à la manipulation par pointeur. C'est ce que nous montre l'exemple de programme d'affichage de chaîne proposé ci-dessous :

```
// DisplayString - affiche un tableau de caractères via un
//                 pointeur et via les indices du tableau
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // déclare une chaîne
    char* szString = "Randy";
    cout << "La chaîne contient '" << szString << "'" << endl;

    // affiche szString en tant que tableau
    cout << "Affiche la chaîne en tant que tableau : ";
    for(int i = 0; i < 5; i++)
    {
      cout << szString[i];
    }
    cout << endl;

    // affiche la chaîne en utilisant un pointeur
    cout << "Affiche la chaîne en utilisant un pointeur : ";
    char* pszString = szString;
    while(*pszString)
    {
      cout << *pszString;
      pszString++;
    }
    cout << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La première boucle affiche la chaîne `szString` en parcourant sa plage d'indices. Elle s'arrête au bout de cinq parcours (ce qui correspond à la longueur de la chaîne).

La seconde boucle affiche la même chaîne, mais en se servant cette fois d'un pointeur. On définit la variable `pszString` comme étant égale à l'adresse du premier caractère du tableau. La boucle est parcourue jusqu'à ce que le caractère pointé par `pszString` devienne égal à `false`, en d'autres termes, jusqu'à ce que ce caractère soit le terminateur `nul`.

La valeur entière 0 est interprétée comme valant `false`. Toutes les autres valeurs sont considérées comme vraies (`true`).

Le programme affiche le caractère pointé par `pszString`, puis il incrémente cette adresse de façon à se placer sur le caractère suivant. Et on recommence.

L'incrémentation et le traitement d'un adressage peuvent être combinés en une seule expression (et c'est généralement la méthode utilisée). Exemple :

```
cout << *pszString++;
```

L'affichage produit par notre programme se présente ainsi :

```
La chaîne contient 'Randy'
Affiche la chaîne en tant que tableau : Randy
Affiche la chaîne en utilisant un pointeur : Randy
Appuyez sur une touche pour continuer...
```

Parmi les exemples à télécharger sur notre site, vous trouverez également une variante basée sur des pointeurs du programme de concaténation de chaînes développé dans le Chapitre 7. Je vous laisse le soin de l'étudier.

Justification de la manipulation de chaîne par pointeur

La nature parfois ésotérique des manipulations de chaînes de caractères basées sur des pointeurs risque de désorienter le lecteur qui se demandera quels sont les avantages que lui offre une version basée sur le pointeur `char*` par rapport à la version indexée, plus facile à lire.

La réponse tient à la fois à l'histoire du C++ et à la nature humaine. Lorsque le C, l'ancêtre du C++, fut inventé, les compilateurs étaient assez frustes. Ils étaient incapables de procéder aux optimisations sophistiquées que réalisent les compilateurs modernes. Aussi compliquée qu'elle puisse paraître, une instruction comme `*pszString++` peut être convertie en un nombre incroyable-

ment réduit d'instructions en langage machine, même par le compilateur le plus obtus.

Comparés aux processeurs actuels, ceux d'autrefois étaient terriblement lents ; l'enregistrement de quelques instructions de programmation n'était pas une mince affaire. Ces fonctionnalités donnèrent au C un gros avantage par rapport aux autres langages de cette époque, notamment sur le Fortran qui ne proposait aucune arithmétique de pointeur.

En plus de la recherche de l'efficacité, les programmeurs aiment bien développer des programmes élégants, sans aucune redite qui les alourdirait. Une fois qu'un programmeur en C++ a appris à écrire des programmes compacts, certes quelque peu énigmatiques mais ô combien efficaces, il n'aura plus aucune envie de revenir aux tableaux et à leurs indices.

Il est inutile de créer des expressions C++ complexes pour améliorer l'efficacité du programme. Il n'y a en effet aucune relation évidente entre le nombre d'instructions C++ et la quantité d'instructions en langage machine générées.

Appliquer des opérateurs à des types de pointeur autres que char

Il n'est pas difficile de vous convaincre par vous-même que `szTarget + n` pointe sur `sztarget[n]` lorsque `szTarget` est un tableau de caractères de type `char`. Car après tout, un élément `char` occupe un seul octet. Si `szTarget` est stocké à l'adresse 0x100, le sixième élément se trouve à 0x105.

Le fait que l'ajout d'un pointeur fonctionne exactement de la même manière pour un tableau d'entiers `int` n'est pas évident car les `int` exigent quatre octets, soit quatre fois plus qu'un `char` (du moins dans le cas d'un processeur 32 bits Intel). Si le premier élément d'un tableau `intArray` se trouve à 0x100, le sixième élément se trouvera à 0x114, soit 0x100 + (5 * 4), et non à 0x104.

Heureusement pour nous, `array + n` pointe sur `array[n]` quelle que soit la taille d'un élément du tableau `array`. Le C++ (n'oublions pas de le remercier au passage) tient automatiquement compte de la taille des éléments.

Là encore, notre bonne vieille analogie immobilière fonctionne à merveille. La troisième maison à partir du 123 Rue Principale porte le numéro 126, quelle que soit la taille des maisons (et même s'il y a un immeuble entre deux).

Pointeurs et tableaux

Il existe quelques menues différences entre gestion des indices de tableau et utilisation d'un pointeur. Par exemple, le tableau alloue un espace pour les données, ce que le pointeur ne fait pas :

```
void arrayVsPointer()
{
   // allocation d'espace pour 128 caractères.
   char charArray[128];

   // allocation d'espace pour un pointeur, mais
   // pas pour l'objet vers lequel il pointe.
   char* pArray;
}
```

Ici, `charArray` contient 128 caractères ; `pArray` n'occupe que quatre octets, c'est-à-dire l'espace de stockage exigé par un pointeur.

Cherchez l'erreur dans le listing qui suit :

```
void arrayVsPointer()
{
   // Celle-ci fonctionne bien :
   char charArray[128];
   charArray[10] = '0';
   *(charArray + 10) = '0';

   // Celle-ci ne fonctionne pas :
   char* pArray;
   pArray[10] = '0';
   *(pArray + 10) = '0';
}
```

Les expressions `charArray[10]` et `(charArray + 10)` sont équivalentes et admises. Les deux expressions impliquant `pArray` sont équivalentes sur le plan de la syntaxe, mais elles n'ont pas de sens. Alors qu'elles sont toutes deux légales en C++, le pointeur non initialisé `pArray` contient des valeurs aléatoires ; il n'a pas été initialisé pour pointer vers un tableau tel que `charArray`. C'est pourquoi `pArray[10]` et son équivalent `*(pArray + 10)` référencent des données incohérentes.

Une erreur de référencement de la mémoire avec une variable de pointeur non initialisé est en général interceptée par le processeur lors de l'exécution du

programme, ce qui entraîne la redoutable erreur de *violation de segment* que votre application favorite, sous votre système d'exploitation plus ou moins favori, se fait un plaisir d'afficher de temps en temps. Ce problème n'est généralement pas dû au processeur ou au système d'exploitation, mais au programme.

Une autre différence entre un pointeur et l'adresse d'un tableau réside dans le fait que `charArray` est une constante alors que `pArray` n'en est pas une. C'est pourquoi la boucle `for` ci-dessous, utilisée pour initialiser le tableau `charArray`, ne fonctionne pas :

```
void arrayVsPointer()
{
char charArray[10];
for (int i = 0 ; i < 10; i++)
{
   *charArray = '0';  // ...ceci se conçoit
   charArray++;       // ...mais pas cela !
}
}
```

L'expression `charArray++` n'est pas plus justifiée que `10++`. Voici la version correcte :

```
void arrayVsPointer()
{
char charArray[10];
char* pArray = charArray;
for (int i = 0; i < 10; i++)
{
   *pArray = '\0';    // ceci marche bien !
   pArray++;
}
}
```

Déclarer et utiliser des tableaux de pointeurs

Si des pointeurs peuvent désigner des tableaux, le contraire devrait aussi être vrai. Les tableaux de pointeurs sont des types de tableau particulièrement intéressants.

Puisque les tableaux peuvent recevoir d'autres types de données, rien n'empêche qu'ils contiennent des pointeurs. La ligne qui suit déclare un tableau de pointeurs vers des entiers `int` :

```
int* pInts[10];
```

D'après la déclaration ci-dessus, `pInts[0]` est un pointeur vers une valeur entière `int`. De ce fait, ce qui suit est vrai :

```
void fn()
{
   int n1;
   int* pInts[3];
   pInts[0] = &n1;
   *pInts[0] = 1;
```

ou bien :

```
void fn()
{
   int n1, n2, n3;
   int* pInts[3] = (&n1,&n2,&n3);
   for (int i = 0; i < 3; i++)
   {
      *pInts[i] = 0;
   }
}
```

ou encore :

```
void fn()
{
   int* pInts[3] = {(new int)},
                    new int),
                    new int)};
   for (int i = 0; 1 < 3; i++)
   {
      *pInts[i] = 0;
   }
}
```

Ce dernier listing déclare trois objets `int` sur le tas (*heap*).

Les tableaux de pointeurs sont surtout utilisés pour référencer des tableaux de chaînes de caractères. Les deux exemples qui suivent montrent en quoi ces tableaux s'avèrent utiles.

Utiliser des tableaux de chaînes de caractères

Si le C++ supporte les tableaux de pointeurs, il devrait être possible de créer des tableaux de pointeurs pointés sur des tableaux. Cette récursivité peut être menée aussi loin que vous le désirez (des tableaux de pointeurs pointant sur des tableaux de pointeurs qui pointent...). Les tableaux de pointeurs pointés sur des chaînes de caractères méritent un intérêt tout particulier. Rappelez-vous qu'une chaîne n'est rien de moins qu'un type de tableau de caractère spécial.

Supposons que nous ayons besoin d'une fonction qui retourne le nom du mois correspondant à un argument entier qui lui a été passé. Par exemple, si le programme reçoit comme valeur 1, il réagit en retournant un pointeur vers la chaîne `"Janvier"`. Si c'est 2, il retournera un pointeur vers `"Février"`, et ainsi de suite. Le mois 0 est supposé non valide, de même que toute valeur supérieure à 12.

Cette fonction pourrait s'écrire ainsi :

```
// int2month() - Retourne le nom du mois.
char* int2month(int nMonth)
{
   char* pszReturnValue;

   switch(nMonth)
   {
      case 1: pszReturnValue = "Janvier";
              break;
      case 2: pszReturnValue = "Février";
              break;
      case 3: pszReturnValue = "Mars";
              break;
      // ...et ainsi de suite...
      default: pszReturnValue = "Non valide";
   }
   return pszReturnValue;
}
```

La commande de contrôle `switch()` est l'équivalent d'une succession d'instructions `if`.

Une solution plus élégante consiste à utiliser la valeur entière d'un mois comme index dans un tableau de pointeurs vers les noms des mois. En pratique, le listing se présente ainsi :

```
//  Int2Month - renvoie le nom des mois
//              à l'aide d'un pointeur
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// int2month() - Retourne le nom du mois
char* int2month(int nMonth)
{
    // Détection d'une valeur hors de la plage admise
    if (nMonth < 1 || nMonth > 12)
    {
        return "Non valide";
    }

    // nMonth est valide - Retour du nom du mois
    char* pszMonths[] = {"Non valide",
                         "Janvier",
                         "Février",
                         "Mars",
                         "Avril",
                         "Mai",
                         "Juin",
                         "Juillet",
                         "Août",
                         "Septembre",
                         "Octobre",
                         "Novembre",
                         "Décembre"};
    return pszMonths[nMonth];
}

int main(int nArgs, char* pszArgs[])
{
    for(int i = 0; i<=12; i++)
    {
        cout << "Le mois de rang " << i << " est "
             << int2month(i) << "\n";
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
```

```
    system("PAUSE");
    return 0;
}
```

Dans ce listing, `int2month()` vérifie d'abord si le nombre est compris dans une plage allant de 1 à 12 (dans l'exemple précédent, cette tâche avait été confiée à la clause `default` de l'instruction `switch`). Si `nMonth` est valide, la fonction l'utilise comme décalage dans un tableau contenant les noms des mois.

Cette technique est particulièrement utile pour écrire des programmes destinés à être exécutés dans des langues différentes. Par exemple, un programme peut déclarer un tableau de pointeurs `ptrMonths` vers un calendrier Julien et l'initialiser lors de l'exécution en français, anglais ou encore allemand. Dans ces conditions, et quelle que soit la langue employée, `ptrMonths[1]` pointera toujours vers le premier mois de l'année.

Accéder aux arguments de main()

Le premier argument de `main()` est un tableau de pointeurs vers des chaînes. Ces chaînes contiennent les arguments du programme lui-même. Il s'agit des chaînes qui apparaissent avec le nom du programme au moment où vous exécutez celui-ci. Supposons par exemple que j'aie entré la commande suivante à la suite de l'invite de MS-DOS :

```
MonProg fichier.txt /w
```

MS-DOS exécute le programme du fichier `MonProg.exe` en lui passant les arguments `fichier.txt` et `/w`.

Si vous n'avez jamais vu l'invite de MS-DOS, ne me jetez pas la première pierre. L'analogie avec Windows est indiquée un peu plus loin.

La variable `pszArgs` passée à `main()` est un tableau de pointeurs vers les arguments du programme, tandis que `nArgs` contient le nombre d'arguments.

Voyons un exemple simple :

```
// PrintArgs - écrit les arguments du programme
//             vers la sortie standard.
#include <cstdio>
#include <cstdlib>
```

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

int main(int nNumberofArgs, char* pszArgs[])
{
    // affiche un message d'avertissement
    cout << "Les arguments du programme " << pszArgs[0] << " sont :\n";

    // écrit maintenant le reste des arguments
    for (int i = 1; i < nNumberofArgs; i++)
    {
        cout << i << " : " << pszArgs[i] << "\n";
    }

    // terminé
    cout << "C'est tout !\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Comme toujours, la fonction `main()` accepte deux arguments. Le premier est un entier que j'ai nommé `nNumberofArgs`. Cette variable contient le nombre d'arguments passés au programme. Le deuxième argument est un tableau de pointeurs de type `char[]*` que j'ai appelé `pszArgs`. Chacun de ces éléments `char*` pointe vers un argument passé au programme.

Accéder aux arguments du programme (style MS-DOS)

Si j'exécute le programme `PrintArgs` comme suit :

```
PrintArgs arg1 arg2 arg3 /w
```

à partir de la ligne de commande d'une fenêtre MS-DOS, `nNumberofArgs` sera égal à 5 (un par argument). Le premier argument est le nom du programme ; de ce fait, `pszArgs[0]` pointe vers PrintArgs. Les éléments restant de `pszArgs` pointent vers les arguments du programme. L'élément `pszArgs[1]` pointe par exemple vers arg1, `pszArgs[2]` vers arg2, et ainsi de suite. Comme MS-DOS n'accorde aucune signification particulière à `/w`, cette chaîne est passée comme argument à traiter par le programme.

Accéder aux arguments du programme (style Dev-C++)

Vous pouvez aussi ajouter des arguments à votre programme lorsque vous l'exécutez à partir de la fenêtre de Dev-C++. Pour cela, ouvrez le menu Debug et choisissez la commande Paramètres. Saisissez ce que vous voulez, puis lancez le programme à l'aide de la commande Exécuter du menu portant le même nom (ou en appuyant sur Ctrl+F10). Le résultat obtenu est semblable à ce que donne le style MS-DOS.

Accéder aux arguments du programme (style Windows)

Windows lui aussi sait passer des arguments pour communiquer avec votre programme. Essayez l'expérience suivante. Compilez normalement votre programme. Retrouvez l'exécutable ainsi obtenu dans l'Explorateur Windows (sur ma machine, le dossier correspondant est `C:\Dev-Cpp\Nuls\Chap09`).

Sélectionnez maintenant un nom de fichier et faites-le glisser sur l'icône de l'exécutable (n'importe lequel fera l'affaire, cela n'a aucune importance puisque nous n'effectuons aucun traitement sur les arguments). Vlan ! Le programme `PrintArgs` se lance et le nom du fichier que vous venez de faire glisser apparaît.

Refaites un autre essai, mais en sélectionnant cette fois un groupe de fichiers (faites glisser la souris tout en maintenant son bouton gauche enfoncé, ou procédez par séries de Ctrl+clic ou encore de Maj+clic, bref, les techniques habituelles). Le nom de chaque fichier va apparaître dans la sortie. Par exemple :

```
Les arguments du programme C:\Dev-Cpp\Nuls\Chap09\PrintArgs.exe sont :
1:C:\Program Files\WinZip\WZINST.HLP
2:C:\Program Files\WinZip\WZCAB.DLL
3:C:\Program Files\WinZip\WZCAB3.DLL
4:C:\Program Files\WinZip\WZ32.DLL
5:C:\Program Files\WinZip\WZQKPICK.EXE
6:C:\Program Files\WinZip\WZQKSTRT.RTF
C'est tout !
Appuyez sur une touche pour continuer...
```

Notez que chaque nom de fichier correspond bien à un argument et un seul, et ce, même s'il contient des espaces. Remarquez aussi que Windows passe le chemin d'accès complet aux fichiers.

La technique du glisser et déposer est un procédé facile pour passer des arguments au démarrage d'un programme.

Chapitre 10

Débogage du code C++

Dans ce chapitre :

- Différencier les types d'erreurs.
- Comprendre les messages de "plantage".
- Techniques de débogage en mode Write.
- Maîtriser les outils de débogage.

Vous aurez sans doute remarqué que vos programmes ne fonctionnent pas toujours du premier coup. À vrai dire, il m'est rarement arrivé (jamais ?) d'écrire un programme C++ quelque peu étoffé sans que des erreurs soient apparues lors de la première exécution.

Vous avez deux possibilités : abandonner purement et simplement un programme qui refuse de fonctionner, ou trouver et corriger ces erreurs. Dans ce chapitre, nous supposons que vous avez choisi de rechercher et éradiquer les bogues de vos logiciels.

Identifier les types d'erreurs

Il existe deux types d'erreurs : celles que le compilateur C++ sait détecter et celles qu'il ne voit pas passer.

Les erreurs que le C++ intercepte sont appelées *erreurs de compilation.* Elles sont relativement faciles à corriger car le compilateur attire généralement votre attention sur la nature du problème. Il arrive parfois que la description qu'il affiche ne soit pas tout à fait exacte (il est très facile de tromper un compi-

lateur, même involontairement) mais une fois que vous connaîtrez les bizarreries de votre environnement C++, vous comprendrez plus facilement ce qu'il vous raconte.

Les erreurs que le C++ n'a pas pu ou su détecter se manifestent après le démarrage du programme. Elles sont appelées *erreurs d'exécution*. Ces problèmes sont beaucoup plus difficiles à localiser que les erreurs de compilation, car vous ne disposez que de très peu d'indices pour savoir ce qui ne va pas, hormis les informations plus ou moins cohérentes que le programme aurait éventuellement pu afficher. Incohérence. C'est bien le mot qui convient.

Vous avez le choix entre deux techniques pour détecter les bogues. L'une consiste à placer des instructions de sortie aux endroits stratégiques du programme. Vous pourrez ainsi vous faire une idée de ce qui se passe au fur et à mesure que ces instructions sont exécutées. L'autre méthode consiste à recourir à un programme distinct, le *débogueur*. Un débogueur permet de contrôler un programme pendant son exécution.

Ces deux techniques seront abordées dans ce chapitre.

La résolution des problèmes par la technique Write

L'ajout d'instructions de sortie au code source C++ pour déterminer ce qui se passe dans le programme est parfois appelé "technique Write". Ce nom remonte à l'époque héroïque du Fortran (et du Cobol) ; Write est, dans ce langage, le nom de la commande de sortie sur le périphérique standard (les plus anciens ont connu un temps où ce n'était pas l'écran).

Le petit programme "bogué" qui suit montre comment utiliser la technique Write.

Il est supposé lire une série de chiffres entrés au clavier et calculer leur moyenne. Il contient hélas deux erreurs : l'une plante le programme, l'autre donne des résultats erronés.

```
// ErrorProgram1 - ce programme effectue la moyenne d'une
//                 série de nombres. Il contient au moins
//                 un bogue fatal.
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Ce programme EST ECRIT POUR SE PLANTER !\n"
         << endl;
    int nSum;
    int nNums;

    // accumule les nombres saisis jusqu'à
    // ce que l'utilisateur entre un nombre
    // négatif, après quoi il retourne la moyenne.
    nNums = 0;
    while(true)
    {
        // entrée d'un autre nombre à ajouter
        int nValue;
        cout << "Entrez un autre nombre : ";
        cin  >> nValue;
        cout << endl;

        // si le nombre entré est négatif...
        if (nValue < 0)
        {
            // ...afficher la moyenne.
            cout << "La moyenne vaut : "
                 << nSum/nNums
                 << endl;
            break;

        }

        // nombre non négatif : l'ajouter à
        // l'accumulateur.
        nSum += nValue;
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Après avoir tapé ce code source (il est téléchargeable depuis le site de First Interactive), construisez le fichier exécutable `ErrorProgram1.exe` (appuyez sur Ctrl+F9).

Exécutez le programme en appuyant sur Ctrl+F10. Entrez les valeurs 1, 2 et 3 puis -1 pour terminer les entrées. Attendez-vous à un choc. Au lieu de produire la valeur plus qu'attendue de 2 (car même moi je serais capable de calculer mentalement la moyenne de 1, 2 et 3) le programme se termine par un message d'erreur totalement ésotérique (voir la Figure 10.1).

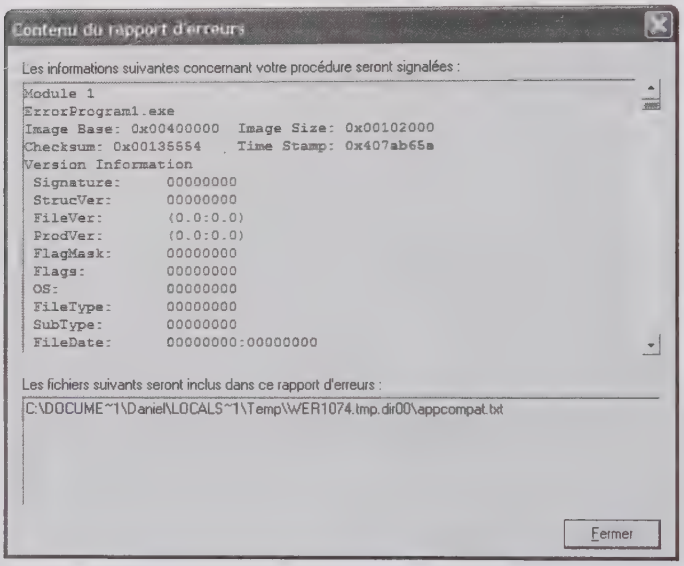


Figure 10.1
La première version de ErrorProgram se termine soudainement au lieu de sortir les résultats attendus.

La Figure 10.1 montre ce que vous obtenez sous Windows XP, du moins si vous avez eu le courage de demander des informations complémentaires dans la fenêtre qui vous annonce brutalement combien Windows est désolé pour vous.

Recherche du bogue n°1

Le message d'erreur de la Figure 10.1 en met plein la vue ; en fait, les informations qu'il fournit sont pour nous inutiles (il doit bien y avoir quelqu'un pour qui elles sont utiles...). Bref, Windows XP ne nous aide en rien. De ce point de vue, on pourrait dire qu'il protège presque trop l'utilisateur. Le seul renseignement précis est l'adresse interne du programme où l'erreur s'est produite, mais cela ne nous est d'aucun secours. Avec une version plus ancienne de Windows, vous auriez peut-être eu la "chance" de découvrir que le problème était dû à une *division par zéro* (il faut bien que je vous aide un peu).

Une division par zéro, on comprend bien ce que cela veut dire. Enfin, en apparence. Supposons par exemple que le programme se soit fourvoyé et qu'il exécute des instructions qui ne lui appartiennent pas (ce qui arrive plus souvent que vous pourriez le croire). Le processeur en arriverait à exécuter une division par zéro (d'où le message), mais l'origine réelle du problème serait simplement masquée. Un programme incohérent est comme un train qui aurait déraillé : rien n'arrête sa course sauf un obstacle vraiment très gros.

Si nous regardons notre code, nous y trouvons une seule division :

```
cout << "La moyenne vaut : "
     << nSum/nNums
     << endl;
```

Ce n'est pas parce que ne voyons qu'une seule division qu'il s'agit du seul endroit où cette opération est effectuée. Le compilateur C++ pourrait parfaitement avoir généré, de son propre chef, une division en réponse à d'autres instructions du programme. De surcroît, la librairie standard du C++ est pleine de divisions.

On peut raisonnablement estimer que, très probablement, la valeur de `nNums` était égale à zéro au moment où l'opération a été évaluée. Cette variable est censée compter le nombre de valeurs saisies. Pour en apprendre, vous pouvez ajouter une instruction `cout` dans la boucle `while` afin d'afficher la valeur courante de `nNums` :

```
while(true)
{
    // affichage
    cout << "nNums = " << nNums << endl;

    // ...le reste du programme est inchangé...
```

Et voilà maintenant ce que vous allez obtenir :

```
Ce programme EST ECRIT POUR SE PLANTER !

nNums = 0
Entrez un autre nombre : 1

nNums = 0
Entrez un autre nombre : 2
```

```
nNums = 0
Entrez un autre nombre : 3

nNums = 0
Entrez un autre nombre :
```

La variable nNums est bien initialisée à 0. Mais où se trouve son incrémentation ? Il n'y en a pas, et c'est bien là qu'est l'erreur. Il est maintenant clair que nNums *devrait* avoir été incrémenté lors de chaque passage dans la section chargée de lire les entrées au clavier. Pour corriger le problème, j'ai transformé le while en une boucle for de la façon suivante :

```
for (int nNums = 0; ;nNums++)
```

Recherche du bogue n°2

Vous êtes venu à bout du bogue n°1. Maintenant, recompilez puis exécutez de nouveau le programme en utilisant les mêmes valeurs : 1, 2, 3 et -1 qui l'avaient planté précédemment. Cette fois, pas de fenêtre radicale. Mais, même sans grandes connaissances mathématiques, on se rend compte que la sortie n'a aucun sens :

```
Ce programme EST ECRIT POUR SE PLANTER !

Entrez un autre nombre : 1

Entrez un autre nombre : 2

Entrez un autre nombre : 3

Entrez un autre nombre : -1

La moyenne vaut : 1456814
Appuyez sur une touche pour continuer...
```

Apparemment, ni nSum, ni nNums (ni les deux) n'ont été calculés correctement. Pour vous en sortir, vous devez connaître les valeurs de ces variables. Ce qui vous aiderait, ce serait de connaître aussi la valeur de nValue, car elle est utilisée pour calculer nSum.

Pour connaître les valeurs de nSum, nNums et nValue, modifiez la boucle for de la manière suivante (cette version du programme est téléchargeable depuis le site de First Interactive) :

```
// ErrorProgram2 - Ce programme effectue la moyenne d'une
//                 série de nombres. Il contient au moins
//                 un bogue fatal.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Ce programme fournit des sorties incorrectes."
         << endl;

    // accumule les nombres saisis jusqu'à
    // ce que l'utilisateur entre un nombre
    // négatif, après quoi il retourne la moyenne.
    int nSum;

    for (int nNums = 0; ;nNums++)
    {
        // entrée d'un autre nombre à ajouter
        int nValue;
        cout << "Entrez un autre nombre : ";
        cin  >> nValue;
        cout << endl;

        // si le nombre entré est négatif...
        if (nValue < 0)
        {
            // ...afficher la moyenne
            cout << "\nLa moyenne vaut : "
                 << nSum/nNums
                 << "\n";
            break;

        }
        // affichage des informations critiques
        cout << "nSum = " << nSum   << "\n";
        cout << "nNums= " << nNums  << "\n";
        cout << "nValue= "<< nValue << "\n";
        cout << endl;

        // Nombre non négatif. L'ajouter à
```

```
        // l'accumulateur.
        nSum += nValue;
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Notez l'ajout d'instructions de sortie qui affichent les valeurs de `nSum`, `nNums` et `nValue` à chaque itération dans la boucle.

Le listing ci-dessous montre le résultat produit par l'exécution des habituelles entrées 1, 2, 3 et -1. Il apparaît que, dès la première boucle, la valeur de `nSum` n'est pas crédible. En fait, à ce stade de la première boucle, le programme doit pourtant ajouter une nouvelle valeur à `nSum`. Et la valeur initiale de celui-ci devrait être nulle.

```
Ce programme fournit des sorties incorrectes.
Entrez un autre nombre : 1

nSum = 4370436
nNums= 0
nValue= 1

Entrez un autre nombre : 2

nSum = 4370437
nNums= 1
nValue= 2

Entrez un autre nombre : 3

nSum = 4370439
nNums= 2
nValue= 3

Entrez un autre nombre : -1


La moyenne vaut : 1456814
Appuyez sur une touche pour continuer...
```

Un examen attentif de ce programme révèle que `nSum` est déclaré mais n'est pas du tout initialisé. La solution réside donc dans la déclaration de `nSum` de la manière suivante :

```
int nSum = 0;
```

Aussi longtemps qu'une variable n'a pas été initialisée, sa valeur est indéterminée.

Une fois que vous estimez avoir localisé le problème, nettoyez le programme comme suit (cette version du programme est aussi téléchargeable) :

Le programme a été reconstruit puis testé avec la suite 1, 2, 3 et -1. Il produit cette fois la moyenne attendue de 2 :

```
Entrez un autre nombre : 1

Entrez un autre nombre : 2

Entrez un autre nombre : 3

Entrez un autre nombre : -1


La moyenne vaut : 2
Appuyez sur une touche pour continuer...
```

Faites d'autres tests avec des valeurs différentes pour vous convaincre que le programme fonctionne maintenant parfaitement.

Recourir au déboqueur

Pour des programmes de petite taille, la technique Write fonctionne plutôt bien. L'ajout d'instructions est relativement simple et, comme le programme est rapidement reconstruit, la durée du cycle de débogage est suffisamment courte. Cette approche ne pose pas de problème aussi longtemps que vous vous en tenez à de petits programmes, comme ceux de ce livre.

Mais, dans une application plus ambitieuse, le programmeur ne saura généralement pas par où il doit commencer à insérer des instructions de sortie. À la longue, l'ajout d'instructions de débogage, d'exécution, de nouvelles instructions et les tests qui s'en suivent finissent par devenir extrêmement fastidieux.

De plus, à chaque modification d'une instruction de sortie, il faut reconstruire entièrement le programme. Pour un projet de grande envergure, le temps de compilation peut être assez significatif (il arrive que la construction dure toute une nuit).

Enfin, la résolution des problèmes de pointeurs est quasiment impossible par la technique Write. L'affichage d'un pointeur en hexadécimal ne vous aidera pas et, sitôt que vous essayez de "déréférencer" le pointeur, le programme se plante.

Une autre technique, plus sophistiquée, est basée sur un utilitaire particulier appelé *débogueur*. Cette approche évite les inconvénients de la technique Write ; elle implique cependant l'apprentissage d'un nouvel outil, le débogueur.

Un débogueur, c'est quoi ?

C'est un outil que l'on trouve en particulier dans les environnements Dev-C++ et Microsoft Visual C++ .NET (comme d'ailleurs dans la plupart des environnements de développement). Ils sont tous différents mais ils travaillent selon les mêmes principes.

Le programmeur contrôle le débogueur au travers de commandes qui sont toutes disponibles dans l'interface de l'éditeur. Elles sont accessibles via des menus ou des raccourcis clavier.

Le débogueur permet de contrôler l'exécution d'un programme. Il peut le parcourir pas à pas, s'arrêter en un point ou encore examiner les valeurs contenues dans les variables. Il faut voir un débogueur à l'œuvre pour en apprécier la puissance.

Choisir un débogueur

Contrairement au langage C++ qui est dûment standardisé, chaque débogueur possède ses propres jeux de commandes. Fort heureusement, celles-ci sont fondamentalement les mêmes. Celles dont vous aurez besoin se trouvent aussi bien dans l'incontournable environnement Visual C++ .NET de Microsoft que dans Dev-C++. De plus, les commandes de débogage sont toutes accessibles via des menus ou des touches de fonction. Le Tableau 10.1 indique les raccourcis clavier des deux environnements.

Tableau 10.1 : Les commandes des débogueurs de Visual C++ et de Dev-C++.

Commande	Visual C++	Dev-C++
Déboguer	F5	F8
Avancer (pas à pas interne)	F11	Maj+F7
Pas à pas	F10	F7
Continuer	F5	Ctrl+F7
Voir la variable	<menu seulement>	<Ajouter variable>
Ajouter une variable	<menu seulement>	F4
Basculer breakpoint (point d'arrêt)	Ctrl+B ou cliquer	Ctrl+F5
Réinitialiser le programme	Maj+F5	Ctrl+Alt+F2

Dans Dev-C++, vous pouvez aussi ajouter ou supprimer un point d'arrêt en cliquant à hauteur de la ligne voulue dans le bandeau situé à gauche de la fenêtre d'édition.

Dans le restant de ce chapitre, les commandes de débogage seront citées par leur nom. Référez-vous au Tableau 10.1 pour connaître les raccourcis clavier correspondants.

Lancer un programme de test

Le meilleur moyen d'apprendre comment corriger un programme à l'aide d'un débogueur, c'est de travailler pas à pas sur un programme bogué. Dans celui qui suit (il peut être téléchargé depuis le site de First Interactive), plusieurs problèmes doivent être découverts puis réparés.

```
// ConcatenateErr - concaténation de deux chaînes séparées
//                  par un " - ". Cette version plante.
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string.h>

using namespace std;
```

```
void stringEmUp(char* szTarget,
                char* szSource1,
                char* szSource2,
                int nLength);

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Ce programme concatène deux chaînes.\n"
         << "(Cette version se plante.)" << endl;
    char szStrBuffer[256];

    // on  créée deux chaînes de même longueur...
    char szString1[16];
    strncpy(szString1, "Ceci est une chaîne", 16);
    char szString2[16];
    strncpy(szString2, "CECI EST UNE CHAINE", 16);

    // ...puis on les colle
    stringEmUp(szStrBuffer,
               szString1,
               szString2,
               16);

    // affichage du résultat
    cout << "<" << szStrBuffer << ">" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

void stringEmUp(char* szTarget,
                char* szSource1,
                char* szSource2,
                int nLength)
{
    strcpy(szTarget, szSource1);
    strcat(szTarget, " - ");
    strcat(szTarget, szSource2);
}
```

Compilez le programme ; la construction s'effectue sans incident. Exécutez-le. Au lieu de produire la sortie attendue, vous devriez voir très brièvement s'ouvrir une fenêtre de console, puis plus rien. Que s'est-il passé ? Mystère !

Votre seul espoir de trouver la solution, c'est de vous plonger dans les profondeurs du débogueur.

Parcourir un programme pas à pas

Le mieux que vous ayez à faire pour détecter un problème, c'est d'exécuter le programme en mode débogage. Le débogueur est parfois à même de vous fournir des informations utiles.

La première fois que j'ai exécuté mon exemple de programme sous Dev-C++ en utilisant le débogueur (via le raccourci F8), j'ai reçu un message d'erreur indiquant : "Une violation d'accès (erreur de segmentation) est apparue dans votre programme".

C'est déjà mieux que rien, mais il vous faut de plus amples renseignements pour localiser le problème.

Une erreur de segmentation indique généralement un problème avec un pointeur quelconque.

Avant de faire un nouvel essai, vous devez refermer la fenêtre du message puis réinitialiser le programme. Vous disposez pour cela de trois méthodes : choisir la commande Arrêter l'exécution dans le menu Debug, le bouton de même nom dans le panneau Debug affiché en bas de la fenêtre de Dev-C++, ou enfin le raccourci clavier Ctrl+Alt+F2. De cette façon, tous les paramètres internes du débogueur sont remis à zéro.

Pensez toujours à réinitialiser le débogueur avant de relancer un programme que vous testez. C'est de toute façon nécessaire pour revenir au point de départ de l'exécution, et sans douleur si le programme s'y trouvait déjà.

Pour mieux voir où se situe exactement le problème, un procédé courant consiste à exécuter simplement une partie du programme. C'est ce que vous permet de faire le débogueur à l'aide d'un *point d'arrêt* (breakpoint), c'est-à-dire d'une marque indiquant qu'il faut stopper l'exécution à cet endroit et rendre la main au programmeur.

Définissez un point d'arrêt sur la première instruction exécutable (le `cout` qui suit `main()`). Pour cela, cliquez sur la ligne et choisissez dans le menu Debug la commande Basculer breakpoint. Vous pouvez aussi utiliser le raccourci Ctrl+F5 ou encore cliquer à hauteur de la ligne sur le bandeau noir qui se trouve à gauche de la fenêtre d'édition. Une coche s'affiche dans ce bandeau et l'instruction est surlignée, ce qu'illustre la Figure 10.2.

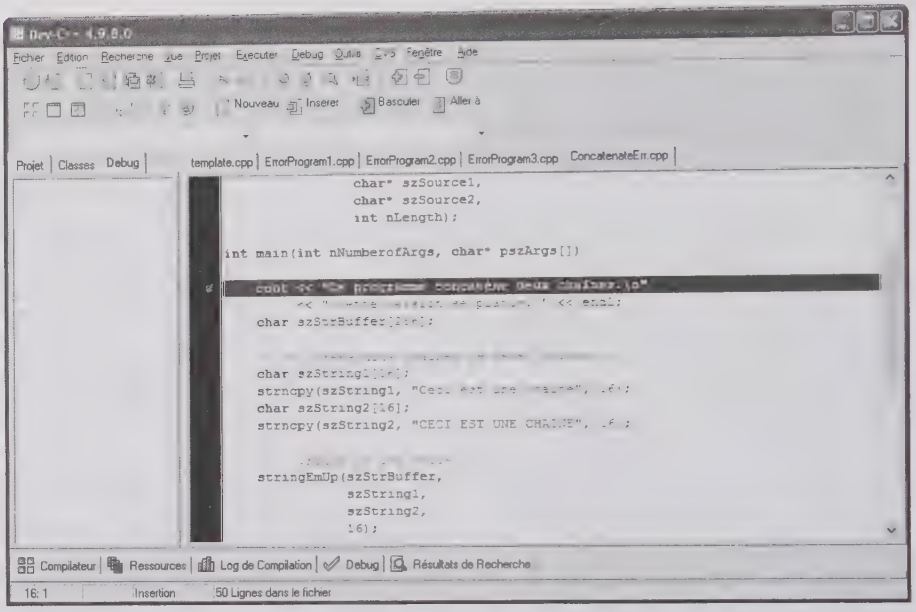


Figure 10.2
Un point d'arrêt a été inséré dans le programme.

Exécutez maintenant une nouvelle fois le programme sous le contrôle du débogueur. Vous pouvez pour cela passer par le menu Debug (commande Debugger), par l'onglet et le bouton portant le même nom en bas de la fenêtre ou encore appuyer directement sur la touche F8. Le programme se lance normalement, mais il s'arrête immédiatement après la première ligne (et donc juste avant d'exécuter l'instruction sur laquelle vous avez défini un point d'arrêt). La ligne courante apparaît alors sur un fond bleu.

À ce stade, vous pouvez entrer dans le mode Pas à pas, toujours à partir du menu Debug ou de la barre d'outils correspondante. Le raccourci clavier associé à ce mode est F7. La ligne courante est exécutée et le marqueur bleu se déplace sur l'instruction suivante. Vous remarquerez que les déclarations sont ignorées. Il ne s'agit en effet pas de commandes exécutables : elles servent uniquement à réserver de l'espace en mémoire pour les variables. Si vous regardez le contenu de la console, vous constaterez qu'elle affiche bien ce qu'a envoyé `cout` sur le dispositif de sortie standard (voir la Figure 10.3).

Exécutez encore deux fois le mode Pas à pas pour vous déplacer jusqu'à l'appel de la fonction `StringEmUp()`. Et maintenant, appuyez encore un coup sur F7. Crac ! Revoici le plantage ignoble ! Au moins, vous savez maintenant que le problème se trouve quelque part à l'intérieur de la fonction `StringEmUp()`.

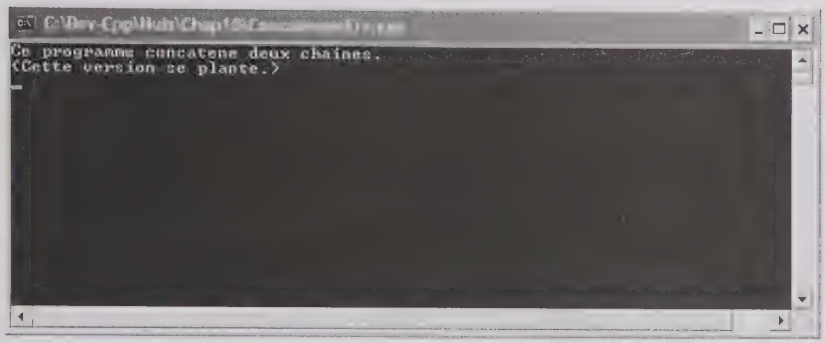


Figure 10.3
Activez à tout moment la fenêtre de console pour voir ce qui est affiché par le programme.

Lorsqu'un programme plante à l'intérieur d'une fonction, soit celle-ci est boguée, soit les arguments qui lui ont été passés sont incorrects.

La commande Pas à pas traite une fonction comme s'il s'agissait d'une instruction unique. Mais une fonction contient tout un ensemble d'instructions C++. Vous devez donc exécuter chaque ligne de code qui se trouve dans la fonction pour comprendre ce qui se passe. Ce mode de mise au point est accessible via la commande Avancer (toujours dans le menu Debug ou la barre d'outils correspondante) dont le raccourci est Maj+F7.

Réinitialisez le programme à l'aide de la commande Arrêter l'exécution (ou du raccourci Ctrl+Alt+F2). Vous voudriez bien sûr gagner un peu de temps en accédant directement à la ligne où la fonction est appelée. Pour cela, cliquez sur la marque du point d'arrêt actuel (dans la colonne affichée à gauche de la fenêtre d'édition). La coche disparaît et la ligne est réaffichée normalement. Cliquez maintenant dans la même colonne, juste à hauteur de l'instruction qui appelle la fonction. Vous venez de poser un nouveau point d'arrêt (voir la Figure 10.4).

Vous pouvez poser autant de points d'arrêt que vous le souhaitez dans un même programme (essayez tout de même de rester raisonnable).

Relancez maintenant le débogueur. L'exécution s'arrête maintenant juste avant d'appeler la fonction. Appuyez une fois sur Maj+F7 pour passer en mode Avancer. Le curseur saute à la première ligne d'instruction contenue dans la fonction (voir la Figure 10.5).

Vous savez que le moment du plantage n'est pas loin. Il serait plus facile de comprendre ce qui se passe si vous pouviez voir la valeur des arguments passés à la fonction. C'est justement le rôle de la commande Ajout Variable (c'est ce que l'on appelle souvent un *espion*). Cette technique permet d'afficher l'état d'une variable chaque fois que le programme est arrêté. Pour cela, il suffit

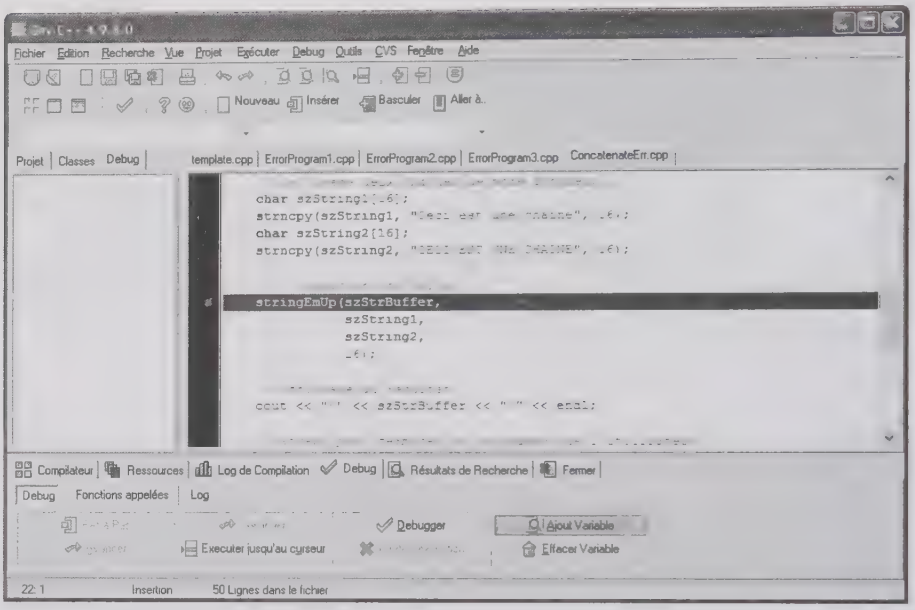


Figure 10.4
Le programme va s'exécuter jusqu'au point d'arrêt.

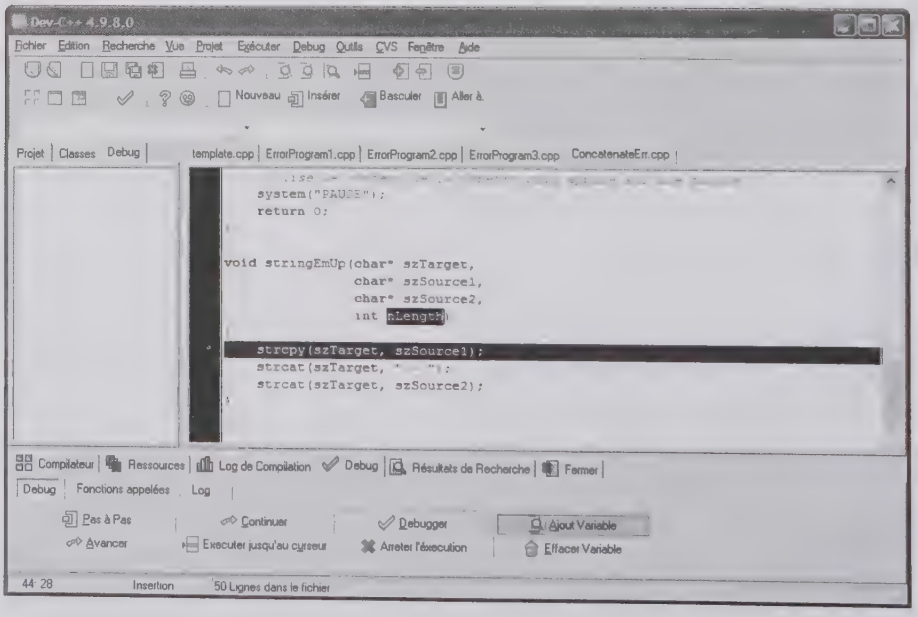


Figure 10.5
En mode Avancer, le contrôle se déplace sur la première ligne d'instruction trouvée dans la fonction.

de sélectionner le nom de la variable sur l'affichage et d'appuyer sur F4 (exactement comme sur la Figure 10.5). Vous pouvez aussi placer le curseur dans le nom de la variable et appuyer sur F4. Le débogueur vous suggèrera alors l'appellation voulue.

Mais il y a encore mieux. Si vous activez l'onglet Debug (à droite de la fenêtre principale) et que vous cliquez sur un nom de variable, celle-ci s'affiche automatiquement dans le panneau (au bout d'une ou deux secondes). Sur la Figure 10.6, j'ai utilisé ce procédé pour afficher tous les arguments de la fonction.

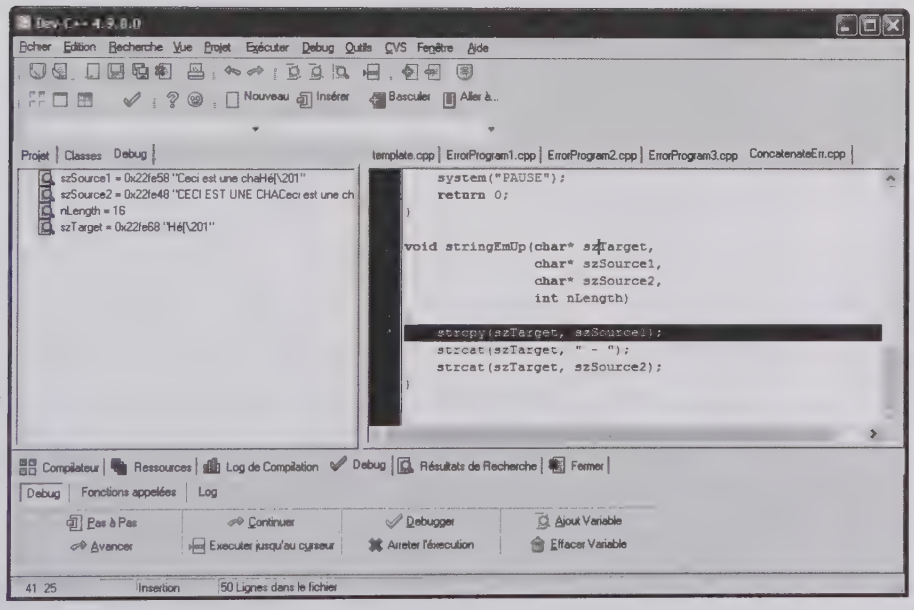


Figure 10.6
Les espions permettent au programmeur de vérifier la valeur des variables.

Le nombre hexadécimal qui suit la plupart des noms de variable indique leur adresse en mémoire. Rien de bien utile dans notre cas. Si nous regardons de près cet affichage, il est clair qu'il y a un problème avec la façon dont les chaînes sont mémorisées. Manifestement, et d'ailleurs très classiquement, c'est une histoire de pointeur.

Vous n'avez pas encore trouvé ? Je vais vous aider un peu. La longueur des deux chaînes n'est pas de seize caractères, mais de dix-sept caractères ! C'est pourquoi le programme principal a échoué en tentant d'allouer de la place pour le caractère de terminaison. Il se termine donc dès que vous tentez

d'exécuter la première instruction de la fonction `StringEmUp()`, c'est-à-dire l'appel à `strcopy()`.

La longueur d'une chaîne inclut toujours le caractère de terminaison.

Nous pouvons maintenant corriger notre programme. Cette fois, nous laisserons C++ calculer la taille de la chaîne pour qu'il ajuste l'espace nécessaire. La nouvelle version, `Concatenate2.cpp`, devrait fonctionner correctement :

```
// Concatenate2 - concaténation de deux chaînes séparées
//                par un " - ".
//               (cette version ne doit pas planter)
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string.h>

using namespace std;
void stringEmUp(char* szTarget,
                char* szSource1,
                char* szSource2);

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Ce programme concatène deux chaînes.\n"
         << "(Cette version ne devrait pas planter.)" << endl;
    char szStrBuffer[256];

    // on  créée deux chaînes...
    char szString1[] = "Ceci est une chaîne";
    char szString2[] = "CECI EST UNE CHAINE";

    // ...puis on les colle
    stringEmUp(szStrBuffer,
               szString1,
               szString2);

    // affichage du résultat
    cout << "<" << szStrBuffer << ">" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

```
void stringEmUp(char* szTarget,
                char* szSource1,
                char* szSource2)
{
    strcpy(szTarget, szSource1);
    strcat(szTarget, " - ");
    strcat(szTarget, szSource2);
}
```

Et voici maintenant la sortie que vous devriez obtenir :

```
Ce programme concatène deux chaînes.
(Cette version ne devrait pas planter.)
<Ceci est une chaîne - CECI EST UNE CHAINE>
Appuyez sur une touche pour continuer...
```

Félicitations ! Vous voilà devenu un expert du débogage.

Troisième partie

Programmer avec classe

Dans cette partie...

La grande différence entre le C++ et d'autres langages informatiques, c'est qu'il supporte la programmation orientée objet. La *programmation orientée objet* est un de ces termes qu'on entend à tout bout de champ dans le monde des ordinateurs (je sais, *.com* l'a certainement supplanté). Les langages, les éditeurs et les bases de données sont paraît-il tous "orientés objet". C'est parfois vrai, mais c'est souvent faux.

Pourquoi cet engouement pour tout ce qui est "orienté objet" ? Lisez et vous le comprendrez.

Chapitre 11

Analyse d'un programme orienté objet

Dans ce chapitre :

- Cuisiner des nachos.
- Vue d'ensemble de la programmation orientée objet.
- Présentation des abstractions et des classifications.
- Découvrir pourquoi la programmation orientée objet est si importante.

Qu'est-ce qu'exactement la programmation orientée objet ? Cette technique de programmation, que d'aucuns appellent familièrement POO, découle de deux principes que vous avez appris avant même d'avoir quitté vos couches-culottes : l'abstraction et la classification. Avant de développer ces notions, permettez-moi vous raconter une petite histoire.

Une histoire de four à micro-ondes

Parfois, quand mon fils et moi regardons un match de foot (ce qui n'arrive que quand ma femme ne trouve pas la télécommande), je m'empiffre de nachos : je pose des frites dans un plat, j'y ajoute des haricots, du fromage, ce qu'il faut de jalapeños, puis je mets tout ça dans le four à micro-ondes pendant cinq minutes.

Pour utiliser mon four à micro-ondes, j'ouvre la porte, je pose le plat à l'intérieur et j'appuie sur quelques boutons. Après plusieurs minutes, les nachos

sont prêts (j'évite de rester près du four pendant qu'il fonctionne de crainte que mes yeux ne se mettent à briller dans le noir).

Voyons maintenant tout ce que je dois faire pour utiliser mon four à micro-ondes :

- Je n'ai pas à intervenir à l'intérieur du four pour le faire fonctionner. Il est en effet équipé d'une interface : le panneau frontal avec ses boutons et le petit afficheur de la minuterie, qui me permettent de tout régler comme je l'entends.
- Je n'ai pas à reprogrammer le logiciel qui pilote le petit circuit intégré du four, même si j'ai cuisiné un autre plat la dernière fois que je l'ai utilisé.
- Je n'ai pas à regarder dans le boîtier du four.
- Même si j'étais un fabricant de fours à micro-ondes et que je savais tout ce qui se passe dedans, notamment au niveau de son programme, je l'utiliserais pour chauffer mes nachos sans me poser de question sur la façon dont tout cela peut bien fonctionner.

Ces observations ne brillent pas par leur profondeur. Pour limiter la quantité de choses qui vous incombent, vous choisissez en effet de travailler à un certain niveau de détails en mettant de côté ou en reportant ce qui est secondaire. En programmation orientée objet (POO), le niveau de détails auquel vous travaillez est appelé *niveau d'abstraction*. En d'autres termes, j'abstrais des détails internes du four à micro-ondes.

Quand je prépare des nachos, mon four se présente sous la forme d'une boîte. Aussi longtemps que je ne l'utilise qu'au travers de son interface (le panneau), rien ne devrait entraîner un fonctionnement imprévu de cet engin ou un plantage, ou pire : la transformation de mes nachos en pâte complètement brûlée et fumante.

La préparation fonctionnelle des nachos

Supposons que je demande à mon fils d'écrire un algorithme qui explique comment Papa fait ses nachos. Une fois qu'il a compris ce que j'attends de lui (mouline, mouline... crouic, crouic...), il écrira sans doute : "Ouvrir une boîte de haricots, râper du fromage, découper les jalapeños", et ainsi de suite. Quand il s'agira d'exposer cette décoction aux micro-ondes, il écrira certainement un truc du genre "Cuire au micro-ondes pendant cinq minutes" (si c'est un bon jour...).

Cette description est claire, nette et précise. Mais ce n'est pas comme cela qu'un programmeur digne de ce nom coderait un programme qui prépare des nachos. Un programmeur *fonctionnel* vit dans un univers dévolu à des objets appelés "four à micro-ondes" et autres appareils électroménagers. Il est quelque peu obnubilé par des organigrammes qui se ramifient à l'infini. Pour lui, la résolution fonctionnelle de la préparation des nachos passe par un flux continu de contrôles qui s'écoulent au travers de mes doigts et du panneau avant de s'éparpiller dans les profondeurs du four. Très vite, on retrouvera ce flux au niveau des données qui contrôlent la durée et la vitesse de rotation du magnétron et du plateau, et le moment où un joyeux "ding" indique que c'est prêt.

Dans un monde comme celui-ci, il est difficile de penser en termes de niveaux d'abstraction. Il n'y a pas d'objet, pas d'abstraction derrière lesquels cacher la complexité de toute cette procédure.

La préparation de nachos orientés objet

Dans une approche orientée objet de la recette des nachos, je commence par identifier les types d'objets du problème : des frites, des haricots, du fromage et le four. Puis, j'entreprends la modélisation logicielle de ces objets, sans trop m'attarder sur la manière dont ils seront utilisés dans le programme final.

Pendant que j'effectue cette tâche, je suis sensé travailler (et penser) au niveau des objets de base. Je dois penser à la manière de rendre le four utilisable, mais je n'ai pas encore à penser au processus logique qui conduit à la préparation des nachos. Après tout, les concepteurs du four n'ont pas besoin de réfléchir au problème spécifique qui consiste à préparer un casse-croûte. Leur travail, c'est de concevoir et de réaliser un four qui fonctionne.

Après avoir dûment codé et testé les objets dont j'ai besoin, je peux passer au prochain niveau d'abstraction et commencer à penser à la préparation des nachos, et non à la fabrication d'un four à micro-ondes. Arrivé à ce point, il m'est facile de traduire les instructions de mon fils directement en langage C++.

Classer les fours à micro-ondes

La classification est un concept aussi important que l'abstraction. Si je demande à mon fils "Les micro-ondes, ça te dit quoi ?", il me répondra sans doute "C'est ce qu'il y a dans un four...". Et je lui demanderai "Oui, mais, c'est quoi un four ?" et il dira : "C'est un appareil de cuisine qui...". Là, si je lui

demande ce qu'est un appareil de cuisine, il me répondra certainement : "Pourquoi est-ce que tu poses autant de questions stupides ?"

Les réponses de mon fils au sujet de ce qu'est pour lui notre four à micro-ondes sont fondées sur l'idée générale qu'il se fait des fours de ce type. De plus, à ses yeux, un four à micro-ondes n'est jamais qu'un type de four particulier, qui n'est lui-même qu'un type particulier d'appareil électroménager installé dans la cuisine.

En programmation orientée objet, mon four à micro-ondes est une *instance* de la classe des fours à micro-ondes. Celle-ci est elle-même une *sous-classe* de la classe four, qui est à son tour une sous-classe de la classe des appareils électroménagers.

L'être humain aime classer. Tout dans le monde qui nous entoure est assujetti à la taxinomie. Nous nous adonnons à ce vice pour réduire le nombre de choses dont nous devons nous rappeler. Souvenez-vous la première fois que vous avez vu une voiture dont nous tairons pudiquement la marque. La pub disait "Révolutionnaire ! Du jamais vu !" Mais vous et moi n'avons pas été dupes, nous savions bien qu'il n'en était rien. Bon, OK, elle était bien belle mais enfin, il y en a d'autres, et ce n'est jamais qu'une bagnole, avec tout ce qu'on y trouve : un volant, des sièges, un moteur, des freins, etc. Je parie même qu'on peut la conduire sans consulter le manuel !

Je n'ai donc pas à encombrer mon esprit avec tout ce que cette voiture a de commun avec d'autres. Tout ce que j'ai à me rappeler, ce sont les caractéristiques qui la distinguent des autres, car il y en a (notamment l'affichette qui indique son prix). Les voitures sont une sous-classe des véhicules à roues, à laquelle appartiennent aussi les camions et les camionnettes. Les véhicules à roues sont peut-être une sous-classe des véhicules, qui comprennent les navires et les avions. Et ainsi de suite...

Pourquoi classer ?

Oui, pourquoi vouloir tout classer ? Ça n'apporte que des problèmes. Mais comme le genre humain a adopté cette approche fonctionnelle depuis la nuit des temps, pourquoi la remettre maintenant en question ?

Il semblerait plus facile de concevoir et construire un four à micro-ondes spécialement pour résoudre le problème des nachos, plutôt que de construire un objet "four" distinct, générique. Supposons que je doive concevoir un four à micro-ondes destiné à la préparation des nachos, et uniquement des nachos. Il serait inutile de l'équiper d'un panneau de commandes et un bouton Marche serait amplement suffisant. Comme la durée de cuisson de mes nachos est

toujours la même, je pourrais me passer de tous ces sacrés boutons poétiquement nommés Décongélation ou encore Minuterie. Il suffirait d'y fourrer mon plat. Un quart de mètre cube entièrement consacré aux nachos...

Je pourrais même me dispenser complètement du concept de "four à micro-ondes". Tout ce qu'il me faut, c'est un four, quel qu'il soit. Dans ma recette, je précise : "Mettre les nachos dans la boîte, connecter le fil électrique rouge au fil noir, porter le magnétron à environ 3 000 volts. Remarquez un léger bourdonnement. Ne pas vous tenir trop près si vous avez l'intention d'avoir un jour des enfants". Bref, ce genre de littérature...

Mais l'approche fonctionnelle n'est pas dénuée d'inconvénients :

- **Elle est trop complexe.** Je n'ai pas du tout envie de mêler des considérations techniques sur la fabrication des fours à la préparation de mes nachos. Si je ne puis définir les objets en les débarrassant de la gangue de détails où ils sont embourbés, je vais devoir me confronter à la complexité de tous ces problèmes en même temps.

- **Elle manque de souplesse.** Je serai sans doute amené un jour à remplacer mon four à micro-ondes par un autre d'un modèle différent. Je dois être capable de continuer à préparer mes nachos du moment que l'interface reste la même. Si un objet n'a pas été clairement décrit et défini, il est impossible d'envisager efficacement son remplacement par un autre.

- **La réutilisation est impossible.** Les fours permettent de cuisiner des plats différents. Je n'ai pas du tout l'intention de devoir créer un nouveau four chaque fois que je trouverai une nouvelle recette. Après avoir résolu un problème une première fois, je veux bénéficier de la possibilité de réutiliser la solution dans d'autres programmes.

Dans les autres chapitres de cette partie, nous allons voir comment la programmation orientée objet permet de résoudre tous ces problèmes.

Chapitre 12

Ajouter des classes au C++

Dans ce chapitre :

- Grouper des données en classes.
- Déclarer et définir des membres de classe.
- Accéder aux membres d'une classe.

Les programmes gèrent souvent des groupes de données : le nom d'une personne, un rang, un numéro de série et autres informations de ce genre. Une seule de ces valeurs est insuffisante pour décrire une personne ; seule leur agrégation donne du sens. Une structure simple, comme par exemple un tableau, est suffisante pour contenir des valeurs autonomes. Mais elles ne sont pas adaptées à la gestion de groupes de données. Les bons vieux tableaux ne conviennent pas du tout au stockage de données complexes, comme les numéros de cartes de crédit que les multinationales s'efforcent de protéger des assauts de hordes de pirates boutonneux prépubères.

Pour des raisons qui seront plus claires d'ici peu, j'appelle de telles données groupées un *objet*. Nous avons vu dans le Chapitre 11 qu'un four à micro-ondes est un objet. De ce point de vue, vous êtes aussi un objet. Votre nom, votre profession, votre numéro de carte de crédit : objets que tout cela !

Présentation des classes

Nous avons besoin d'une structure capable de contenir les différents types de données nécessaires pour décrire un seul objet. Dans notre petit exemple, un

seul objet contiendrait simultanément le prénom, le nom ainsi que le numéro de la carte de crédit.

Le C++ appelle *classe* la structure qui combine les différents éléments de données en un seul objet.

Le format d'une classe

Une classe susceptible de décrire le nom de quelqu'un et son numéro de carte de crédit se présenterait sous la forme suivante :

```
// La classe d'un ensemble de données
class NameDataSet
{
   public:
      char firstName [128];
      char lastName  [128];
      int  creditCard;
};

// Une instance unique de cet ensemble de données
NameDataSet nds;
```

Une définition de classe commence par le mot-clé `class` suivi du nom de la classe et d'une paire d'accolades ouverte et fermée.

L'instruction après l'accolade ouverte est le mot-clé `public`. (Ce terme est expliqué un peu plus loin. Dans les chapitres à venir, nous aborderons aussi les options de `public`, comme `private`. C'est pourquoi, *public* doit rester privé jusqu'à ce que je puisse rendre *private* public.)

Il est aussi possible d'utiliser le synonyme `struct`. Les mots-clés `class` et `struct` sont identiques, sauf que pour *`struct`*, la déclaration *`public`* est implicite. Mais je vous conseille de faire comme la plupart des programmeurs : restez-en au terme `class` (pour des raisons que vous comprendrez mieux un peu plus loin dans ce chapitre).

Après le mot-clé `public` se trouvent les entrées qui décrivent l'objet. La classe `NameDataSet` contient des définitions pour le nom et le prénom ainsi que pour le numéro de carte de crédit. Comme vous l'avez deviné, le nom et le prénom sont des tableaux de caractères ; le numéro de carte de crédit est un nombre entier.

Une déclaration de classe comprend les données indispensables à la description d'un objet unique.

La dernière ligne du listing déclare la variable `nds` comme unique entrée de la classe `NameDataSet`. De ce fait, `nds` pourrait être une entrée décrivant une certaine personne.

La variable `nds` est une *instance* de la classe `NameDataSet`. Vous *instanciez* cette classe pour créer `nds`. Enfin, `firstName` et les autres variables sont des *membres* ou *propriétés* de la classe.

Accéder aux membres d'une classe

La syntaxe suivante permet d'accéder aux propriétés d'un objet spécifique :

```
NameDataSet nds;
nds.creditCard = 10;
cin >> nds.firstName;
cin >> nds.lastName;
```

Ici, `nds` est une instance de la classe `NameDataSet` (par exemple, un objet `NameDataSet` particulier). L'entier `nds.creditCard` est une propriété de l'objet `nds`. Le type de `nds.creditCard` est un entier `int` alors que celui de `nds.firstName` est `char[]`.

On est en plein jargon de programmeur... Voyons ce qui se passe ici. Cet extrait de listing déclare un objet `nds` que nous utiliserons pour décrire un client. Pour quelque bonne raison, le programme affecte à la carte de crédit de cette personne le numéro 10 (il est manifestement factice, mais cela vaut mieux que de mettre son propre numéro de carte bancaire).

Ensuite, le programme lit le prénom et le nom de la personne à partir de l'entrée par défaut.

J'utilise un tableau de caractères et non la classe `string` pour gérer le nom.

Jusqu'à présent, le programme est capable de se référer au seul objet `nds` sans avoir à s'occuper de ses éléments distinctifs (le prénom, le nom et le numéro de carte de crédit) aussi longtemps qu'il n'en a pas besoin.

Le programme qui suit montre comment on peut utiliser la classe `NameDataSet` :

```
// DataSet - mémorise des données associées
//           dans un tableau d'objets
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string.h>
using namespace std;

// NameDataSet - enregistre des informations sur le nom,
//               le prénom et le numéro de carte de crédit
class NameDataSet
{
  public:
    char firstName[128];
    char lastName [128];
    int  creditCard;
};

// prototypes des fonctions
bool getData(NameDataSet& nds);
void displayData(NameDataSet& nds);

int main(int nNumberofArgs, char* pszArgs[])
{
    // alloue de l'espace pour 25 jeux de données
    const int MAX = 25;
    NameDataSet nds[MAX];

    // charge les noms, prénoms et numéros
    // de carte de crédit
    cout << "Lecture du nom et de la carte bancaire\n"
         << "Entrez 'quitter' to terminer"
         << endl;
    int index = 0;
    while (getData(nds[index]) && index < MAX)
    {
        index++;
    }

    // affiche les noms et numéros entrés
    cout << "\nInformations saisies : " << endl;
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i]);
    }

    // attend pour terminer le programme que l'utilisateur
```

```
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

// getData - remplit un objet NameDataSet
bool getData(NameDataSet& nds)
{
    cout << "\nEntrez le prénom : ";
    cin  >> nds.firstName;

    // si on veut terminer (la casse n'intervient pas)...
    if (stricmp(nds.firstName, "quitter") == 0)
    {
        return false;
    }

    cout << "Entrez le nom : ";
    cin  >> nds.lastName;

    cout << "Entrez le numéro de la carte bancaire : ";
    cin  >> nds.creditCard;

    return true;
}

// displayData - affiche un ensemble de données
void displayData(NameDataSet& nds)
{
    cout << nds.firstName
         << " "
         << nds.lastName
         << "/"
         << nds.creditCard
         << endl;
}
```

La fonction `main()` alloue vingt-cinq objets de la classe `NameDataSet`. Ensuite, `main()` demande à l'utilisateur ce qu'il attend de lui, après quoi il entre dans une boucle au cours de laquelle les entrées effectuées au clavier sont lues grâce à la fonction `getData()`. La boucle se termine si `getData()` retourne un 0 (`false`) ou si le nombre maximum d'objets, c'est-à-dire vingt-cinq, a été créé. Ces mêmes objets qui viennent d'être lus sont ensuite passés à `displayData(NameDataSet)` afin d'être affichés.

La fonction `getData()` accepte un objet `NameDataSet` comme argument d'entrée auquel il affecte le nom `nds`.

Ignorez pour le moment l'esperluette (&). Nous en reparlerons dans le Chapitre 14.

Ensuite, `getData()` lit la chaîne contenue dans l'entrée standard `firstName`. Si la fonction `stricmp()` ne trouve aucune différence entre le nom saisi et "quitter", elle retourne 0 à `main()`, indiquant ainsi que le moment est venu de quitter. (La fonction `stricmp()` compare deux chaînes sans différencier leur casse. Pour elle, "quitter" et "QUITTER" sont identiques, de même que toutes les autres variantes possibles de majuscules et de minuscules.) Sinon, la fonction continue son travail en lisant le nom et le numéro de carte de crédit à partir de l'objet `nds`.

La fonction `displayData()` affiche séparément chaque membre de l'objet `NameDataSet` avec des délimiteurs.

Une exécution de ce programme pourrait donner ceci :

```
Lecture du nom et de la carte bancaire
Entrez 'quitter' to terminer

Entrez le prénom : Stephen
Entrez le nom : Davis
Entrez le numéro de la carte bancaire : 123456

Entrez le prénom : Marshall
Entrez le nom : Smith
Entrez le numéro de la carte bancaire : 567890

Entrez le prénom : Quitter

Informations saisies :
Stephen Davis/123456
Marshall Smith/567890
Appuyez sur une touche pour continuer...
```

Le programme affiche d'abord une petite explication. J'ai commencé par entrer mon prénom dès la première invite (quelle modestie !). Comme mes parents n'ont pas eu l'idée incongrue de me prénommer "Quitter", le programme se poursuit. J'ai ensuite tapé mon nom et le supposé numéro de carte de crédit. Lors de la passe suivante, j'ai entré un certain Marshall Smith et son véritable numéro de carte (tu vas rigoler en lisant ton relevé bancaire, Marshall). Enfin, j'ai tapé le fatidique "Quitter" qui fait sortir de la boucle de saisie. Le programme ne fait ensuite que restituer les informations que je viens d'entrer.

Chapitre 13

Rendre les classes opérationnelles

Dans ce chapitre :

- Ajouter des propriétés actives aux classes.
- Déclarer et définir une fonction membre.
- Accéder aux fonctions membres d'une classe.
- Surcharger des fonctions membres.

Les programmeurs utilisent les classes pour réunir des éléments de données liés en un seul objet. Ci-dessous, la classe `Savings` associe un solde de compte d'épargne à un unique numéro de compte :

```
class Savings
{
   public:
    unsigned accountNumber;
    float balance;
};
```

Chaque instance de `Savings` contient les mêmes deux éléments de données :

```
void fn(void)
{
   Savings a;
   Savings b;
```

```
    a.accountNumber = 1; // Ceci n'est pas identique à...
    b.accountNumber = 2; // ...cela.
}
```

La variable `a.accountNumber` est différente de la variable `b.accountNumber`. Le solde de mon compte d'épargne est en effet différent du vôtre, même si dans les deux cas le terme est le même (solde). Encore que dans mon cas particulier, le terme "découvert" serait plus juste.

Activer les objets

Les classes servent à simuler des objets du monde réel. Cette méthode vous permet de réfléchir en termes d'entités et non plus de lignes de code. Plus les objets C++ sont proches de ceux de la réalité, plus il est facile de les programmer. Tout ça a l'air simple, mais `Savings` ne reproduit toutefois pas très bien le fonctionnement d'un compte d'épargne.

Simuler des objets du monde réel

Les objets du monde réel ont assurément des propriétés de types de données tels que des numéros de compte ou des soldes. C'est pourquoi la classe `Savings` est un bon point de départ pour décrire un objet "vrai". Mais les objets de notre réalité peuvent aussi effectuer des actions : les fours cuisent, les comptes d'épargne produisent des intérêts, ou alors ils vous infligent de substantielles pénalités à chaque retrait.

Un programme fonctionnel "effectue des actions" au travers de fonctions. Un programme en C++ peut par exemple appeler `strcmp()` pour comparer deux chaînes, ou encore `max()` pour retourner le plus grand de deux nombres. Vous apprendrez dans le Chapitre 24 que même les flux d'entrée/sortie `cin >>` et `cout <<` sont des variantes particulières de la fonction d'appel `call`.

La classe `Savings` nécessite ses propres propriétés actives si nous voulons qu'elle constitue une représentation efficace du monde réel :

```
class Savings
{
    public:
        float deposit(float amount)
        {
```

```
            balance += amount;
            return balance;
        }

        unsigned int accountNumber;
        float  balance;
};
```

En plus du numéro de compte et du solde, cette version de `Savings` comprend une fonction `deposit()`. Elle permet à `Savings` de contrôler son propre avenir. Une classe `FourMicroOndes` pourrait avoir une fonction `cuisson()` ; dans cet esprit, la classe `Savings` est dotée d'une fonction `accumulateInterest()`. Quand à la classe `CD()`, elle possède une fonction `penalizeForEarlyWithdrawal()` (N.D.T. : "pénalisation pour retrait anticipé").

Les fonctions définies dans une classe sont appelées *fonctions membres*.

À quoi bon utiliser des fonctions membres ?

Pourquoi devrions-nous nous compliquer l'existence avec des fonctions membres ? Qu'est-ce qu'elles apportent de plus ? Examinons ce listing :

```
class Savings
{
   public:
    unsigned accountNumber;
    float balance;
};
unsigned deposit(Savings& s, unsigned amount)
{
 s.balance += amount;
 return s.balance;
}
```

Ignorez l'esperluette (`&`) pour le moment. Nous y reviendrons dans le Chapitre 14.

Ici, `deposit()` implémente la fonction "dépôt sur le compte d'épargne". Cette solution fonctionnelle s'appuie sur une fonction extérieure, `deposit()`, pour implémenter une activité de gestion de comptes qui manque à `Savings`. La tâche est en effet réalisée, mais en infraction avec les règles de la programmation orientée objet.

Un four à micro-ondes est équipé de composants internes qui "savent" comment chauffer, décongeler et rendre les plats croustillants. Les membres de données d'une classe sont comme les pièces d'un four : les fonctions membres d'une classe exécutent des fonctions comparables à celles de la cuisson.

Quand je prépare des nachos, je n'ai pas à raccorder entre eux selon un certain schéma les différents éléments internes du four pour que ce dernier fonctionne. Je veux qu'il en soit de même pour mes classes. Il faut qu'elles sachent manipuler leurs petites salades internes sans aucune intervention extérieure.

Les fonctions membres de `Savings`, comme `deposit()`, peuvent être écrites sous la forme de fonctions externes. J'ai la possibilité de réunir toutes les fonctions nécessaires pour que le compte d'épargne fonctionne tout seul. Un four à micro-ondes peut bien sûr fonctionner après que l'on a dénudé et soudé des fils électriques. Mais il n'est pas question qu'il en soit ainsi pour le four que je viens d'acheter ou pour mes classes. Je veux une classe `Savings` utilisable pour mon programme bancaire sans avoir à me demander ce qui se passe à l'intérieur.

Ajouter une fonction membre

Deux aspects doivent être pris en considération lors de l'ajout d'une fonction membre à une classe : la création de la fonction membre et le nom qu'elle aura (ça a l'air tout bête, n'est-ce pas ?).

Créer une fonction membre

Pour illustrer les fonctions membres, nous commencerons par représenter un étudiant "virtuel" en définissant une classe `Student`. Voici une des représentations possibles :

```
class Student
{
    public:
        // ajout d'un cours à l'enregistrement
        float addCourse(int hours, float grade)
        {
            // calcule la somme de tous les cours
            // fois la moyenne générale des notes
            float weightedGPA;
            weightedGPA = semesterHours * gpa;
```

```
            // ajout dans le nouveau cours
            semesterHours += hours;
            weightedGPA += grade * hours;
            gpa = weightedGPA / semesterHours;

            // retourne la nouvelle moyenne
            return gpa;
        }

        int  semesterHours;
        float gpa;
};
```

La fonction `addCourse(int, float)` est appelée "fonction membre de la classe `Student`". Il s'agit en principe d'une propriété de la classe tout comme les membres de données `semesterHours` et `gpa` (N.D.T. : GPA sont les initiales de *Grade Point Average,* c'est-à-dire "moyenne générale des notes").

Il n'existe aucun nom particulier pour les fonctions ou les données qui ne sont pas membres d'une classe, mais je les appellerai néanmoins *non-membres.*

Il n'est pas nécessaire que les fonctions membres précèdent les membres de données, comme dans cet exemple. L'ordre peut être quelconque, mais je préfère placer les fonctions en premier.

Pour des raisons historiques, les fonctions membres sont aussi appelées *méthodes.* Ce terme provient de l'un des premiers langages orientés objet, mais il n'a aucune signification particulière en C++. Le mot a néanmoins acquis une certaine popularité dans les cercles de la POO, car il est plus facile à dire que "fonction membre" (sans compter qu'il en impose davantage). Donc, si à table vos collègues commencent à parler de "méthode de la classe", remplacez simplement "méthode" par "fonction membre" et corrigez-les sans hésiter. Comme le terme "méthode" n'est que peu pertinent en C++, je ne l'utiliserai pas ici.

Nommer les membres de classe

Une fonction membre est un peu comme le membre d'une famille. Le nom complet de la fonction `addCourse(int, float)` est `Student::addcourse(int, float)`, tout comme mon nom complet est Stephen Davis. Le diminutif de la fonction est `addCourse(int, float)`, tout comme mon prénom est Stephen. Le nom au début du nom complet indique que la fonction est un membre de la

classe `Student`. Les deux-points (`::`) entre le nom de classe et le nom de la fonction ne sont que des séparateurs. Le nom Davis à la fin de mon nom complet indique que je suis un membre de la famille Davis.

Le "nom complet" est aussi appelé *nom étendu*.

Vous pouvez définir une fonction `addCourse(int, float)` qui n'a rien à voir avec `Student` ; il existe bien des Stephen qui n'ont rien de commun avec ma famille (je dis ça en toute connaissance de cause car je connais quelques Stephen qui n'ont vraiment pas envie d'être confondus avec ma famille et surtout avec moi).

Vous pourriez par exemple vous retrouver avec une fonction `Prof::addCourse(int, float)`, voire même `Golf::addCourse()`. Une fonction `addCourse(int, float)`, dépourvue de nom de classe, n'est jamais qu'une bonne vieille convention pour une fonction non-membre.

Le nom étendu pour la fonction non-membre est `::addCourse(int, float)` (remarquez les deux-points sans nom de famille devant).

Appeler une fonction membre

Avant de voir comment une fonction membre est appelée, souvenez-vous de la manière dont on accède à un membre de données :

```
class Student
{
 public:
  int semesterhours;
  float gpa;
};

Student s;
void fn(void)
{
  // Accès aux membres de données de s
  s.semesterHours = 10;
  s.gpa     = 3.0;
}
```

Remarquez que vous devez spécifier un objet avec le nom de membre. Autrement dit, ce qui suit est complètement erroné :

```
Student s;
void fn(void)
{
  // Rien de cela n'est admis
  semesterHours = 10; // Membre de quel objet de
                      // quelle classe ?
  Student::semesterHours = 10; // OK, je connais
         // la classe, mais toujours pas l'objet.
}
```

Accéder à une fonction membre

Rappelez-vous que les fonctions membres agissent fonctionnellement de la même manière que les membres de données. La fonction `CallMemberFunction` qui suit montre comment invoquer la fonction membre `addCourse()` :

```
//  CallMemberFunction - définit et invoque une fonction
//                       qui est un membre de la classe Student
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
    public:
        // ajoute un cours complet à l'enregistrement
        float addCourse(int hours, float grade)
        {
            // calcule la somme de tous les cours
            // fois la moyenne générale des notes
            float weightedGPA;
            weightedGPA = semesterHours * gpa;

            // ajout dans le nouveau cours
            semesterHours += hours;
            weightedGPA += grade * hours;
            gpa = weightedGPA / semesterHours;

            // retourne la nouvelle moyenne
            return gpa;
        }
```

```
        int  semesterHours;
        float gpa;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.semesterHours = 10;
    s.gpa     = 3.0;

    // valeurs avant l'appel
    cout << "Avant   : s = (" << s.semesterHours
         << ", "
         << s. gpa
         << ")"
         << endl;

    // soumet les membres de données de l'objet s
    // à la fonction membre addCourse()
    s.addCourse(3, 4.0); // appelle la fonction membre

    // les valeurs sont maintenant modifiées
    cout << "Ensuite : s = (" << s.semesterHours
         << ", "    << s. gpa
         << ")"     << endl;

    // accède à un autre objet juste pour voir
    Student t;
    t.semesterHours = 6;
    t.gpa      = 1.0;    // pas terrible
    t.addCourse(3, 1.5); // la situation ne s'améliore pas


    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La syntaxe d'un appel à une fonction membre ressemble à un hybride entre la syntaxe d'accès à un membre de données et celle d'un appel de fonction. La partie à droite du point est la même que celle d'un appel de fonction conventionnel, mais un objet se trouve à gauche.

De même que la phrase "mi-vitesse" ne prend son sens qu'en la décomposant, "`s` est l'objet sur lequel agit `addCourse()`" ou, exprimé différemment, `s` est l'étudiant auquel un cours est ajouté. Il est impossible d'ajouter les heures

semestrielles sans savoir à quel étudiant il faut les attribuer. Et vous ne pouvez ajouter un étudiant à un cours si vous ne savez pas de quel étudiant il s'agit.

L'appel d'une fonction membre sans un objet n'a pas plus de sens que de référencer un membre de données sans un objet.

Accéder aux autres membres à partir d'une fonction membre

Je vous vois venir. Vous vous répétez : "Accéder à un membre sans un objet n'a pas de sens, accéder à un membre sans un objet n'a pas de sens, accéder...". Au moment même où vous avez bien intégré cette règle, vous vous penchez sur la fonction membre `Student::addCourse()` et là, *paf !* le choc : `addCourse()` accède à d'autres membres de classe sans référence à un objet. Comment est-ce possible ? (N.D.T. : Fernand Raynaud aurait répondu : "C'est étudié pour".)

Alors, en fin de compte, on peut ou on ne peut pas ? Croyez-moi, on ne peut pas. Quand vous référencez un membre de `Student` depuis `addCourse()`, cette référence est contre l'objet `Student` avec lequel l'appel à `addCourse()` a été fait. Pas clair ? Revenons à l'exemple `CallMemberFunction`. Les sections critiques sont reprises ici :

```
int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.semesterHours = 10;
    s.gpa       = 3.0;
    s.addCourse(3, 4.0); // appelle la fonction membre

    Student t;
    t.semesterHours = 6;
    t.gpa       = 1.0;    // pas terrible
    t.addCourse(3, 1.5); // la situation ne s'améliore pas

    system("PAUSE");
    return 0;
}
```

Lorsque `addCourse()` est invoqué par l'objet `s`, toutes les autres références de membre non qualifiées de `addCourse()` se réfèrent aussi à `s`. Par conséquent, la référence à `semesterHours` dans `addCourse()` concerne `s.semesterHours` et `gpa` représente `s.gpa`. Mais quand la fonction `addCourse()` est invoquée avec l'objet `Student t`, ces mêmes références s'appliquent dans ce cas à `t.semesterHours` et à `t.gpa`.

L'objet avec lequel la fonction membre a été appelée est l'objet "courant" ; toutes les références non qualifiées aux membres de classe se réfèrent à cet objet. Dit autrement, les références non qualifiées aux membres de classe faites par une fonction membre sont toujours contre l'objet courant.

Nommer l'objet courant

Comment une fonction membre sait-elle ce qu'est l'objet courant ? Il n'y a rien de sorcier : l'adresse de l'objet est passée à la fonction membre en tant que premier argument implicite caché. Autrement dit, il se produit la conversion suivante :

```
s.addCourse(3, 2.5) est identique à
Student::addCourse(&s, 3, 2.5)
```

Notez que vous ne pouvez pas utiliser la seconde syntaxe ; c'est simplement une façon de montrer comment le C++ la voit.

À l'intérieur de la fonction, le pointeur implicite vers l'objet courant a un nom, au cas où vous devriez y faire référence. C'est une logique de type "Quel objet ? *Cet* objet". Vous suivez ? Le type de ce "*cet*" est toujours un pointeur vers un objet de la classe appropriée.

Chaque fois qu'une fonction membre fait référence à un autre membre de la même classe sans fournir explicitement un objet, C++ suppose l'existence de *cet* objet. Vous pouvez y faire explicitement référence si vous le désirez. La fonction `Student::addCourse()` aurait dans ce cas pu être écrite ainsi (sachant que *cet* se dit *this* en anglais) :

```
float Student::adCourse(int hours, float grade)
{
  float weightedGPA;
  weightedGPA = this->semesterHours * cet->gpa;

  // Ajout d'un nouveau cours :
  this->semesterHours += hours;
  weightedGPA += hours * grade;
  this->gpa = weightedGPA / this->semesterHours;
  return this-> gpa;
}
```

L'effet est le même, que vous ayez explicitement inclus *cet* objet (`this->`), comme dans l'exemple ci-dessus, ou qu'il l'ait été implicitement comme vous l'aviez fait auparavant.

La résolution de portée

Les signes :: entre un membre et sa classe forment ce que l'on appelle un *opérateur de résolution de portée* car ils indiquent la portée de la classe à laquelle le membre appartient. Le nom de classe qui précède ces signes est l'équivalent d'un nom de famille, tandis que le nom de fonction qui les suit serait le prénom. L'ordre est similaire à celui de l'état civil officiel : le nom d'abord, puis le prénom.

Vous pouvez utiliser l'opérateur :: pour décrire une fonction non-membre à l'aide d'un nom de classe vide. Il est par exemple possible de se référer à la fonction non-membre `addCourse` sous la forme `::addCourse(int, float)` si vous le préférez. C'est comme une fonction sans domicile fixe.

Normalement, l'opérateur :: est facultatif, sauf dans quelques circonstances. Par exemple :

```
// addCourse - combine les variables hours et grade dans
//             un grade pondéré.
float addCourse(int hours, float grade)
{
    return hours * grade;
}

class Student
{
  public:
    int  semesterHours;
    float gpa;

    // ajout d'un cours complet à l'enregistrement
    float addCourse(int hours, float grade)
    {
        // appel d'une fonction externe pour le calcul de
        // la moyenne pondérée.
        float weightedGPA = addCourse(semesterHours, gpa);

        // ajout dans le nouveau cours
        semesterHours += hours;

        // utilisation de la même fonction pour calculer
        // la note pondérée de ce nouveau cours.
        weightedGPA += addCourse(hours, grade);
        gpa = weightedGPA / semesterHours;
```

```
        // retourne le nouveau gpa.
        return gpa;
    }
};
```

Ici, je veux que la fonction membre `Student::addCourse()` appelle la fonction non-membre `::addCourse()`. Mais, sans l'opérateur `::`, un appel à `addCourse()` depuis `Student` se réfère à `Student::addCourse()`.

Une fonction membre peut utiliser un nom court lorsqu'elle se réfère à un autre membre de la même classe. C'est comme dans la famille Davis : on s'appelle par notre prénom tout en se comprenant.

Dans ce cas, l'absence du nom de classe fait que la fonction s'appelle elle-même, ce qui n'est généralement pas une bonne chose. Le préfixe `::` dirige l'appel vers la version globale, comme il se doit :

```
class Student
{
  public:
    int  semesterHours;
    float gpa;

    // ajout d'un cours complet à l'enregistrement
    float addCourse(int hours, float grade)
    {
        // appel d'une fonction externe pour calculer
        // la note pondérée.
        float weightedGPA = ::addCourse(semesterHours, gpa);

        // ajout dans le nouveau cours
        semesterHours += hours;

        // utilisation de la même fonction pour calculer
        // la note pondérée de ce nouveau cours.
        weightedGPA += ::addCourse(hours, grade);
        gpa = weightedGPA / semesterHours;

        // retour du nouveau gpa
        return gpa;
     }
};
```

C'est un peu comme si je criais "Stephen" chez moi ; personne ne peut se douter qu'il s'agit de moi-même car on sait seulement que je m'appelle Davis. Pour les gens, c'est mon nom par défaut. Si j'étais censé appeler un autre Stephen n'appartenant pas à ma famille, je devrais dire "Stephen Smith" ou "Stephen Jones"... C'est ce que fait l'opérateur de résolution de portée.

Le nom étendu d'une fonction comporte ses arguments. Nous avons maintenant ajouté le nom de la classe à laquelle appartient la fonction.

Définir une fonction membre dans une classe

Une fonction membre peut être définie dans une classe ou à part. Lorsqu'une définition de classe est définie dans la classe, elle ressemble à celle du fichier inclus `Savings.h` :

```
// Savings - définit une classe incluant la possibilité
//           de faire un dépôt

class Savings
{
    public:
        float deposit(float amount)
        {
            balance += amount;
            return balance;
        }

        unsigned int accountNumber;
        float  balance;
};
```

L'ajout des fichiers `#include` se fait plutôt simplement. Un programme peut maintenant inclure la définition de classe (en même temps que la définition de la fonction membre), comme le prouve l'exemple qui suit :

```
//  SavingsClassInline - invoque une fonction membre qui est
//                       déclarée et définie dans la classe
//                       Savings
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;
#include "Savings.h"

int main(int nNumberofArgs, char* pszArgs[])
{
    Savings s;
    s.accountNumber = 123456;
    s.balance = 0.0;

    // ajoute quelque chose au compte
    cout << "Je mets 10 sur le compte " << s.accountNumber << endl;
    s.deposit(10);
    cout << "La balance vaut " << s.balance << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Voilà qui est génial car, mis à part le programmeur de la classe `Savings.h`, toute autre personne n'a plus qu'à se concentrer sur le dépôt d'argent sans se soucier des détails de la cuisine bancaire. Tout cela ne concerne plus que les fichiers inclus.

La directive `#include` insère le contenu lors du processus de compilation. Le compilateur C++ "voit" votre code comme si le fichier inclus `Savings.h` en faisait partie intégrante.

Conserver une fonction membre hors d'une classe

Pour de plus grandes fonctions, placer le code directement dans la définition de classe peut entraîner des définitions particulièrement vastes et encombrantes. Pour éviter cela, le C++ vous permet de définir des fonctions membres hors d'une classe.

Une fonction définie en dehors de la classe est dite *outline*. Ce terme s'oppose à la notion de fonction intégrée, *inline*, définie à l'intérieur de la classe.

Fonctions membres inline

Les fonctions membres définies dans une classe sont par défaut *inline*, littéralement *intégrées* (à moins qu'elles n'aient été spécifiquement mises *outline* par un commutateur du compilateur ou parce qu'elles contiennent une boucle). Le plus souvent, c'est parce qu'une fonction membre définie dans la classe est habituellement très petite et que ces petites fonctions se prêtent particulièrement bien à l'intégration.

Le contenu d'une fonction *inline* est inséré partout où elle est appelée. Une fonction *inline* s'exécute plus rapidement car le processeur n'est pas obligé de sauter à l'emplacement où la fonction est définie. En contrepartie, une fonction *inline* exige davantage de mémoire car elle est recopiée lors de chaque appel au lieu de l'être une fois pour toutes.

Il existe une bonne raison, certes plus technique, d'intégrer les fonctions membres définies à l'intérieur d'une classe. Rappelez-vous que les structures du langage C sont normalement définies dans des fichiers inclus qui sont ensuite insérés dans le code source C++ qui en a besoin. De tels fichiers ne devraient pas contenir de données ou de fonctions car ils sont compilés à plusieurs reprises. L'inclusion d'une fonction *inline* est toutefois admise car, à l'instar d'une macro-commande, elle subit en quelque sorte une expansion sur place dans le fichier source. Le problème cité plus haut est évité en acceptant les fonctions membres par défaut définies dans des intégrations de classes.

S'il est écrit hors d'une déclaration de classe, le fichier `Savings.h` peut exposer la fonction `deposit()` sans la définir, comme ceci :

```
// Savings2 - définit une classe incluant la possibilité
//           de faire un dépôt
class Savings
{
  public:
    // declare la fonction membre sans la définir
    float deposit(float amount);
    unsigned int accountNumber;
    float  balance;
};
```

La définition de la fonction `deposit()` doit être incluse dans un des fichiers sources qui composent le programme. Pour simplifier la question, je définis les

fonctions dans le fichier qui contient également `main()` (il s'agira ici de `SavingsClassOutline.cpp`).

```
//  SavingsClassOutline - invoque une fonction membre qui est
//                        déclarée et définie dans la classe
//                        Savings2 mais définie dans un autre fichier
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;
#include "Savings2.h"

// définit la fonction membre Savings::deposit()
// (elle est normalement contenue dans un fichier séparé
// qui est ensuite combiné avec le reste du code)
float Savings::deposit(float amount)
{
    balance += amount;
    return balance;
}

// le programme principal
int main(int nNumberofArgs, char* pszArgs[])
{
    Savings s;
    s.accountNumber = 123456;
    s.balance = 0.0;

    // ajoute quelque chose au compte
    cout << "Je mets 10 sur le compte " << s.accountNumber << endl;
    s.deposit(10);
    cout << "La balance vaut " << s.balance << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La définition de classe ne contient qu'une simple déclaration, un prototype, pour la fonction `deposit()`. Le code de cette dernière apparaît ailleurs. La déclaration du prototype de la fonction membre dans la structure est analogue à n'importe quelle autre déclaration de même type (et tout autant obligatoire).

Le nom `deposit()` était tout à fait suffisant lorsque la fonction était définie à l'intérieur de la classe. Dès l'instant où cette définition est placée en dehors de la classe, l'utilisation du nom étendu devient indispensable.

La surcharge des fonctions membres

Les fonctions membres peuvent être surchargées de la même manière que les fonctions conventionnelles (si vous avez oublié ces notions, relisez le Chapitre 6). Rappelez-vous cependant que le nom de la classe fait partie du nom étendu ; c'est pourquoi les fonctions suivantes sont toutes légales :

```
class Student
{
 public:
  // grade - retourne la moyenne générale (GPA) des notes.
  float grade();
  // grade - définit la note et retourne à la valeur
  //         précédente.
    float grade(float newGPA);
  // ...Membres de données et autres choses...
};
class Slope
{
 public:
  // grade --- retourne le pourcentage des notes
    float grade();
  // ...C'est là que vous placez votre code...
};

// grade - retourne la lettre équivalente à une note
char grade(float value)

int main(int argcs, char* pArgs[])
{
  Student s;
  s.grade(3.5);     // Student::grade(float)
  float v = s.grade(); // Student::grade()

  char c = grade(v);   // ::grade(float)

  Slope o;
  float m = o.grade(); // Slope::grade()
  return 0;
}
```

Chaque appel fait par `main()` est annoté dans les commentaires avec le nom étendu de la fonction appelée.

Quand vous appelez des fonctions surchargées, ce ne sont pas seulement les arguments de la fonction qui enlèvent toute ambiguïté à l'appel, mais aussi le type de l'objet (s'il existe) avec lequel la fonction est appelée. Le terme *désambiguïsation* appartient au jargon propre à la programmation orienté objet qui se traduit par "décider au moment de la compilation quelle fonction surchargée doit être appelée". D'autres disent aussi "différencier".

Dans l'exemple, les deux premiers appels aux fonctions membres `Student::grade(float)` et `Student::grade()` sont différenciés par leurs listes d'arguments et par le type d'objet utilisé. L'appel à `s.grade()` appelle `Student::grade()` car `s` est du type `Student`.

Le troisième appel n'a pas d'objet, de sorte que son absence d'ambiguïté dénote la fonction non-membre `::grade(float)`.

L'appel final est fait avec un objet de type `Slope`. Il doit se référer à la fonction membre `Slope::grade()`.

Chapitre 14

Créer des pointeurs vers des objets

Dans ce chapitre :

- Examen d'un objet fait de tableaux d'objets.
- Diriger des pointeurs vers des pointeurs d'objets.
- Rigueur avec les pointeurs.
- Naviguer à travers des listes d'objets.

Les programmeurs C++ ont toujours généré des tableaux avec diverses choses : des tableaux de nombres entiers, de nombres flottants... Et pourquoi pas des tableaux d'étudiants ? Après tout, les étudiants sont tout le temps disposés en rangs dans les amphis, et les tableaux – noirs –, ils connaissent... Le concept d'objet Étudiant, ou `Student` dans cet ouvrage, nous convient parfaitement. Pourquoi nous en priver ?

Définir des tableaux et des pointeurs vers des objets simples

Un tableau est une suite d'objets identiques, comme les maisons dans une rue par exemple. Chaque élément d'un tableau comporte un indice, qui correspond en fait au numéro d'ordre de l'élément à partir du début du tableau. Rappelons que le premier élément du tableau se trouve à la position 0.

Les tableaux C++ sont déclarés en utilisant des crochets entre lesquels figure le nombre d'éléments qu'ils peuvent contenir :

```
int array[10];  // Déclaration d'un tableau de 10 éléments
```

Il est possible d'accéder à chacun des éléments d'un tableau en comptant, dans notre métaphore, le nombre de maisons à partir du coin de la rue :

```
array[0] = 10;   // Affecte 10 au premier élément
array[9] = 20;   // Affecte 20 au dernier élément
```

Le programme affecte d'abord la valeur 10 au premier élément du tableau : la maison située à zéro emplacement du coin de la rue. Le programme affecte ensuite 20 au dernier élément du tableau, la neuvième maison à partir du coin.

Rappelez-vous une fois pour toutes que le C++ numérote les indices en partant de 0 et en allant jusqu'à la taille du tableau moins 1.

Pour rester dans la métaphore immobilière, le nom du tableau représente le nom de la rue et les numéros des maisons représentent un indice du tableau (N.D.T. : il est bien sûr présumé qu'ils se suivent un par un, et non par numéro pair ou impair). Dans le même esprit, les variables peuvent être identifiées grâce à leur adresse unique dans la mémoire de l'ordinateur. Ces adresses peuvent être calculées et stockées à toutes fins utiles.

```
int variable;              // Déclaration d'un objet entier
int* pVariable = &variable  // Stockage de l'adresse dans
                           // pVariable
*pVariable = 10;           // Affectation de 10 à l'entier
                           // sur lequel pointe pVariable
```

Le pointeur `pVariable` est déclaré en tant que conteneur de l'adresse de *variable*. L'affectation place 10 dans l'entier `int` sur lequel pointe `pVariable`.

Si nous utilisons une fois de plus notre analogie immobilière, nous obtenons :

- `variable` est une maison ;
- `pVariable` est l'équivalent d'un feuillet sur lequel aurait été griffonnée l'adresse de la maison ;
- L'affectation finale délivre le message 10 à la maison dont l'adresse est écrite dans `pVariable`, tout comme le ferait le préposé des postes (un facteur, n'ayons pas peur des mots). Encore que, contrairement à mon facteur, l'ordinateur ne se trompe jamais d'adresse.

Les notions de tableaux de variables simples (intrinsèques) sont étudiées dans le Chapitre 7, tandis que les Chapitres 8 et 9 décrivent les pointeurs en détails.

Déclarer des tableaux d'objets

Les tableaux d'objets fonctionnent de la même manière que les tableaux de simples variables. Examinons ce listing :

```
// ArrayOfStudents - définit un tableau d'objets Student
//                   et accès à un élément qu'il contient.
//                   Ce programme ne fait rien de particulier.
class Student
{
  public:
    int  semesterHours;
    float gpa;
    float addCourse(int hours, float grade);
};

// une fonction quelconque, juste un exemple
void someFn()
{
    // déclaration d'un tableau de 10 étudiants
    Student s[10];

    // affectation au 5e étudiant d'une
    // moyenne générale - gpa - de 5.0
    s[4].gpa = 5.0;

    // ajout d'un autre cours au 5e étudiant
    // cette fois il a échoué ; que cela lui serve de leçon
    s[4].addCourse(3, 0.0);
}
```

Nous avons ici un tableau d'objets nommé Student où `s[4]` se réfère au cinquième objet `Student` du tableau. Par extension, `s[4].gpa` se réfère à la moyenne générale (GPA) du cinquième étudiant. Plus loin, `s[4].addCourse()` ajoute un cours à l'objet "5e étudiant".

Déclarer des pointeurs vers des objets

Les pointeurs vers des objets fonctionnent de la même manière que les pointeurs vers des types simples :

```
// ObjPtr - définit et utilise un pointeur vers un objet Student
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int  semesterHours;
    float gpa;
    float addCourse(int hours, float grade){return 0.0;};
};

int main(int argc, char* pArgs[])
{
    // création d'un objet Student
    Student s;
    s.gpa = 3.0;

    // création d'un pointeur vers un objet Student
    Student* pS;

    // fait pointer le pointeur vers notre objet Student
    pS = &s;
    cout << "s.gpa   = " << s.gpa   << "\n"
         << "pS->gpa = " << pS->gpa << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le programme déclare une variable `s` de type `Student`. Il déclare ensuite un pointeur `pS` qui est du type "pointeur vers un objet Student", aussi écrit `Student*`. On initialise également la valeur de l'un des membres de données de `s`. Après quoi, l'adresse de `s` est affectée à `pS`. Finalement, on fait référence au même objet soit par son nom (`s`), soit en utilisant le pointeur vers cet objet (`pS`). L'étrange notation `pS->gpa` est expliquée dans la section suivante.

Déréférencer un pointeur d'objet

Par analogie avec les pointeurs dirigés vers des variables simples, vous pourriez penser que ce qui suit se réfère au GPA (moyenne générale) de l'étudiant s :

```
int main(int argc, char* pArgs[])
{
  // ce qui suit est incorrect :
  Student s;
  Student* pS = &s;  // création d'un pointeur vers s

  // accès au membre gpa de l'objet pointé par pS
  // (ceci ne fonctionne pas)
  *pS.gpa = 3.5;

  return 0;
}
```

Comme le précise le commentaire, ce code ne fonctionne pas. Le problème, c'est que l'opérateur "point" (.) est évalué avant le pointeur "astérisque" (*).

Les programmeurs en C++ utilisent des parenthèses pour s'affranchir de l'ordre dans lequel les opérations sont effectuées. Dans l'expression ci-dessous, elles forcent par exemple l'addition à être calculée avant la multiplication :

```
int i = 2 * (1 + 3);  // l'addition est calculée
                      // avant la multiplication
```

Les parenthèses ont le même effet sur les variables de pointeur :

```
int main(int argc, char* pArgs[])
{
  Student s;
  Student* pS = &s;  // création d'un pointeur vers s

  // accès au membre gpa de l'objet sur lequel pointe pS
  // (ceci fonctionne comme prévu)
  (*pS).gpa = 3.5;

  return 0;
}
```

Le pointeur *pS évalue l'objet Student vers lequel pS est pointé ; .gpa se réfère donc bien au membre gpa de cet objet.

Tirer des pointeurs fléchés

L'utilisation de l'opérateur "astérisque" avec des parenthèses est parfaite pour déréférencer les pointeurs dirigés sur des objets ; cependant, même le technicien le plus émérite admettra que cette syntaxe est quelque peu alambiquée.

Le C++ propose un opérateur plus commode pour accéder aux membres d'un objet tout en évitant les lourdes expressions de pointeurs d'objets. L'opérateur -> est défini ainsi :

```
ps->gpa est l'équivalent de (*pS).gpa
```

L'opérateur "flèche" est presque exclusivement utilisé car il est plus facile à lire. Les deux formes n'en sont pas moins équivalentes.

Passer des objets à des fonctions

La passation de pointeurs à des fonctions est un autre moyen de vous amuser avec les variables de pointeur.

Appeler une fonction avec une valeur d'objet

Comme vous le savez, le C++ passe les arguments aux fonctions par référence lorsque leur type se voit accoler le caractère esperluette (&). Mais, par défaut, les arguments sont passés aux fonctions par valeur. Un problème de mémoire ? Reportez-vous aux Chapitres 6 et 8 !

Les objets de classe complexes, définis par l'utilisateur, sont aussi passés par valeur, comme de vulgaires entiers. Voyons cela de plus près grâce à l'exemple suivant :

```
// PassObjVal - essayer de changer la valeur d'un objet
//              dans une fonction échoue si l'objet est
//              passé par valeur
#include <cstdio>
#include <cstdlib>
```

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

class Student
{
  public:
    int  semesterHours;
    float gpa;
};

void someFn(Student copyS)
{
    copyS.semesterHours = 10;
    copyS.gpa           = 3.0;
    cout << "Valeur de copyS.gpa = "
         << copyS.gpa << "\n";
}

int main(int argc, char* pArgs[])
{
    Student s;
    s.gpa = 0.0;

    // affiche la valeur de s.gpa avant d'appeler someFn()
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // passe l'adresse de l'objet existant
    cout << "Appel de someFn(Student)\n";
    someFn(s);
    cout << "Retour de someFn(Student)\n";

    // la valeur de s.gpa reste 0
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La fonction `main()` crée un objet `s` qu'elle passe ensuite à la fonction `someFn()`.

Ce n'est pas l'objet `s` lui-même qui est passé, mais une copie de `s`.

L'objet `copyS`, dans `someFn()`, s'active en tant que copie exacte de la variable `s` provenant de `main()`. Toute modification de `copyS` faite dans `someFn()` n'a

aucun effet rétroactif sur la variable s de `main()`. Vérifions-le à l'aide de la sortie produite par ce programme :

```
Valeur de s.gpa = 0
Appel de someFn(Student)
Valeur de copyS.gpa = 3
Retour de someFn(Student)
Valeur de s.gpa = 0
Appuyez sur une touche pour continuer...
```

Appeler une fonction avec un pointeur d'objet

La plupart du temps, le programmeur veut que les modifications réalisées dans la fonction soient répercutées là où elle est appelée. Pour cela, il doit passer soit l'adresse d'un objet, soit une référence à ce dernier. En tout cas, pas l'objet lui-même. Le programme ci-dessous utilise une approche par adresse :

```
// PassObjPtr - change le contenu d'un objet dans une
//              fonction en lui passant un pointeur vers l'objet
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int  semesterHours;
    float gpa;
};

void someFn(Student* pS)
{
    pS->semesterHours = 10;
    pS->gpa           = 3.0;
    cout << "Valeur de pS->gpa = "
         << pS->gpa << "\n";
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.gpa = 0.0;

    // affiche la valeur de s.gpa avant d'appeler someFn()
```

```
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // passe l'adresse de l'objet existant
    cout << "Appel de someFn(Student*)\n";
    someFn(&s);
    cout << "Retour de someFn(Student*)\n";

    // la valeur de s.gpa est maintenant 3.0
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Le type d'argument de `someFn()` est un pointeur vers un objet `Student` (soit `Student*`) et non l'objet `Student` lui-même. Ceci est illustré par la manière dont le programme appelle `someFn()`, passant l'adresse de `s` au lieu de la valeur de `s`. Cette méthode permet à la fonction de modifier les données de l'objet. Conceptuellement, ceci est très proche de l'écriture d'une adresse postale sur un bout de papier `pS` qui est ensuite confié à `someFn()`. Cette dernière utilise la syntaxe fléchée pour déréférencer le pointeur `pS`.

L'affichage produit par `PassObjPtr` est nettement plus satisfaisant (du moins pour moi) :

```
Valeur de s.gpa = 0
Appel de someFn(Student*)
Valeur de pS->gpa = 3
Retour de someFn(Student*)
Valeur de s.gpa = 3
Appuyez sur une touche pour continuer...
```

Appeler une fonction à l'aide de l'opérateur de référence

L'opérateur de référence rencontré dans le Chapitre 9 fonctionne aussi avec les objets définis par l'utilisateur. Modifions notre code pour mieux comprendre ce qui se passe :

```
// PassObjRef - change le contenu d'un objet dans une
//              fonction en utilisant une référence
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    int  semesterHours;
    float gpa;
};

// comme plus haut, mais on utilise ici des références
void someFn(Student& refS)
{
    refS.semesterHours = 10;
    refS.gpa      = 3.0;
    cout << "Valeur de refS.gpa = "
         << refS.gpa << "\n";
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Student s;
    s.gpa = 0.0;

    // affiche la valeur de s.gpa avant d'appeler someFn()
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // passe l'adresse de l'objet existant
    cout << "Appel de someFn(Student*)\n";
    someFn(s);
    cout << "Retour de someFn(Student&)\n";

    // la valeur de s.gpa est maintenant 3.0
    cout << "Valeur de s.gpa = " << s.gpa << "\n";

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Dans cet exemple, le C++ passe une référence à `s` plutôt qu'une copie. Les modifications faites dans `someFn()` sont répercutées dans `main()`.

Ce qui se produit ici, c'est que le C++ conserve la trace de l'adresse de `s` lorsqu'il est passé à une fonction par référence. Cela revient donc au même que passer l'adresse de l'objet.

Pointeurs ou références : des soucis en trop ?

Les pointeurs et les références ont leurs avantages respectifs. D'accord. Mais pourquoi se prendre la tête avec tout cela ? N'est-il pas plus simple de passer directement l'objet lui-même ?

L'une des réponses à cette question a déjà été apportée dans ce chapitre. Vous ne pouvez pas modifier l'objet dans une fonction où il est passé par valeur, puisque celle-ci ne reçoit qu'une copie de la structure du dit objet. C'est clair, net et définitif.

Voici une seconde raison. Certains objets sont gros (je veux dire *vraiment* obèses). Les passer par valeur signifie les copier en totalité dans la mémoire de la fonction.

La zone de mémoire utilisée pour transmettre des arguments à une fonction est dénommée la *pile d'appel*.

Si la fonction en appelle une autre, l'objet devra à nouveau être copié, et ainsi de suite. Au bout d'un certain temps, vous pouvez vous retrouver avec des dizaines de clones de l'objet initial. Non seulement vous consommez inutilement de la mémoire, mais en plus la vitesse d'exécution du programme peut s'en trouver considérablement ralentie, genre démarrage de Windows, vous voyez ce que je veux dire ?

Le problème est encore pire que cela. Nous y reviendrons dans le Chapitre 18.

Retour au Heap

Le problème qui affecte les simples types de pointeurs concerne aussi les pointeurs d'objets de classe. Vous devez notamment vous assurer que le pointeur que vous utilisez pointe véritablement vers un objet valide. Évitez par exemple de retourner une référence vers un objet défini localement dans la fonction :

```
MyClass* myFunc()
{
  // Ce qui suit ne fonctionne pas :
  MyClass mc;
  MyClass* pMC = &mc;
  return pMC;
}
```

Au retour de `myFunc()`, l'objet `mc` est hors de portée. Le pointeur retourné par `myFunc()` n'est pas valable dans la fonction appelante.

Les problèmes de portée sont discutés dans le Chapitre 9.

L'allocation de l'objet sur le *heap* (le tas) résout le problème :

```
MyClass* myFunc()
{
  MyClass* pMC = new MyClass
  return pMC;
}
```

Le heap sert à allouer des objets dans diverses situations.

Pointeurs ou références ?

Je déteste pointer des références et référencer des pointeurs. Mais au fait pourquoi les deux sont-ils nécessaires ? Avez-vous vraiment besoin des deux ?

Il est vrai que C# et bon nombre d'autres langages ne se servent pas des pointeurs. Cependant, les variables de pointeurs sont un constituant profondément enraciné dans ce bon vieux standard C++ (je ne parle pas ici des spécificités de Visual Studio .NET). Les traditions les plus fortes ne sont pas toujours les pires...

Pourquoi ne pas utiliser uniquement des références ?

La syntaxe utilisée pour manipuler une référence est semblable à celle dont on se sert avec les objets normaux. Pourquoi alors ne pas s'en tenir là et oublier jusqu'à l'existence des pointeurs ?

Les objets et leur adresse sont des choses différentes. Il est fréquent que la syntaxe d'une référence devienne plus complexe que son équivalent sous forme de pointeur. Regardons l'exemple qui suit :

```
class Student
{
  public:
    int semesterHours;
    float gpa;
    Student  valFriend;
    Student& refFriend;
    Student* ptrFriend;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // on déclare ici une référence sur le heap
    // (ce n'est pas trop compliqué)
    Student& student = *new Student;
    student.gpa = 10;

    // idem
    Student& studentFriend = *new Student;
    studentFriend.gpa = 20;

    // on copie maintenant la valeur d'un objet
    // Student dans le second
    student.valFriend = studentFriend;

    // ceci ne marche pas du tout
    Student& refFriend;
    refFriend = studentFriend;

    // par contre, ceci fonctionne
    student.ptrFriend = &studentFriend;

    return 0;
}
```

J'ai modifié la classe `Student` de façon à ce qu'un étudiant puisse faire référence à son meilleur copain. Pour cela, j'ai essayé d'utiliser un type de variable passé par référence. J'ai créé dans `main()` deux étudiants pour tenter de lier le premier objet `Student` à son compagnon `studentFriend`.

Dans le corps du programme, la première affectation copie le contenu de l'ami dans le membre de données. L'objet `Student` contient en quelque sorte deux corps. La deuxième affectation ne marche pas du tout : C++ ne peut pas faire la différence entre affecter un objet à une variable de référence et l'affecter à l'objet lui-même. Seule la troisième affectation fonctionne correctement. L'objet `student` pointe vers l'adresse de `studentFriend`, ce qui est exactement le but recherché.

Liaisons par listes chaînées

Après le tableau, la seconde structure la plus courante est appelée une *liste*. Il existe différents types et tailles de listes. Mais la forme la plus répandue est la *liste chaînée*. Ici, chaque objet pointe vers le membre suivant, formant ainsi une espèce de chaîne qui se développe en mémoire. Il suffit que le programme pointe le dernier élément de la liste vers un objet pour que celui-ci soit incorporé dans la chaîne. Il n'est donc pas indispensable de déclarer la taille de la liste chaînée au début du code, car elle peut s'agrandir (et se réduire) autant que nécessaire en ajoutant (ou supprimant) des objets.

Le coût de cette souplesse est la vitesse d'accès. Contrairement à ce qui se passe avec les tableaux, vous ne pouvez plus accéder directement au dixième élément (par exemple). Non : il vous faut partir du début de la liste et tirer dix fois sur la corde à nœuds pour arriver jusqu'à l'objet voulu.

Les listes chaînées ont une autre caractéristique en plus de leur expansion dynamique (ce qui est bien) et des difficultés d'accès aux objets (ce qui l'est beaucoup moins) : elles font un usage intensif de pointeurs. D'un point de vue pédagogique, elles représentent donc un outil rêvé pour s'exercer à la pratique des variables de pointeurs.

Une liste chaînée peut être comparée à un groupe d'enfants qui se tiennent par la main pour traverser une rue. Chaque objet contient un lien vers le prochain objet de la chaîne (un autre enfant). La maîtresse joue le rôle du "pointeur de tête", qui désigne le premier élément de la liste (elle donne la main au premier élève).

Toutes les classes ne sont pas susceptibles d'être utilisées pour créer une liste chaînée. Une classe "chaînable" est déclarée comme suit :

```
class LinkableClass
{
  public:
    LinkableClass* pNext;

    // autres membres de la classe
};
```

Le pointeur `pNext` est l'élément déterminant d'un objet de la classe `LinkableClass`. De prime abord, cela paraît bizarre : une classe qui contient un pointeur vers elle-même ? En fait, cette syntaxe indique que la classe `LinkableClass` contient un pointeur vers un autre objet qui appartient lui aussi à la `LinkableClass`.

Le pointeur `pNext` est comparable aux mains qui se tiennent afin de former la chaîne d'enfants. La liste des enfants est faite d'un certain nombre d'objets, tous de type "enfant". Chaque enfant donne la main à (pointe vers) un autre enfant.

Le pointeur de tête est simplement un pointeur de type `LinkableClass*`. Pour en rester à l'analogie du groupe d'élèves, l'institutrice pointe vers un objet de la classe Enfant (il est intéressant de remarquer que la maîtresse n'est pas un enfant : le pointeur de tête n'est pas de type `LinkableClass*`).

```
LinkableClass* pHead = (LinkableClass*)0;
```

Initialisez toujours un pointeur à 0. Le zéro, généralement appelé "nul" quand il s'agit des pointeurs, est universellement reconnu comme étant le "non-pointeur". Dans tous les cas, se référer à l'adresse 0 stoppe toujours immédiatement le programme. La conversion explicite (*cast*) de l'entier 0 en `LinkableClass*` n'est pas nécessaire. Le C++ considère en effet que 0 peut appartenir à tous les types ; il en fait une sorte de "pointeur universel". Mais, à mon avis, la conversion est une bonne habitude à prendre.

Le pointeur vers le premier membre d'une liste chaînée est appelé le *pointeur de tête*. Le pointeur vers le dernier membre (s'il existe) est donc logiquement le *pointeur de queue*. D'où le nom `pHead` dans la ligne de code précédente. (en torturant la langue, et puisque le "p" doit se prononcer en anglais comme "pie", `pHead` est donc une tête de pie, et du coup `tHead` ne serait rien d'autre qu'une queue de pie...).

Pour voir comment une liste chaînée fonctionne en pratique, jetez un coup d'œil à cette petite fonction qui ajoute au début de la liste l'argument qui lui a été passé :

```
void addHead(LinkableClass* pLC)
{
   pLC->pNext = pHead;
   pHead = pLC;
}
```

Ici, le pointeur `pNext` de l'objet est dirigé vers le premier membre de la liste. C'est comme prendre la main du premier enfant de la chaîne. La seconde ligne dirige le pointeur de tête vers l'objet passé en argument. La maîtresse lâche la main de l'enfant que je tiens et elle prend la mienne. Je suis maintenant le premier gamin de la chaîne.

Effectuer d'autres opérations sur une liste chaînée

L'ajout d'un objet en tête d'une liste chaînée est l'opération la plus simple. Parcourir les éléments d'une telle liste vous aidera à mieux en comprendre le fonctionnement :

```
// navigation dans une liste chaînée
LinkableClass* pL = pHead;
while(pL)
{
    // on effectue ici une certaine opération
    // ... un certain code ...
    // puis on passe à l'entrée suivante
    pL = pL->pNext;
}
```

Ce programme initialise le pointeur `pL` vers le premier objet d'une liste d'objets `LinkableClass` via le pointeur `pHead` (on attrape la main du premier enfant). Il entre ensuite dans une boucle `while`. Si le pointeur `pL` n'est pas détruit (nul, ou `false`), il désigne donc un certain objet de la classe `LinkableClass`. On entre alors dans la boucle, où le programme peut effectuer les traitements voulus sur l'objet ainsi pointé.

L'affectation `pL = pL->pNext` "déplace" le pointeur `pL` vers l'objet (l'enfant) suivant de la liste. Retour au début du `while` : si `pL` est nul, c'est que nous avons

épuisé toute la liste (je veux dire par là qu'il n'y a plus d'autres enfants dans la file, pas qu'ils sont tous épuisés).

Connexion ! (le programme LinkedListData)

Le programme `LinkedListData` présenté ci-dessous implémente une liste d'objets contenant des noms de personnes. Il pourrait d'ailleurs facilement traiter n'importe quel autre type d'information : numéro de Sécurité sociale, poids ou taille, cursus scolaire, relevé de compte bancaire, etc. Je me suis contenté du nom pour ne pas compliquer inutilement les choses.

```
// LinkedListData - Stockage de noms dans
//                  une liste chaînée d'objets.
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string.h>
using namespace std;

// NameDataSet - mémorise le nom d'une personne
//               (ou toute autre information utile)
class NameDataSet
{
  public:
    char szName[128];

    // lien vers la prochaine entrée de la liste
    NameDataSet* pNext;
};

// pointeur vers la première entrée de la liste
NameDataSet* pHead = 0;

// add - ajout d'un nouveau membre à la liste chaînée
void add(NameDataSet* pNDS)
{
    // pointe l'entrée courante vers le début de la liste...
    pNDS->pNext = pHead;

    // pointe le pointeur de tête vers l'entrée courante
    pHead = pNDS;
}

// getData - lit un nom ; retourne Nul s'il n'y a plus
//           rien à lire
```

```
NameDataSet* getData()
{
    // lit le premier nom
    char nameBuffer[128];
    cout << "\nEntrez un nom : ";
    cin  >> nameBuffer;

    // si le nom entré est 'quitter'...
    if ((stricmp(nameBuffer, "quitter") == 0))
    {
        // ...retourne un Nul pour terminer la saisie
        return 0;
    }

    // prochaine position à remplir
    NameDataSet* pNDS = new NameDataSet;

    // remplit le nom et met à zero le pointeur de chaînage
    strncpy(pNDS->szName, nameBuffer, 128);
    pNDS->szName[127] = '\0';  // la chaîne doit être terminée
    pNDS->pNext = 0;

    // renvoie l'adresse de l'objet créé
    return pNDS;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Lecture de noms de personnes\n"
         << "Tapez 'quitter' terminer la saisie\n";

    // créée un (autre) objet NameDataSet
    NameDataSet* pNDS;
    while (pNDS = getData())
    {
        // l'ajoute au début de la liste des
        // objets NameDataSet
        add(pNDS);
    }

    // itération à travers la liste pour afficher les
    // objets (stop si la prochaine adresse est NUL)
    cout << "Liste des noms :\n";
    pNDS = pHead;
    while(pNDS)
    {
        // affiche l'entrée courante
```

```
        cout << pNDS->szName << "\n";

        // passe à l'entrée suivante
        pNDS = pNDS->pNext;
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Bien qu'il soit assez long, le programme `LinkedListData` est relativement simple à comprendre si vous le prenez point par point. La structure `NameDataSet` réserve de la place pour un nom de personne et pour un lien vers l'objet suivant de la classe. Nous avons donc bien à faire à une liste chaînée. J'ai indiqué plus haut que cette classe contiendrait d'autres membres dans une véritable application.

La fonction `main()` commence par appeler `getData()` dans une boucle pour lire une autre entrée `NameDataSet` faite par l'utilisateur. Si ce dernier a tapé "quitter", `getData()` retourne un nul et la boucle se termine.

La fonction `stricmp()` compare deux chaînes sans se soucier des majuscules et des minuscules.

La fonction `getData()` demande un nom à l'utilisateur. Elle se contente de lire ce que celui-ci tape sur son clavier, en espérant simplement qu'il saura faire preuve de sagesse (on se limite à 128 caractères, mais aucun contrôle n'est effectué).

Tant que l'utilisateur ne tape pas "quitter", le programme crée un nouvel objet `NameDataSet`, le remplit avec le nom qui a été saisi, puis il annule (ou réinitialise) le pointeur `pNext`.

Ne laissez jamais des pointeurs de chaîne non initialisés. Appliquez toujours la bonne vieille maxime des programmeurs : "Dans le doute, mettre à zéro".

Finalement, `getData()` renvoie l'adresse de l'objet à `main()`.

La fonction `main()` ajoute chaque objet renvoyé par `getData()` au début de la liste chaînée pointée par la variable globale `pHead`. Lorsque `getData()` retourne un nul, on quitte la boucle `while` initiale pour entrer dans une seconde section. Celle-ci est chargée de parcourir la liste complète pour afficher chaque objet.

La seconde boucle `while` se termine une fois le dernier objet traité (dans ce cas, la valeur du pointeur `pNext` devient nulle).

Le programme affiche les noms dans l'ordre inverse de leur saisie. Cela vient du fait que chaque nouvel objet est inséré au début de la liste. On aurait tout aussi bien pu ajouter les objets en fin de liste. Cela demande simplement un peu plus de code. Vous trouverez en téléchargement sur le site de l'éditeur une variante qui propose cette autre méthode (`LinkedListData2.cpp`).

Une lueur d'espoir : des conteneurs dans la librairie C++

Je crois que tout le monde devrait marcher avant de courir, réfléchir à la façon de résoudre un problème d'arithmétique avant de se précipiter sur une calculette, et écrire un programme gérant des listes chaînées avant d'utiliser des classes toutes faites.

Cela étant dit, le Chapitre 27 décrit d'autres classes de listes fournies par l'environnement de C++. Ces classes sont généralement désignées sous le nom de *conteneurs*, car elles contiennent d'autres objets. Les tableaux et les listes chaînées ne sont que deux exemples parmi d'autres de ces classes conteneurs.

Chapitre 15

La protection des membres : ne pas déranger

Dans ce chapitre :

- Déclarer des membres protégés.
- Accéder à des membres protégés depuis une classe.
- Accéder à des membres protégés de l'extérieur d'une classe.

Le Chapitre 12 a présenté la notion de classe. Ce chapitre décrit le mot-clé `public` comme s'il faisait partie d'une déclaration de classe, ce que vous serez amené à faire. Vous y trouverez aussi une alternative à *public*.

Protéger les membres

Les membres d'une classe peuvent être marqués comme "membres protégés", ce qui les rend inaccessibles depuis l'extérieur de la classe. L'alternative, c'est d'en faire des membres publics qui, eux, sont accessibles de partout.

Le mot "inaccessible" doit être compris dans un sens restrictif. N'importe quel programmeur peut éditer le code source, supprimer la protection puis faire ce qu'il (ou elle) veut. Et un pirate informatique compétent peut donc changer le code d'une section protégée. La vérité, c'est que tout cela sert surtout au programmeur à se protéger de lui-même en évitant d'accéder par inadvertance à une classe.

Pourquoi protéger des membres ?

Pour comprendre le rôle de la protection, revenons d'abord aux objectifs de la programmation orientée objet.

Supposons par exemple que vous projetiez de développer le logiciel d'un four à micro-ondes – ou de ce que vous voulez – en lui donnant une interface simple avec le monde extérieur ; ce logiciel serait ensuite placé dans un boîtier pour que n'importe qui n'aille pas farfouiller dedans. C'est ce terme de "boîtier" qui illustre la notion de protection. La protection permet de :

- Mettre à l'abri le contenu de la classe des fonctions externes. C'est ce que fait notre "boîtier" de four à micro-ondes.

- Rendre la classe responsable de la maintenance de son propre état interne. Il n'est pas raisonnable de demander à une classe d'assumer cette responsabilité s'il est possible à d'autres d'entrer dedans et de manipuler ce qui s'y trouve (pas plus qu'il n'est honnête de demander à un fabricant de fours à micro-ondes d'être responsable des bricolages que je pourrais faire dans le câblage interne).

- Limiter l'interface de la classe vis-à-vis du monde extérieur. Il est plus facile d'apprendre à utiliser une classe dont l'interface est limitée (aux membres publics). Les membres protégés sont cachés à l'utilisateur et n'exigent aucun apprentissage. L'interface du four est la classe ; ce principe est appelé *abstraction* (voir le Chapitre 11).

- Réduire le niveau d'interconnexion entre la classe et le restant du code. En limitant les interconnexions, il est plus facile de remplacer une classe par une autre ou d'utiliser la classe dans un autre programme.

Votre esprit ô combien cartésien et fonctionnel vous fait très certainement dire : "Je n'ai que faire de ces fonctions bizarroïdes. Donnez-moi une règle qui me dise quels membres doivent être publics, et donc accessibles à toutes les fonctions, et ceux qui ne doivent pas l'être".

En théorie, cela devrait être possible, mais la pratique répond que non. Au début, les gens sont pleins de bonnes intentions. Mais si le langage ne décourage pas au minimum l'accès direct aux membres protégés, ces bonnes intentions ne résisteront pas à la pression : on veut tous voir le produit fini sortir au plus vite.

Le fonctionnement des membres protégés

L'ajout du mot-clé `public` à une classe rend ses membres subséquents publics ; des fonctions non-membres peuvent alors y avoir accès. L'ajout du mot-clé `protected` protège les membres de la classe qui suivent ; ils deviennent donc inaccessibles aux non-membres de la classe. Il est possible de passer d'un mode public à un mode protégé aussi souvent que vous le désirez.

Supposons que vous disposiez d'une classe nommée `Student`. Dans l'exemple qui suit, les capacités d'un étudiant digne de ce nom sont définies (notez l'absence des fonctions `spendMoney()` – "dilapider son argent" – et `drinkBeer()` – "boire de la bière" – ; nous avons à faire à un étudiant "virtuel", bien sous tous rapports). Bref, les fonctions sont :

`addcourse(int hours, float grade)` : ajoute un cours (heures, notes).

`grade()` : retourne la moyenne courante des notes.

`hours()` : retourne le nombre d'heures effectuées pour l'obtention du diplôme.

Les autres membres de `Student` peuvent être déclarés comme "protégés" afin de maintenir secrètes certaines révélations indiscrètes des fonctions.

```
class Student
{
  public:
    // grade - retourne la moyenne courante (gpa)
    float grade()
    {
        return gpa;
    }

    // hours - retourne le nombre d'heures semestriel
    int hours()
    {
        return semesterHours;
    }
    // addCourse - Ajoute un autre cours à l'enregistrement de
    //             l'étudiant
    float addCourse(int hours, float grade);

    // les membres ci-dessous sont tenus à l'écart des autres
  protected:
    int semesterHours; // heures effectuées pour l'obtention
```

```
                        // du diplôme
    float gpa;          // moyenne des notes
};
```

Ici, seuls les autres membres de `student` peuvent accéder à `semesterHours` et `gpa`. C'est pourquoi ce qui suit ne peut pas fonctionner :

```
Student s;
int main(int argcs, char* pArgs[])
{
  // augmente ma moyenne (mais pas trop, sinon personne
  // n'y croira)
  s.gpa = 3.5;          // <- génère une erreur de compilation.
  float gpa = s.grade(); // <- cette fonction publique lit une
                         // copie de la valeur, mais vous ne
                         // pouvez la changer ici
  return 0;
}
```

La tentative de modification de la valeur `gpa` est sanctionnée par une erreur de compilation.

Pour l'élégance d'un programme, il est recommandé de ne pas se fier aux valeurs par défaut et de spécifier dès le début de la classe ce qui est public et ce qui est privé. La plupart du temps, les membres publics sont déclarés en premier car ils forment l'interface de la classe. Les membres protégés sont enregistrés par la suite.

Les membres de classe peuvent aussi être protégés en les déclarant comme étant privés (*private*). C'est en fait le mode par défaut des classes qui démarrent ainsi. Dans ce livre, j'utilise exclusivement le terme `protected` car il correspond au concept le plus générique.

Créer un argument pour les membres protégés

Maintenant que nous en savons un peu plus sur l'utilisation des membres protégés dans une véritable classe, revenons aux arguments qui servent à les utiliser.

La protection de l'état interne d'une classe

Protéger le membre gpa empêche l'application d'attribuer une valeur arbitraire à la moyenne. Il est possible d'ajouter des cours à l'application, mais pas d'en modifier la moyenne générale.

Si l'application a une raison légitime de modifier directement cette moyenne, la classe peut fournir une fonction membre à cette fin en procédant comme suit :

```
class Student
{
  public:
    // la même chose qu'avant
    float grade()
    {
        return gpa;
    }
    // nous autorisons ici la modification de la moyenne
    float grade(float newGPA)
    {
        float oldGPA = gpa;
        // seulement si la nouvelle valeur est valide
        if (newGPA > 0 && newGPA <= 4.0)
        {
            gpa = newGPA;
        }
        return oldGPA;
    }
    // ... e reste est pareil, y compris les membres de données
  protected:
    int  semesterHours; // Heures consacrées à l'obtention
                        // du diplôme
    float gpa;
};
```

L'ajout de la fonction membre `float grade()` permet à l'application de définir la valeur de gpa. Remarquez cependant que la classe n'a toujours pas rendu complètement le contrôle. L'application ne peut pas mettre gpa à n'importe quelle valeur ; il faut qu'elle soit comprise entre 0 et 4.0.

De ce fait, `Student` a fourni un accès à un membre de données interne sans renoncer à sa responsabilité, qui est de s'assurer que l'état interne de la classe est valide.

Utiliser une classe avec une interface limitée

Une classe fournit une interface limitée. Pour utiliser une classe, il vous suffit de connaître ses membres publics, ce qu'ils font et ce que sont leurs arguments. Ces informations peuvent réduire considérablement le nombre de choses que vous devez apprendre – et retenir – pour utiliser une classe.

Au fur et à mesure que les conditions changent ou que des bogues sont découverts, vous serez amené à modifier le fonctionnement interne d'une classe. Une fois que vous aurez caché ce fonctionnement interne, les modifications de détails n'exigeront généralement pas de modifications dans le code d'application externe.

Une seconde raison, peut-être encore plus importante, est que les êtres humains (je ne parle pas des autres) ne peuvent retenir qu'un nombre limité de choses à un instant donné. L'emploi d'une interface de classe définie de façon stricte permet au programmeur d'oublier les détails cachés. De même, le programmeur qui construit la classe n'a pas besoin de se concentrer avec la même intensité sur tout ce qui concerne l'utilisation des fonctions de la classe par le code externe.

Permettre aux fonctions non-membres d'accéder à des membres protégés

Il vous arrivera parfois de vouloir autoriser une fonction non-membre à accéder aux membres protégés d'une classe. Il faut pour cela désigner la fonction comme "amie" de la classe en utilisant le mot-clé `friend`.

Une fonction externe exige parfois un accès direct à un membre de données. Sans amitié, le programmeur serait contraint de déclarer ce membre comme `public`, ce qui permettrait aussi à quiconque d'y accéder.

L'amitié du C++, c'est comme de permettre à un voisin de venir jeter un coup d'œil chez vous pendant que vous êtes en vacances. Confier la clé de la maison à quelqu'un d'extérieur à la famille n'est généralement pas recommandé, mais c'est mieux que de laisser la porte ouverte. Bien sûr, il faut savoir se limiter aux amis les plus proches et ne pas confier un trousseau de clés à n'importe qui !

La déclaration amie apparaît dans la classe qui contient le membre protégé. Elle est comparable à une déclaration de prototype dans la mesure où elle inclut le nom étendu et le type de retour. Dans l'exemple qui suit, la fonction `initialize()` peut accéder librement à l'ensemble du contenu de `Student` :

```
class Student
{
    friend void initialize(Student*);
  public:
    // les mêmes membres publics qu'auparavant
  protected:
    int  semesterHours; // heures effectuées pour l'obtention
                        // du diplôme
    float gpa;
};
// la fonction qui suit est un ami de Student, capable de
// ce fait d'accéder à tous les membres protégés.
void initialize(Student *pS)
{
    pS->gpa = 0;            // ceci est maintenant autorisé, ce
    pS->semesterHours = 0;  // qui n'était pas le cas auparavant.
}
```

Une même fonction peut être déclarée comme amie de deux classes à la fois. Bien que cela puisse s'avérer pratique, cette solution tend à lier les deux classes ensemble ; or, cette méthode est normalement considérée comme mauvaise car elle rend une classe dépendante de l'autre. Mais ce n'est pas une démarche négative à condition que les deux classes soient conçues pour travailler conjointement, comme dans ce listing :

```
class Student;   // déclaration préalable
class Teacher
{
    friend void registration(Teacher& t, Student& s);
  public:
    void assignGrades();
  protected:
    int   noStudents;
    Student *pList[100];
};
class Student
{
    friend void registration(Teacher& t, Student& s);
  public:
    // les mêmes membres publics qu'auparavant
  protected:
    Teacher *pT;
    int  semesterHours; // heures effectuées pour l'obtention
                        // du diplôme
    float gpa;
```

```
};

void registration(Teacher& t, Student& s)
{
    // initialise l'objet Student
    s.semesterHours = 0;
    s.gpa = 0;

    // si il reste de la place...
    if (t.noStudents < 100)
    {
        // ...ajoute l'étudiant en fin de liste
        t.pList[t.noStudents] = &s;
        t.noStudents++;
    }
}
```

Dans cet exemple, la fonction `registration()` accède à la fois aux classes `Student` et `Teacher` afin de les lier au moment de l'inscription sans pour autant être une fonction membre de l'une ou de l'autre.

Remarquez que la première ligne de l'exemple déclare la classe `Student` mais aucun de ses membres. Rappelez-vous qu'il s'agit là d'une déclaration en avant qui définit simplement le nom d'une classe de manière à ce que d'autres classes, comme `Teacher`, puissent définir un pointeur vers cette entité. Des références en avant sont nécessaires lorsque deux classes se réfèrent l'une à l'autre.

Une fonction membre d'une classe peut déclarer un ami appartenant à une autre classe. Par exemple :

```
class Teacher
{
    // ... un autre membre ...
  public:
    void assignGrades();
};
class Student
{
    friend void Teacher::assignGrades();
  public:
    // les mêmes membres publics que précédemment ...
  protected:
    int semesterHours;  // heures effectuées pour
                        // l'obtention du diplôme
```

```
    float gpa;
};
void Teacher::assignGrades()
{
    // on peut accéder ici aux membres
    // protégés de Teacher
}
```

Contrairement à l'exemple non-membre, la fonction membre `assignGrades()` doit être déclarée avant que la classe `Student` puisse l'utiliser en tant qu'amie.

Une classe entière peut être amie d'une autre. Dans ce cas, chaque membre de cette classe est un ami, comme dans :

```
class Student;   // déclaration en avant
class Teacher
{
 protected:
  int   noStudents;
  Student *pList[100];
 public:
  void assignGrades();
};
class Student
{
  friend class Teacher; // toute la classe est un ami
 public:
  // mêmes membres publics qu'auparavant...
 protected:
  int  semesterHours; //Heures effectuées pour
                      // l'obtention du diplôme.
  float gpa;
};
```

Maintenant, n'importe quelle fonction membre de `Teacher` a accès aux membres protégés de `Student`. La déclaration d'une classe en tant qu'amie d'une autre rend les deux inséparables.

Chapitre 16

Construire et démolir des objets : constructeurs et destructeurs

Dans ce chapitre :

- Créer et détruire des objets.
- Déclarer des constructeurs et des destructeurs.
- Appeler des constructeurs et des destructeurs.

Les objets d'un programme sont construits et jetés de la même manière que les objets du monde réel. Si une classe doit être responsable de son comportement, il faut qu'elle puisse avoir un certain contrôle sur les processus qui la concernent. Il se trouve que le C++ est doté des mécanismes idoines. Mais avant tout, voyons ce qu'est la création d'un objet.

Créer des objets

Certaines personnes utilisent les termes *classe* et *objet* sans beaucoup de discernement. Quelle est la différence ? Quelles sont leurs relations ?

Je peux créer une classe Chien qui décrit les caractéristiques propres au meilleur ami de l'homme (N.D.T. : mais l'homme est le meilleur ami du chat). J'ai deux chiens chez moi. De ce fait, ma classe Chien comportera deux

instances : Trudie et Scooter (enfin bon, je pense qu'il y a deux instances, parce que ça fait quelques jours que je n'ai pas vu Scooter).

Une *classe* décrit un type de chose. Un *objet* est l'une de ces choses, autrement dit une instance d'une classe. La classe est Chien et les objets sont Trudie et Scooter. Chaque chien est un objet distinct, mais il n'y a qu'une seule classe Chien, quel que soit le nombre de toutous que je possède.

Les objets sont créés et détruits, mais les classes se contentent d'exister. Mes petits copains Trudie et Scooter vont et viennent, mais la classe Chien est immuable (l'évolution des espèces mise à part).

Différents types d'objets sont créés à différents moments. Les *objets globaux* sont créés lorsque le programme commence à être exécuté. Les *objets locaux* sont créés au moment où le programme rencontre leur déclaration.

Un objet global est un objet déclaré en dehors d'une fonction. Un objet local est déclaré à l'intérieur d'une fonction ; il est par conséquent local à la fonction. Dans l'exemple qui suit, la variable `moi` est globale et la variable `pasMoi` est locale à la fonction `ChoisisEnUn()` :

```
int moi;
void ChoisisEnUn()
{
  int pasMoi;
}
```

Selon les règles du langage C, les objets globaux sont tous initialisés à zéro lorsque le programme commence son exécution. Les objets déclarés localement à une fonction n'ont pas de valeur de départ particulière. Ce n'est généralement pas acceptable pour toutes les classes.

Le C++ permet à une classe de définir une fonction membre spéciale qui est appelée automatiquement lorsqu'un objet de cette classe est créé. Cette fonction membre, appelée *constructeur*, doit initialiser l'objet à un état initial valide. De plus, la classe peut définir un *destructeur* qui gère la fin de vie de l'objet. Ces deux fonctions forment le sujet de ce chapitre.

Utiliser des constructeurs

Un constructeur est une fonction membre qui est automatiquement appelée lorsqu'un objet d'une certaine classe est créé. Sa tâche principale consiste à initialiser l'objet à une valeur de départ qui soit légale pour cette classe. À

charge pour les autres fonctions membres de s'assurer que cette valeur *reste* ensuite légale.

De la nécessité des constructeurs

Vous pourriez initialiser un objet comme au niveau de la déclaration ; c'est d'ailleurs ce que ferait un programmeur en C. Par exemple :

```
struct Student
{
    int  semesterHours;
    float gpa;
};
void fn()
{
    Student s1 = {0, 0.0};

    // ou
    Student s2;
    s2.semesterHours = 0;
    s2.gpa = 0.0;

    // ...continuation de la fonction...
}
```

Vous pourriez équiper la classe d'une fonction d'initialisation que l'application appellerait aussitôt l'objet créé. Comme cette fonction d'initialisation est un membre de la classe, elle aurait accès aux membres protégés. La solution se présente sous cette forme :

```
class Student
{
  public:
    void init()
    {
        semesterHours = 0;
        gpa = 0.0;
    }
    // ...autres membres publics...
  protected:
    int  semesterHours;
    float gpa;
};
void fn()
```

```
{
    Student s;    // créée l'objet...
    s.init();     // ...puis l'initialise
    // ...la fonction continue...
}
```

Le seul problème avec cette solution, c'est qu'elle abroge la responsabilité de la classe, qui est de rechercher ses propres membres de données. Autrement dit, la classe doit compter sur l'application pour appeler la fonction `init()`. Si elle ne le fait pas, l'objet est rempli d'informations incohérentes et nul ne sait ce qui pourrait se produire.

Il nous faut donc un moyen de prendre nous-mêmes la responsabilité de l'appel à la fonction `init()` depuis l'extérieur du code de l'application et de transmettre cet appel au compilateur. Chaque fois qu'un objet est créé, le compilateur peut insérer un appel à la fonction spéciale `init` afin de l'initialiser. C'est ça, un constructeur !

Travailler avec les constructeurs

Le constructeur est une fonction membre spéciale qui est automatiquement appelée lorsqu'un objet est créé. Il porte le même nom que la classe ; le compilateur sait ainsi quelle fonction membre est le constructeur (les concepteurs du C++ auraient pu choisir une autre règle, du genre "le constructeur doit être appelé `init()`" ; cela n'aurait fait aucune différence du moment que le compilateur arrive à reconnaître le destructeur). De plus, un constructeur n'a aucun type de retour pas même `void`, puisqu'il est appelé automatiquement (d'ailleurs, si un constructeur retournait quoi que ce soit, il n'y aurait pas d'emplacement prévu pour recevoir cette donnée). Enfin, un constructeur ne peut pas être appelé manuellement.

Construction d'un objet unique

Avec un constructeur, la classe `Student` apparaît comme suit :

```
//  Constructor - exemple qui invoque un constructeur
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
class Student
{
  public:
    Student()
    {
        cout << "Construction de student" << endl;
        semesterHours = 0;
        gpa = 0.0;
    }
  protected:
    int  semesterHours;
    float gpa;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Fabrication d'un nouvel objet Student" << endl;
    Student s;

    cout << "Placement d'un nouvel objet sur le tas" << endl;
    Student* pS = new Student;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Au moment de la déclaration de `s`, le compilateur insère un appel au constructeur `Student::Student()`. Allouer un nouvel objet `Student` sur le tas a le même effet, comme le montre la sortie produite par le programme :

```
Fabrication d'un nouvel objet Student
Construction de student
Placement d'un nouvel objet sur le tas
Construction de student
Appuyez sur une touche pour continuer...
```

Ce constructeur simple a été écrit en tant que fonction membre *inline*. Des constructeurs peuvent aussi être écrits sous la forme de fonction *outline*, comme par exemple dans :

```
class Student
{
  public:
    Student();
    // ... autres membres publics ...
  protected:
    int  semesterHours;
    float gpa;
};
Student::Student()
{
    cout << "Construction de student" << endl;
    semesterHours = 0;
    gpa = 0.0;
}
```

L'affichage produit par ce programme peut vous "prouver" que les constructeurs fonctionnent comme prévu. Mais, pour juger de l'effet réel, rien ne vaut une exécution en mode pas à pas dans le débogueur (revoyez à ce sujet le Chapitre 10).

Essayez d'ajouter une fonction `main()` à cet exemple (vous en trouverez une version en téléchargement sous le nom `Constructor2.cpp`). Exécutez le code pas à pas en vous arrêtant sur la déclaration `Student s`. Sélectionnez la commande Avancer. Magique : le contrôle saute à pieds joints vers `Student::Student()`. Lorsque la fonction se termine, le contrôle retourne à l'instruction qui suit la déclaration.

Il peut arriver que le constructeur soit exécuté globalement et non ligne par ligne. Dans ce cas, il vous suffira de poser un point d'arrêt à l'intérieur du constructeur. Cela marche à tous les coups.

Construction d'objets multiples

Chaque élément d'un tableau doit être construit séparément. Avec quelques modifications, notre programme `Constructor` s'adapte à un groupe (ou plutôt un tableau) d'étudiants :

```
//  ConstructArray - invoque un constructeur
//                   sur un tableau d'objets
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

class Student
{
  public:
    Student()
    {
        cout << "Construction de student" << endl;
        semesterHours = 0;
        gpa = 0.0;
    }
    // ... autres membres publics ...
  protected:
    int  semesterHours;
    float gpa;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Fabrication d'un tableau de 5 objets Student" << endl;
    Student s[5];

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Ce qui génère la sortie suivante :

```
Fabrication d'un tableau de 5 objets Student
Construction de student
Construction de student
Construction de student
Construction de student
Construction de student
Appuyez sur une touche pour continuer...
```

Construction d'un duplex

Si une classe contient un membre de donnée qui est un objet provenant d'une autre classe, le constructeur de cette dernière est automatiquement appelé. Étudiez le programme suivant, `ConstructMembers`. J'ai inséré des instructions de sortie de façon à ce que vous puissiez voir l'ordre dans lequel les objets sont invoqués.

```
//  ConstructMembers - les objets membres d'une classe
//                     sont tous construits avant que le
//                     constructeur de la classe conteneur
//                     soit appelé
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Course
{
  public:
    Course()
    {
        cout << "Construction de cours" << endl;
    }
};

class Student
{
  public:
    Student()
    {
        cout << "Construction de student" << endl;
        semesterHours = 0;
        gpa = 0.0;
    }
  protected:
    int  semesterHours;
    float gpa;
};
class Teacher
{
  public:
    Teacher()
    {
        cout << "Construction de teacher" << endl;
    }
  protected:
    Course c;
};
class TutorPair
{
  public:
    TutorPair()
    {
```

```
        cout << "Construction de tutorpair" << endl;
        noMeetings = 0;
    }
  protected:
    Student student;
    Teacher teacher;
    int   noMeetings;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "Fabrication de l'objet TutorPair" << endl;
    TutorPair tp;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

L'exécution du programme se traduit par la sortie suivante :

```
Fabrication de l'objet TutorPair
Construction de student
Construction de cours
Construction de teacher
Construction de tutorpair
Appuyez sur une touche pour continuer...
```

La création de l'objet `tp` dans `main()` invoque automatiquement le constructeur de `TutorPair`. Mais avant que le contrôle ne lui soit effectivement passé, les constructeurs des deux objets membres (`student` et `teacher`) sont appelés.

Le constructeur de `Student` est exécuté d'abord car il est déclaré en premier ; puis c'est le constructeur de `Teacher` qui est appelé.

Le membre `Course.c` de la classe `Teacher` est construit en même temps que l'objet `Teacher`. Le constructeur `Course` est invoqué à cet effet. Chaque objet d'une classe doit se construire lui-même avant d'appeler le constructeur de la classe elle-même. Sinon, le constructeur principal ne saurait rien de l'état de ses membres de données.

Une fois tous les membres de données traités, le contrôle retourne à l'accolade ouverte et le constructeur de `TutorPair` est autorisé à construire le reste de l'objet.

Il ne servirait à rien de rendre `TutorPair` responsable de l'initialisation de `Student` et de `Teacher`. Chaque classe est en effet responsable de l'initialisation de ses propres objets.

Comprendre les destructeurs

Un objet qui a été créé peut être détruit ("tu es poussière et tu redeviendras poussière"). Si une classe peut avoir un constructeur qui crée des objets, elle devrait aussi avoir une fonction membre spéciale qui se charge de faire le ménage. Ce membre est appelé le *destructeur.*

Pourquoi un destructeur est nécessaire

Si une classe alloue des ressources au constructeur, ces ressources doivent pouvoir être désallouées avant que l'objet cesse d'exister. Par exemple, si le constructeur ouvre un fichier, ce fichier doit être fermé avant de quitter la classe ou le programme lui-même. Ou encore, si le constructeur alloue de la mémoire sur le heap, cette mémoire doit être libérée avant que l'objet s'en aille. Le destructeur permet à la classe d'effectuer automatiquement ces tâches de nettoyage sans avoir à compter sur l'application pour appeler les fonctions membres correctes.

Travailler avec les destructeurs

Les membres destructeurs ont le même nom qu'une classe, mais leur nom est précédé d'un caractère tilde (~). Remarquez que le C++ fait bien les choses, puisque le tilde est le symbole de l'opérateur logique `NOT` ("pas"). Compris ? Un destructeur est un "non-constructeur". *Très* subtil. À l'instar d'un constructeur, un destructeur n'a pas de type de retour. Par exemple, la classe `Student` à laquelle un destructeur a été ajouté apparaît sous la forme :

```
class Student
{
 public:
  Student()
  {
```

```
    semesterHours = 0;
    gpa = 0.0;
  }
~Student()
  {
    // ...tout  ce qui est actif est traité ici...
  }
  // ...Autres membres publics
protected:
  int semesterHours;
  float gpa;
};
```

Le destructeur est appelé automatiquement lorsqu'un objet est supprimé (ou *détruit* en jargon C++). Cette phrase évoque curieusement un cercle vicieux dans la mesure où un destructeur est appelé quand un objet est détruit. S'il ne s'agit pas de mémoire placée sur le tas (heap), il serait plus juste de dire "quand un objet se retrouve hors de portée". Un objet local se retrouve hors de portée quand la fonction se termine. Un objet global ou statique devient hors de portée quand le programme s'achève.

Et pour la mémoire heap ? Un pointeur peut se trouver hors de portée, mais pas la mémoire heap. Par définition, il s'agit d'une mémoire qui n'est pas associée à une fonction donnée. Un objet qui a été alloué sur le tas est détruit lorsque la mémoire correspondante est libérée grâce à la commande `delete`. Le programme qui suit, `DestructMembers`, illustre cela.

```
//  DestructMembers - construit et détruit un ensemble
//                    de membres de données
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Course
{
  public:
    Course()  { cout << "Construction de course" << endl; }
    ~Course() { cout << "Destruction de course" << endl;  }
};

class Student
{
  public:
```

```
    Student()
    {
        cout << "Construction de student" << endl;
        semesterHours = 0;
        gpa = 0.0;
    }
    ~Student() { cout << "Destruction de student" << endl; }
  protected:
    int  semesterHours;
    float gpa;
};
class Teacher
{
  public:
    Teacher()
    {
        cout << "Construction de teacher" << endl;
        pC = new Course;
    }
    ~Teacher()
    {
        cout << "Destruction de teacher" << endl;
        delete pC;
    }
  protected:
    Course* pC;
};
class TutorPair
{
  public:
    TutorPair()
    {
        cout << "Construction de tutorpair" << endl;
        noMeetings = 0;
    }
    ~TutorPair() { cout << "Destruction de tutorpair" << endl; }
  protected:
    Student student;
    Teacher teacher;
    int   noMeetings;
};

TutorPair* fn()
{
    cout << "On construit l'objet TutorPair dans la fonction fn()"
         << endl;
    TutorPair tp;
```

```
    cout << "Allocation de TutorPair sur le tas" << endl;
    TutorPair*  pTP = new TutorPair;

    cout << "Retour de fn()" << endl;
    return pTP;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // appelle la fonction fn() puis renvoie l'objet
    // TutorPair retourné au tas
    TutorPair* pTPReturned = fn();
    cout << "Retour de l'objet heap au tas" << endl;
    delete pTPReturned;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La fonction `main()` invoque une fonction `fn()` qui définit l'objet `tp`. Ceci vous permet de d'observer comment la variable devient hors de portée lorsque le contrôle ne se trouve plus à l'intérieur de la fonction. `fn()` alloue aussi de la mémoire sur le tas. Celle-ci est renvoyée à `main()` pour y être libérée.

L'exécution de ce programme donne la sortie suivante :

```
On construit l'objet TutorPair dans la fonction fn()
Construction de student
Construction de teacher
Construction de course
Construction de tutorpair
Allocation de TutorPair sur le tas
Construction de student
Construction de teacher
Construction de course
Construction de tutorpair
Retour de fn()
Destruction de tutorpair
Destruction de teacher
Destruction de course
Destruction de student
Retour de l'objet heap au tas
```

```
Destruction de tutorpair
Destruction de teacher
Destruction de course
Destruction de student
Appuyez sur une touche pour continuer...
```

Chaque constructeur est appelé tout à tour au fur et à mesure de la construction de l'objet `TutorPair`. On part du membre de donnée le plus "petit", pour remonter progressivement vers le constructeur `TutorPair::TutorPair()`.

Deux objets `TutorPair` sont créés. Le premier, `tp`, est défini localement dans la fonction `fn()`. Le second, `pTP`, est alloué sur le tas. L'objet `tp` devient hors de portée et est détruit lorsque la fonction n'a plus la main. La mémoire allouée sur le tas, dont l'adresse est retournée par `fn()`, n'est détruite que quand `main()` appelle explicitement la commande `delete`.

La séquence des destructeurs exécutés lorsqu'un objet est détruit est invoquée dans l'ordre inverse de l'appel des constructeurs correspondants.

Chapitre 17

Élaborer des arguments constructifs

Dans ce chapitre :

- Créer des constructeurs argumentés.
- Surcharger un constructeur.
- Créer des objets à l'aide des constructeurs.
- Appeler des constructeurs membres.
- Ordre de construction et de destruction.

Une classe représente un type d'objet du monde réel. Par exemple, nous avons utilisé la classe `Student` pour représenter les propriétés d'un étudiant, à savoir son nom et son numéro de Sécurité sociale. À l'instar des étudiants, les classes sont autonomes. En revanche, les classes sont responsables de leur propre état et de leur nourriture : une classe doit se maintenir tout le temps en état valide.

Le constructeur par défaut, présenté dans le Chapitre 16, n'est pas suffisant. Par exemple, un constructeur par défaut peut initialiser l'identificateur d'un étudiant à 0 pour s'assurer qu'il ne contient pas une valeur aléatoire. Mais un identificateur d'étudiant nul n'est probablement pas valide (pas l'étudiant, l'identificateur). C'est à la classe de veiller à ce que son identificateur – l'ID – soit initialisé à une valeur légale lorsque l'objet est créé.

Les programmeurs C++ ont besoin de constructeurs qui acceptent certains types d'arguments de façon à initialiser un objet avec autre chose que sa valeur

par défaut. Le présent chapitre est consacré aux constructeurs possédant des arguments.

Élaborer des constructeurs avec des arguments

Le C++ permet au programmeur de définir un constructeur doté d'arguments. Par exemple :

```
class Student
{
  public:
    Student(char *pName);

    // ...la classe continue...
};
```

Justifier les constructeurs

Une opération aussi limpide que l'ajout d'arguments à un constructeur ne devrait pas avoir à être justifiée, mais je le ferai quand même. D'abord, autoriser des arguments dans les constructeurs est commode. C'est idiot, pour un programmeur, de construire un objet par défaut et d'être obligé d'appeler immédiatement une fonction d'initialisation pour que des données y soient placées. Un constructeur avec des arguments ressemble à une journée de courses dans un centre commercial : il offre un service complet.

Une autre bonne raison d'attribuer des arguments à un constructeur, c'est qu'il n'est pas toujours possible de construire un objet par défaut qui soit réputé convenable. Rappelez-vous que le constructeur a pour mission de construire un objet légal (au sens où le définit la classe). Si un objet par défaut n'est pas légal, le constructeur ne peut pas faire son travail.

Par exemple, un compte bancaire sans numéro de compte n'est sans doute pas légal (le C++ n'en a cure, mais la banque tiquera certainement). Nous pourrions construire un objet `CompteBancaire` sans numéro et demander ensuite que l'application utilise quelque autre fonction membre pour initialiser le numéro de compte avant son utilisation. Mais ce procédé va à l'encontre de nos règles car il force la classe à dépendre de l'application en ce qui concerne son initialisation.

Utiliser un constructeur

D'un point de vue conceptuel, l'ajout d'un argument est simple. Un constructeur est une fonction membre et les fonctions membres peuvent avoir des arguments. Par conséquent, les constructeurs peuvent avoir des arguments.

Rappelez-vous cependant que vous n'appelez pas le constructeur comme une fonction normale. De ce fait, le seul moyen de passer des arguments à un constructeur est de le faire lorsque l'objet est créé. Par exemple, le programme qui suit crée un objet `s` de la classe `Student` en appelant le constructeur `Student(char*)`. L'objet `s` est détruit lorsque l'on revient à la fonction `main()`.

```
//  ConstructorWArg - fournit un constructeur avec arguments
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

const int MAXNAMESIZE = 40;

class Student
{
  public:
    Student(char* pName)
    {
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
        semesterHours = 0;
        gpa = 0.0;
    }

  // ...autres membres publics...
  protected:
    char  name[MAXNAMESIZE];
    int   semesterHours;
    float gpa;
};

int main(int argcs, char* pArgs[])
{
    Student s("O. Danny Boy");
    Student* pS = new Student("E. Z. Rider");

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
```

```
    system("PAUSE");
    return 0;
}
```

Le constructeur ressemble à ceux du Chapitre 16, sauf en ce qui concerne l'ajout de l'argument `pName` de type `char*`. Le constructeur initialise les membres de données à leur valeur vide de démarrage, sauf pour le membre de données `name` qui tire sa valeur initiale de `pName`.

L'objet est créé dans `main()`. L'argument à passer au constructeur apparaît dans la déclaration de `s`, juste à droite du nom de l'objet. De ce fait, l'étudiant `s` reçoit le nom "Danny" dans cette déclaration. L'accolade fermée appelle le destructeur pour rayer de la carte ce pauvre Danny.

Les arguments du constructeur apparaissent à la suite du nom de la classe lorsque l'objet est alloué en mémoire heap.

Dans ce chapitre, de nombreux constructeurs violent la règle qui stipule que "les fonctions de plus de trois lignes ne doivent pas être mises en mode *inline*". J'ai néanmoins décidé de les mettre inline car j'estime qu'elles seront ainsi plus lisibles pour vous. Ne suis-je pas plein de sollicitude ?

Surcharger le constructeur

Puisque j'en suis à faire dans ce chapitre des parallèles entre les constructeurs et d'autres fonctions plus normales, je vais aller encore plus loin : les constructeurs peuvent être surchargés.

La surcharge d'une fonction signifie que deux fonctions sont définies par le même nom court, mais avec des types d'arguments différents. Reportez-vous au Chapitre 6 pour en savoir plus sur la surcharge.

Le C++ choisit le constructeur qui convient selon les arguments figurant dans la déclaration de l'objet. Par exemple, la classe `Student` pourrait avoir trois constructeurs en même temps, comme le démontre l'exemple de programme que voici :

```
//  OverloadConstructor - fournit à la classe plusieurs
//                        façons de créer des objets en
//                        surchargeant le constructeur
//
#include <cstdio>
```

```
#include <cstdlib>
#include <iostream>
#include <strings.h>

using namespace std;

const int MAXNAMESIZE = 40;

class Student
{
  public:
    Student()
    {
        cout << "Construction d'un objet Student sans nom" << endl;
        semesterHours = 0;
        gpa = 0.0;
        name[0] = '\0';
    }
    Student(char *pName)
    {
        cout << "Construction de l'objet Student " << pName << endl;
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
        semesterHours = 0;
        gpa = 0;
    }
    Student(char *pName, int xfrHours, float xfrGPA)
    {
        cout << "Construction de l'objet Student " << pName << endl;
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
        semesterHours = xfrHours;
        gpa = xfrGPA;
    }
    ~Student()
    {
        cout << "Destruction de l'objet Student" << endl;
    }
  protected:
    char  name[40];
    int   semesterHours;
    float gpa;
};

int main(int argcs, char* pArgs[])
{
    // ces instructions invoquent trois constructeurs différents
```

```
    Student noName;
    Student freshman("Marian Haste");
    Student xferStudent("Pikumup Andropov", 80, 2.5);

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Comme l'objet `noName` est sans arguments, il est construit à l'aide du constructeur `Student::Student()`. Ce constructeur est appelé "constructeur par défaut" ou "constructeur *vide*" (je préfère la deuxième appellation, mais comme la première est la plus usitée, je l'utiliserai dans le livre. Je suis vraiment esclave de la mode...). L'objet `freshMan` est produit par un constructeur qui n'a qu'un argument `char*` tandis que `xferStudent` utilise un constructeur à trois arguments.

Remarquez la similitude entre les trois constructeurs. Le nombre d'heures semestrielles et la moyenne générale sont mis à zéro si seul le nom est fourni. En dehors de cela, il n'y a pas de différence entre les deux derniers constructeurs. Le dernier serait même inutile si vous pouviez vous contenter de spécifier une valeur par défaut pour ces deux arguments.

C++ vous permet d'indiquer une valeur par défaut pour un argument dans une déclaration de fonction afin d'anticiper le cas où cet argument ne serait pas présent. En ajoutant des paramètres par défaut au dernier constructeur, nos trois constructeurs peuvent être combinés en un seul. Par exemple, la classe ci-dessous répond intelligemment à ce souci :

```
//  ConstructorWDefaults - des constructeurs multiples peuvent
//                         souvent être combinés avec la présence
//                         d'arguments par défaut
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>
using namespace std;

const int MAXNAMESIZE = 40;

class Student
```

```
{
  public:
    Student(char *pName  = "sans nom",
            int xfrHours = 0,
            float xfrGPA = 0.0)
    {
        cout << "Construction de l'objet Student " << pName << endl;
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
        semesterHours = xfrHours;
        gpa = xfrGPA;
    }
    ~Student()
    {
        cout << "Destruction de l'objet Student" << endl;
    }

   // ...autres membres publics...
  protected:
    char  name[MAXNAMESIZE];
    int   semesterHours;
    float gpa;
};

int main(int argcs, char* pArgs[])
{
    // ces instructions invoquent trois constructeurs différents
    Student noName;
    Student freshman("Marian Haste");
    Student xferStudent("Pikumup Andropov", 80, 2.5);

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Maintenant, les trois objets sont créés à l'aide d'un même constructeur ; des paramètres par défaut sont fournis aux arguments non existants de `noName` et `freshMan`.

Dans les anciennes versions du C++, il n'était pas possible de créer un constructeur par défaut en fournissant des paramètres d'initialisation pour tous les arguments. Ce constructeur par défaut devait être un constructeur explicite distinct. Bien que cette restriction ait été levée dans la version standard (qui

semble être fondée sur de bonnes bases), certains vieux compilateurs risquent encore de l'imposer.

Les défaillances des constructeurs par défaut

Pour autant que le C++ soit concerné, chaque classe doit avoir un constructeur ; sinon, il vous sera impossible de créer le moindre objet de cette classe. Si vous ne fournissez pas un constructeur à vos classes, le C++ devrait peut-être se contenter de générer une erreur, mais ce n'est pas ce qu'il fait. Afin d'assurer la compatibilité avec le code C existant, qui ne sait rien des constructeurs, le C++ produit automatiquement un constructeur par défaut (une sorte de constructeur par défaut fourni par défaut) qui met tous les membres de données de l'objet à un zéro binaire.

Si votre classe a déjà un constructeur, le C++ s'abstient d'en produire un de sa propre initiative (comme il est évident que le programme que vous tapez n'est pas du C, le C++ ne se sent pas obligé de faire du travail en plus rien que pour garantir la compatibilité).

Il en résulte que si vous définissez un constructeur pour votre classe mais qu'il vous faut aussi un constructeur par défaut, vous devez le définir par vous-même.

Quelques lignes de programmation contribueront à démontrer cela. Ceci est légal :

```
class Student
{
  // Les mêmes données que précédemment, mais
  // sans constructeurs.
};

int main(int argcs, char* pArgs[])
{
   Student noName;
   return 0;
}
```

Ici, `noName` est déclaré sans arguments, de sorte que le C++ appelle le constructeur par défaut afin de construire un objet `Student`. Celui-ci se contente d'initialiser tous les membres de données de `Student`.

En revanche, le fragment de code qui suit ne se compile pas correctement :

```
class Student
{
  public:
    Student(char *pName);
};

int main(int argcs, char* pArgs[])
{
  Student noName;
  return 0;
}
```

L'ajout, apparemment inoffensif, du constructeur `Student(char*)` empêche le C++ de produire automatiquement un constructeur `Student` pour construire l'objet `noName`.

Éviter le piège de la déclaration d'objet

Voyons de nouveau comment les objets `Student` ont été déclarés dans l'exemple `ConstructorWDefaults` :

```
Student noName;
Student freshMan("Smell E. Fish");
Student xfer("Upp R. Classman", 80, 2.5);
```

Tous les objets `Student` hormis `noName` sont déclarés au constructeur avec des parenthèses de part et d'autre des arguments. Pourquoi `noName` est-il déclaré sans parenthèses ?

Pour être clair et cohérent, vous pourriez penser à une déclaration du style :

```
Student noName();
```

Malheureusement, cette syntaxe ne produit pas l'effet attendu. Au lieu de déclarer qu'un objet `noName` d'une classe `Student` doit être construit avec le constructeur par défaut, elle déclare une fonction qui retourne par valeur un objet de la classe `Student`. Surprise !

Les deux déclarations qui suivent démontrent la ressemblance du nouveau format C++ de déclaration d'un objet avec celle d'une fonction (à mon avis, c'est le fruit d'une erreur, mais qu'est-ce que j'en sais ?). La seule différence réside dans le fait que la déclaration d'une fonction contient des types entre parenthèses alors que la déclaration d'objet contient des objets :

Éviter le piège de la déclaration d'objet (*suite*)

```
Student ceciEstUneFonction(int);
Student ceciEstUnObjet(10);
```

Si les parenthèses sont vides, rien ne distingue un objet d'une fonction. Pour conserver la compatibilité avec le C, le langage C++ a choisi de considérer comme une fonction une déclaration dont les parenthèses sont vides. Une solution plus sûre aurait consisté à forcer le mot-clé `void` dans une fonction, mais la compatibilité avec les programmes écrits en C n'aurait plus été assurée.

Construire des membres de classe

Dans les exemples précédents, tous les membres de données possédaient un type simple, comme `int` et `float`. Ces types sont suffisants pour affecter une valeur à une variable à l'intérieur du constructeur. Mais que se passe-t-il par exemple si la classe contient des membres de données provenant d'une classe définie par l'utilisateur ?

Construire un membre de données complexe

Les membres d'une classe ont le même problème que n'importe quelle autre variable. Cela n'a aucun sens pour un objet `Student` d'avoir un certain identificateur par défaut égal à zéro (l'étudiant peut être nul, pas son numéro d'examen). Étudions l'exemple suivant :

```
//  ConstructingMembers - une classe peut passer des arguments
//                        aux constructeurs des membres
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>
using namespace std;

const int MAXNAMESIZE = 40;

int nextStudentId = 0;
```

```
class StudentId
{
  public:
    StudentId()
    {
        value = ++nextStudentId;
        cout << "Affectation de l'ID de l'objet Student "
             << value << endl;
    }
  protected:
    int value;
};

class Student
{
  public:
    Student(char* pName)
    {
        cout << "Construction de l'objet Student " << pName << "\n";
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
        semesterHours = 0;
        gpa = 0.0;
    }

  // ...autre membres publics...
  protected:
    char  name[MAXNAMESIZE];
    int   semesterHours;
    float gpa;
    StudentId id;
};

int main(int argcs, char* pArgs[])
{
    Student s("Chester");

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Un numéro d'identification (ID) est affecté à chaque étudiant lorsque l'objet `Student` est construit. Dans cet exemple, ces identificateurs sont gérés séquentiellement par la variable globale `nextStudentId`.

La classe `Student` contient un membre `id` de la classe `StudentId`. Le constructeur de `Student` ne peut affecter une valeur à ce membre `id` car `Student` n'a pas accès aux membres protégés de `StudentId`. Vous pourriez faire de `Student` un ami (*friend*) de `StudentId`, mais ceci enfreindrait le principe "Vous vous occupez de vos affaires et je m'occupe des miennes". D'une façon ou d'une autre, il vous faut appeler le constructeur de `StudentId` lorsque `Student` est construit.

Dans ce cas, le C++ le fait automatiquement à votre place : il appelle le constructeur par défaut `StudentId::StudentId()` sur `id`. Ceci se produit après que le constructeur `Student` a été appelé mais avant que le contrôle soit passé à la première instruction du constructeur (Procédez à une exécution pas à pas avec votre débogueur pour voir ce que je veux dire. Comme d'habitude, veillez à avoir forcé les fonctions *inline* en *outline*.) La sortie de ce programme simple donne :

```
Affectation de l'ID de l'objet Student 1
Construction de l'objet Student Chester
Appuyez sur une touche pour continuer...
```

Remarquez que le message du constructeur `StudentId` apparaît avant la sortie du constructeur `Student`. À propos, avec tous ces constructeurs qui font des sorties, vous devez vous imaginer que les constructeurs doivent absolument afficher quelque chose. Or, la plupart des constructeurs n'émettent pas le moindre son...

Si le programmeur ne propose pas de constructeur, le constructeur par défaut fourni par C++ appelle automatiquement les constructeurs par défaut de n'importe quel membre de données. Le destructeur de la classe appelle automatiquement le destructeur de n'importe quel membre de données qui en possède un. Le destructeur fourni par C++ en fait de même.

Tout cela est bien beau si l'on s'en tient au constructeur par défaut. Mais que se passerait-il si nous voulions appeler un autre constructeur ? Où mettrions-nous l'objet ? Pour simuler cette situation, supposons qu'au lieu de calculer l'identificateur de l'étudiant, celui-ci est fourni au constructeur `Student` qui le passe à son tour au constructeur de la classe `StudentId`.

Permettez-moi d'abord de vous montrer ce qui ne fonctionne pas. Considérons l'extrait de programme qui suit (une version plus complète, `ConstructSeparateID`, est disponible en téléchargement) :

```
class Student
{
  public:
   Student(char *pName  = "no name", int ssId = 0)
   {
      cout << "Construction de l'objet Student " << pName << "\n";
      strncpy(name, pName, sizeof(name));
      name[sizeof(name) - 1]  = '\0';
      // A NE PAS ESSSAYER. CELA NE MARCHE PAS !
      StudentId id(ssId);    // construit un ID pour un étudiant
   }
  protected:
   char  name[40];
   StudentId id;
};
```

Le constructeur de `StudentId` a été modifié pour accepter une valeur externe (la valeur par défaut est nécessaire pour que l'exemple se compile, pour des raisons qui deviendront claires d'ici peu). À l'intérieur du constructeur de `Student`, le programmeur (c'est moi) a (intelligemment) essayé de construire un objet `StudentId` nommé `id`.

L'examen de la sortie du programme révèle un problème :

```
Affectation de l'ID de l'objet Student : 0
Construction de l'objet Student Randy
Affectation de l'ID de l'objet Student : 1234
Destruction de l'objet StudentID 1234
Message provenant de main
Appuyez sur une touche pour continuer...
```

Le premier problème, c'est que le constructeur de `StudentId` semble avoir été appelé deux fois, une fois avec 0, une autre fois avec le chiffre 1234 attendu. Nous remarquons ensuite que l'objet 1234 est détruit avant que le message provenant de `main()` ne soit affiché. Apparemment, l'objet `StudentId` est détruit à l'intérieur même du constructeur `Student`.

L'explication de ce comportement assez bizarre est limpide. Le membre de données `id` existe déjà au moment où l'on entre dans le corps du constructeur. Au lieu de construire le membre de données existant `id`, la déclaration fournie au constructeur crée un objet local ayant le même nom. Cet objet local est détruit lorsque l'on revient du constructeur.

D'une manière ou d'une autre, il nous faudra un mécanisme particulier indiquant qu'il faut construire le membre existant et non en créer un nouveau. Ce mécanisme doit apparaître avant l'accolade ouverte, avant que les membres de données aient été déclarés. Le C++ fournit un constructeur adapté à cette situation. C'est ce que nous montre le programme `ConstructDataMembers` :

```
//  ConstructDataMember - construit un membre de données avec
//                        une valeur autre que par défaut
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>
using namespace std;

const int MAXNAMESIZE = 40;

class StudentId
{
  public:
    StudentId(int id = 0)
    {
        value = id;
        cout << "Affectation de l'ID de l'objet Student : " << value << endl;
    }

  protected:
    int value;
};

class Student
{
  public:
    Student(char *pName  = "no name", int ssId = 0)
      : id(ssId)
    {
        cout << "Construction de l'objet Student "
             << pName << endl;
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1]  = '\0';
    }
    ~Student()
    {
        cout << "Destruction de l'objet Student" << endl;
    }

  protected:
```

```
    char  name[40];
    StudentId id;
};

int main(int argcs, char* pArgs[])
{
    Student s("Chester", 1234);
    cout << "Message provenant de main" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Remarquez tout particulièrement la première ligne du constructeur. Il y a là quelque chose que vous n'avez pas rencontré auparavant. Le deux-points signifie que ce qui suit sont des appels aux constructeurs des membres de données de la classe courante. Pour le compilateur C++, cette ligne se lit "Construit le membre `id` en utilisant l'argument `ssId` du constructeur `Student`." Tous les membres de données qui ne sont pas appelés de la sorte sont construits avec le constructeur par défaut.

Ce nouveau programme produit la sortie attendue :

```
Affectation de l'ID de l'objet Student : 1234
Construction de l'objet Student Chester
Message provenant de main
Appuyez sur une touche pour continuer...
```

Vous constaterez en particulier que le destructeur n'est pas invoqué. Chester est bien vivant !

Construire un membre de données constant

Voici un autre sujet de réflexion : comment initialiser un membre qui a été déclaré constant (`const`) ? Souvenez-vous qu'une variable `const` est initialisée lorsqu'elle est déclarée et qu'elle ne peut plus être modifiée par la suite. Comment un constructeur peut-il alors affecter une valeur à un membre de données `const` ? Le problème est résolu avec la même syntaxe "deux-points" (:) que celle qui est utilisée pour initialiser les objets complexes.

```
class Mammifère
{
  public:
    Mammifère(int ndp) : nombreDePattes(ndp) {}
  protected:
    const int nombreDePattes;
};
```

Indépendamment des questions liées à la station débout, aux amputations et au fameux mouton à cinq pattes, tout le monde sait qu'un mammifère est un animal possédant quatre pattes. Cette information avérée devrait donc, en bonne logique, être déclarée `const` lorsque l'objet est créé. C'est ce que fait la déclaration ci-dessus. Le membre `nombreDePattes` ne peut plus être changé une fois qu'il a été déclaré et initialisé.

Les programmeurs utilisent fréquemment la syntaxe " : " pour initialiser des membres de données même si ce ne sont pas des constantes. Ce n'est pas une nécessité, juste une pratique courante.

L'ordre de construction

S'il existe de multiples objets, tous avec des constructeurs, les programmeurs ne se soucient en général pas de l'ordre dans lequel les choses sont construites. Cependant, cet ordre peut jouer un rôle si l'un ou l'autre de ces constructeurs induit des effets de bord.

Voici les règles qui régissent l'ordre de construction :

- Les objets locaux et statiques sont construits dans l'ordre où leurs déclarations sont appelées.
- Les objets statiques ne sont construits qu'une seule fois.
- Tous les objets globaux sont construits avant `main()`.
- Les objets globaux ne sont pas construits dans un ordre particulier.
- Les membres sont construits dans l'ordre de leur déclaration dans les classes.
- Les destructeurs sont appelés dans l'ordre inverse des constructeurs.

Une variable *statique* est une variable qui est locale par rapport à une fonction mais qui conserve sa valeur d'un appel à la fonction à un autre. Une variable *globale* est déclarée en dehors d'une fonction.

Examinons maintenant de plus près les règles qui viennent d'être énoncées.

Les objets locaux se construisent en ordre

Les objets locaux sont construits dans l'ordre dans lequel le programme rencontre leurs déclarations. C'est normalement le même que celui dans lequel les objets apparaissent, à moins que votre fonction effectue des sauts vers des déclarations particulières (à propos, les sauts aux déclarations sont à éviter car ils compliquent la lecture du listing ainsi que la tâche du compilateur).

Les objets statiques ne se construisent qu'une seule fois

Les objets statiques sont semblables aux autres variables locales, sauf qu'ils ne sont construits qu'une seule fois. Le C++ doit attendre que le contrôle passe pour la première fois par la déclaration d'objets statiques pour procéder à la construction. Voyons ce programme très élémentaire :

```
//  ConstructStatic - les objets statiques ne sont
//                    construits qu'une seule fois
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class DoNothing
{
  public:
     DoNothing(int initial)
     {
          cout << "DoNothing construit avec la valeur "
               << initial
               << endl;
     }
};
void fn(int i)
{
```

```
    cout << "La fonction fn passe une valeur de " << i << endl;
    static DoNothing dn(i);
}

int main(int argcs, char* pArgs[])
{
    fn(10);
    fn(20);
    system("PAUSE");
    return 0;
}
```

L'exécution de ce programme donne les résultats suivants :

```
La fonction fn passe une valeur de 10
DoNothing construit avec la valeur 10
La fonction fn passe une valeur de 20
Appuyez sur une touche pour continuer...
```

Remarquez que le message de la fonction `fn()` apparaît deux fois, mais le message du constructeur de `NeFaitRien` n'apparaît que la première fois où `fn()` est appelé.

Tous les objets globaux sont construits avant main()

Toutes les variables globales se trouvent à portée aussitôt que le programme démarre. De ce fait, tous les objets globaux sont construits avant que le contrôle ne soit passé à `main()`.

L'initialisation des variables globales peut provoquer bien des prises de tête au moment du débogage. Certains débogueurs essayent d'exécuter le programme jusqu'à `main()` dès que le programme a été chargé et avant qu'ils ne rendent le contrôle à l'utilisateur. Cela peut poser problème car le code du constructeur de tous les objets globaux a déjà été exécuté au moment où vous reprenez le contrôle de votre programme. Si l'un de ces constructeurs présente un bogue fatal, vous n'aurez aucune chance de localiser l'erreur (du moins, via le débogueur). Dans ce cas, le programme semble expirer avant même d'avoir commencé à vivre !

Il existe différentes approches de ce problème. L'une consiste à tester chaque constructeur sur des objets locaux avant de les utiliser sur des objets globaux.

Si cela ne suffit pas à trouver la solution, vous pouvez ajouter des instructions de sortie au début de chaque constructeur suspect. La dernière sortie que vous obtiendrez proviendra probablement du constructeur défectueux.

Les objets globaux ne se construisent pas dans un ordre particulier

Il est facile de se représenter l'ordre de construction des objets locaux, car cet ordre est celui du déroulement du programme. Mais avec les objets globaux, aucune continuité de ce genre n'impose un quelconque ordre. Tous les objets globaux sont mis à portée simultanément (vous vous en souvenez ?). D'accord, rétorquerez-vous, mais qu'est-ce qui empêche le compilateur de commencer au début du fichier et de parcourir la liste des objets globaux ?

Ce serait parfait pour un seul fichier (et je présume que c'est ce que font la plupart des compilateurs). Malheureusement, bon nombre des logiciels du "vrai" monde sont réalisés à partir de plusieurs fichiers qui sont compilés séparément puis assemblés. Comme le compilateur n'a aucun contrôle sur la façon dont ces fichiers sont assemblés, il ne peut non plus agir sur l'ordre dans lequel les objets globaux (qui sont éparpillés un peu partout) sont construits. C.Q.F.D.

La plupart du temps, le travail se fait tout seul. Mais, de temps en temps (disons assez souvent pour qu'il soit nécessaire d'en parler dans un livre), les variables globales sont susceptibles de provoquer des bogues particulièrement difficiles à localiser.

Voyons l'exemple suivant :

```
class Student
{
  public:
    Student (int id) : studentId(id) {}
    const int studentId;
};
class Tutor
{
  public:
    Tutor(Student& s) : tutoredId(s.studentId) {}
    int tutoredId;
};

// définition d'un étudiant
// set up a student
```

```
Student randy(1234);

// affectation de cet étudiant à un assistant
Tutor   jenny(randy);
```

Ici, le constructeur de `Student` affecte un identificateur (ID) d'étudiant. Le constructeur de `Tutor` enregistre l'ID de l'étudiant à aider. Le programme déclare l'étudiant `randy` et affecte à cet étudiant l'assistant (ou le professeur) `jenny`.

Le problème, c'est que ce programme suppose implicitement que `randy` est construit avant `jenny`. Supposons que ce soit le contraire : `jenny` serait alors construit(e) avec un bloc de mémoire qui n'a pas encore été converti en objet `Student` et, par conséquent, l'identificateur de l'étudiant contiendrait des informations incohérentes.

L'exemple précédent n'est pas très difficile à comprendre. Il est seulement un peu tiré par les cheveux. Néanmoins, les problèmes découlant des objets globaux construits sans ordre particulier peuvent se produire de façon très subtile. Pour les éviter, n'autorisez pas le constructeur d'un objet global à faire référence au contenu d'un autre objet global.

Les membres sont construits dans l'ordre où ils ont été déclarés

Les membres d'une classe sont construits dans l'ordre dans lequel ils sont déclarés à l'intérieur de cette classe. Ce n'est pas aussi évident qu'il y paraît. Examinons cet exemple :

```
class Student
{
  public:
    Student (int id, int age) : sAge(age), sId(id){}
    const int sId;
    const int sAge;
};
```

Dans cet exemple, `sId` est construit avant `sAge` bien qu'il apparaisse en deuxième position dans la liste d'initialisation du constructeur. Le seul cas où vous pourriez probablement détecter une différence dans l'ordre de construction est celui où les deux membres de données sont des instances d'une classe qui a un constructeur présentant des effets de bord.

Les destructeurs agissent dans l'ordre inverse des constructeurs

Enfin, quel que soit l'ordre dans lequel les constructeurs se mettent à l'ouvrage, vous pouvez avoir la certitude que les destructeurs sont appelés dans l'ordre exactement inverse (cela fait du bien de savoir qu'en C++ il existe au moins une règle qui s'applique sans "mais", "si", "sauf" ou "à condition que").

Chapitre 18

Copier le constructeur de copie

Dans ce chapitre :

- Présentation du constructeur de copie.
- Faire des copies.
- Copies automatiques.
- Copies de surface et copies profondes.
- Comment éviter toutes ces copies.

Le constructeur est une fonction spéciale que le C++ appelle automatiquement lorsqu'un objet est créé afin de lui permettre de s'initialiser de lui-même. Le Chapitre 16 a présenté la notion de constructeur, tandis que le Chapitre 17 a été consacré à d'autres types de constructeurs. Nous allons aborder ici une variante particulière, appelée *constructeur de copie*.

Copier un objet

Une copie est un constructeur que le C++ utilise pour créer des copies d'objets. Elle se présente sous la forme `X::X(X&)` où `X` est le nom de la classe. Autrement dit, le constructeur de la classe `X` prend en argument une référence à un objet de la classe `X`. Je sais bien que tout cela semble sans intérêt, mais laissez-moi une chance de vous expliquer pourquoi le C++ ne saurait se passer de cette chose.

À quoi le constructeur de copie peut-il servir ?

Imaginez un moment ce qui se passerait si vous appeliez une fonction comme celle-ci :

```
void fn(Student fs)
{
  // ... même scénario, autre argument...
}
int main(int argcs, char*, pArgs[])
{
  Student ms;
  fn(ms);
  return 0;
}
```

Lors de l'appel à `fn()`, le C++ passe une copie de l'objet `ms` et non l'objet lui-même.

Le C++ passe les arguments aux fonctions par valeur.

Essayez de vous représenter ce que signifie "créer une copie d'un objet". Le C++ prend d'abord un constructeur pour créer l'objet, voire une copie d'un objet existant. Il pourrait copier l'objet existant dans le nouvel objet, octet par octet, mais qu'en serait-il si nous ne voulions pas d'une simple copie physique ? Que se passerait-il si nous avions besoin d'autre chose ? Vous devez être capable de spécifier comment la copie doit être construite.

Un constructeur de copie est donc nécessaire dans l'exemple ci-dessus pour créer une copie de l'objet `ms` sur la pile durant l'appel à la fonction `fn()`. Ce constructeur de copie particulier sera `Student::Student(Student&)`. Dites-le trois fois rapidement pour exorciser les démons.

Utiliser le constructeur de copie

Le meilleur moyen de comprendre le fonctionnement du constructeur de copie, c'est de le voir en action. Observons attentivement le programme `CopyConstructor` ci-dessous :

```
//  CopyConstructor - exemple de constructeur de copie
//
#include <cstdio>
```

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

const int MAXNAMESIZE = 40;

class Student
{
  public:
    // constructeur classique
    Student(char *pName = "no name", int ssId = 0)
    {
        strcpy(name, pName);
        id = ssId;
        cout << "Construction de "  << name << endl;
    }

    // constructeur de copie
    Student(Student& s)
    {
        strcpy(name, "Copie de ");
        strcat(name, s.name);
        id = s.id;
        cout << "Construction de "  << name << endl;
    }

    ~Student()
    {
        cout << "Destruction de " << name << endl;
    }

  protected:
    char name[MAXNAMESIZE];
    int  id;
};

// fn - reçoit son argument par valeur
void fn(Student copy)
{
    cout << "Dans la fonction fn()" << endl;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Student chester("Chester", 1234);
    cout << "Appel de fn()" << endl;
    fn(chester);
```

```
    cout << "Retour de fn()" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

L'exécution de ce programme donne la sortie suivante :

```
Construction de Chester
Appel de fn()
Construction de Copie de Chester
Dans la fonction fn()
Destruction de Copie de Chester
Retour de fn()
Appuyez sur une touche pour continuer...
```

Le constructeur normal génère le premier message lors de la déclaration qui se trouve au début de `main()`. Celle-ci envoie ensuite le message "Appel de" juste avant d'appeler la fonction `fn()`. Au cours de ce processus, le C++ invoque le constructeur de copie afin qu'il réalise une copie de `Chester`. C'est cette copie qui est passée à `fn()`, qui affiche simplement un message indiquant où l'on se trouve à ce stade. La copie est détruite au retour de `fn()`. L'objet original, `Chester`, est détruit à son tour à la fin de `main()`.

Le constructeur de copie est signalé par des commentaires pour vous permettre de mieux suivre le processus. Il ressemble à un constructeur normal, sauf qu'il prend ses entrées d'un autre objet plutôt que de plusieurs arguments séparés. Il affecte d'abord la chaîne `Copie de` dans son champ `name`. Il copie ensuite le nom provenant de l'objet source `s` dans l'objet courant. Le constructeur affiche alors le résultat avant de se terminer.

La première ligne affichée montre l'objet `Chester` en cours de construction. La troisième ligne signale que le constructeur de copie est en train de générer une copie de l'objet `Student`. Cet affichage est d'ailleurs la seule action effectuée ici. Enfin, la copie est détruite dès que l'on quitte la fonction, avec sortie du message correspondant.

Le constructeur de copie automatique

Le constructeur de copie est aussi important que le constructeur par défaut. Disons suffisamment important pour que C++ estime qu'aucune classe ne devrait être écrite sans ce constructeur. Si vous ne fournissez pas un constructeur de copie de votre cru, le C++ le génère à votre place (ce comportement est différent de la génération du constructeur par défaut que le C++ ne produit que si vous n'en avez pas défini pour votre classe).

Le constructeur de copie fourni par le C++ effectue une copie membre par membre de chaque membre de données. Dans les anciennes versions du langage, la copie se faisait bit à bit. La différence réside dans le fait qu'une copie membre par membre appelle n'importe quel constructeur de copie qui pourrait exister pour les membres d'une classe, ce que ne fait pas la version bit à bit.

L'exemple qui suit illustre les effets de cette différence (vous remarquerez que la classe `Student` est exactement la même que dans l'exemple précédent, `CopyConstructor` :

```
//  DefaultCopyConstructor - démontre que le constructeur de
//                           copie invoque les constructeurs
//                           de copie de chaque membre
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

const int MAXNAMESIZE = 40;

class Student
{
  public:
    // constructeur classique
    Student(char *pName = "no name", int ssId = 0)
    {
        strcpy(name, pName);
        id = ssId;
        cout << "Construction de "  << name << endl;
    }

    // constructeur de copie
    Student(Student& s)
    {
```

```
        strcpy(name, "Copie de ");
        strcat(name, s.name);
        id = s.id;
        cout << "Construction de "  << name << endl;
    }

    ~Student()
    {
        cout << "Destruction de " << name << endl;
    }

  protected:
    char name[MAXNAMESIZE];
    int  id;
};

class Tutor
{
  public:
    Tutor(Student& s) : student(s) // invoque le constructeur de
    {                              // copie sur le membre student
        cout << "Construction de l'objet Tutor" << endl;
        id = 0;
    }
  protected:
    Student student;
    int id;
};

void fn(Tutor tutor)
{
    cout << "Dans la fonction fn()" << endl;
}

int main(int argcs, char* pArgs[])
{
    Student chester("Chester");
    Tutor tutor(chester);
    cout << "Appel de fn()" << endl;
    fn(tutor);
    cout << "Retour de fn()" << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

L'exécution de ce programme donne la sortie suivante :

```
Construction de Chester
Construction de Copie de Chester
Construction de l'objet Tutor
Appel de fn()
Construction de Copie de Copie de Chester
Dans la fonction fn()
Destruction de Copie de Copie de Chester
Retour de fn()
Appuyez sur une touche pour continuer...
```

La création de l'objet `Chester` appelle le constructeur "classique" de `Student` qui sort la première ligne. L'objet `Tutor` est créé en appelant le constructeur de copie de `Student` de façon à obtenir son propre membre de données `Student`. Cette opération provoque l'affichage des deux lignes suivantes.

Le programme passe ensuite une copie de l'objet `Tutor` à `fn()`. Comme je n'ai pas fourni de constructeur de copie pour la classe `Tutor`, le programme invoque celui que C++ propose par défaut. De cette façon, nous pouvons obtenir une copie afin de la passer à `fn()`.

Le constructeur de copie par défaut de `Tutor` appelle le constructeur de copie de chaque membre de données. Dans le cas de la "classe" `int`, cela se limite à dupliquer la valeur. Dans le cas de la classe `Student`, vous avez déjà vu ce qui se passe. C'est son constructeur de copie qui provoque l'affichage du message "Construction de Copie de Copie de Chester". Le destructeur correspondant est exécuté au moment où l'on quitte `fn()`.

Copies de surface contre copies profondes

Il semble évident qu'un constructeur de copie se doit d'effectuer des copies membre par membre. À part placer des choses aussi bêtes que les mots "Copie de" avant un nom d'étudiant, en quoi une copie autre que membre par membre pourrait-elle nous être utile ?

Imaginez ce qui se passerait si le constructeur allouait une entité, par exemple de la mémoire sur le tas. Si le constructeur de copie se contentait de dupliquer cet élément sans effectuer sa propre allocation, il s'en suivrait une situation gênante : deux objets penseraient avoir un accès exclusif au même élément. C'est encore pire lorsque le destructeur est appelé pour les deux objets et que chacun essaie de récupérer à son profit la même entité. Pour voir concrètement ce que cela donne, examinez l'exemple que voici :

```
//  ShallowCopy - effectuer une copie de surface bit à bit
//                (shallow copy) n'est pas correct lorsque
//                la classe contient des entités actives
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>
using namespace std;

class Person
{
  public:
    Person(char *pN)
    {
        cout << "Construction de " << pN << endl;
        pName = new char[strlen(pN) + 1];
        if (pName != 0)
        {
            strcpy(pName, pN);
        }
    }
    ~Person()
    {
        cout << "Destruction de " << pName << endl;
        strcpy(pName, "MEMOIRE DEJA DETRUITE");
        // delete pName;
    }
 protected:
    char *pName;
};

void fn()
{
    // création d'un nouvel objet
     Person p1("Ceci_est_un_nom_vraiment_long");

    // copie le contenu de p1 dans p2
    Person p2(p1);
}

int main(int argcs, char* pArgs[])
{
    cout << "Appel fn()" << endl;
    fn();
    cout << "Retour de fn()" << endl;

    // attend pour terminer le programme que l'utilisateur
```

```
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Ce programme génère l'affichage suivant :

```
Appel de fn()
Construction de Ceci_est_un_nom_vraiment_long
Destruction de Ceci_est_un_nom_vraiment_long
Destruction de MEMOIRE DEJA DETRUITE
Retour de fn()
Appuyez sur une touche pour continuer...
```

Ici, le constructeur de `Person` alloue de la mémoire heap afin d'y stocker le nom d'une personne, au lieu de s'en tenir à une certaine limite arbitraire imposée par un tableau de longueur fixe. Cependant, le destructeur copie un message dans ce tampon de mémoire plutôt que de le remettre sur le tas. La partie principale du programme appelle la fonction `fn()`, qui crée simplement une personne `p1` puis en fait une copie appelée `p2`. Les deux objets sont automatiquement détruits à la sortie de la fonction.

Lorsque vous exécutez ce programme, le constructeur n'affiche qu'un seul message. Ce n'est pas étonnant, car le constructeur de copie par défaut fourni par C++ pour engendrer `p2` n'effectue aucune sortie. Comme `p1` et `p2` se retrouvent hors de portée, vous n'obtenez pas les deux messages de sortie auxquels vous vous attendiez. Le premier destructeur indique bien `Destruction de Ceci_est_un_nom_vraiment_long`. Mais le second nous apprend que la mémoire a déjà été détruite.

Si vous vouliez vraiment supprimer le nom (en retirant le commentaire devant `delete pName;`), le programme deviendrait instable après la seconde destruction et pourrait même totalement planter.

Le constructeur est appelé une seule fois. Il alloue un bloc de mémoire contenant le nom de la personne. Le constructeur de copie fourni par le C++ duplique cette adresse dans le nouvel objet, mais sans allouer un nouveau bloc de mémoire.

L'origine du problème est illustrée sur la Figure 18.1. L'objet `p1` est copié dans le nouvel objet `p2`, mais ce n'est pas le cas des entités actives. Par conséquent, `p1` et `p2` finissent par pointer sur le même élément (de la mémoire heap en

l'occurrence). On appelle cela une *copie de surface* car, en copiant les membres eux-mêmes, elle ne fait qu'"effleurer la surface des choses".

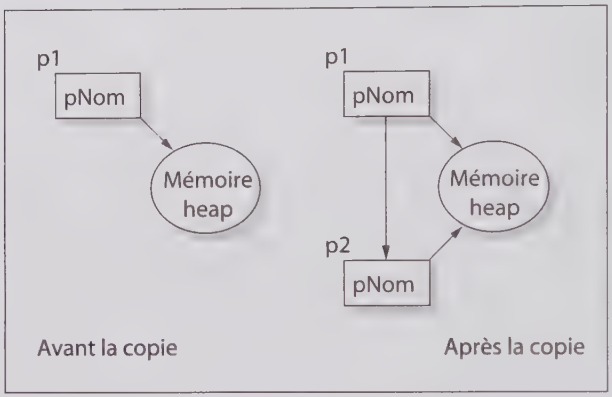


Figure 18.1
Copie de surface de p1 vers p2.

La solution à ce problème est indiquée visuellement sur la Figure 18.2. Vous devez disposer d'un constructeur de copie qui alloue ses propres éléments actifs au nouvel objet. Ajoutez-en un à `Person` et voyez ce que cela donne. Le listing ci-dessous propose un constructeur de copie approprié pour la classe `Person` (le programme complet ainsi modifié, et renommé `DeepCopy`, est disponible en téléchargement) :

```
class Person
{
  public:
   Person(char *pN)
   {
      cout << "Construction de " << pN << endl;
      pName = new char[strlen(pN) + 1];
      if (pName != 0)
      {
          strcpy(pName, pN);
      }
   }
   // le constructeur de copie alloue un nouveau bloc heap
   Person(Person &p)
   {
      cout << "Copie de " << p.pName
           << " dans son propre bloc" << endl;
      pName = new char[strlen(p.pName) + 1];
      if (pName != 0)
```

```
    {
       strcpy(pName, p.pName);
    }
  }
  ~Person()
  {
     cout << "Destruction de " << pName << endl;
     strcpy(pName, "MEMOIRE DEJA DETRUITE");
     // delete pName;
  }
 protected:
  char *pName;
};
```

Vous constatez ici que le constructeur de copie alloue son propre bloc de mémoire pour le nom, puis il copie le contenu du nom d'objet source dans ce nouveau bloc (voir la Figure 18.2). La *copie profonde* est ainsi nommée parce qu'elle peut atteindre et copier tous les éléments courants (certes, cette analogie est un peu tirée par les cheveux, mais c'est comme cela qu'ils s'expriment).

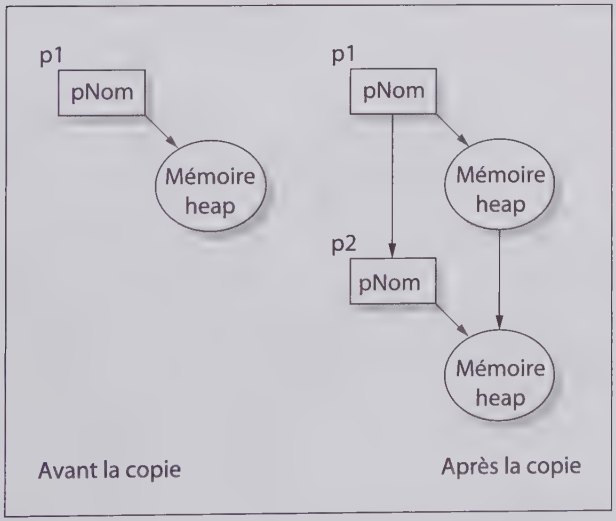


Figure 18.2
La copie en profondeur de p1 vers p2.

La sortie du programme se présente maintenant ainsi :

```
Appel de fn()
Construction de Ceci_est_un_nom_vraiment_long
Copie de Ceci_est_un_nom_vraiment_long dans son propre bloc
Destruction de Ceci_est_un_nom_vraiment_long
Destruction de Ceci_est_un_nom_vraiment_long
Retour de fn()
Appuyez sur une touche pour continuer...
```

Le destructeur de `Person` indique maintenant que les pointeurs de chaîne dans `p1` et `p2` n'adressent plus le même bloc de données. N'oubliez surtout pas que le message affiché par ce destructeur n'a d'autres vertus que pédagogiques. En réalité, il ne fait rien de particulier !

Les objets temporaires

Des copies d'objets sont générées en C++ lorsque ceux-ci sont passés par valeur (c'est ce que nous avons vu précédemment). C'est le cas le plus évident, mais ce n'est pas le seul. C++ crée en effet des copies d'objets dans d'autres conditions.

Considérons une fonction qui retourne un objet par valeur. Il faut alors que C++ le duplique en utilisant le constructeur de copie. Le morceau de code qui suit illustre cette situation :

```
Student fn();  // retourne l'objet par valeur
int main (int argcs, char* pArgs[])
{
   Student s;
   s = fn();   // l'appel à fn() crée un objet temporaire

   // Combien de temps dure un objet temporaire
   // retourné par fn() ?
   return 0;
}
```

La fonction `fn()` retourne un objet par valeur. Cet objet est finalement copié dans `s`, mais où diable réside-t-il en attendant ?

Le C++ crée un objet temporaire dans lequel il place l'objet retourné (le même résultat peut aussi être atteint par d'autres moyens). "C'est bien beau." direz-vous, "Mais comment le C++ saura-t-il à quel moment il doit le détruire ?" (C'est ce qu'on appelle poser les bonnes questions au bon moment.)

Dans cet exemple, cela n'a pas grande importance puisque vous en aurez fini avec l'objet temporaire lorsque le constructeur l'aura copié dans s. Mais qu'en est-il si s est défini en tant que référence ? Par exemple :

```
int main(int argcs, char* pArgs[])
{
   Student& refS = fn();
   // ...et ensuite ?...
   return 0;
}
```

Maintenant, la durée de vie de l'objet devient un sujet d'importance, car refS existe pour la fonction tout entière. Un objet temporaire créé par le compilateur est valide tout au long de l'existence de l'expression étendue dans laquelle il a été défini, et pas davantage.

Dans la fonction suivante, je signale l'endroit à partir duquel l'objet temporaire n'est plus valide :

```
Student fn1();
int fn2(Student&);
int main(int argcs, char* pArgs[])
{
    int x;
    // Création d'un objet Student en appelant fn1().
    // Passation de cet objet à la fonction fn2().
    // La fonction fn2() retourne un entier
    // utilisé pour des calculs quelconques.
    // Pendant tout ce temps, l'objet temporaire
    // retourné par fn1() reste valide.
    x = 3 * fn2(fn1()) + 10;

    // L'objet temporaire retourné par fn1() n'est
    // désormais plus valide.
    // ...ce que vous voulez ici...
    return 0;
}
```

L'exemple de référence précédent n'est pas valide car l'objet risque de disparaître avant refS, laissant alors refS se référer à un non-objet.

Éviter en permanence tout ce qui est temporaire

Vous avez dû vous rendre compte que la copie de tous ces objets de-ci de-là peut prendre du temps. Comment faire si vous ne voulez pas faire des copies de tout ? La solution la plus directe consiste à passer les objets aux fonctions (et à les faire retourner par celles-ci) uniquement par référence. Cette manipulation vient à bout de la majorité des cas.

Mais qu'en est-il si vous n'êtes toujours pas persuadé que C++ n'est pas en train de construire astucieusement des objets temporaires dont vous ne savez rien ? Ou si votre classe alloue des éléments uniques que vous ne voulez pas voir copier ? Que faire dans ces cas-là ?

Vous pouvez simplement ajouter une instruction de sortie à votre constructeur de copie. La présence de ce message vous préviendra qu'une copie vient d'être faite.

Une autre approche consiste à déclarer le constructeur de copie en mode protégé, comme dans :

```
class Student
{
  protected:
  Student(Student& s){}

  public:
  // ... le reste est normal...
};
```

Ceci empêche toute fonction extérieure, y compris C++, de construire une copie de vos objets `Student`. Mais cela n'affecte en rien la possibilité, pour les fonctions membres, de créer des copies. Si personne ne peut invoquer le constructeur de copie, aucune copie ne sera générée. Et c'est tout.

Constructeur de copie et référence

Le fait que le constructeur de copie soit utilisé pour créer des objets temporaires et des copies sur la pile répond à un de ces détails qui vous empoisonnent l'existence. Considérez le morceau de programme suivant :

```
class Student
{
  public:
    Student(Student s)
    {
        // ...ce que vous voulez...
    }
};

void fn(Student fs) {}

void fn()
{
    Student ms;
    fn(ms);
}
int main(int argcs, char* pArgs[])
{
            Student ms;
            fn(ms);
            return 0;
}
```

Remarquez que l'argument du constructeur de copie n'est plus référentiel. En fait, une telle déclaration n'est absolument pas légale. Le compilateur Dev-C++ génère une horrible série de messages d'erreur bien peu informatifs. Un autre compilateur C++ du domaine public affiche une explication qui nous renseigne un peu mieux :

```
Error: invalid constructor; you probably meant 'Student(const Student&)'
```

Soit :

```
Erreur : constructeur non valide ;
vous avez probablement voulu dire "Student(const Student&)"
```

Pourquoi l'argument au constructeur devrait-il être référentiel ? Examinez attentivement le programme. Quand `main()` appelle la fonction `fn()`, le compilateur C++ utilise le constructeur de copie pour créer une copie de l'objet `Student` sur la pile. Cependant, le constructeur de copie lui-même requiert un objet de la classe `Student`. Pas de problème, car le compilateur peut appeler le constructeur de copie afin qu'il crée un objet `Student` pour le

constructeur de copie. Mais bien sûr, cette opération exige un autre appel au constructeur de copie, et il en va ainsi jusqu'à ce que le compilateur finisse éventuellement par s'écrouler, victime d'épuisement par saturation.

Chapitre 19

Les membres statiques

Dans ce chapitre :

- Comment déclarer des membres de données statiques ?
- Quelles sont les fonctions des membres statiques ?
- Pourquoi ma fonction de membre statique ne peut-elle pas appeler les autres fonctions membres ?

Par défaut, les membres de données sont alloués "par objet". Par exemple, chaque étudiant a un nom.

Vous pouvez aussi déclarer un membre de manière à ce qu'il soit partagé par tous les objets d'une classe, en le déclarant comme membre statique. Le terme *statique* s'applique à la fois aux membres de données et aux fonctions membres bien que leur signification soit légèrement différente. Ce chapitre explique ces différences en commençant par les membres de données statiques.

Définition d'un membre statique

En les déclarant comme *statiques,* les membres de données sont définis comme étant communs à tous les objets d'une classe. On les appelle alors *membres de données statiques* (ils seraient vexés qu'on leur donne un autre nom).

Voici pourquoi vous en avez besoin

La plupart des propriétés sont des propriétés de l'objet. Dans l'exemple rebattu (usé jusqu'à la corde ?) de l'étudiant, des propriétés telles que le prénom, le nom, le numéro de Sécurité sociale, et ainsi de suite, sont bien sûr spécifiques à chaque étudiant. Certaines propriétés sont cependant partagées

par tous les étudiants. C'est le cas du nombre d'étudiants inscrits, de la meilleure note ou du pointeur vers le premier étudiant d'une liste chaînée.

Il est assez facile de stocker ce type d'informations dans une variable globale commune, ordinaire et très courante. Nous pourrions par exemple utiliser une modeste variable `int` pour conserver une trace du nombre d'objets `Student`. Le problème, c'est que les variables globales sont en dehors de la classe ; ce serait comme laisser le fil qui alimente mon four à micro-ondes débranché. Le courant passe toujours, mais il ne mène à rien ; et si mon chien se prend les pattes dans la prise sous tension, je serai obligé de le décoller du plafond avec un démonte-pneu, ce qui ne me ferait pas plaisir (ni au chien d'ailleurs).

Si une classe est censée être responsable de son propre état, les variables globales doivent être amenées dans la classe, tout comme le courant électrique doit être acheminé dans le four à micro-ondes, et non dans les pattes du chien. C'est là le concept sous-jacent aux membres statiques.

Il vous arrivera d'entendre parler des membres statiques sous le nom de *membres de classe* car ils sont partagés par tous les objets d'une classe. Par comparaison, les membres normaux sont alors appelés *membres d'instance* ou *membres objets,* car chaque objet reçoit sa propre copie de ces membres.

Utiliser les membres statiques

Un membre de données statique est toujours déclaré par la classe de stockage `static`. Par exemple :

```
class Student
{
  public:
    Student(char *pName = "no name") : name(pName)
    {
        noOfStudents++;
    }

    ~Student()
    {
        noOfStudents--;
    }

    static int noOfStudents;
    string name;
};
```

```
Student s1;
Student s2;
```

Le membre de données `noOfStudents` fait partie de la classe `Student` mais il n'appartient ni à `s1`, ni à `s2`. C'est-à-dire que pour chaque objet de la classe `Student`, il existe un nom `name` distinct, mais il y a seulement un nombre d'étudiants `noOfStudents` que tous les étudiants (`Student`) doivent se partager.

"Oui mais alors," vous demandez-vous, "si la place pour `noOfStudent` n'a été allouée à aucun des objets de la classe `Student`, où l'a-t-elle été ?" La réponse est : "Elle n'a pas été allouée". Vous devez le faire vous-même, avec l'instruction qui suit :

```
int Student::noOfStudents = 0;
```

Cette syntaxe un peu curieuse alloue de l'espace pour le membre de données statique et l'initialise à zéro. Les membres de données statiques doivent être globaux : une variable statique ne peut pas être locale à une fonction.

Spécifier le nom de la classe est obligatoire pour tout membre qui apparaît en dehors des limites de sa classe.

L'effort à faire pour allouer manuellement de l'espace peut sembler étonnant. Mais n'oubliez pas que les définitions des classes se trouvent éparpillées dans divers fichiers qui sont inclus par de multiples modules de code source. Le C++ doit savoir dans quels fichiers sources `.CPP` il doit allouer de l'espace pour les variables statiques. Ce problème ne se pose pas avec les variables non statiques, puisque la mémoire est alors allouée au niveau de chacun des objets créés.

Référencer les membres de données statiques

Les règles d'accès pour les membres statiques sont les mêmes que celles auxquelles se soumettent les membres normaux. À l'intérieur d'une classe, les membres statiques sont référencés comme n'importe quel autre membre de classe. Les membres statiques publics peuvent être référencés depuis l'extérieur de la classe, ce qui n'est pas possible avec les membres statiques dûment protégés. Les deux types de référence figurent dans l'extrait de programme qui suit :

```
class Student
{
 public:
  Student()
  {
   noOfStudents++; // référence depuis l'intérieur de la classe
   // ...la suite ici...
  }

  static int noOfStudents;
   // ...et voici d'autres instructions...
};

void fn(Student& s1, Student& s2)
{
  // référence statique publique
  cout << "Nombre d'étudiants "
       << s1.noOfStudents  // référence depuis l'extérieur de
                           // la classe
       << endl;
}
```

Dans `fn()`, `noOfStudents` est référencé par l'objet `s1`. Mais `s1` et `s2` partagent le même membre `noOfStudents`. Comment saurais-je que je dois choisir `s1` ? Pourquoi ne pas utiliser `s2` à la place ? Il n'y a pas de différence. Vous pouvez référencer un membre statique avec n'importe quel objet de sa classe. Par exemple :

```
// ...classe définie comme auparavant...
void fn(Student& s1, Student& s2)
{
  // ce qui suit donne des résultats identiques.
  cout << "Nombre d'étudiants "
       << s1.noOfStudents << endl;
  cout << "Nombre d'étudiants "
       << s2.noOfStudents << endl;
}
```

En fait, vous n'avez même pas besoin d'un objet. Vous pouvez utiliser directement le nom de la classe, si vous le préférez, en procédant ainsi :

```
// ...classe définie comme auparavant...
void fn(Student& s1, Student& s2)
{
```

```
  // ce qui suit donne le même résultat
  cout << "Nombre d'étudiants "
       << Student::noOfStudents
       << endl;
}
```

Si vous vous servez d'un nom d'objet pour accéder à un membre statique, le C++ n'utilisera que la classe de cet objet.

Ce n'est qu'un détail technique, mais qu'il est peut-être utile de signaler : l'objet utilisé pour référencer un membre statique n'est pas évalué même si c'est une expression. Prenons le cas suivant :

```
class Student
{
  public:
   static int noOfStudents;
   Student& nextStudent();
   // ...ce que vous voulez ici...
};

void fn(Student& s)
{
  cout << s.nextStudent(), noOfStudents << endl;
}
```

La fonction membre `nextStudent()` n'est pas véritablement appelée. Pour accéder à `noOfStudents`, le C++ a uniquement besoin du type de retour et il peut l'obtenir sans avoir à évaluer une expression. Ceci est vrai même si `nextStudent()` doit faire d'autres choses, comme laver les carreaux ou cirer vos chaussures. Rien de tout cela ne sera fait. Mais bien que cet exemple soit obscur, c'est ainsi que cela se passe. Et c'est ce que vous obtiendrez si vous essayez de fourrer trop de choses dans une seule expression.

Utilisations des membres de données statiques

Les membres de données statiques peuvent servir à une foule de choses. Voyons-en simplement quelques applications. Vous pouvez d'abord les utiliser pour compter le nombre d'objets qui circulent dans le coin. Dans la classe `Student`, par exemple, le décompte est initialisé à zéro, le constructeur l'incrémente et le destructeur le décrémente. À tout moment, le membre statique

contient le compte exact du nombre d'objets `Student` existants, y compris les objets temporaires (et donc pas nécessairement le nombre d'étudiants).

Un usage semblable consiste à utiliser un membre statique comme drapeau qui indique si une action particulière s'est produite. Par exemple, une classe `Radio` peut avoir besoin d'initialiser du matériel avant d'envoyer une première commande `réglage`, mais pas avant les `réglages` subséquents. Un drapeau servira à indiquer qu'il s'agit du premier `réglage`. Des drapeaux peuvent aussi être créés lorsqu'une erreur s'est produite.

Un autre usage classique consiste à placer dans un membre statique le pointeur vers le premier membre d'une liste chaînée (si le terme *pointeur de tête* ne vous dit rien, reportez-vous au Chapitre 14). Des membres statiques peuvent en effet allouer des "données communes" que partageront tous les objets de toutes les fonctions. Notez toutefois qu'une utilisation exagérée de cette mémoire commune n'est vraiment pas recommandée car elle complique la recherche des erreurs.

Déclarer des fonctions membres statiques

Les fonctions membres peuvent elles aussi être déclarées comme statiques. Cela peut être pratique si vous voulez associer une action à une classe sans qu'elle dépende d'un objet particulier. Par exemple, la fonction membre `Pigeon::Vole()` concerne un certain pigeon, ce qui n'est évidemment pas le cas du plus violent `Pigeon::TirAu()`.

À l'instar des membres de données statiques, ces fonctions sont associées à une classe et non à un objet particulier de cette classe. Cela signifie que, tout comme une référence à un membre de données statique, une référence à une fonction membre statique n'exige aucun objet. Si un objet est présent, seul son type est utilisé.

De ce fait, dans le listing qui suit, les deux appels à la fonction membre statique `number()` sont légaux :

```
// CallStaticMember - présente deux méthodes pour appeler
//                    une fonction membre statique
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
class Student
{
  public:
    Student(char* pN = "no name")
    {
        pName = new char[strlen(pN) + 1];
        if (pName)
        {
            strcpy(pName, pN);
        }
        noOfStudents++;
    }
    ~Student() { noOfStudents--; }
    static int number() { return noOfStudents; }

  // ...inchangé...
  protected:
    char* pName;
    static int noOfStudents;
};
int Student::noOfStudents = 0;

int main(int argcs, char* pArgs[])
{
    Student s1("Chester");
    Student s2("Scooter");
    cout << "Nombre d'objets Student : "
         << s1.number() << endl;
    cout << "Nombre d'objets Student : "
         << Student::number() << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Remarquez comment la fonction membre peut accéder au membre de données statique. Une fonction membre statique n'est toutefois pas directement associée à un objet, de sorte qu'elle ne bénéficie d'aucun accès par défaut aux membres non statiques. C'est pourquoi ce qui suit n'est pas légal :

```
class Student
{
  public:
```

```
    // ce qui suit n'est pas légal
    static char *sName()
   {
    return pName;  // Quel nom ? Il n'y a pas d'objet.
   }

    // ...le reste inchangé...
 protected:
  char* pName;
  static int noOfStudents;
}
```

Cela ne signifie en rien que les fonctions membres statiques n'ont pas accès aux membres de données statiques. C'est ce que nous prouve la fort utile fonction `findName()` présentée ci-dessous et qui trouve un objet spécifique dans une liste chaînée (revoyez à ce sujet le Chapitre 14). L'essentiel du code prenant en charge le fonctionnement de la liste chaînée est laissé, cher lecteur, chère lectrice, à votre sagacité. Bref, c'est un exercice (moi aussi, je déteste ce mot !). Voici donc le projet de fonction `findName()` :

```
class Student
{
  public:
    Student(char *pName)
    {
        // ...construit l'objet et l'ajoute à la
        //    liste des objets Student...
    }

    // findName - retourne l'étudiant spécifié
    static Student *findName(char *pName)
    {
        // ...parcours de la liste en partant du premier
        //    objet pointé par pHead et en avançant dans
        //    la liste grâce à pNext jusqu'à ce que
        //    l'objet voulu soit trouvé...
    }

  protected:
    static Student *pHead;
    Student *pNext;
    char* pName;
};
Student* Student::pHead = 0;
```

La fonction de recherche `findName()` a accès à `pHead` car elle est partagée par tous les objets. Du fait qu'elle est un membre de la classe `Student`, la fonction `findName()` a aussi accès à `pNext`, ce qui lui permet de naviguer dans la liste jusqu'à ce que l'objet voulu soit trouvé. Le morceau de code qui suit montre comment de telles fonctions membres statiques peuvent être utilisées :

```
int main(int argcs, char* pArgs[])
{
  Student s1("Randy");
  Student s2("Jenny");
  Student s3("Kinsey");
  Student *pS = Student::findName("Jenny");
  return 0;
}
```

Mais qu'est-ce que ce this ?

J'ai mentionné le mot-clé `this` plusieurs fois dans ce livre, mais attardons-nous un peu dessus. Le terme `this` désigne un pointeur vers l'objet "courant" à l'intérieur d'une fonction membre. Il est utilisé lorsque aucun autre nom d'objet n'a été spécifié. Dans une fonction membre normale, `this` est le premier argument implicite de la fonction. Par exemple :

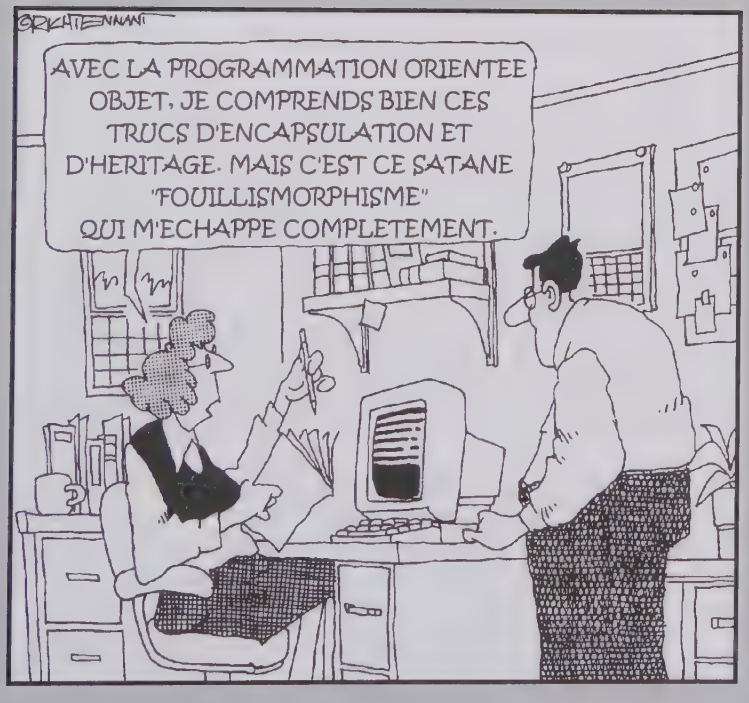
```
class SC
{
 public:
    void nFn(int a);  // comme SC::nFn(SC *this, int a)
    static void sFn(int a);  // comme SC::sFn(int a)
};

void fn(SC& s)
{
    s.nFn(10);  // converti en SC::nFn(&s, 10);
    s.sFn(10);  // converti en SC::sFn(10);
}
```

La fonction `nFn` est interprétée pratiquement comme si elle avait été déclarée sous la forme `void SC::nFn(SC *this, int a)`. L'appel à `nFn()` est converti par le compilateur sous cette forme, l'adresse de `s` étant passée comme premier argument (en fait, vous ne pouvez pas écrire l'appel de cette manière ; c'est seulement le compilateur qui procède ainsi).

Les références aux autres membres non statiques à l'intérieur de `SC::nFn()` utilisent automatiquement l'argument `this` comme pointeur vers l'objet courant. Lorsque `SC::sFn()` a été appelée, aucune adresse d'objet n'a été passée. De ce fait, elle ne dispose pas du pointeur `this` pour référencer des fonctions non statiques. C'est pourquoi nous disons qu'une fonction membre statique n'est associée à aucun objet courant.

Quatrième partie

Les héritages de classe



Dans cette partie...

Après nos réflexions de la troisième partie du livre sur la philosophie orientée objet, il apparaît que deux caractéristiques majeures du monde réel ne sont manifestement pas partagées par ce que la programmation fonctionnelle peut nous offrir comme solutions.

La première de ces fonctionnalités est la capacité à traiter des objets séparément. Reprenons la métaphore du four à micro-ondes. Le four est équipé d'une interface (le panneau en façade) qui me permet de le contrôler sans me soucier le moins du monde de ce qui se passe à l'intérieur. Ceci est vrai même si je sais tout sur le fonctionnement de cet engin de mort (et c'est loin d'être mon cas).

La seconde de ces fonctionnalités inspirées par le monde réel est la capacité à catégoriser des objets, c'est-à-dire à reconnaître et exploiter leurs similitudes. Si ma recette exige un four de n'importe quel type, le four à micro-ondes devrait convenir puisque c'est justement... un four.

J'ai déjà expliqué par quels mécanismes le C++ implémente la première fonctionnalité grâce aux classes. Pour supporter le deuxième aspect de la programmation orientée objet, le C++ fait appel à un concept appelé *héritage*, qui étend la notion de classe.

L'héritage est le sujet principal de cette quatrième partie.

Chapitre 20

Hériter d'une classe

Dans ce chapitre :

- Définition de l'héritage.
- Hériter d'une classe de base.
- Construire la classe de base.
- Les relations IS_A et HAS_A.

Ce chapitre est consacré à l'*héritage* qui est la capacité d'une classe à pouvoir hériter des propriétés ou des capacités d'une autre classe.

L'héritage est un concept courant. Je suis un être humain (sauf peut-être le matin quand je me réveille). J'ai hérité de certaines propriétés de la classe `Humain`, notamment la capacité de converser (plus ou moins...) intelligemment et ma dépendance à l'air, à l'eau et à une nourriture riche en hydrates de carbone (dont je crains bien d'être un peu trop dépendant). Ces propriétés ne sont pas uniques aux humains. La classe `Humain` hérite sa dépendance à l'air, à l'eau et à la nourriture de la classe `Mammifère` qui elle-même en a héritée de la classe `Animaux`.

La capacité de transmettre des propriétés est l'une des plus puissantes qui soient. Elle permet de décrire des choses d'une manière économique. Si par exemple mon fils me demande "C'est quoi un canard ?", je peux lui répondre que "C'est un oiseau qui cancane". Contrairement à ce que vous pourriez croire, cette réponse véhicule une grande quantité d'informations. Il sait ce qu'est un oiseau et il sait maintenant que les caractéristiques d'un oiseau s'appliquent au canard, qui présente en outre une intéressante propriété : il cancane (reportez-vous au Chapitre 12 pour trouver d'autres causeries sur des propriétés aussi profondes que celle-là).

Les langages orientés objet expriment cette relation en permettant à une classe d'hériter d'une autre. De ce fait, les langages OO savent générer un modèle plus

proche du monde réel (le monde réel, on y revient toujours !) que les modèles générés par les langages qui ne supportent pas l'héritage.

Le C++ permet à une classe d'hériter d'une autre classe de la manière suivante :

```
class Student
{};

class GraduateStudent : public Student
{
};
```

Un étudiant de troisième cycle (`GraduateStudent`) est l'héritier de tous les membres de la classe `Student`. Par conséquent, cet étudiant EST UN étudiant (avec C++, nous dirons `IS_A`, les majuscules soulignant l'importance de cette relation). Bien entendu, `GraduateStudent` peut parfaitement contenir d'autres membres que ceux de `Student` afin de caractériser ce qui est spécifique à un étudiant de troisième cycle.

Ai-je besoin d'héritage ?

La notion d'héritage a été introduite dans le C++ pour plusieurs raisons. La principale, bien sûr, c'est la capacité d'exprimer une relation d'héritage (j'y reviendrai dans un moment). Une autre raison, certes plus mineure, est que cela permet de réduire la saisie au clavier. Supposons que vous ayez une classe `Student` et qu'on vous demande d'ajouter une nouvelle classe appelée `GraduateStudent`. L'héritage peut réduire considérablement le nombre de choses que vous avez à placer dans cette classe. La classe `GraduateStudent` devra simplement contenir tout ce qui décrit les différences entre un étudiant en général (`Student`) et un diplômé ou un thésard en particulier (`GraduateStudent`).

On en déduit immédiatement un autre mot à connaître : *réutilisation*. Les petits génies du développement de logiciels se sont vite rendu compte que ce n'était pas la peine de réinventer la roue à chaque nouveau projet et de retaper les mêmes éléments de programmation. Une classe bien conçue doit être réutilisable. C'est clair.

Ceci nous amène à un autre bénéfice de l'héritage. Imaginons un héritage provenant de quelque classe existante. Vous découvrez par la suite que la classe de base présente un bogue, ou simplement qu'elle ne fait pas exactement tout ce dont a besoin la sous-classe. Modifier la classe de base peut provoquer des effets de bord gênants, voire rendre inutilisable le code qui l'utilise. Créer une nouvelle sous-classe qui répare la fonctionnalité incorrecte

permettra de résoudre la difficulté sans créer de problèmes supplémentaires aux autres.

IS_A belle ?

Pour prendre conscience de son environnement, l'être humain s'adonne à la taxonomie. Fido appartient à une catégorie spécifique de chiens, eux-mêmes membres particuliers de la famille des canidés, qui appartiennent à l'ordre des mammifères, et ainsi de suite. C'est de cette façon que se forme notre compréhension du monde environnant.

Pour prendre un autre exemple un étudiant est un type (un peu spécial) de personne. Ceci dit, je connais plein de choses sur les étudiant(e)s (américain(e)s, du moins). Je sais qu'ils/elles ont un numéro de Sécurité sociale, qu'ils/elles regardent trop la télé et qu'ils/elles fantasment un maximum sur le sexe opposé (surtout les mâles). Je sais tout cela parce qu'il s'agit de propriétés communes à tous ces gens.

En C++, nous dirions que la classe `Étudiant` hérite de la classe `Personne`. Nous dirions aussi que `Personne` est la *classe de base* de `Étudiant` et que, corollairement, `Étudiant` est une *sous-classe* de `Personne`. Enfin, nous dirions que `Étudiant` IS_A ("est une") `Personne` (cette relation est communément exprimée en lettres majuscules). Le C++ partage cette terminologie avec d'autres langages orientés objet.

Remarquez que, bien que `Étudiant` IS_A `Personne`, l'inverse n'est pas vrai. Une `Personne` IS not A ("n'est pas") un `Étudiant` (une déclaration comme celle-ci est toujours très générale ; il se peut évidemment qu'une `Personne` en particulier soit un `Étudiant`). Beaucoup de gens appartenant à la classe `Personne` ne sont pas membres de la classe `Étudiant`. De plus, la classe `Étudiant` possède des propriétés qui ne sont pas partagées par la classe `Personne`. Par exemple, un `Étudiant` a une moyenne semestrielle, ce qui n'est pas le cas d'une `Personne` en général.

La propriété d'héritage est transitive. Par exemple, si je définis une nouvelle classe `Diplômés` comme sous-classe de `Étudiant`, la classe `Diplômés` doit aussi être une `Personne`. Elle doit l'être selon la règle qui veut que si un `Diplômé` IS_A ("est un") `Étudiant` et qu'un `Étudiant` IS_A ("est une") `Personne`, il en découle *ipso facto* que `Diplômé` IS_A ("est une") `Personne`. Socrate, Socrate...

Comment une classe fait-elle pour hériter ?

Et revoilà l'exemple de la classe `GraduateStudent` ! Nous allons la remplir avec quelques membres :

```
//  InheritanceExample - démontre une relation d'héritage
//                       dans laquelle le constructeur de la
//                       sous-classe passe des arguments au
//                       constructeur de la classe de base
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>
using namespace std;

// Advisor - classe vide
class Advisor {};

const int MAXNAMESIZE = 40;
class Student
{
  public:
    Student(char *pName = "no name")
       : average(0.0), semesterHours(0)
    {
        strncpy(name, pName, MAXNAMESIZE);
        name[MAXNAMESIZE - 1] = '\0';
        cout << "Construction de l'objet Student "
             << name
             << endl;
    }

    void addCourse(int hours, float grade)
    {
        cout << "Ajoute un module pour " << name << endl;
        average = (semesterHours * average + grade);
        semesterHours += hours;
        average = average / semesterHours;
    }

    int hours( ) { return semesterHours;}
    float gpa( ) { return average;}

  protected:
    char name[MAXNAMESIZE];
```

```
    int  semesterHours;
    float average;
};

class GraduateStudent : public Student
{
  public:
    GraduateStudent(char *pName, Advisor& adv, float qG = 0.0)
         : Student(pName), advisor(adv), qualifierGrade(qG)
    {
        cout << "Construction de l'objet GraduateStudent "
             << pName
             << endl;
    }

    float qualifier( ) { return qualifierGrade; }

  protected:
    Advisor advisor;
    float qualifierGrade;
};

int main(int nNumberofArgs, char* pszArgs[])
{
    Advisor advisor;

    // création de deux types d'étudiants
    Student llu("Cy N Sense");
    GraduateStudent gs("Matt Madox", advisor, 1.5);

    // ajoute un module à leur moyenne générale
    llu.addCourse(3, 2.5);
    gs.addCourse(3, 3.0);

    // affiche qualificateur de Matt
    cout << "Qualificateur de Matt = "
         << gs.qualifier()
         << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Ce programme crée et utilise deux objets, l'un de la classe `Student` et l'autre de la sous-classe `GraduateStudent`. Il affiche la sortie montrée ci-dessous :

```
Construction de l'objet Student Cy N Sense
Construction de l'objet Student Matt Madox
Construction de l'objet GraduateStudent Matt Madox
Ajoute un module pour Cy N Sense
Ajoute un module pour Matt Madox
Qualificateur de Matt = 1.5
Appuyez sur une touche pour continuer...
```

Utiliser une sous-classe

La classe `Student` a été définie de façon tout à fait conventionnelle. En revanche, `GraduateStudent` est un peu différente ; sur la première ligne, le deux-points suivi de `public Student` déclare `GraduateStudent` comme étant une sous-classe de `Student`.

L'apparition du mot-clé `public` implique qu'il y a là sans doute un héritage protégé. Certes. Mais les héritages *protégés* sont hors du propos de ce livre.

Les programmeurs adorent inventer de nouveaux mots, ou détourner le sens de termes existants. Pire. Ils poussent même parfois le vice jusqu'à créer de nouveaux mots auxquels ils donnent une seconde signification. Les phrases qui suivent décrivent en réalité la même relation :

- `GraduateStudent` est une sous-classe de `Student`.
- `Student` est la classe de base (ou classe parent) de `GraduateStudent`.
- `GraduateStudent` hérite de `Student`.
- `GraduateStudent` étend `Student`.

En tant que sous-classe de `Student`, `GraduateStudent` hérite de tous ses membres. Par exemple, un `GraduateStudent` possède un nom (`name`) puisque ce membre est défini dans la classe de base. Mais une sous-classe peut aussi ajouter ses propres membres, par exemple `qualifierGrade`. Car après tout, et de façon presque littérale, `gs` IS_A `Student` et même un peu plus que cela.

La fonction `main()` déclare deux objets. L'un, `llu` est de type `Student`. L'autre, `gs`, est de type `GraduateStudent`. Elle accède ensuite à la fonction membre `addCourse()` pour les deux types d'étudiants et enfin à la fonction `qualifier` qui est un membre spécifique à la sous-classe.

Construire une sous-classe

Bien qu'une sous-classe ait accès aux membres protégés de la classe de base et est à même de les initialiser, elle est totalement responsable de sa propre initialisation.

Avant que le contrôle ne passe à l'accolade ouvrante du constructeur de `GraduateStudent`, le constructeur par défaut de `Student` est appelé (aucun autre constructeur n'ayant été indiqué). Si `Student` était l'héritier d'une autre classe (telle que `Person`), le constructeur de celle-ci serait appelé avant que le constructeur de `Student` ne prenne le contrôle. À la manière d'un immeuble, l'objet est construit en commençant par la fondation (la classe de base), puis en montant chaque étage de la structure (de classe, s'entend) l'un après l'autre.

Comme dans le cas des objets membres, vous avez souvent besoin de passer des arguments au constructeur de la classe de base. L'exemple déclare le constructeur de la sous-classe de la façon suivante :

```
GraduateStudent(char *pName, Advisor& adv, float qG = 0.0)
         : Student(pName), advisor(adv), qualifierGrade(qG)
{
    // le code de construction se trouve ici
}
```

Ici, le constructeur de `GraduateStudent` appelle le constructeur de `Student` en lui passant l'argument `pName`. Le C++ initialise alors les membres `advisor` et `qualifierGrade` avant d'exécuter les instructions qui se trouvent entre les accolades ouvrante et fermante du constructeur.

Le constructeur par défaut de la classe de base est exécuté dans le cas où la sous-classe ne fait pas explicitement référence à un constructeur différent. Dans le petit exemple qui suit, la classe de base `Pig` est construite avant tout autre membre de `LittlePig`, bien que cette dernière n'appelle pas explicitement ce constructeur :

```
class Pig
{
  public:
    Pig() : pHouse(null)
    {}
  protected:
    House* pHouse;
};
class LittlePig : public Pig
```

```
{
  public:
    LittlePig(float volStraw, int numSticks, int numBricks)
      : straw(volStraw), sticks(numSticks), bricks(numBricks)
    { }

  protected:
    float straw;
    int sticks;
    int bricks;
};
```

Il ne reste plus qu'à trouver le grand méchant loup...

Dans cet esprit, le constructeur de copie d'une classe de base est lui aussi automatiquement invoqué.

Détruire une sous-classe

D'après la règle selon laquelle les destructeurs sont appelés dans l'ordre inverse des constructeurs, le destructeur de `GraduateStudent` reçoit le contrôle en premier. Une fois qu'il a fait sa sale besogne, le contrôle passe au destructeur de `Advisor` puis à celui de `Student`. Si `Student` avait été basé sur une classe "supérieure" `Person`, le destructeur de `Person` aurait ensuite pris le contrôle. Ainsi va le monde.

Tout cela est logique. Une parcelle de mémoire est d'abord convertie en objet `Student`. Ce n'est qu'après cela que le constructeur de `GraduateStudent` transforme ce modeste étudiant en `GraduateStudent`. Le destructeur inverse simplement le processus.

Obtenir une relation HAS_A

Remarquez que la classe `GraduateStudent` comprend des membres des classes `Student` et `Advisor`, mais d'une façon différente. En définissant un membre de données de la classe `Advisor`, vous savez que `Student` a en lui tous les membres de données de `Advisor` ; mais il ne vous est toutefois pas possible d'affirmer qu'un étudiant de troisième cycle est un "conseiller éducatif" ou un "tuteur" (*advisor*). Vous diriez plutôt, en jargon anglo-informatico-C++, qu'un `GraduateStudent` HAS_A ("a un") `Advisor`. Où est la différence avec un héritage ?

Prenons l'exemple d'une voiture. Vous pouvez en toute logique affirmer qu'une voiture appartient à la sous-classe des véhicules. Elle hérite donc des propriétés de cette catégorie. Par ailleurs, une voiture a un moteur. Quand vous achetez une automobile, vous pouvez en déduire sans trop vous tromper que vous achetez par la même occasion un moteur (à moins que vous n'alliez à la casse où j'ai trouvé mon dernier tacot).

Si vous êtes invité à un rallye et que vous arrivez dans votre petit bijou, vos amis n'auront pas de raison de se plaindre (même si quelqu'un d'autre est venu sur une bicyclette) car une voiture IS_A ("est un") véhicule. Mais si vous venez à pied en poussant une brouette avec un moteur dedans, vos amis auront de bonnes raisons de se moquer de vous car un moteur n'est *pas* un véhicule. Un moteur ne possède pas certaines des propriétés indispensables à tout véhicule qui se respecte, comme par exemple une pendule électronique (évidemment hors d'usage).

D'un point de vue informatique, la relation HAS_A est toute aussi simple. Voyez ce listing :

```
class Vehicule {};
class Moteur {};
class Voiture : public Vehicule
{
  public:
    Moteur moteur;
};

void VehiculeFn(Vehicule& v);
void MoteurFn(Moteur& m);

int main(int nNumberofArgs, char* pszArgs[])
{
    Voiture voiture;
    VehiculeFn(voiture);     // ceci est autorisé
    MoteurFn(voiture);       // ceci ne l'est pas
    MoteurFn(voiture.moteur);// c'est autorisé aussi
    return 0;
}
```

L'appel `VehiculeFn(voiture)` est admis car `voiture` IS_A ("est un") `Vehicule`. L'appel `MoteurFn(voiture)` ne l'est pas car `voiture` n'est pas un moteur, bien qu'il contienne un `Moteur`. Si vous voulez passer la partie `moteur` de `voiture` à la fonction, il faut l'exprimer explicitement, comme dans l'appel `MoteurFn(voiture.moteur)`.

Chapitre 21

Les fonctions de membre virtuelles sont-elles réelles ?

Dans ce chapitre :

- Découvrir le fonctionnement du polymorphisme.
- Pourquoi les nachos polymorphes sont-ils plus sûrs ?
- Redéfinir les fonctions membres dans une sous-classe.
- Considérations particulières sur le polymorphisme.

Le nombre et le type des arguments d'une fonction se trouvent dans son nom complet, ou *étendu*. Il est ainsi possible de donner le même nom à deux fonctions à condition que le nom étendu soit différent, comme dans :

```
void uneFn(int);
void uneFn(char*);
void uneFn(char*, double);
```

Dans les trois cas, le nom court de ces fonctions est `uneFn()`. En revanche, les noms étendus sont différents : on a `uneFn(int)`, qui n'est pas pareille que `uneFn(char*)`, et ainsi de suite. Le C++ est parfaitement capable de retrouver la bonne fonction uniquement à partir des arguments qui lui sont passés lors de l'appel.

Comme le type de retour ne fait pas partie du nom étendu, vous ne pouvez pas avoir deux fonctions portant le même nom et qui ne diffèrent que par le type d'objet renvoyé.

Les fonctions membres peuvent être surchargées. Le nombre et le type des arguments ne sont pas les seuls éléments d'un nom étendu ; le nom de la classe en fait aussi partie.

Mais l'héritage suscite de nouvelles interrogations. Exemple : que se passe-t-il si, dans une classe de base, une fonction porte le même nom qu'une fonction de la sous-classe ? Voyons cela de plus près :

```
class Student
{
 public:
  float calcTuition();
};

class GraduateStudent : public Student
{
 public:
  float calcTuition();
};

int main(int argcs, char* pArgs[])
{
  Student s;
  GraduateStudent gs;
  s.calcTuition();    // appelle Student::calcTuition()
  gs.calcTuition();   // appelle GraduateStudent::calcTuition()
  return 0;
}
```

Comme pour toutes les situations de surcharge, lorsque le programmeur se réfère à `calcTuition()` (un simple cours de calcul), le C++ doit déterminer quelle est la fonction visée. De toute évidence, si les deux fonctions diffèrent au niveau de leur type d'arguments, il n'y a pas de problème. Même si les arguments étaient identiques, le nom de classe devrait être suffisant pour résoudre l'appel, et c'est ce qui se passe dans cet exemple. L'appel `s.calcTuition()` se réfère à `Student::calcTuition()` car `s` est déclaré localement comme étant `Student`, alors que `gs.calcTuition()` ne peut concerner que `GraduateStudent::calcTuition`.

Mais qu'en serait-il si la classe exacte de l'objet ne pouvait pas être déterminée au moment de la compilation ? Pour montrer comment ceci peut se produire, nous allons quelque peu modifier le programme précédent :

```
//  OverloadOverride - étude de surcharge de fonctions
//                     au moment de la déclaration et
//                     lors de la compilation
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    float calcTuition()
    {
        cout << "Nous sommes dans Student::calcTuition" << endl;
        return 0;
    }
};

class GraduateStudent : public Student
{
  public:
    float calcTuition()
    {
        cout << "Nous sommes dans GraduateStudent::calcTuition"
             << endl;
        return 0;
    }
};

void fn(Student& x)
{
    x.calcTuition();   // à quelle fonction calcTuition()
                       // fait-on référence ?
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // passe un objet de la classe de base à la fonction
    // (pour vérifier la déclaration)
    Student s;
    fn(s);
```

```
    // passe maintenant une version spécialisée
    // de la classe de base
    GraduateStudent gs;
    fn(gs);

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Vous obtenez alors la sortie suivante :

```
Nous sommes dans Student::calcTuition
Nous sommes dans Student::calcTuition
Appuyez sur une touche pour continuer...
```

Au lieu d'invoquer `calcTuition()` directement, l'appel est maintenant effectué au travers d'une fonction intermédiaire, `fn()`. Selon la manière dont `fn()` est appelée, `x` peut être un objet `Student` ou `GraduateStudent`.

Un `GraduateStudent` IS_A `Student`, comme vous l'avez appris à la fin du chapitre précédent (relisez-le si vous avez déjà tout oublié).

Ici, l'argument `x` passé à `fn()` est déclaré comme étant une référence à `Student`.

Le passage par référence est nettement plus efficace que le passage par valeur. Reportez-vous au Chapitre 18 pour réviser la copie des objets.

Nous voudrions que `x.calcTuition` appelle `Student::calcTuition()` lorsque `x` est un `Student`, mais qu'il appelle `GraduateStudent::calcTuition()` si `x` est un objet `GraduateStudent`. Ce serait tout à fait sympathique si C++ avait l'intelligence de s'en charger.

Normalement, le compilateur décide à quelle fonction se réfère un appel au moment de la compilation. Quand vous cliquez sur le bouton qui invite le compilateur C++ à reconstruire votre programme exécutable, celui-ci doit fureter dans le code source pour déterminer, lors de chaque appel et selon les arguments utilisés, la fonction à laquelle vous pensiez.

Dans le cas de figure décrit ici, le type d'argument déclaré pour `fn()` n'est pas suffisamment descriptif. Bien que cet argument soit déclaré `Student`, il pourrait tout aussi bien s'agir d'un objet `GraduateStudent`. Aucune décision ne peut être prise tant que vous n'exécutez pas véritablement le programme. Ce n'est que

quand la fonction est appelée pour de bon que le C++ peut étudier son type d'argument et décider s'il s'agit d'un `Student` pur et dur ou d'un `GraduateStudent`.

Le type que vous avez rencontré jusqu'à présent est appelé "type de déclaration" ou "type à la compilation". Le type déclaré de `x` est `Student` dans les deux cas car c'est ce qu'indique la déclaration effectuée dans `fn()`. L'autre type, que vous pourriez appeler "type véritable", est le type dynamique (celui qui est produit lors de l'exécution). Dans notre exemple, le type dynamique de `x` est `Student` lorsque `fn()` est appelé avec `s` et `GraduateStudent` lorsque `fn()` est appelé avec `gs`. Plus on est de fous, plus on s'amuse.

La capacité de décider, lors de l'exécution, laquelle, parmi plusieurs fonctions membres surchargées, doit être appelée selon le type dynamique se nomme *polymorphisme,* du grec *poly* (plusieurs), *morphe* (forme) et *isme* (terme mis à toutes les sauces, comme "machinisme", "moi-moi-isme", etc.) On parle aussi de *liaison dynamique.* Le C++ supporte le polymorphisme, ce qui n'est pas surprenant (sinon, ce sujet ne serait même pas évoqué dans ce livre). Le fait de décider, au moment de la compilation, quelle fonction membre surchargée doit être appelée est dit *liaison statique,* par opposition aux liaisons dynamiques.

La surcharge polymorphique d'une fonction de classe de base est appelée *redéfinition.* Ce nouveau nom est utilisé pour distinguer d'une surcharge normale ce cas de figure plus complexe.

Pourquoi vous avez besoin du polymorphisme

C'est le polymorphisme qui donne toute sa puissance à la programmation orientée objet. Il y joue un si grand rôle qu'un langage qui ne supporterait pas le polymorphisme ne pourrait pas être considéré comme "orienté objet" (c'est un peu comme les labels de normalisation, du genre AFNOR ou ISO, qui sont décernés par des instances accréditées).

Les langages qui supportent les classes mais pas le polymorphisme sont appelés "langages basés objet". C'est le cas du langage Ada. (N.D.T. : langage développé à la fin des années 1970 par le Français Jean Ichbiah pour le département de la Défense américain. Ada était le prénom d'Augusta Ada Byron qui avait programmé la célèbre machine analytique, un ordinateur mécanique inventé au milieu du XIX^{e} siècle par Charles Babbage.)

Sans le polymorphisme, l'héritage serait peu de chose. Permettez-moi de vous proposer un autre exemple qui illustre ceci. Supposons que j'aie écrit un programme qui ait rencontré un gros succès, intitulé – on se demande bien pourquoi – "Student". Après de longs mois de conception, de programmation

et de tests, les collègues et la critique l'encensent (il est même question de créer un prix Nobel de programmation spécialement pour moi, mais je tairai modestement cette rumeur).

Quelque temps après, mon directeur me demande d'ajouter à ce programme la possibilité de gérer des étudiants de troisième cycle qui sont semblables, mais pas identiques, à des étudiants normaux (ils clameront d'ailleurs sans doute qu'ils ne sont pas semblables du tout). Bref, le directeur ne sait rien du niveau de profondeur du programme où la fonction `someFunction()` appelle `CalcTuition()`. D'ailleurs, cela ne l'intéresse pas. Il y a d'ailleurs bien des choses qu'il ne connaît pas. Il vaut mieux me poser directement les questions.

La fonction `someFunction()` appelle la fonction membre `calcTuition()` de la façon suivante :

```
void someFunction(Student& s)
 {
  // ...on passe par là...
  s.calcTuition();
  // ...et on continue ici...
}
```

Si le C++ ne supportait pas les liaisons dynamiques, il me faudrait éditer `someFunction()` comme suit (par exemple) pour ajouter une classe `GraduateStudent` :

```
#define STUDENT 1
#define GRADUATESTUDENT 2
void someFunction(Student& s)
{
  // ...on passe par là...
  // ajout d'un type de membre qui indique
  // le véritable type de l'objet
  switch (s.type)
  {
    case STUDENT:
      s.Student::calcTuition();
      break;

    case GRADUATESTUDENT:
     s.GraduateStudent::calcTuition();
     break;
  }
  // ...et on continue ici...
}
```

Je devrais aussi ajouter la variable `type` à la classe, de même que l'affectation `type = STUDENT` au constructeur de `Student` et l'affectation `type = GRADUATES-TUDENT` au constructeur de `GraduateStudent`. La valeur de `type` indiquerait alors le type dynamique de `s`. Il me faudrait ensuite recopier le test qui figure dans le petit bout de code ci-dessus à chaque emplacement où une fonction membre redéfinie est appelée.

Tout cela n'a pas l'air si mal, sauf sur trois points. Le premier, c'est qu'il n'y a ici qu'une seule fonction. Supposons que `calcTuition()` soit appelé depuis de multiples endroits et que cette fonction ne soit pas la seule différence entre les deux classes. Il y a peu de chances que vous trouviez tous les emplacements à modifier. Du moins avant la prochaine année scolaire.

Deuxièmement, je dois éditer (lisez "interrompre") le code qui a été débogué et le reprendre, au risque d'introduire des éléments qui fonctionneront mal. L'édition peut s'avérer longue et ennuyeuse, ce qui a généralement pour effet de faire baisser mon attention. Chacune de mes modifications pourrait alors contenir des erreurs ou ne pas coller avec le code existant. Qui sait ?

Enfin, après avoir fini de tout (ré)éditer, (re)déboguer et (re)tester, je me (re)trouve avec deux versions à suivre (à moins de pouvoir laisser tomber la version originale). Ceci suppose l'édition de deux sources si des bogues se manifestent (au secours !) et une sorte de système de pointage pour vérifier si les corrections ont bien été faites dans les deux versions.

Et que se passerait-il si mon directeur me demandait d'ajouter encore une autre classe (il adore ce genre de choses) ? Il me faudra non seulement répéter le processus, mais aussi travailler sur trois copies.

Grâce au polymorphisme, il y a de bonnes chances pour que le travail se limite à l'ajout d'une nouvelle sous-classe, suivi d'une recompilation. J'aurai peut-être à modifier la classe de base elle-même, mais du moins un seul endroit du programme sera concerné. Les modifications du code de l'application seront largement minimisées.

D'un point de vue plus philosophique, le polymorphisme s'impose pour une raison encore plus importante. Vous vous rappelez comment j'avais mis les nachos au four ? En procédant ainsi, j'agissais en tant que liaison dynamique. La recette dit : "Faites chauffer les nachos au four". Elle ne dit pas "Si le four est du type à micro-ondes, faites ceci, s'il est de type classique, faites cela, s'il est du type à convection, faites autrement". La recette (le code) me fait confiance à moi (la liaison dynamique) pour décider ce qu'est l'action de chauffer (la fonction membre) lorsqu'elle est appliquée au four (une instance particulière de la classe `Four`) ou par l'une quelconque de ses variantes (sous-classes), en l'occurrence le four à micro-ondes (`FourMicroOndes`). C'est ainsi que les gens pensent. Et un langage informatique adapté au mode de pensée du commun

des mortels permet au modèle de programmation de décrire le monde réel avec une plus grande exactitude.

Comment le polymorphisme fonctionne-t-il ?

Tout langage informatique est susceptible de supporter les liaisons dynamiques ou statiques. Les ancêtres, comme le C, ne connaissaient que les liaisons statiques. Certains langages plus récents, tels que Java, ne supportent que les liaisons dynamiques. Le C++ joue les traits d'union et supporte les deux formes.

Mais vous serez peut-être étonné d'apprendre que l'option par défaut du C++ est la liaison statique. Les raisons en sont simples, même si elles datent un peu. Tout d'abord, C++ s'efforce comme d'habitude de conserver une compatibilité ascendante avec le C. D'autre part, le polymorphisme ajoute une petite surcharge à tous les appels de fonction, que ce soit en termes de stockage de données ou de code nécessaire pour exécuter ces appels. Les créateurs du C++ savaient bien que toute surcharge supplémentaire par rapport au C aurait été prise comme prétexte pour ne pas l'adopter comme langage de prédilection. C'est pourquoi ils ont choisi par défaut la liaison statique, plus efficace.

Voyons une dernière raison. Il peut être utile pour la personne qui a programmé une certaine classe de décider si telle ou telle fonction membre pourra être redéfinie dans un futur plus ou moins proche. Cet argument est suffisamment fort pour que le nouveau langage C# de Microsoft permette d'ajouter un drapeau aux fonctions afin de déclarer qu'elles ne sont pas redéfinissables (mais elles le restent par défaut).

Pour qu'une fonction membre devienne polymorphe, le programmeur doit y placer un drapeau qui prend la forme du mot-clé `virtual`. Par exemple :

```
class Student
{
  public:
    virtual float calcTuition()
    {
        cout << "Nous sommes dans Student::calcTuition" << endl;
        return 0;
    }
};
```

C'est le mot-clé `virtual` qui indique au C++ que `calcTuition()` est une fonction membre polymorphe. Autrement dit, déclarer que `calcTuition()` est virtuel signifie que tout appel se fera dynamiquement s'il existe un doute quelconque quant au type dynamique de l'objet qui l'appelle au moment de l'exécution.

Pour préciser cela, observez l'exemple suivant :

```
// PolyTest - exemple de liaison dynamique
#include <cstdio>
#include <iostream>
using namespace std;

class Base
{
  public:
   virtual void fn()
   {
      cout << "Dans la classe Base" << endl;
   }
};
class SubClass : public Base
{
  public:
   virtual void fn()
   {
      cout << "Dans la classe SubClass" << endl;
   }
};

void test(Base &b)
{
   b.fn();          // la liaison est dynamique
}

int main()
{
   Base bc;
   SubClass sc;
   cout << "Appelle test(bc)\n";
   test(bc);
   cout << "Appelle test(sc)\n";
   test(sc);

   // attend pour terminer le programme que l'utilisateur
   // lise le contenu de la fenêtre puis appuie sur une touche
```

```
    system("PAUSE");
    return 0;
}
```

Ici, `fn()` est appelé par la fonction intermédiaire `test()`. Lorsqu'un objet de classe `Base` est passé à `test()`, `b.fn()` appelle `Base::fn()`. Mais lorsqu'un objet de la classe `SubClass` est passé à `test()`, le même appel est dirigé vers `SubClass::fn()`.

L'exécution du programme génère la sortie suivante :

```
Appelle test(bc)
Dans la classe Base
Appelle test(sc)
Dans la classe SubClass
Appuyez sur une touche pour continuer...
```

Si vous êtes à l'aise avec le débogueur de votre environnement C++, vous avez vraiment intérêt à exécuter le programme précédent pas à pas.

Une fonction virtuelle n'a à être déclarée que dans la classe de base. La "virtualité" est en effet automatiquement convoyée vers les sous-classes. Dans ce livre toutefois, j'ai respecté la programmation standard en déclarant (virtuellement) partout la fonction comme étant virtuelle.

Vous voulez un autre exemple de polymorphisme ? Téléchargez les programmes du livre. Vous y trouverez notre célèbre `PolymorphicNachos`, la recette des nachos polymorphes (éventuellement comestibles, mais ce n'est pas garanti).

Quand est-ce qu'une fonction n'est pas virtuelle ?

Le fait que vous pensiez qu'un appel de fonction particulier est une liaison dynamique ne signifie aucunement que c'est vraiment le cas.

Ce qu'il faut surveiller de près, c'est le fait que les fonctions membres en question sont déclarées de façon identique, y compris leur type de retour. Si elles ne sont pas déclarées avec les mêmes arguments dans les sous-classes, les fonctions membres ne sont pas redéfinies de façon polymorphe, qu'elles aient été ou non marquées comme virtuelles.

Il y a une exception à cette règle. Si la fonction membre dans la classe de base retourne un pointeur ou une référence à un objet de cette même classe, une fonction membre redéfinie dans une sous-classe a le droit de renvoyer à son tour un pointeur ou une référence à un objet de la dite sous-classe. En d'autres termes, la fonction `makeACopy()` dans le code ci-dessous est polymorphe bien que les deux "versions" (celle de la classe de base et celle de sa sous-classe) renvoient un type différent :

```
class Base
{
  public:
    // retourne une copie de l'objet courant
    Base* makeACopy()
    {
      // ...fait ce qu'il faut pour créer une copie
    }
};

class SubClass : public Base
{
  public:
    // retourne une copie de l'objet courant
    SubClass* makeACopy()
    {
        // ... fait ce qu'il faut pour créer une copie
    };
};

void fn(Base& bc)
{
    BaseClass* pCopy = bc.makeACopy();

    // on continue...
}
```

En pratique, c'est tout à fait naturel. Une fonction `makeACopy()` devrait retourner un objet de type `SubClass` même s'il redéfinit `BaseClass::makeACopy()`.

À propos du virtuel

Quelques petites choses doivent rester présentes à l'esprit lorsque l'on utilise des fonctions virtuelles.

En premier lieu, les fonctions membres statiques ne peuvent pas être déclarées comme étant virtuelles. En effet, elles ne sont pas appelées avec un objet et, par voie de conséquence, il n'existe pas d'objet dynamique susceptible de motiver une décision quant au type de liaison.

Deuxièmement, la spécification d'un nom de classe dans l'appel force celui-ci à se lier statiquement, que la fonction soit virtuelle ou non. Par exemple, l'appel ci-dessous s'effectue vers `Base::fn()` car c'est ce que le programmeur a indiqué, et cela même si `fn()` est déclaré comme étant virtuel :

```
void test(Base& b)
{
  b.base::fn();  // cet appel n'est pas lié dynamiquement
}
```

Enfin, les constructeurs ne peuvent pas être virtuels parce qu'il n'existe pas d'objet (fini) pouvant être utilisé pour déterminer le type. Au moment où un constructeur est appelé, la mémoire occupée par l'objet n'est qu'une masse informe. C'est seulement après que le constructeur a fini sa tâche que l'objet est devenu un membre de la classe en bonne et due forme.

Le destructeur, lui, devrait pratiquement toujours être déclaré comme virtuel. Sinon, vous courez le risque de détruire improprement l'objet, comme dans ce cas de figure :

```
class Base
{
 public:
   ~Base();
};

class SubClass : public Base
{
 public:
   ~SubClass();
};

void finishWithObject(Base* pHeapObject)
{
  // ...travail avec l'objet...
  // le retourne maintenant dans le heap
  delete pHeapObject;  // appelle ~Base() quel que
 }                     // soit le type dynamique de
                       // pHeapObject
```

Si le pointeur passé à `finishWithObject()` pointe réellement sur une sous-classe, le destructeur `SubClass` n'est pas appelé correctement. En effet, puisqu'il n'a pas été déclaré comme étant virtuel, il est toujours lié statiquement. Déclarer le destructeur comme virtuel résout ce problème.

Et si vous ne voulez pas déclarer le destructeur comme virtuel ? Il n'y a qu'un cas où c'est possible. Les fonctions virtuelles présentent une légère surcharge. Permettez-moi d'être plus précis. Quand le programmeur définit la première fonction virtuelle dans une classe, le C++ ajoute un pointeur supplémentaire caché (pas un pointeur par fonction, juste un seul, que la classe contienne une ou plusieurs fonctions virtuelles). Une classe dépourvue de fonctions virtuelles (et qui n'en hérite pas à partir de la classe de base) ne reçoit pas ce pointeur.

Un pointeur, cela n'à l'air de rien, et ce n'est effectivement pas grand chose sauf si ces deux conditions sont vraies :

- La classe n'a pas beaucoup de membres de données (de sorte qu'un pointeur n'est pas négligeable comparé à ce qui s'y trouve).
- Vous avez l'intention de créer beaucoup d'objets de cette classe (sinon, la surcharge n'aurait pas de raison d'être).

Si ces deux conditions sont réunies et si la classe ne contient encore aucune fonction membre virtuelle, vous pouvez ne pas déclarer le destructeur comme étant virtuel.

À l'exception de ce cas précis, vous devez toujours déclarer les destructeurs comme étant virtuels, même si une classe n'est pas (jusqu'à présent) sous-classée. Vous ne savez jamais si quelqu'un n'a pas l'intention d'utiliser un jour votre classe comme classe de base pour un de ses projets. Si un destructeur n'est pas déclaré comme virtuel, signalez-le dans un commentaire !

Chapitre 22

Les classes dérivées

Dans ce chapitre :

- Dériver les propriétés communes dans une classe de base.
- Utiliser des classes abstraites pour y stocker les informations de dérivation.
- Déclarer des classes abstraites.
- Hériter d'une classe abstraite.
- Programme, modules et fichier de projet.

Le concept d'héritage permet à une classe d'hériter des propriétés d'une classe de base. L'héritage sert à beaucoup de choses, notamment à payer les frais de scolarité de mon fils. Il peut aussi réduire le temps de programmation en évitant les répétitions de code inutiles. L'héritage permet à un programme de réutiliser les classes existantes dans d'autres applications grâce à la redéfinition des fonctions.

Le principal avantage de l'héritage tient à sa capacité de mettre en évidence les relations entre les classes. Elles s'effectuent au travers de la relation IS_A : un `FourAMicroOndes` IS_A ("est un") `Four`, et ainsi de suite.

La dérivation est fabuleuse pour peu que vous trouviez les bonnes corrélations. Par exemple, la relation entre un four à micro-ondes et un four conventionnel semble aller de soi. En revanche, affirmer que le four à micro-ondes est une variété particulière de grille-pain risque d'attirer quelques remarques concernant votre santé mentale. Il est vrai que tous deux chauffent des aliments, qu'ils fonctionnent à l'électricité et qu'on les trouve généralement dans une cuisine, mais les similitudes s'arrêtent là. Le fait est qu'un four à micro-ondes ne peut pas griller du pain (j'ai essayé, et ce n'est pas bon).

L'identification des classes inhérentes à la résolution d'un problème et la mise en évidence des relations entre ces classes est un processus appelé *dérivation* (voilà qui rappelle les cours de maths au lycée).

La dérivation

Dans cette section, nous allons voir comment vous pouvez utiliser l'héritage pour simplifier vos programmes. Nous partirons pour cela d'un exemple simplifié de compte en banque.

Supposons que l'on vous demande d'écrire une application bancaire qui implémente deux types de comptes : compte chèques courant (que nous appellerons *checking*) et compte épargne (*savings*).

Pour épargner la vie de quelques arbres, l'application complète n'est pas reproduite ici. Vous la trouverez donc en téléchargement avec les autres exemples de ce livre.

Je pourrais parler de ces classes jusqu'à en devenir tout bleu. Mais les programmeurs orientés objet se doivent d'être concis dans la description des points essentiels d'une classe. C'est pourquoi ils se servent de représentations graphiques (peut-être aussi parce qu'ils n'aiment pas beaucoup écrire). La Figure 22.1 montre comment on pourrait organiser les deux classes `Checking` et `Savings`. Bien sûr, il existe d'autres méthodes pour schématiser ce type de problème, mais celle-ci me convient parfaitement.

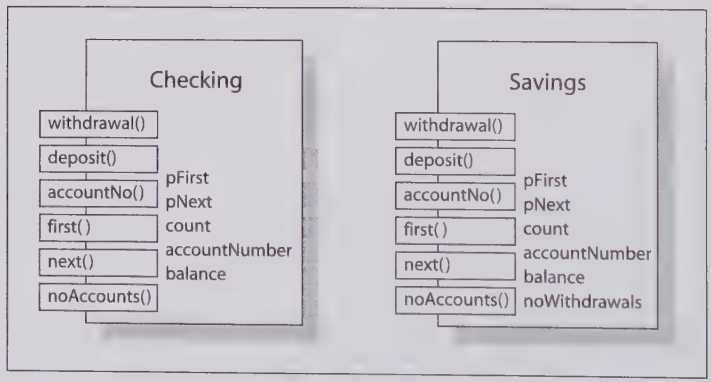


Figure 22.1
Les classes indépendantes Checking et Savings.

Voici quelques informations sur le contenu des figures et la manière de les lire :

- Les grands cadres qui projettent une ombre portée sont des classes dont le nom est indiqué en haut.
- Les noms qui se trouvent dans les petites boîtes sont des fonctions membres.

- Les noms qui ne sont pas dans des boîtes sont des membres de données.
- Les noms qui dépassent partiellement des cadres sont accessibles publiquement ; ils sont donc visibles par des fonctions qui ne font pas partie de la classe ou de ses descendants. Les membres qui se trouvent entièrement dans le grand cadre ne sont pas accessibles à l'extérieur de la classe.
- Une flèche épaisse représente la relation IS_A.
- Une flèche mince représente la relation HAS_A.

Une `Voiture` `IS_A` `Véhicule` et une Voiture `HAS_A` `Moteur`.

Vous remarquez dans la Figure 22.1 que les classes `Checking` et `Savings` ont beaucoup de choses en commun. Par exemple, les deux ont des fonctions membres pour les retraits (`withdrawal()`) et les dépôts (`deposit()`). Mais, comme ces deux classes ne sont pas identiques, elles doivent rester distinctes (dans une véritable application bancaire, elles seraient très différentes de la représentation donnée ici). Il devrait toutefois exister un moyen d'éviter ces répétitions.

Nous pourrions faire en sorte qu'une de ces classes hérite de l'autre. Comme `Savings` possède plus de membres que `Checking`, ce serait donc `Savings` qui hériterait de `Checking`. Cet arrangement est illustré sur la Figure 22.2, où `Savings` hérite de tous les membres. La classe est complétée par l'ajout du membre de données `noWithdrawals` et par la redéfinition de la fonction `withdrawal()`. Cette redéfinition est indispensable puisque que les règles du retrait d'argent sur un compte épargne sont différentes de celles du compte chèques (elles ne s'appliquent pas à moi parce que je n'ai plus rien à retirer).

Laisser `Savings` hériter de `Checking` allège le travail, mais cette solution n'est pas totalement satisfaisante. Le problème principal est que, à l'instar de la photo agrafée sur mon permis de conduire, cette solution ne représente pas la vérité. Cette relation d'héritage impliquerait qu'un compte épargne n'est qu'un type particulier de compte chèques, ce qui n'est pas le cas.

"Oui et alors ?", rétorquerez-vous, "L'héritage fonctionne et il nous évite des efforts". C'est vrai, mais mes réserves sont plus que de simples clauses de style. Cette représentation erronée sème en effet la confusion au niveau de la programmation, pour aujourd'hui comme pour demain. Un prochain jour, un programmeur, peu familiarisé avec les subtilités de votre programme, devra le lire et comprendre ce qu'il fait. Les mauvaises représentations sont difficiles à comprendre et à corriger.

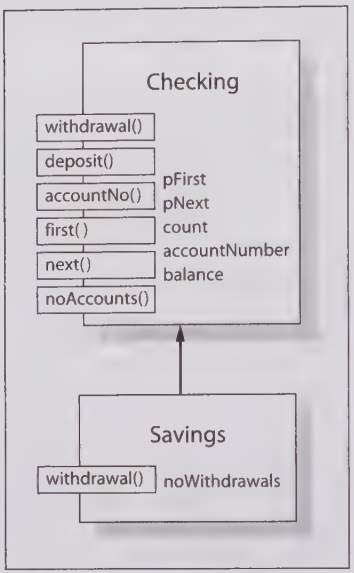


Figure 22.2
Savings a été implémenté comme sous-classe de Checking.

De plus, des représentations fausses ou approximatives peuvent engendrer des problèmes qui n'apparaîtront que par la suite. Supposons par exemple que la banque change les règles concernant les comptes chèques. Imaginons qu'elle ait l'idée (invraisemblable) de facturer les opérations si, au cours du mois, le solde courant tombe en dessous d'une certaine valeur.

Ce genre de modification peut facilement être géré en modifiant légèrement la classe `Checking`. Il suffirait d'ajouter un nouveau membre de données afin d'effectuer le suivi du solde minimal au cours du mois. Allons-y et appelons ce membre `minimumBalance`.

Mais nous avons maintenant un problème : comme `Savings` hérite de `Checking`, elle récupère du même coup ce nouveau membre de données, ce qui ne lui servira à rien puisque le solde minimum n'affecte en rien le compte d'épargne. Il se contente être là. Rappelez-vous que chaque objet de type `Checking` reçoit ce membre `minimumBalance`. Rien de bien gênant en soi, mais cela ajoute à la confusion.

De telles modifications finissent par s'accumuler. Aujourd'hui, c'était un membre de données supplémentaire, demain, ce sera une fonction membre à modifier ou à ajouter. Peu à peu, la classe `Savings` se remplira inutilement d'une foule de bagages supplémentaires qui n'ont d'utilité que pour les comptes chèques.

Prise de frénésie, la banque décide maintenant de modifier certaines règles de gestion des comptes épargne. Pour ce faire, nous devons modifier une certaine fonction dans `Checking`. De tels changements dans la classe de base se propagent aussitôt vers les sous-classes, à moins que la fonction ne soit déjà redéfinie dans la sous-classe `Savings`. Supposons que la banque décide d'offrir un grille-pain pour chaque dépôt effectué sur un compte chèques (eh oui, ça se pourrait !). À l'insu de la banque (et du programmeur), les clients qui déposeraient de l'argent sur leur compte épargne recevraient aussi automatiquement un grille-pain. Un instant d'inattention et toute modification effectuée dans `Checking` risque d'apparaître inopportunément dans `Savings`.

Comment éviter ces problèmes ? Affirmer que `Checking` est un cas particulier de `Savings` modifie, mais ne résout pas, le problème. Ce qu'il nous faut, c'est une troisième classe (que nous nommerons `Account`, autrement dit *compte*) qui incorpore tout ce qui est commun aux comptes chèques et aux comptes épargne. Cette relation est schématisée sur la Figure 22.3.

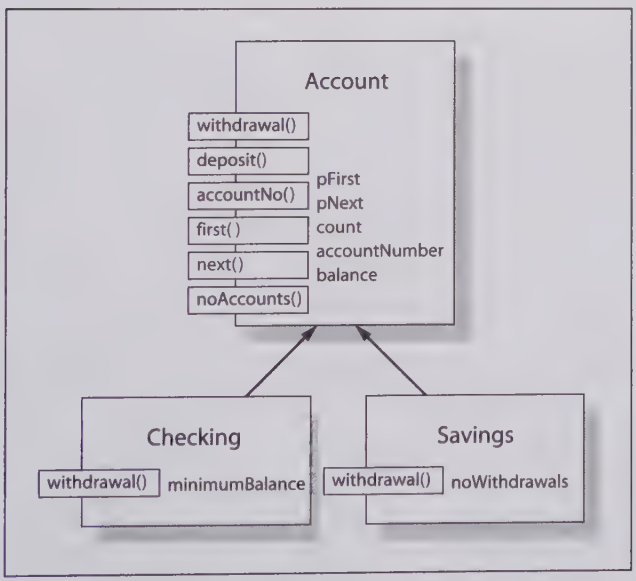


Figure 22.3
Les classes Checking et Savings sont maintenant basées sur une classe commune nommée Account.

Comment l'ajout d'un nouveau compte résout-il le problème ? Tout d'abord, la création de ce nouveau compte est plus conforme à ce qui se passe dans le monde réel (quel qu'il soit). Il existe en effet réellement quelque chose que l'on appelle un compte (enfin, je crois). Le compte chèques et le compte épargne sont des cas particuliers de ce concept plus général.

De plus, la classe `Savings` est préservée des changements apportés à la classe `Checking` et inversement. Si la banque institue une modification fondamentale pour tous les comptes, il suffira d'adapter `Account` pour que toutes les sous-classes héritent automatiquement du changement. Mais si la banque ne change les règles que pour les comptes chèques, il suffira de modifier la classe `Checking` sans rien changer à `Savings`.

Ce procédé de modification des propriétés communes à des classes similaires est appelé *dérivation*. C'est une fonctionnalité importante des langages orientés objet, et ceci pour toutes les raisons décrites jusqu'à présent plus une : la réduction des redondances. Au risque de me répéter, je rappelle que les redondances sont à éviter. Elles n'ont pas leur place ici.

La dérivation n'est légitime que si la relation d'héritage correspond à une réalité. Dériver ensemble les classes `Souris` et `Manette` parce que toutes deux sont des dispositifs de pointage est légitime. Mais dériver les classes `Souris` et `Affichage` uniquement parce que toutes deux effectuent des appels de bas niveau au système d'exploitation ne l'est pas.

La dérivation peut produire, et généralement produit, de multiples niveaux d'abstraction. Par exemple, un programme développé pour une banque digne de ce nom pourrait avoir une structure de classes semblable à celle qui est illustrée sur la Figure 22.4.

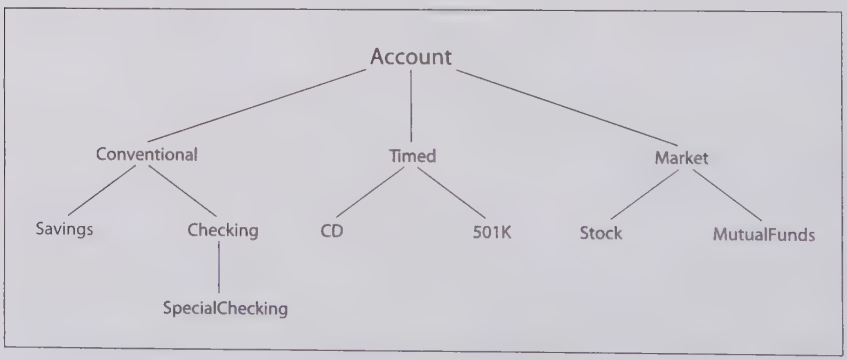


Figure 22.4
Une hiérarchie de comptes bancaires plus développée.

Vous remarquez qu'une autre classe a été insérée entre le duo `Checking`/`Savings` et la classe plus générale `Account`. Appelée `Conventional`, cette classe incorpore des caractéristiques communes aux comptes classiques. D'autres types de comptes ont été prévus, par exemple pour les porteurs d'actions (branche `Market`). Mais ce sont des gens que je ne connais pas personnellement.

De telles arborescences de classes sont courantes et recommandables dès lors que la relation qu'elles expriment correspond à la réalité. Notez toutefois qu'il n'existe aucune hiérarchie correcte, unique, pour un ensemble de classes donné.

Supposons que la banque autorise le détenteur d'un compte à accéder à distance aux comptes `Checking` et `Stock`. Les retraits sur d'autres types de comptes ne peuvent être effectués que sur place, à la banque. Bien que la structure de classe de la Figure 22.4 semble naturelle, celle de la Figure 22.5 se justifie tout autant (sinon mieux) à partir de cette nouvelle information. Le programmeur doit décider de la structure de classes correspondant le mieux aux données à traiter et qui conduit à l'implémentation la plus naturelle, la plus claire et la plus efficace possible.

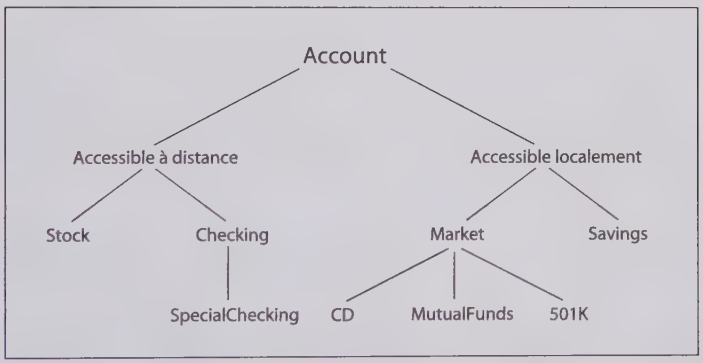


Figure 22.5
Une autre version de la hiérarchie de classes de la Figure 22.4.

L'implémentation de classes abstraites

Aussi satisfaisante intellectuellement que puisse être la dérivation, elle n'en induit pas moins un problème qui lui est inhérent. Revenons aux classes des comptes bancaires, plus spécifiquement à la classe de base commune `Account`. Essayez d'imaginer pendant une minute comment vous définiriez les différentes fonctions membres définies dans cette classe.

La plupart des fonctions membres de `Account` ne posent pas de problème car les deux types de compte les implémentent de la même manière. Mais la question posée par `Account::withdrawal()` est différente. Les règles de retrait sur un compte épargne ne sont en effet pas les mêmes que pour un compte chèques. Nous devons donc implémenter `Savings::withdrawal()` d'une toute autre manière que `Checking::withdrawal()`. Mais comment sommes-nous censés rédiger `Account::withdrawal()` ?

Demandons au directeur de la banque. La conversation pourrait prendre cette tournure :

- "Quelles sont les règles pour les retraits sur les comptes ?" demandez-vous benoîtement.
- "Sur quel type de compte ? Épargne ou compte chèques ?" interroge le directeur.
- "Eh bien... sur un compte...", répondez-vous un peu embêté, "Sur n'importe quel compte...".
- Le directeur prend un air absent (un air un peu compte, diraient certains).

Là où ça coince, c'est que la question n'a aucun sens. Il n'existe aucun objet "n'importe quel compte". Dans cet exemple, tous les comptes sont soit "chèque", soit "épargne". La notion de compte est une abstraction qui dérive des propriétés communes aux deux classes concrètes. Elle est incomplète car il lui manque la propriété essentielle `withdrawal()` qui sert à effectuer les retraits. (En entrant dans les détails, vous découvrirez d'autres propriétés qui manquent à un compte tout simple.)

Une *classe abstraite* n'existe que via des sous-classes. Par opposition, une *classe concrète* est une classe (dérivée) qui n'est pas abstraite. C'est trop beau.

Permettez-moi d'emprunter un exemple dans le règne animal. Vous pouvez observer différentes espèces d'animaux à sang chaud, dont la femelle porte les petits, et en conclure qu'il y a un concept appelé *mammifère*. Vous pouvez dériver des classes de mammifères, comme les canidés, les félins et les hominidés. Il est toutefois impossible sur terre de trouver un pur mammifère, c'est-à-dire un archétype qui n'appartiendrait à aucune sous-espèce particulière de mammifère. Le mammifère est un concept de haut niveau inventé par l'homme : il n'en existe aucune instance.

Remarquez que je formule cette assertion avec une certaine assurance et pourtant, du temps a passé depuis que j'ai écrit cela. Or les scientifiques découvrent tous les jours de nouveaux animaux. Mais aucun n'a découvert un concept de mammifère. Jamais un savant n'a eu l'occasion de déclarer : "Ce nouvel animal est un mammifère et rien de plus. Rien qu'un mammifère." Le problème avec ce genre d'affirmation, c'est que tout animal a inévitablement des propriétés que d'autres mammifères ne possèdent pas. Et même si ce n'est pas le cas aujourd'hui, on trouvera bien des différences dans l'avenir.

Afin de se conformer au mieux à cette situation, le C++ est doté de fonctionnalités permettant de définir incomplètement les classes abstraites, ce qui le rend capable de décrire un concept aussi incomplet que celui de mammifère.

Notion de classe abstraite

Une *classe abstraite* est une classe possédant une ou plusieurs fonctions virtuelles pures. Génial ! Voilà qui nous avance bien !

Allons-y. Une fonction *virtuelle pure* est une fonction membre virtuelle qui est signalée comme n'ayant pas d'implémentation, probablement parce que personne ne sait comment l'implémenter à partir des informations disponibles dans la classe (classes de base y compris).

Demander comment implémenter exactement la fonction `withdrawal()` dans la classe `Account` ne veut rien dire. Toutefois, le concept de retrait sur un compte est sensé. Un programmeur en C++ peut écrire une fonction `withdrawal()` qui représente le concept du retrait d'argent, mais qui n'a pas de corps de fonction car nous ne savons pas comment l'implémenter. Une telle fonction est appelée "fonction virtuelle pure" (ne me demandez pas pourquoi on l'appelle ainsi).

La syntaxe de la déclaration d'une fonction virtuelle pure est exposée dans la classe `Account` ci-dessous :

```
// Account - cette classe est une classe abstraite
class Account
{
  protected:
    Account(Account& c);  // évite de  faire des copies
  public:
    Account(unsigned accNo, float initialBalance = 0.0F);

    // fonctions d'accès
    unsigned int accountNo( );
    float acntBalance( );
    static int noAccounts( );

    // fonctions de transaction
    void deposit(float amount);

    // ce qui suit est une fonction virtuelle pure
    virtual void withdrawal(float amount) = 0;

  protected:
```

```
    // conserve les comptes dans une liste chaînée afin
    // qu'il n'y ait aucune limite au nombre de comptes.
    static int count;     // nombre de comptes
    unsigned  accountNumber;
    float   balance;
};
```

Le `= 0` après la déclaration de `withdrawal()` indique que le programmeur n'a pas l'intention de définir cette fonction. La déclaration est un substitut pour les sous-classes. Les sous-classes de `Account` sont censées redéfinir cette "coquille vide" avec une fonction concrète. Le programmeur doit fournir une implémentation pour chaque fonction membre qui n'est pas déclarée comme étant virtuelle pure.

À mon avis, cette notation est idiote et je ne l'apprécie pas plus que vous. Mais comme elle a été placée là pour y rester, vous devez apprendre à vivre avec. Il y a une raison, sinon une justification, à cette notation : chaque fonction virtuelle doit avoir une entrée dans une table spéciale. Cette entrée contient l'adresse de la fonction. L'entrée d'une fonction virtuelle pure est zéro. D'autres langages procèdent différemment.

Une classe abstraite ne peut pas être instanciée avec un objet. C'est-à-dire qu'il vous est impossible de créer un objet à partir d'une classe abstraite. Par exemple, la déclaration qui suit n'est pas légale :

```
void fn()
{
  // déclaration d'un compte avec 100 unités.
  Account acnt(1234, 100.00);  // ceci n'est pas légal
  acnt.withdrawal(50);         // à votre avis, que fera
                               // cet appel ?
}
```

Si cette déclaration était admise, l'objet qui en résulterait serait incomplet. Il lui manquerait des fonctionnalités. Par exemple, que ferait l'appel ci-dessus ? Rappelez-vous que `Account::withdrawal()` n'a pas d'existence réelle.

Les classes abstraites servent de classes de base pour d'autres classes. La classe `Account` contient toutes les propriétés associées à un compte bancaire générique. Vous pouvez créer d'autres types de comptes bancaires qui hériteront de `Account`, mais ils ne pourront pas être instanciés avec un objet.

Obtenir une honnête classe à partir d'une classe abstraite

La sous-classe d'une classe abstraite reste abstraite jusqu'à ce que toutes les fonctions virtuelles pures aient été redéfinies. La classe `Savings` n'est pas abstraite car elle redéfinit la fonction virtuelle pure `withdrawal()` avec une déclaration irréprochable. Un objet de la classe `Savings` sait comment il doit exécuter `withdrawal()` lorsqu'il a été appelé à cet effet. Il en va de même pour la classe `Checking` : elle n'est pas virtuelle parce que la fonction `withdrawal()` redéfinit la fonction virtuelle pure qui se trouve dans la classe de base.

Une sous-classe d'une classe abstraite peut néanmoins rester abstraite. Examinons les classes suivantes :

```
class Display
{
  public:
    virtual void initialize( ) = 0;
    virtual void write(char *pString) = 0;
};

class SVGA : public Display
{
  // redéfinit les deux fonctions membres avec des fonctions "réelles"
  virtual void initialize( );
  virtual void write(char *pString);
};

class HWVGA : public Display
{
  // redéfinit la seule fonction que nous connaissons jusqu'à présent
  virtual void write(char *pString);
};

class ThreedVGA : public HWVGA
{
  virtual void initialize( );
};

void fn( )
{
  SVGA mc;
  ThreedVGA  vga;
  // ...ce que la fonction choisit de faire à partir d'ici...
}
```

La classe `Display`, censée représenter des moniteurs de PC, a deux fonctions virtuelles pures : `initialize()` et `write()`. Vous ne pouvez implémenter aucune des deux pour un adaptateur *en général*. Les différents types de carte vidéo ne s'initialisent ou n'écrivent pas de la même manière.

L'une des sous-classes, `SVGA`, n'est pas abstraite. Il s'agit d'un type d'adaptateur vidéo spécifique que l'on sait comment programmer. C'est pourquoi la classe `SVGA` a redéfini `initialize()` ainsi que `write()` de façon appropriée pour cet adaptateur.

L'autre sous-classe, `HWVGA` n'est pas abstraite non plus. Là encore, nous savons (peut-être) comment programmer une carte VGA accélérée. Mais ici, il existe un niveau d'abstraction entre la classe générique `Display` et le cas particulier de l'affichage `ThreeVGA` qui correspond à un type spécial de carte vidéo 3D.

Pour notre propos, nous supposerons que le mode d'écriture est identique pour toutes les cartes VGA accélérées, mais que chacune doit être initialisée à sa façon (ce n'est pas obligatoirement vrai, mais nous ferons comme si ça l'était). Pour exprimer ce mode d'écriture commun, la classe `HWVGA` redéfinit l'implémentation de la fonction `write()`, sans oublier évidemment toutes les autres propriétés partagées par les cartes HWVGA. Nous ne redéfinissons toutefois pas la fonction membre `initialize()`, puisque ce n'est pas une propriété commune à toutes ces cartes HWVGA.

C'est pourquoi la classe `HWVGA` reste une abstraction : la fonction `write()` y est redéfinie, mais pas la fonction `initialize()`.

Comme `ThreeVGA` hérite de `HWVGA`, elle n'a à redéfinir que la seule fonction membre manquante, `initialize()`, pour compléter la définition de l'adaptateur `Display`. La fonction `fn()` est de ce fait libre d'instancier et d'utiliser un objet `ThreeVGA`.

La redéfinition de la dernière fonction virtuelle pure avec une fonction membre normale rend la classe complète (non abstraite). Seules les classes non abstraites peuvent être instanciées avec un objet.

Passer des classes abstraites

Comme vous ne pouvez pas instancier une classe abstraite, il paraît curieux de pouvoir déclarer un pointeur ou une référence vers cette classe. Grâce au polymorphisme, ce n'est pas aussi aberrant qu'il y paraît. Voyez plutôt ce petit bout de code :

```
void fn(Account *pAccount);  // ceci est légal
void otherFn( )
{
    Savings s;
    Checking c;

    // ceci est légitime car Savings IS_A Account
    fn(&s);
    // idem
    fn(&c);
}
```

Ici, `pAccount` est déclaré en tant que pointeur vers un compte `Account`. Il est cependant clair que quand la fonction est appelée, elle recevra l'adresse de quelque objet d'une sous-classe non abstraite telle que `Savings` ou `Checking`.

Tous les objets reçus par `fn()` seront soit de la classe `Savings`, soit de la classe `Checking` (ou quelque future classe non abstraite de `Account`). La fonction est assurée que vous ne passerez jamais un objet issu de la classe `Account`, puisqu'il est effectivement impossible d'en créer un.

Déclarer des fonctions virtuelles pures : est-ce vraiment nécessaire ?

Si la fonction `withdrawal()` ne peut pas être définie, pourquoi ne pas la laisser de côté ? Pourquoi ne pas la déclarer uniquement dans `Savings` et dans `Checking` où elle pourrait aussi être définie tout en restant hors de portée de `Account` ? Vous pouvez le faire dans beaucoup de langages orientés objet. Mais le C++ tient à s'assurer que vous savez ce que vous êtes en train de faire.

Rappelez-vous que la déclaration d'une fonction formalise son nom étendu, y compris ses arguments, tandis qu'une définition inclut le code à exécuter lorsque la fonction est appelée.

Je peux apporter quelques modifications mineures au programme `Account` pour illustrer ce problème :

```
class Account
{
    // comme auparavant, mais sans
    // la déclaration de withdrawal()
};
```

```
class Savings : public Account
{
  public:
    virtual void withdrawal(float amnt);
};

void fn(Account *pAcc)
{
    // on retire de l'argent
    pAcc->withdrawal(100.00F);
               // cet appel n'est pas autorisé car
               // withdrawal( ) n'est pas un membre
               // de la classe Account
};

int main( )
{
    Savings s;  // ouvre un compte
    fn(&s);

    // ...on continue...
}
```

Supposons que vous ouvriez un compte d'épargne `s`. Vous passez ensuite l'adresse de ce compte à la fonction `fn()` qui tente d'effectuer un retrait. Mais, comme la fonction `withdrawal()` n'est pas un membre de `Account`, le compilateur génère une erreur.

Voyons comment les fonctions virtuelles pures corrigent le problème. Voici la même situation, mais avec `Account` déclaré comme classe abstraite :

```
class Account
{
  public:
    // comme auparavant, mais déclaration de
    // withdrawal en tant que fonction virtuelle pure
    virtual void withdrawal(float amnt) = 0;
};

class Savings : public Account
{
  public:
    virtual void withdrawal(float amnt);
};
```

```
void fn(Account *pAcc)
{
    // on retire de l'argent
    pAcc->withdrawal(100.00F); // maintenant, cela marche
};

int main( )
{
    Savings s;  // ouvre un compte
    fn(&s);
    // ...on continue...
}
```

La situation est la même, sauf que la classe `Account` inclut la fonction membre `withdrawal()`. Désormais, lorsque le compilateur essaie de voir si `pAcc->withdrawal()` est défini, il trouve comme prévu la définition de `Account::withdrawal()`. Le compilateur est content. Vous êtes content. Moi aussi je suis content (mais, franchement, un match de football et une bonne bière suffisent à mon bonheur).

Une fonction virtuelle pure est un "réservoir vide" dans la classe de base que la sous-classe redéfinira avec sa propre implémentation. Sans ce substitut, il n'y a pas de redéfinition possible.

Dérivation du code source C++

Dériver un problème a un côté physique. Les classes qui ont été dérivées en partant du magma de concepts qui constituent un programme devraient être déplacées dans leur propre "espace".

Le programmeur peut diviser une même application en une série de fichiers distincts, ou *modules*. Ces fichiers sources individuels sont compilés séparément au cours du processus de construction afin de générer le programme final. Les modules peuvent être alloués à des groupes distincts que l'on appelle *espaces de noms*.

Combiner des modules qui ont été compilés chacun de leur côté pour former un seul exécutable se dit *lier* (ou éditer les liens). Ce travail relève du *linker*, ou *éditeur de liens*.

Il existe nombre de raisons qui justifient le découpage des programmes en un ensemble de parties plus faciles à gérer. Tout d'abord, le partage en modules

offre un niveau plus élevé d'*encapsulation*. Les classes érigent des murs autour de leurs membres internes afin d'obtenir un degré suffisant de sécurité. Les programmes peuvent faire la même chose pour leurs fonctions.

L'encapsulation est l'un des avantages offerts par la programmation orientée objet.

Ensuite, il est plus facile de comprendre, et donc d'écrire et de mettre au point, un programme composé de modules bien organisés qu'un énorme fichier source unique qui contiendrait toutes les classes et les fonctions utilisées dans l'application.

Il faut aussi parler de la réutilisation. C'est un de mes arguments favoris pour vendre C++. Il est extrêmement difficile d'assurer la maintenance d'une classe si vous la recopiez bêtement dans chacun de vos programmes. Il est évidemment préférable de partager le module qui la contient entre toutes les applications qui en ont besoin.

Enfin, le temps a une valeur non négligeable. Certes, construire les exemples de ce livre sous Dev-C++ (ou sous Visual C++) n'est pas bien long, surtout avec un ordinateur aussi puissant que le vôtre. Mais les programmes du commerce contiennent parfois des millions de lignes de code source. Les reconstruire peut alors prendre la journée entière. Il serait évidemment intolérable de s'engager dans une telle procédure sous le prétexte qu'on vient de corriger une toute petite ligne. En réalité, l'essentiel du temps est passé à compiler des fichiers sources dans lesquels rien n'a été modifié. Il est bien plus rapide de ne recompiler que les modules qui ont été mis à jour, puis de lier tout ce petit monde.

Les espaces de noms séparés autorisent un niveau encore plus élevé d'encapsulation. Ils devraient combiner un ensemble de modules répondant à un certain objectif. Par exemple, toutes les fonctions mathématiques pourraient être associées dans un espace de nom appelé `Math`.

Dans ce qui suit, nous allons passer aux travaux pratiques en construisant un programme simpliste, appelé `SeparateModules`, qui contiendra une classe `Student`, une autre classe `GraduateStudent` et un module `main()` permettant de les tester toutes les deux.

Découper le programme Student

Vous allez commencer par décider du découpage de `SeparateModules` en unités logiques. Remarquez d'abord que `Student` est en soi une entité. Elle ne dépend d'aucune autre fonction (mis à part celles de C++). Il semble donc cohérent de

placer cette classe dans son propre module. Comme elle sera utilisée à plusieurs reprises, nous définirons deux fichiers séparés : un pour les partie déclarations (`student.h`) et un autre pour l'implémentation (`Student.cpp`). Par convention, le fichier inclus (`include`) porte le nom de la classe principale qu'il définit, mais en caractères minuscules. Dans l'idéal, ce fichier ne devrait comporter qu'une seule classe. Ceci permet de n'inclure que les fichiers nécessaires à un programme donné.

Historiquement, tous les fichiers inclus portaient l'extension `.h`. Ceci a été changé dans le standard C++ actuel. Les fichiers système inclus (comme par exemple `iostream`) ne possèdent plus du tout d'extension. Mais la plupart des programmeurs continuent à ajouter l'extension `.h` à leurs inclusions. Cela permet au moins de les repérer plus facilement.

Le fichier `student.h` va donc se présenter ainsi :

```
// Student - un étudiant de base
#ifndef _STUDENT_
#define _STUDENT_

namespace Schools
{
    class Student
    {
      public:
        Student(char* pszName, int nID);
        virtual char* display();
      protected:
        // nom de l'étudiant
        char* pszName;
        int   nID;
    };
}
#endif
```

La directive `#ifndef` est un contrôle destiné au préprocesseur, comme `#include`. La ligne `#ifndef _STUDENT_` demande d'inclure les lignes qui suivent seulement si l'argument `_STUDENT_` est défini. C'est justement le cas la première fois que `student.h` est inclus. Mais la ligne qui suit, `#define _STUDENT_`, définit maintenant l'argument. Le résultat de tout cela est que `student.h` n'est traité qu'une seule fois, indépendamment du nombre d'inclusions que l'on peut rencontrer dans un fichier donné.

Définir un espace de nom

Le fichier `student.h` crée ensuite un espace de nom : `Schools`.

Un *espace de nom* est une collection de classes, plus ou moins appariées, et qui offrent quelque similitude logique. Dans cet exemple, j'ai l'intention de regrouper toutes mes classes, qu'il s'agisse des étudiants de base, des diplômés, des cours ou encore des modules, dans un seul et même espace de nom, `Schools`.

Les classes qui relèvent du même espace de nom sont un peu comme les membres d'une famille. On peut s'y appeler par son prénom. En d'autres termes, une classe d'un espace de nom a le droit de faire directement référence à une autre classe de cet espace. En revanche, les classes externes (qui ne font pas partie de la famille) doivent spécifier l'espace de nom. Vous verrez comment procéder un peu plus loin, dans l'application `SeparatedMain`.

Une autre raison justifie la séparation des modules en espaces de noms : éviter les collisions. Ainsi, la classe `Lion` de l'espace de nom `Animaux` ne peut pas interférer avec celle qui porte le même nom dans l'espace `Constellations`.

Implémentation de la classe Student

J'ai placé l'implémentation de la classe `Student` dans le fichier `Student.cpp` :

```
// Student - implémente les méthodes de la classe Student
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
#include "student.h"

namespace Schools
{
    Student::Student(char* pszNameArg, int nIDArg)
        : nID(nIDArg)
    {
        pszName = new char[strlen(pszNameArg) + 1];
        strcpy(pszName, pszNameArg);
    }

    // display - renvoie une description de l'étudiant
    char* Student::display()
    {
```

```
        // copie le nom de l'étudiant dans un bloc du tas
        // que nous pouvons retourner à l'appelant
        char* pReturn = new char[strlen(pszName) + 1];
        strcpy(pReturn, pszName);
        return pReturn;
    }
}
```

Le constructeur de `Student` recopie le nom et l'identificateur qui lui sont passés. La méthode virtuelle `display()` renvoie une chaîne qui décrit l'objet `Student`. La compilation de `Student.cpp` produit un fichier intermédiaire qui pourra ensuite être combiné rapidement avec d'autres fichiers intermédiaires pour former un programme exécutable complet.

Pour des raisons historiques, ce fichier intermédiaire porte comme extension `.o` (pour "fichier objet") dans la plupart des environnements C++.

Découper le programme GraduateStudent

Le prochain module (pratiquement) indépendant est `GraduateStudent`. En bonne logique, on pourrait songer à incorporer la classe `GraduateStudent` dans `Student.cpp`. Mais il est parfaitement possible que certains programmes aient besoin d'utiliser la classe `Student` en tant qu'abstraction (quoi que les étudiants sont plus souvent distraits qu'abstraits), sans s'intéresser le moins du monde aux différences entre les étudiants de première année et ceux qui sont en troisième cycle.

J'ai me suis efforcé de faire dans la simplicité (après tout, le sujet est suffisamment difficile comme cela sans qu'il soit nécessaire d'en rajouter). Le fichier inclus (`graduateStudent.h`) se présente ainsi :

```
// graduateStudent - un type particulier d'étudiant
#ifndef _GRADUATE_STUDENT_
#define _GRADUATE_STUDENT_

#include "student.h"
namespace Schools
{
    class GraduateStudent : public Student
    {
      public:
        // un constructeur trivial
        GraduateStudent(char* pszName, int nID)
```

```
                : Student(pszName, nID){}
        // démonstration de fonction virtuelle
        virtual char* display();
    };
}

#endif
```

Vous remarquerez que `student.h` est inclus dans `graduateStudent.h`. C'est normal, puisque la classe `GraduateStudent` dépend de la définition de `Student`.

Le fichier source `GraduateStudent.cpp` code la méthode `display()`, qui est la seule fonction membre restant à implémenter :

```
// GraduateStudent - un type particulier d'étudiant
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include "graduateStudent.h"
namespace Schools
{
    char* GraduateStudent::display()
    {
        // récupère la description d'un étudiant de base...
        char* pFirst = Student::display();

        // ...pour lui ajouter le texte suivant
        char* pSecond = "-G";

        // concaténation des deux chaînes
        char* pName = new char[strlen(pFirst) +
                               strlen(pSecond) + 1];
        strcpy(pName, pFirst);
        strcat(pName, pSecond);

        // ne pas oublier de libérer la chaîne retournée par
        // Student::display() sur le tas avant de renvoyer
        // notre nouvelle chaîne à l'appelant
        delete pFirst;
        return pName;
    }
}
```

La version GraduateStudent de display() ajoute les caractères "-G" à la fin de la chaîne renvoyée par l'objet Student courant. Elle alloue pour cela un tableau de caractères suffisamment grand pour contenir ces nouvelles informations.

Ne supposez jamais par avance que le tampon d'origine contient assez de place pour y ajouter des caractères supplémentaires.

Le programme copie le contenu de la chaîne originale dans le tableau qui vient d'être alloué. Il y ajoute alors les caractères "-G". La fonction display() doit libérer du tas l'espace alloué par Student::display() avant de se terminer.

Oublier de rendre des tampons au tas constitue ce que l'on appelle une *fuite de mémoire*. Un programme qui comporte des fuites de mémoire commence par s'exécuter normalement. Puis, il ralentit de plus en plus au fur et à mesure que la mémoire disponible se réduit. Il peut même finir par se bloquer. Les fuites de mémoire sont très difficiles à détecter (rien de commun avec les fuites de robinet).

Implémenter une application

Les deux classes, Student et GraduateStudent, ont été enregistrées dans des fichiers sources différents et incluses dans l'espace de nom Schools. J'ai alors écrit une application très simple pour les invoquer :

```
// SeparatedMain - exemple d'application partagée en
//                 plusieurs sections - ici, la partie main()
#include <cstdio>
#include <cstdlib>
#include <iostream>

#include "graduateStudent.h"
#include "student.h"

using namespace std;
//using namespace Schools;
using Schools::GraduateStudent;

int main(int nArgc, char* pszArgs[])
{
    Schools::Student s("Sophie Moore", 1234);
    cout << "Student = " << s.display() << endl;

    GraduateStudent gs("Greg U. Waite", 5678);
    cout << "Student = " << gs.display() << endl;
```

```
    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Vous retrouvez ici les deux fichiers inclus `student.h` et `graduateStudent.h`. Ceci permet à l'application d'accéder à nos deux classes.

Je sens que vous allez me faire remarquer que l'inclusion de `graduateStudent.h` devrait provoquer automatiquement celle de `student.h`. Mais ne vous laissez pas aller aux évidences trop simplistes. Vous pourriez parfaitement avoir besoin d'accéder uniquement à la classe `Student`, que `graduateStudent.h` soit ou non inclus. De toute façon, la directive `#ifndef` de `student.h` vous assure que le contenu de ce fichier ne sera pas traité deux fois par le compilateur C++.

`SeparatedMain` n'est pas membre de l'espace de nom `Schools`. Lorsque `main()` fait référence à la classe `Student`, C++ ne sait donc *a priori* rien de l'espace de nom d'où elle provient. Il nous faut donc lever cette ambiguïté en spécifiant entièrement le couple espace de nom/classe. C'est ce que fait la déclaration `Schools::Student`.

D'un autre côté, le programmeur peut préciser ses intentions dès le début du module. La phrase :

```
using Schools::GraduateStudent;
```

indique à C++ que toute mention du nom `GraduateStudent` se rapporte à l'espace de nom `Schools`.

Plus largement, il est possible d'accéder à tous les membres de cet espace de nom en ajoutant la commande `using namespace Schools`. La version suivante de la fonction `main()` sera construite avec succès :

```
using namespace Schools;

int main(int nArgc, char* pszArgs[])
{
    Student s("Sophie Moore", 1234);
    cout << "Student = " << s.display() << endl;

    GraduateStudent gs("Greg U. Waite", 5678);
```

```
    cout << "Student = " << gs.display() << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Vous avez utilisé (peut-être sans vous en rendre compte) l'instruction `using namespace std` dès le début de ce livre. Les modules qui composent la librairie standard du C++ sont membres de l'espace de nom `std`.

Le fichier de projet

Plein de fébrilité, j'ouvre le fichier `SeparatedMain.cpp` dans le compilateur et je demande à construire le programme. La compilation du module semble s'effectuer sans problème, mais une erreur se produit au cours de l'édition des liens (*Linker error*). Manifestement, C++ ne sait pas ce qu'est un étudiant (`Student`). Il vous faut donc lui expliquer d'une façon ou d'une autre que `Student.cpp` et `GraduateStudent.cpp` doivent être liés à `SeparatedMain.cpp` pour créer le programme exécutable.

La plupart des environnements C++ (y compris Dev-C++ et Visual C++ .NET) combinent les modules via ce que l'on appelle un *fichier de projet*. Même si le format de ces fichiers est différent sous Dev-C++ et Visual C++ .NET, le principe reste fondamentalement le même.

Créer un fichier de projet sous Dev-C++

Voyons pas à pas comment créer un projet sous Dev-C++ :

1. **Choisissez Fichier->Nouveau->Projet. Sélectionnez le type de projet Console Application, puis saisissez `SeparateModules` dans le champ Nom (voir la Figure 22.6).**

2. **Cliquez sur OK. Dev-C++ va ouvrir une fenêtre pour sauvegarder le fichier.**

3. **Sélectionnez le dossier de destination. Ici, j'ai ouvert `\Nuls\Chap22`. Dev-C++ crée alors un projet appelé `SeparateModules.dev` qui est associé au module par défaut `main.cpp`.**

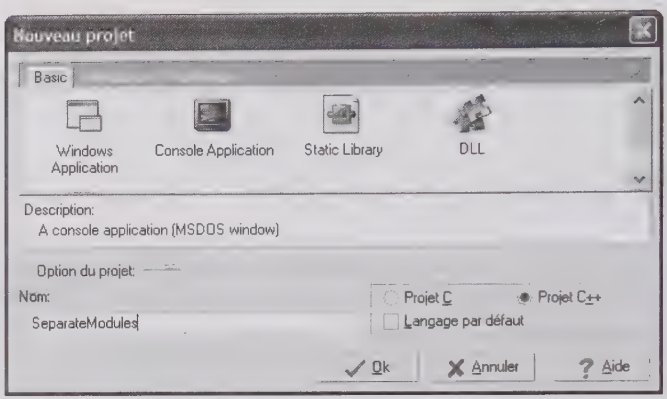


Figure 22.6 Définissez le type et le nom du projet.

4. **Puisque vous avez déjà votre propre fonction `main()`, il est inutile de conserver `main.cpp`. Choisissez Projet->Supprimer du projet. Cliquez sur la ligne `main.cpp` puis sur le bouton OK (voir la Figure 22.7).**

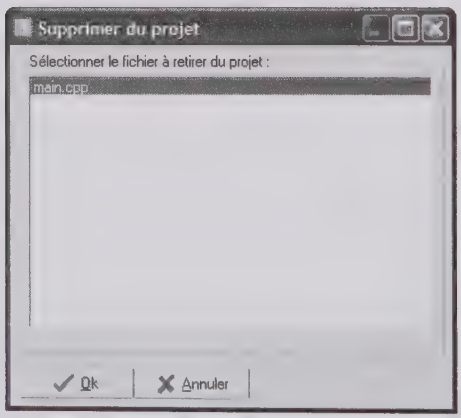


Figure 22.7 Supprimez un module du projet courant.

5. **Si ce n'est pas encore fait, copiez tous les fichiers dont vous avez besoin dans le dossier du projet (c'est-à-dire `\Nuls\Chap22` dans mon cas). Ces fichiers sont `SeparatedMain.cpp`, `Student.cpp`, `GraduateStudent.cpp`, `student.h` et `graduateStudent.h`.**

6. **Choisissez maintenant Projet->Ajouter au projet. Dev-C++ devrait vous proposer le bon dossier. Appuyez sur Ctrl+A pour sélectionner tous les fichiers. Cliquez sur Ouvrir.**

7. **Choisissez alors Exécuter->Tout reconstruire. Vous demandez ainsi à Dev-C++ de compiler tous les modules du projet et de créer un programme exécutable.**

8. **Cliquez sur l'onglet Classes dans le panneau de gauche pour afficher une description détaillée de toutes les classes du programme (voir la Figure 22.8).**

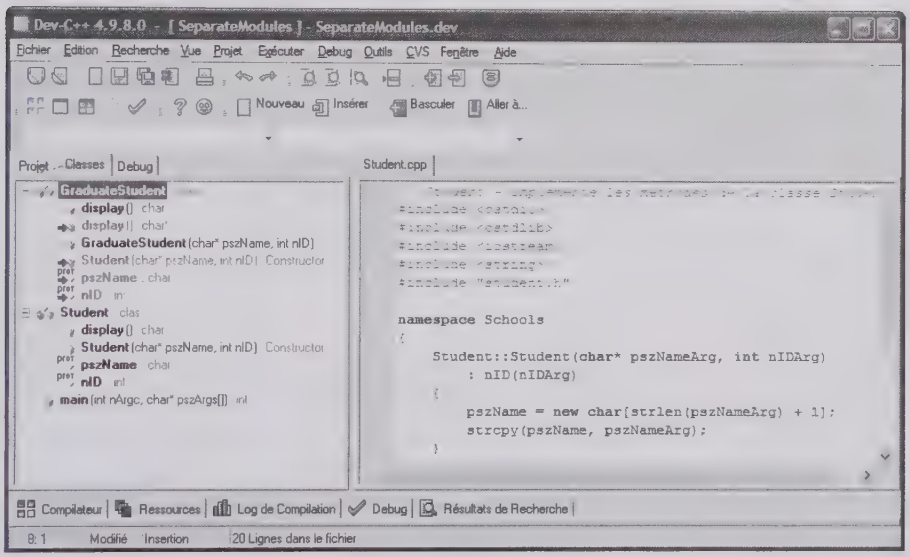


Figure 22.8
L'onglet Classes affiche les membres de chaque classe.

9. **Vous n'obtenez pas toutes ces informations. Il faut vous assurer que l'explorateur de classes est correctement configuré. Pour cela, choisissez la commande Outils->Options de l'éditeur. Cliquez sur l'onglet Explorateur de classes. Activez les options voulues en respectant les indications de la Figure 22.9.**

   Remarquez la façon dont l'explorateur de classes affiche chaque membre. Les fonctions exposent leurs arguments ainsi que le type d'objet retourné. Vous notez également la présence de deux fonctions `display()` sous la classe `GraduateStudent`.

10. **Sous la classe `GraduateStudent`, cliquez sur la ligne `display()` qui est précédée d'une sorte de flèche colorée.**

    Cette action ouvre le fichier `Student.cpp` et place le curseur sur la fonction membre `display()`. Faites de même avec l'autre ligne `display()` de

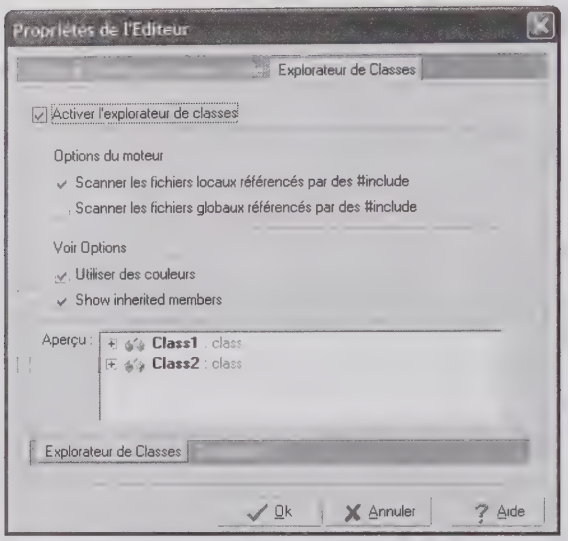


Figure 22.9 Comment configurer l'explorateur de classes.

l'explorateur de classes (toujours dans `GraduateStudent`). Cette fois, l'éditeur active la fonction membre `GraduateStudent::display()` du fichier `GraduateStudent.cpp`.

Les propriétés du projet ont pour l'instant des valeurs par défaut. Vous pouvez les personnaliser de la façon suivante :

11. **Sélectionnez Projet->Options du projet.**

    Sélectionnez par exemple les options de l'éditeur de liens sous l'onglet Compilateur. Cliquez au bout de la ligne Générer des informations de débogage. Choisissez la valeur Yes si vous avez l'intention de vous servir du débogueur de Dev-C++. Validez.

Je vous encourage vivement à décomposer votre programme en de multiples fichiers sources. Cela simplifiera l'édition, l'adaptation et la mise au point de vos applications.

Cinquième partie

Facultatif, mais si important

Dans cette partie...

Le but de ce livre n'est pas de faire de vous un expert du C++, mais de vous donner une solide compréhension des bases de ce langage ainsi que de la programmation orientée objet.

Les parties précédentes ont couvert les notions essentielles qu'il vous faut connaître pour produire un programme C++, orienté objet et bien écrit. Mais l'évidence s'impose : C++ est un langage d'ampleur considérable et il reste de nombreuses fonctionnalités à découvrir. Citons en particulier les entrées et les sorties dans des fichiers ou encore la librairie standard (STL, ou *Standard Template Library*), un vrai monde à découvrir.

Chapitre 23

Un nouvel opérateur d'affectation

Dans ce chapitre :

- Introduction à l'opérateur d'affectation.
- Pourquoi et quand l'opérateur d'affectation est nécessaire.
- Ressemblances entre l'opérateur d'affectation et le constructeur de copie.

Les types de données *intrinsèques* sont ceux qui font partie du langage lui-même : `int`, `float`, `double` et ainsi de suite, plus les divers types de pointeurs. Les Chapitres 3 et 4 décrivent les opérateurs que le C++ fournit pour les types de données intrinsèques. Mais il offre plus en vous permettant de définir en supplément les opérateurs des classes que vous créez vous-même. C'est ce que l'on appelle la *surcharge des opérateurs*.

Normalement, la surcharge de l'opérateur est optionnelle et n'est généralement pas utilisée par les programmeurs débutants. Bon nombre de développeurs expérimentés pensent d'ailleurs que les opérateurs surchargés ne sont pas une aussi bonne idée qu'il y paraît. Cependant, vous devez au moins savoir comment surcharger un opérateur : l'affectation.

Comparaison entre opérateurs et fonctions

Un opérateur n'est rien de plus qu'une fonction intégrée possédant une syntaxe particulière. L'addition ci-dessous :

```
a + b
```

peut (et d'ailleurs devrait) se comprendre comme si elle était écrite :

```
operator+(a, b)
```

Le C++ attribue à chaque opérateur un nom spécial de type fonction. Le *nom fonctionnel* d'un opérateur est le symbole de cet opérateur précédé du mot-clé `operator` et suivi des arguments appropriés. Par exemple, l'opérateur `+` qui additionne deux entiers `int` et produit un autre entier `int` est nommé `int operator+(int, int)`.

Tout opérateur peut être défini dans une classe définie par l'utilisateur. Je peux donc parfaitement créer une syntaxe :

```
Complex operator*(Complex&, Complex&)
```

qui me permettrait de multiplier deux objets de type `Complex`. Le nouvel opérateur peut être conçu avec la même sémantique que celui qu'il surcharge, mais ce n'est pas une obligation.

La surcharge des opérateurs s'appuie sur les règles suivantes :

- Le programmeur ne peut surcharger les opérateurs `.`, `::`, `*` (déréférencement) et `&`.
- Il ne peut pas non plus inventer de nouveaux opérateurs. Par exemple, `x $ y` n'est pas autorisé.
- Le format des opérateurs ne peut pas être changé. Il n'est donc pas possible de définir un opérateur `%i` car `%` est un opérateur unaire.
- La priorité des opérateurs est intangible. Un programme ne peut pas forcer `operator+` à être évalué avant `operator*`.
- Les opérateurs ne peuvent pas être redéfinis lorsqu'ils s'appliquent à des types de données intrinsèques. Par exemple, il est impossible de changer la signification de 1 + 2. En d'autres termes, la surcharge des opérateurs ne concerne que des types de données nouvellement définis.

La surcharge des opérateurs fait partie de ces choses qui ont *a priori* l'air d'une bonne idée et qui n'en sont pas forcément une. Elle provoque souvent plus de problèmes qu'elle n'en résout, à deux exceptions près qui sont le sujet de ce chapitre.

Insérer un nouvel opérateur

Les opérateurs d'insertion (<<) et d'extraction (>>) sont comme leur nom l'indique... des opérateurs qui sont justement surchargés dans diverses classes concernant les entrées et les sorties. Ces définitions se trouvent dans le fichier inclus `iostream` (qui est systématiquement utilisé dans tous les programmes). L'instruction :

```
cout << "une chaîne"
```

devient donc :

```
operator<<(cout, "une chaîne")
```

Nos vieux amis `cout` et `cin` sont des objets prédéfinis, respectivement associés à la console et au clavier. Nous reviendrons sur ces relations dans le Chapitre 24.

La création de copies de surface est un problème profond

Quelle que soit l'opinion que l'on puisse avoir sur la surcharge des opérateurs, il en est un qui est concerné par la plupart des classes que vous créez : l'opérateur d'affectation. Le langage C++ fournit une définition par défaut de `operator=()` pour toutes les classes. Cette définition par défaut effectue une copie membre à membre des objets. Ceci fonctionne parfaitement pour un type de donnée intrinsèque tel que `int` :

```
int i;
i = 10;   // copie "membre à membre"
```

Le même processus par défaut s'applique aux classes définies par l'utilisateur. Dans l'exemple ci-dessous, chaque membre de `source` est copié vers le membre correspondant de `destination` :

```
void fn()
{
```

```
  MaStructure source, destination;
  destination = source;
}
```

L'opérateur d'affectation par défaut fonctionne avec la plupart des classes. En revanche, il n'est pas correct dans le cas de classes qui allouent des ressources, notamment de la mémoire sur le tas. Le programmeur doit surcharger `operator=()` pour gérer le transfert de ces ressources.

L'opérateur d'affectation ressemble assez à un constructeur de copie (voir le Chapitre 18). À l'usage, les deux sont pratiquement identiques :

```
void fn(MaClasse& mc)
{
  MaClasse newMC = mc;   // celui-ci utilise évidemment le
                         // constructeur de copie
  MaClasse newerMC = mc; // moins évident : celui-ci utilise aussi
                         // le constructeur de copie
  MaClasse newestMC      // celui-ci crée un objet par défaut
  newestMC = mc;         // puis le remplit avec l'argument passé
}
```

La création de `newMC` suit le schéma habituel : le nouvel objet est une image de l'original obtenue à l'aide du constructeur de copie `MaClasse(MaClasse&)`. En revanche, il est moins évident que `newerMC` est lui aussi créé par le constructeur de copie. La syntaxe `MaClasse a = b` n'est qu'une autre façon d'écrire `MaClasse a(b)`. En particulier, et en dépit de son apparence, cette déclaration ne met *pas* en jeu l'opérateur d'affectation. En revanche, `newestMC` est créé en utilisant le constructeur par défaut (`void`), puis réécrit en lui affectant l'objet `mc`.

À l'instar du constructeur de copie, un opérateur d'affection devrait être produit chaque fois qu'une copie superficielle n'est pas appropriée. Une règle simple consiste à fournir un opérateur d'affectation pour toutes les classes qui possèdent un constructeur de copie personnalisé.

Précisons cette règle. Le constructeur de copie est utilisé lorsqu'un nouvel objet est créé. L'opérateur d'affectation est utilisé si l'objet situé à sa gauche existe déjà.

Surcharger l'opérateur d'affectation

Le programme qui suit permet de montrer comment on peut surcharger l'opérateur d'affectation. Il contient également un constructeur de copie pour fournir une comparaison.

```
// DemoAssignmentOperator - surcharge de l'opérateur d'affectation
//                          dans une classe définie par l'utilisateur
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

// Name - classe générique servant d'exemple
//        pour l'opérateur d'affectation et
//        le constructeur de copie
class Name
{
  public:
    Name(char *pszN = 0)
    {
        copyName(pszN, "");
    }
    Name(Name& s)
    {
        copyName(s.pszName, " (copie)");
    }
    ~Name()
    {
        deleteName();
    }

    // opérateur d'affectation
    Name& operator=(Name& s)
    {
        // supprime ce qui existe...
        deleteName();
        // ...avant de le remplacer par du neuf
        copyName(s.pszName, " (remplacement)");
        // renvoie une référence à l'objet existant
        return *this;
    }

    // fonction d'accès très simple
    char* out() { return pszName; }
```

```
  protected:
    void copyName(char* pszN, char* pszAdd);
    void deleteName();
    char *pszName;
};

// copyName() - alloue de la mémoire sur le tas pour
//              enregistrer le nom
void Name::copyName(char* pszN, char* pszAdd)
{
    pszName = 0;
    if (pszN)
    {
        pszName = new char[strlen(pszN) +
                           strlen(pszAdd)  + 1];
        strcpy(pszName, pszN);
        strcat(pszName, pszAdd);
    }
}

// deleteName() - libère la mémoire utilisée
void Name::deleteName()
{
    if (pszName)
    {
        delete pszName;
        pszName = 0;
    }
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // crée deux objets
    Name n1("Claudette");
    Name n2("Greg");
    cout << n1.out() << " et "
         << n2.out() << " sont de nouveaux objets"
         << endl;

    // fait maintenant une copie d'un objet
    Name n3(n1);
    cout << n3.out() << " est une copie de "
         << n1.out() << endl;

    // crée un nouvel objet avec le format "="
    // pour accéder au constructeur de copie
```

```
    Name n4 = n1;
    cout << n4.out() << " est aussi une copie de "
         << n1.out() << endl;

    // remplace n2 par n1
    n2 = n1;
    cout << n1.out() << " vient remplacer "
         << n2.out() << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La classe `Name` contient un pointeur vers un nom de personne. Le constructeur alloue la mémoire nécessaire sur le tas. Le constructeur et le destructeur de la classe `Name` sont semblables à ceux qui ont été présentés dans les Chapitres 17 et 18. Le constructeur `Name(char*)` (qui sert également de constructeur par défaut) copie le nom qui lui est passé dans le membre de données `pszName`. Le constructeur de copie `Name(&Name)` duplique le nom de l'objet passé dans celui de l'objet courant en appelant `copyName()`. Le destructeur libère la mémoire occupée par `pszName` en appelant `deleteName()`.

L'opérateur d'affectation `operator=()` est une méthode de la classe. Remarquez qu'il ressemble à un destructeur suivi d'un constructeur de copie. C'est aussi classique. Étudions l'affectation de notre exemple : `n2 = n1`. L'objet `n2` a déjà un nom qui lui est associé (`"Greg"`). Dans l'affectation, vous devez d'abord appeler `deleteName()` pour retourner au heap la mémoire occupée par le nom original. C'est exactement ce qui se passe pour un destructeur. C'est seulement après que vous pouvez appeler `copyName()` afin d'allouer de la mémoire pour y stocker le nouveau nom. Comme un constructeur de copie.

Le constructeur de copie n'a pas besoin d'appeler `deleteName()` car l'objet n'existe pas encore. De ce fait, la mémoire n'a pas encore été affectée à l'objet lorsque le constructeur a été appelé. Le destructeur n'a pas effectué la fonction de copie.

Voyons deux autres remarques concernant l'opérateur d'affectation. Premièrement, le type retourné par `operator=()` est `Name&`. Les expressions qui mettent en jeu cet opérateur ont une valeur et un type, tous deux fournis par la valeur finale de l'argument qui se trouve à main gauche. Dans l'exemple qui suit, la valeur de `operator=()` est 2.0 et son type est `double`.

```
double d1, d2;
void fn(double );
d1 = 2.0;          // la valeur de cette expression est 2.0
```

Il est donc parfaitement licite d'écrire :

```
d2 = d1 = 2.0
fn(d2 = 3.0);  // réalise l'affectation et passe
               // la valeur résultante à fn()
```

La valeur fournie par `d1 = 2.0` et son type (`double`) sont passés à `d2` lors de l'affectation. Dans le second exemple, la valeur de l'affectation (`d2 = 3.0`) est passée à la fonction `fn()`.

Passons au second détail. L'opérateur `operator=()` a été écrit sous la forme d'une fonction membre. L'argument de gauche correspond à l'objet courant (`this`). Contrairement à d'autres opérateurs, l'affectation ne peut pas être surchargée par une fonction non-membre.

Fournir une protection aux membres

Fournir un opérateur d'affectation à votre classe peut ajouter une souplesse considérable au code de votre application. Mais si cela représente trop de travail, ou si vous ne voulez pas que C++ fasse des copies de votre objet, la surcharge de cet opérateur par une fonction protégée évitera que quiconque puisse effectuer par erreur une copie de surface, membre à membre, sans y être autorisé. Par exemple :

```
class Name
{
  // ...comme auparavant...
  protected:
    // copy constructor
    Name(Name&) {}
    // opérateur d'affectation
    Name& operator=(Name& s) { return *this; }
};
```

Avec cette définition, des affectations telles que celle-ci sont évitées :

```
void fn(Name& n)
{
  Name newN;
  newN = n:  // génère une erreur de compilation car
             // la fonction n'a pas accès à op=().
}
```

La fonction `fn()` n'a pas accès à l'opérateur d'affectation désormais protégé. Cette astuce peut vous éviter les problèmes inhérents à la surcharge d'un opérateur d'affectation (mais vous y perdez en souplesse de programmation).

Si votre classe alloue des ressources (par exemple de la mémoire), vous *devez* soit écrire un constructeur de copie et votre propre opérateur d'affectation, soit les protéger tous deux pour empêcher l'utilisation des appels par défaut de C++.

Chapitre 24

Utiliser le flux E/S

Dans ce chapitre :

- Entrées et sorties.
- Le flux E/S comme opérateur surchargé.
- Utiliser un fichier de flux E/S.
- Utiliser un tampon de mémoire de flux E/S.
- En coulisse avec les manipulateurs.

Les programmes que nous avons développés jusqu'ici lisaient leurs entrées à partir de l'objet `cin` et effectuaient leurs sorties au travers de l'objet `cout`. Peut-être n'avez vous pas encore réfléchi spécialement à cela, mais cette technique forme un sous-ensemble de ce que l'on appelle le *flux E/S* (*E* pour entrée, *S* pour sortie).

Dans ce chapitre, je décris le flux d'entrée/sortie plus en détail. Je vous préviens d'emblée qu'il s'agit d'un domaine très vaste qui ne saurait être totalement couvert dans un seul chapitre ; des ouvrages entiers sont consacrés à ce seul sujet. Heureusement pour nous, et pour la grande majorité des programmes, vous n'avez pas besoin de tout savoir.

Plongée dans le flux E/S

Les opérateurs (surchargés !) qui définissent un flux d'entrée/sortie sont définis dans le fichier inclus `iostream` . Ce fichier comprend des prototypes de plusieurs fonctions `operator>>()` et `operator<<()`. Le code de ces instructions est inclus dans la bibliothèque standard à laquelle se relie votre programme C++ (et que nous avons d'ailleurs utilisée dans tous nos exemples). Voici quelques-uns des prototypes qui apparaissent dans `iostream` :

```
// Nous avons en entrée :
istream& operator>>(istream& source, char* pDest);
istream& operator>>(istream& source, int &dest);
istream& operator>>(istream& source, char &dest);
// ...et ainsi de suite...

// Nous avons en sortie :
ostream& operator<<(ostream& destination, char *pSource);
ostream& operator<<(ostream& destination, int source);
ostream& operator<<(ostream& destination, char source);
// ...et ce n'est pas fini...
```

Un peu de jargon : quand il est surchargé pour effectuer des entrées/sorties, `operator>>()` est appelé l'*extracteur* et `operator<<()` est appelé l'*injecteur*. La classe `istream` est la classe de base pour la lecture à partir d'un fichier ou d'un dispositif comme le clavier. C++ ouvre l'objet `istream` dénommé `cin` au démarrage du programme. De la même façon, `ostream` est la classe de base pour les sorties vers des fichiers. L'objet par défaut de `ostream` est `cout`.

Regardez ce qui se passe quand j'écris ceci :

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

void fn(ostream& out)
{
    out << "Mon nom est Stephen\n";
}
int main(int nNumberofArgs, char* pszArgs[])
{
    fn(cout);
    system("PAUSE");
    return 0;
}
```

Le programme passe `cout` à la fonction `fn()`. Celle-ci applique l'opérateur `<<`, également connu sous le nom `operator<<`.

Le C++ détermine que le meilleur prototype possible dans `iostream` est la fonction `operator<<(ostream&, char*)`. Il génère un appel à cette fonction (l'injecteur `char*`), en passant comme argument la chaîne `"Mon nom est Stephen\n"` ainsi que l'objet `ostream cout`. Ce qui se produit, c'est un appel à `operator<<(cout, "Mon nom est Stephen\n")`. La fonction d'injection `char*`, qui

fait partie de la bibliothèque standard C++, se charge d'effectuer la sortie demandée.

ostream et *istream* forment la base d'un ensemble de classes qui connectent le code de l'application au monde extérieur, y compris les entrées et les sorties vers le système de fichiers. Comment le compilateur a-t-il su que `cout` est de la classe `ostream` ? Cet objet, ainsi que quelques autres objets globaux, est aussi déclaré dans `iostream.h`. Les objets listés dans le Tableau 24.1 sont construits automatiquement au démarrage du programme, avant que le contrôle ne soit passé à `main()`. Des sous-classes de `ostream` et de `istream` servent aux échanges avec les fichiers et les tampons internes.

Tableau 24.1 : Les objets de flux d'entrée/sortie standard.

Objet	Classe	Utilisation
`cin`	`istream`	Entrée standard
`cout`	`ostream`	Sortie standard
`cerr`	`ostream`	Sortie d'erreur standard
`clog`	`ostream`	Sortie imprimante standard

Les sous-classes fstream

Les sous-classes `ofstream`, `ifstream` et `fstream` sont définies dans le fichier d'inclusion `fstream.h` ; elles servent à effectuer des entrées et des sorties de flux vers et depuis un fichier du disque dur. Ces trois classes se partagent plusieurs fonctions membres qui contrôlent les entrées et les sorties. Beaucoup sont héritées de `istream` et `ostream`. La liste complète se trouve dans la documentation de votre compilateur ; je vais néanmoins en présenter quelques-unes.

La classe `ofstream`, qui effectue les sorties de fichier, possède plusieurs constructeurs. Le plus utile est :

```
ofstream::ofstream(char *pszFileName,
                   int  mode = ios::out,
                   int  prot = filebuff::openprot);
```

Le premier argument est un pointeur vers le nom du fichier à ouvrir. Le deuxième et le troisième argument spécifient la façon dont le fichier sera

ouvert. Les valeurs légales de mode figurent dans le Tableau 24.2 et ceux de prot dans le Tableau 24.3. Ces valeurs sont des champs binaires auxquels un opérateur OR a été appliqué (revoyez à ce sujet le Chapitre 4). Les classes ios et filebuff sont toutes deux des classes parentes de ostream.

L'expression ios::out fait référence à un membre de données statique de la classe ios.

Tableau 24.2 : Valeurs de mode dans le constructeur ofstream.

Drapeau	Signification
ios::app	Ajoute à la fin de la ligne. Génère une erreur si le fichier n'existe pas.
ios::ate	Ajoute à la fin du fichier, s'il existe.
ios::in	Ouvre le fichier en entrée (implicite pour istream).
ios::out	Ouvre le fichier en sortie (implicite pour ostream).
ios::trunc	Tronque le fichier s'il existe (par défaut).
ios::nocreate	Retourne une erreur si le fichier n'existe pas déjà.
ios::noreplace	Retourne une erreur si le fichier existe.
ios::binary	Ouvre le fichier en mode binaire (alternative au mode texte).

Tableau 24.3 : Valeurs de prot dans le constructeur ofstream.

Drapeau	Signification
filebuf::openprot	Mode de partage de compatibilité
filebuf::sh_none	Exclusif ; pas de partage
filebuf::sh_read	Partage en lecture autorisé
filebuf::sh_write	Partage en écriture autorisé

Par exemple, le programme qui suit ouvre le fichier MonNom.txt puis écrit dedans des informations de première importance dont la véracité ne saurait être mise en cause :

```
// StreamOutput - exemple simple de sortie dans un fichier
#include <fstream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    ofstream my("MonNom.txt");
    my << "Stephen Davis est un type bien, un excellent camarade,\n"
       << "on peut dire un bel homme, et il n'est pas du tout chauve.\n"
       << endl;
    system("PAUSE");
    return 0;
}
```

Le constructeur `ofstream::ofstream(char*)` attend uniquement un nom de fichier. Il fournit des paramètres par défaut pour les autres modes de fichier. Si le fichier `MonNom.txt` existe déjà, il est tronqué. Autrement, `MonNom.txt` est créé. De plus, le fichier est ouvert en mode de partage de compatibilité.

Un deuxième constructeur, `ofstream::ofstream(char*, int)`, permet au programmeur de spécifier d'autres modes d'E/S pour un fichier. Par exemple, si je voulais ouvrir le fichier en mode binaire et ajouter le texte à la fin de ce fichier s'il existe déjà, je créerais l'objet `ostream` comme suit (en mode binaire, les nouvelles lignes ne sont pas converties en retours chariot et sauts de ligne en sortie, et inversement).

```
void fn()
{
  // ouverture du fichier binaire BINFILE pour y écrire ;
  // si ce fichier existe déjà, ajouter le texte à celui
  // existant
  ofstream bfile("BINFILE", ios::binary | ios::ate);

  // ...le programme continue...
}
```

Les objets de flux gèrent une information sur l'état du processus d'entrée/sortie. La fonction membre `bad()` retourne `TRUE` si quelque chose de "mauvais" se produit. C'est-à-dire si le fichier ne peut pas être ouvert, si tel ou tel objet interne est détérioré, ou si la situation tourne simplement très mal. À un niveau moins grave, l'erreur `fail()` indique soit que quelque chose de `bad()` s'est produit, soit que la dernière tentative de lecture a échoué. Si, par exemple, vous essayez de lire un `int` alors que le programme ne peut trouver qu'un `char`, l'erreur se situe au niveau `fail()`, mais pas `bad()`.

La fonction membre `good()` retourne `TRUE` si `bad()` et `fail()` sont tous deux faux (`FALSE`).

La fonction membre `clear()` remet à zéro le drapeau d'erreur pour vous laisser une autre chance.

Le listing qui suit ajoute un contrôle d'erreur (basique) à notre programme précédent :

```
// StreamOutputWithErrorChecking - exemple simple de
//                                 sortie dans un fichier
#include <fstream>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    const static char fileName[] = "MonNom.txt";
    ofstream my(fileName);
    if (my.bad())        // si l'ouverture ne fonctionne pas...
    {
        cerr << "Erreur en ouverture du fichier "
             << fileName
             << endl;
        return 0;            //...on le signale et on quitte
    }
    my << "Stephen Davis est un type bien, un excellent camarade,\n"
       << "on peut dire un bel homme, et il n'est pas du tout chauve.\n"
       << endl;
    if (my.bad())
    {
        cerr << "Erreur d'écriture dans le fichier "
             << fileName
             << endl;
    }
    system("PAUSE");
    return 0;
}
```

Toute tentative de sortie vers un objet `ofstream` qui a une erreur n'a aucun effet si `my.bad()` renvoie vrai.

Le dernier paragraphe signifie qu'aucune sortie n'est possible tant que le drapeau d'erreur n'est pas réinitialisé. Le programme n'essaiera même pas de recommencer l'opération tant que vous n'aurez pas appelé `clear()`.

Le destructeur de la classe `ofstream` ferme automatiquement le fichier. Dans l'exemple précédent, il a été fermé au moment où la fonction s'est terminée.

La classe `ifstream` fonctionne à peu près de la même façon pour les entrées, comme le démontre cet exemple :

```
// StreamInput - exemple simple de lecture à partir
//               d'un fichier en utilisant fstream
#include <fstream>
#include <iostream>
using namespace std;

ifstream* openFile()
{
    ifstream* pFileStream = 0;
    for(;;)
    {
        // ouvre le fichier spécifié par l'utilisateur
        char fileName[80];
        cout << "Entrez le nom d'un fichier contenant des entiers"
             << endl;
        cin >> fileName;

        // ouvre le fichier en lecture ; ne le crée pas
        // s'il n'existe pas encore
        pFileStream = new ifstream(fileName);
        if (pFileStream->good())
        {
            break;
        }
        cerr << "Impossible d'ouvrir " << fileName << endl;
        delete pFileStream;
    }
    return pFileStream;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // demande un canal pour l'accès aux fichier
    ifstream* pFileStream = openFile();

    // arrête s'il n'y a plus de données dans le fichier
    while (!pFileStream->eof())
    {
        // lit une valeur
        int nValue = 0;
        (*pFileStream) >> nValue;
```

```
        // stoppe si la lecture échoue (probablement parce que
        // l'entrée suivante n'est pas un entier, ou parce que
        // nous avons trouvé une nouvelle ligne sans rien après
        if (pFileStream->fail())
        {
            break;
        }

        // sort la valeur qui vient d'être lue
        cout << nValue << endl;
    }

    system("PAUSE");
    return 0;
}
```

La fonction `openFile()` demande à l'utilisateur d'entrer le nom du fichier à ouvrir. La fonction crée un objet `ifstream` portant le nom spécifié. Le fait de créer un tel objet ouvre automatiquement le fichier en lecture. Si tout s'est bien passé, la fonction renvoie un pointeur vers l'objet `ifstream`. Dans le cas contraire, le programme détruit l'objet et refait une nouvelle tentative. La seule façon de sortir de la boucle est soit de donner un nom de fichier valide, soit de quitter le programme.

Il ne faut surtout pas oublier de supprimer l'objet `pFileStream` si l'ouverture échoue. Sinon, vous provoquerez l'une de ces fameuses fuites de mémoire.

Le programme lit des valeurs entières à partir de l'objet pointé par `pFileStream`. Il continue jusqu'à ce que le drapeau `fail()` soit activé ou que l'on atteigne la fin du fichier, ce qui est signalé par la fonction membre `eof()`. Tenter de lire un objet `ifstream` dont le drapeau d'erreur est levé à la suite d'une erreur antérieure provoque un retour immédiat sans aucune lecture.

Insistons une fois de plus. Non seulement une tentative de lecture d'un flux d'entrée qui contient une erreur ne renvoie rien, mais de surcroît le tampon reste inchangé. Le programme peut donc en arriver à la conclusion erronée qu'il vient de relire la même valeur qu'avant. De ce fait, le marqueur de fin de fichier (`eof()`) ne sera jamais activé si le flux d'entrée contient une erreur.

La sortie produite par ce programme ressemble à ceci (ce qui est saisi par l'utilisateur est mis en caractères gras) :

```
Entrez le nom d'un fichier contenant des entiers
poulet
Impossible d'ouvrir poulet
Entrez le nom d'un fichier contenant des entiers
entiers.txt
1
2
3
4
5
6
7
Appuyez sur une touche pour continuer...
```

Lecture directe de flux

Les opérateurs d'extraction et d'injection fournissent un mécanisme commode pour la lecture d'entrées formatées. Mais il existe bien des situations dans lesquelles vous voulez simplement dire : "Donne-le moi quand même, le format ne m'intéresse pas."

Dans un tel contexte, vous disposez de deux méthodes. La fonction `getline()` retourne une chaîne de caractères jusqu'à ce qu'elle rencontre une terminaison quelconque (il s'agit par défaut du retour chariot). Elle supprime ce caractère de fin sans essayer de reformater ou d'interpréter l'entrée.

La fonction membre `read()` est encore plus fondamentale. Elle lit le nombre de caractères que vous avez spécifié, voire moins si le programme rencontre une fin de fichier. La fonction `gcount()` retourne alors la quantité de caractères effectivement lus.

Le programme qui suit utilise à la fois `getline()` et `read()` pour ouvrir un fichier ayant un contenu tout à fait quelconque et envoyer tout cela vers l'affichage :

```
// FileInput - lit des blocs de données dans un fichier
#include <fstream>
#include <iostream>
using namespace std;

ifstream* openFile(istream& input)
{
    for(;;)
```

```
    {
        // ouvre le fichier spécifié par l'utilisateur
        char fileName[80];
        cout << "Entrez le nom d'un fichier" << endl;

        // lit la saisie faite par l'utilisateur de façon
        // à ne pas provoquer de débordement du tampon clavier
        input.getline(fileName, 80);

        // ouvre le fichier en lecture ; ne crée pas le fichier
        // s'il n'existe pas déjà
        ifstream* pFileStream = new ifstream(fileName);
        if (pFileStream->good())
        {
            return pFileStream;
        }
        cerr << "Impossible d'ouvrir " << fileName << endl;
    }
    return 0;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // ouvre un flux d'entrée
    ifstream* pFileStream = openFile(cin);

    // lit des données par blocs de 80 octets
    char buffer[80];
    while (!pFileStream->eof() && pFileStream->good())
    {
        // lit un bloc - 80 est le maximum mais gcount()
        // renvoie le nombre réel d'octets lus
        pFileStream->read(buffer, 80);
        int noBytes = pFileStream->gcount();

        // fait quelque chose avec le bloc
        for(int i = 0; i < noBytes; i++)
        {
            cout << buffer[i];
        }
    }

    system("PAUSE");
    return 0;
}
```

Le programme invoque d'abord `openFile()` pour ouvrir un fichier. Cette version illustre deux points intéressants. Tout d'abord, la fonction lit un objet `istream` exactement comme elle le ferait à partir de `cin`. En fait, la fonction `main()` passe l'objet `cin` à `openFile()`. L'intérêt de cette démarche est qu'une fonction utilisant un objet `istream` arbitraire peut lire le contenu de fichiers sans aucune modification.

Dans `openFile()`, nous faisons appel à la fonction membre `getline()` pour lire une chaîne. L'un des arguments de cette fonction est la taille du tampon. La fonction `getline()` n'effectuera pas de lecture au-delà de ce maximum. Le résultat est placé dans le tampon de caractères `fileName`.

L'utilisation de `getline()` pour lire une saisie au clavier est plus sûre que celle de l'extracteur, qui récupère un simple tableau de caractères et peut continuer la lecture au-delà de la fin de celui-ci. La fonction `getline()` ne peut pas dépasser le nombre de caractères que vous spécifiez.

La fonction `main()` lit des blocs de quatre-vingts caractères provenant de l'objet flux d'entrée retourné par `openFile()`. Le programme contrôle le nombre effectif de caractères de chaque bloc en appelant la fonction `gcount()`. Ce nombre ne peut pas dépasser les quatre-vingts caractères spécifiés dans l'appel à `read()` et il ne sera inférieur que dans le cas où la fin du fichier a été rencontrée. Le programme utilise l'injecteur standard pour afficher le contenu des blocs.

Le programme `FileInput` affiche simplement le contenu du fichier dont vous entrez le nom :

```
Entrez le nom d'un fichier
MonNom.txt
Stephen Davis est un type bien, un excellent camarade,
on peut dire un bel homme, et il n'est pas du tout chauve.

Appuyez sur une touche pour continuer...
```

À propos de endl

La plupart des programmes de ce livre terminent un flux de sortie en insérant l'objet `endl`. En revanche, d'autres affichages envoient un saut de ligne en envoyant un `\n` à l'intérieur du texte à sortir.

Le symbole `\n` représente le caractère de changement de ligne. L'expression `cout << "Voici une ligne\npuis une autre ligne"` affiche deux lignes différentes. L'objet `endl` provoque le même résultat, mais en allant un peu plus loin.

Les disques sont des dispositifs lents (enfin, relativement). Ecrire plus souvent qu'il n'est nécessaire sur un disque peut ralentir considérablement vos programmes. Pour éviter cela, la classe `fstream` place les données à sortir dans un tampon interne. Elle écrit sur le disque le contenu de ce tampon lorsqu'il est plein (on dit alors qu'il est vidé). L'objet `endl`, justement, effectue automatiquement ce vidage. Vous pouvez d'ailleurs faire à tout moment la même chose en appelant la fonction membre `flush()`.

Utiliser les sous-classes strstream

Les classes de flux offrent au programmeur des mécanismes pour décomposer une entrée en variables de type entier, flottant, tableau de caractères, et ainsi de suite. Il existe des classes permettant au programme de "lire" un tableau de caractères en mémoire (c'est en quelque sorte un *flux de chaîne*, à ne pas confondre avec un fût en chêne). Les classes `istringstream` et `ostringstream` sont définies dans le fichier inclus `sstream`.

Les anciennes versions de ces classes sont `istrstream` et `ostrstream`. Elles sont définies dans le fichier inclus `strstream`.

La sémantique employée ici est la même que dans le cas des classes de fichiers correspondantes. Voyons cela dans un exemple qui traite des informations concernant des comptes bancaires lues dans un fichier :

```
// StringStream - lecture et traitement du contenu d'un fichier
#include <fstream>
#include <sstream>
#include <iostream>
using namespace std;

// parseAccountInfo - lit le tampon passé comme s'il s'agissait
//                    d'un véritable fichier - le format se
//                    présente ainsi :
//                    nom, <numéro de compte> <solde du compte>
//                    retourne true si tout s'est bien passé
bool parseString(char* pString, char* pName, int arraySize,
                 long& accountNum, double& balance)
{
    // associe un objet istrstream à la chaîne
    // de caractères reçue en argument
    istringstream inp(pString);

    // lecture jusqu'à la virgule
    inp.getline(pName, arraySize, ',');
```

```
    // et maintenant le numéro du compte
    inp >> accountNum;

    // puis le solde
    inp >> balance;

    // retourne l'état d'erreur courant
    return !inp.fail();
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // ouvre un flux en entrée (fichier)
    ifstream* pFileStream = new ifstream("Comptes.txt");
    if (!pFileStream->good())
    {
        cout << "Impossible d'ouvrir Comptes.txt" << endl;
        return 0;
    }

    // lit une ligne du fichier, la traite et
    // affiche le résultat
    for(;;)
    {
        // ajouter une ligne de séparation
        cout << "=========================" << endl;
        // lit un tampon
        char buffer[256];
        pFileStream->getline(buffer, 256);
        if (pFileStream->fail())
        {
            break;
        }

        // analyse les champs individuels
        char name[80];
        long accountNum;
        double balance;
        bool result = parseString(buffer, name, 80,
                                  accountNum, balance);

        // sort le résultat
        cout << buffer << "\n";
        if (result == false)
        {
            cout << "Erreur de traitement du contenu\n";
```

```
            continue;
        }
        cout << "nom = " << name << ", "
             << "compte = " << accountNum << ", "
             << "solde = " << balance << endl;

        // replace les champs dans un ordre différent
        // (l'insertion de 'ends' s'assure que le
        // tampon est terminé par le caractère null
        ostringstream out;
        out << name << ", "
            << balance << " "
            << accountNum << ends;

        // sort le résultat - istringstream fonctionne aussi
        // avec la classe string mais je continue pour l'instant
        // à travailler avec des tableaux de caractères
        string oString = out.str();
        cout << oString << "\n" << endl;
    }

    system("PAUSE");
    return 0;
}
```

Le programme commence par ouvrir un fichier appelé `Comptes.txt` (également disponible en téléchargement !). Les informations sur les comptes y sont enregistrées sous la forme `nom, numéro_de_compte, solde, \n`. En supposant que le fichier ait été ouvert avec succès, le programme entre dans une boucle en lisant les lignes jusqu'à ce que tout le contenu du fichier ait été traité.

L'appel à `getline()` effectue une lecture jusqu'à la fin de ligne par défaut. Le programme passe la ligne courante à la fonction `parseString()`.

`parseString()` associe un objet `istringstream` à la chaîne de caractères. À l'aide de la fonction membre `getline()`, elle lit les caractères jusqu'à ce qu'elle trouve une virgule (ou que la fin du tampon soit atteinte). Le programme utilise alors les extracteurs standard pour lire les informations sur le numéro du compte (`accountNum`) et le solde (`balance`). Si `inp.fail()` retourne False, c'est que tout a bien marché.

Après l'appel à `parseString()`, `main()` sort le tampon lu à partir du fichier, suivi des valeurs qui ont été analysées. Il utilise ensuite la classe `ostringstream` pour reconstruire un objet `string` montrant les mêmes données, mais présentées différemment.

Voyons à quoi ressemble l'exécution de ce programme :

```
==========
Chester, 12345 56.60
nom = Chester, compte = 12345, solde = 56.6
Chester, 56.6 12345

==========
Arthur,  34567 67.50
nom = Arthur, compte = 34567, solde = 67.5
Arthur, 67.5 34567

==========
Trudie,  56x78 78.90
Erreur de traitement du contenu
==========
Valerie, 78901 89.10
nom = Valerie, compte = 78901, solde = 89.1
Valerie, 89.1 78901

==========
Appuyez sur une touche pour continuer...
```

Réfléchissez une seconde avant de continuer. Remarquez comment le programme a su se resynchroniser après avoir rencontré une erreur dans le fichier d'entrée. Notez aussi combien le cœur du programme est simple. Imaginez à quoi pourrait bien ressembler la fonction `parseString()` si la classe `istringstream` n'existait pas.

La manipulation des manipulateurs

Vous pouvez utiliser le flux d'entrée/sortie pour sortir des nombres et des chaînes de caractères dans leurs formats par défaut. En général, cette méthode convient très bien, mais ce n'est pas toujours le cas.

Par exemple, je n'ai pas vraiment apprécié de voir que le total d'un calcul financier dans un de mes programmes affichait 249.600006 au lieu de 249.6 (ou mieux encore, 249.60). Il doit y avoir un moyen d'obliger le format par défaut à s'afficher comme je le veux... Le C++ ne propose pas un seul moyen pour contrôler le format de sortie, mais deux.

Selon la configuration par défaut de votre compilateur, vous obtiendrez peut-être 249.6 comme sortie alors que vous souhaitez afficher 249.60.

Tout d'abord, le format peut être contrôlé en appelant une série de fonctions membres sur l'objet flux. Par exemple, le nombre de chiffres significatifs à afficher est défini comme suit à l'aide de la fonction `precision()` :

```
#include <iostream>
void fn(float interest, float dollarAmount)
{
  cout << "Montant en dollars = ";
  cout.precision(2);
  cout << dollarAmount;
  cout.precision(4);
  cout << interest
    << "\n";
}
```

Dans cet exemple, la fonction `precision()` définit une précision sur deux décimales juste avant de sortir la valeur `amountDollar`. Elle permet d'obtenir le nombre 249.60, celui que vous cherchiez à afficher. Elle met ensuite la précision à quatre décimales avant de sortir les intérêts.

Une deuxième approche est basée sur ce que l'on appelle des manipulateurs (un nom qui fait penser aux gens qui grenouillent en coulisses à la bourse, n'est-ce pas ? Mais en C++, les manipulateurs sont plus malins...). Les *manipulateurs* sont des objets définis dans le fichier d'inclusion `iomanip` afin de produire le même effet que des appels à une fonction membre (vous devez impérativement inclure `iomanip` pour avoir accès aux manipulateurs). Le seul avantage des manipulateurs, c'est que le programme peut les insérer directement dans le flux au lieu de recourir à un appel de fonction distinct.

Si vous réécrivez l'exemple précédent mais avec des manipulateurs, le programme apparaîtra sous cette forme :

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

void fn(float interest, float dollarAmount)
{
  cout << "Montant en dollars = "
    << setprecision(2) << dollarAmount
    << "\nTaux d'intérêt = "
    << setprecision(4) << interest
    << "\n";
}
```

Le Tableau 24.4 présente les manipulateurs les plus courants avec leur signification.

Tableau 24.4 : Manipulateurs et fonctions de contrôle de format de flux courants.

Manipulateur	Fonction membre	Description
`dec`	`flags(10)`	Met la base en décimal.
`hex`	`flags(16)`	Met la base en hexadecimal.
`oct`	`flags(8)`	Met la base en octal.
`setfill(c)`	`fill(c)`	Définit "c" comme caractère de remplissage.
`setprecision(c)`	`precision(c)`	Met la précision de l'affichage à c.
`setw(n)`	`width(n)`	Définit une largeur de champ de n caractères*.

* Revient à sa valeur par défaut après la sortie du prochain champ.

Soyez prudent avec le paramètre de taille de champ du manipulateur `setw()` et de la fonction `width()`. La plupart des paramètres conservent leur valeur jusqu'à ce qu'ils soient spécifiquement réinitialisés par un nouvel appel, mais ce n'est pas le cas de celui-ci. Il reprend sa valeur par défaut dès que la sortie qui suit se termine. Vous pourriez par exemple supposer que ce code sort deux entiers formatés sur huit caractères :

```
#include <iostream>
#include <iomanip>
void fn()
{
  cout << setw(8)   // la largeur est 8...
       << 10        // ...pour l'entier 10,...
       << 20        // ...mais 20 sort dans le format par défaut
       << "\n";
}
```

Ce que vous voyez en réalité, c'est un entier de huit caractères suivi par un entier de deux caractères. Pour obtenir partout des sorties sur huit caractères, vous devez coder le programme ainsi :

```
#include <iostream>
#include <iomanip>
void fn()
{
  cout << setw(8)   // fixe la largeur...
       << 10
       << setw(8)   //...et la réinitialise.
       << 20
       << "\n";
}
```

Si vous avez plusieurs objets à sortir et que la largeur par défaut ne convient pas, vous devez donc placer un appel à `setw()` pour chaque objet.

Quelle est la meilleure technique ? Les manipulateurs ou les appels aux fonctions membres ? Les fonctions membres donnent un contrôle un peu plus étendu car elles sont plus nombreuses. De plus, les fonctions membres retournent toujours leur configuration antérieure, de sorte que vous savez comment les restaurer (si vous le désirez). Enfin, il existe pour chaque fonction membre une version "requête" qui vous permet de connaître son paramétrage courant sans avoir à le modifier, comme le montre l'exemple suivant :

```
#include <iostream>
void fn(float value)
{
  int precisionPrecedente;
  // ...divers trucs à faire ici...
  // vous pouvez demander ce qu'est la précision courante :
  precisionPrecedente = cout.precision();

  // ou alors, vous pouvez sauvegarder l'ancienne
  // valeur après l'avoir modifiée :
  precisionPrecedente = cout.precision(2);
  cout << value;

  // et maintenant, restauration de la précision à la
  // valeur précédente :
  cout.precision(precisionPrecedente);


  // ...on continue le travail...

}
```

Même avec toutes ces fonctionnalités, les manipulateurs sont couramment employés, probablement parce qu'ils sont clairs et nets. Utilisez ce que vous préférez, mais attendez-vous à voir les deux techniques présentes dans les programmes d'autrui.

Chapitre 25

Gestion des erreurs : les exceptions

Dans ce chapitre :

- Un remarquable moyen de gestion des erreurs d'un programme.
- Trouver ce qui ne va pas à l'aide des bons vieux retours d'erreur.
- Examiner les exceptions : lancement et interception.
- Et on en jette plus encore.

Je sais que c'est dur à admettre, mais il arrive parfois que des fonctions ne fonctionnent pas, même les miennes. La technique classique, pour signaler un dysfonctionnement, consiste à retourner des informations à l'appelant. Le C++ intègre un nouveau mécanisme d'interception et de gestion des erreurs perfectionné : les *exceptions*. Une exception, c'est littéralement un cas dans lequel une règle ou un principe ne s'applique pas. Les exceptions sont aussi une objection à un événement. Les deux définitions sont pertinentes : une exception est une condition inattendue (et présumée inacceptable) qui se produit au cours de l'exécution du programme.

Le mécanisme d'exception est basé sur les mots-clés `try`, `catch` et `throw` (ce sont autant de mots réservés que vous ne pourrez pas utiliser comme variables). En gros, ils agissent ainsi : une fonction *essaie* (`try`) d'englober une certaine partie de code. Si un problème est détecté, le code *lance* (`throw`) un message d'erreur que la fonction appelante doit *intercepter* (`catch`).

Le petit listing qui suit montre le principe de l'exception :

```
//  FactorialException - calcul de factorielle gérant
//                       les exceptions
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// factorial - calcul de factorielle
int factorial(int n)
{
    // des valeurs négatives de n ne peuvent pas être gérées ;
    // il vaut donc mieux vérifier d'abord cette condition
    if (n < 0)
    {
        throw string("Argument de factorielle plus petit que 0");
    }

    // on continue le calcul de la factorielle
    int accum = 1;
    while(n > 0)
    {
        accum *= n;
        n--;
    }
    return accum;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    try
    {
        // ceci fonctionne
        cout << "Factorielle de 3 vaut " << factorial(3) << endl;

        // mais cela génère une exception
        cout << "Factorielle de -1 vaut " << factorial(-1) << endl;

        // et ceci ne sera jamais exécuté
        cout << "Factorielle de 5 vaut " << factorial(5) << endl;
    }
    //  le contrôle passe par ici
    catch(string error)
    {
        cout << "Interception de l'erreur : " << error << endl;

    }
```

```
    catch(...)
    {
        cout << "Interception standard " << endl;
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La fonction `main()` commence par créer un bloc doté du mot-clé `try`. Elle s'y déplace, comme si ce n'était pas un bloc mais du code tout à fait normal, et essaie de calculer la factorielle d'un nombre négatif. Pas du tout induite en erreur, l'intelligente fonction `factorial()` détecte le problème et émet une indication d'erreur à l'aide du mot-clé `throw`. Le contrôle est transmis à l'instruction `catch` qui se trouve juste après l'accolade fermée du bloc `try`. L'appel suivant à `factorial()` n'est pas exécuté.

Un nouveau mécanisme pour les erreurs ?

Qu'est-ce qui ne va pas dans les messages d'erreur qui ressemblent à ceux que retournait le vieux langage Fortran ? Comme les factorielles ne peuvent pas être négatives, j'aurais pu dire : "OK, si `factorial()` détecte une erreur, il retournera un nombre négatif. La vraie valeur révélera la source du problème". En quoi est-ce erroné ? C'est ce qui s'est toujours fait depuis la nuit des temps...

Malheureusement, plusieurs problèmes se manifestent. Premièrement, il est certes vrai qu'une factorielle ne peut être négative, mais la question n'est pas aussi évidente avec d'autres fonctions. Ainsi, vous ne pouvez pas non plus obtenir le logarithme d'un nombre négatif, mais le coup du retour d'une valeur négative n'est pas valide ici car un logarithme *peut* être positif ou négatif.

Deuxièmement, un entier peut être associé à une foule d'informations. Vous pouvez par exemple obtenir -1 pour "l'argument est négatif" et -2 pour "l'argument est trop grand". Mais si l'argument est trop grand, il serait intéressant de savoir quelle était sa valeur, car ceci contribuerait utilement à la mise au point du programme. Or, il n'y a pas de place pour stocker ce type de renseignement.

Troisièmement, le traitement des retours d'erreur est facultatif. Supposons que quelqu'un écrive la fonction `factorial()` de manière à ce qu'elle vérifie dûment l'argument et retourne un nombre négatif si celui-ci est hors limites. Si le code qui appelle cette fonction ne vérifie pas l'éventuel retour d'une erreur, tout cela

n'aura servi à rien. Je sais bien que, quoique j'insiste sur l'absolue nécessité de rechercher les erreurs, ce ne sont pas mes paroles qui vous obligeront à le faire. Une fois les bonnes intentions affichées, on retombe tellement vite dans les mauvaises habitudes...

Même si vous vérifiez les erreurs retournées par `factorial()` ou toute autre fonction, que pourra bien faire ensuite votre code ? Probablement rien de plus qu'afficher un message de votre cru et retourner une autre indication d'erreur à la fonction appelante, qui en fera vraisemblablement autant. Au bout d'un moment, le code commence à prendre cette allure :

```
// appel d'une fonction, vérification d'une d'erreur
// éventuelle, gestion de celle-ci puis retour
errRtn = someFunc();
if (errRtn)
{
  errorOut ("Erreur à l'appel de someFunc()");
  return MY_ERROR_1;
}
errRtn = someOtherFunc();
if (errRtn)
{
  errorOut ("Erreur à l'appel de someOtherFunc()");
  return MY_ERROR_1;
}
```

Ce mécanisme pose plusieurs problèmes :

- Il est très répétitif.
- Il oblige l'utilisateur à inventer et garder trace des nombreux retours d'indications d'erreurs.
- Il mêle le code issu de la gestion d'erreurs au flux de code normal, ce qui embrouille le cheminement logique, correct, dépourvu d'erreur, du programme.

Dans cet exemple simple, ces problèmes ne semblent pas entraîner de complications. Mais la situation change lorsque les appels deviennent de plus en plus complexes. Le résultat, c'est que le code déjà écrit pour la gestion des erreurs n'est plus en mesure de traiter toutes les conditions qui pourraient se présenter.

Le mécanisme d'exception corrige ces défauts en séparant le déroulement d'une section susceptible de provoquer des erreurs du cours normal du code.

De plus, les exceptions rendent la gestion des erreurs obligatoire. Si votre fonction ne gère pas l'exception de lancement, le contrôle passe en revue toute la chaîne des fonctions appelées et ce, jusqu'à ce que le C++ trouve une fonction capable de gérer l'erreur. Ce procédé vous offre de surcroît une souplesse accrue, puisqu'il est possible d'ignorer les erreurs contre lesquelles vous ne pourriez de toute façon rien faire. Seules les fonctions susceptibles de corriger effectivement le problème devront pouvoir intercepter l'exception.

Analyse du mécanisme d'exception

Nous allons examiner de plus près les différentes étapes que le code met en œuvre pour gérer une exception. Lorsqu'un lancement se produit (*throw*), le C++ copie d'abord l'objet ainsi "jeté" dans un emplacement neutre. Il commence ensuite à rechercher la fin du bloc *try* courant.

Si aucun bloc d'essai n'a été trouvé dans la fonction courante, le contrôle passe à la fonction appelante. Une recherche est ensuite effectuée sur cette fonction. Si aucun bloc d'essai n'y a été trouvé, le contrôle passe à la fonction qui l'avait appelé, et ainsi de suite en remontant la pile des fonctions appelantes. Ce processus est appelé *dépilage*.

Il y a un point très important dans ce processus : lorsque chaque niveau est "dépilé", tous les objets qui deviennent hors de portée sont détruits, exactement comme si la fonction avait exécuté une instruction de retour. Ceci évite au programme de perdre des éléments ou de laisser des objets à l'abandon (ou encore pendants).

Lorsque le bloc d'essai a été trouvé, le code recherche la première occurrence de la phrase `catch` (interception) juste après l'accolade fermée du bloc. Si l'objet lancé correspond au type de l'argument spécifié dans l'instruction d'interception, le contrôle passe à la phrase `catch`. Sinon, une vérification est faite sur la prochaine occurrence de `catch`. Si aucune correspondance n'est trouvée, le code recherche le prochain bloc d'essai au niveau immédiatement supérieur, comme dans une sorte de spirale excentrique, jusqu'à ce qu'un `catch` approprié puisse être trouvé. En cas d'échec, le programme s'achève.

Étudions l'exemple suivant :

```
// CascadingException - le programme qui suit peut générer
//                      des avertissements car les variables
//                      f, i and pMsg se sont pas utilisées
//                      - le compilateur essaie de vous donner
//                      un conseil : peut-être ces arguments
```

```
//                          sont-ils en trop
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Obj
{
  public:
    Obj(char c)
    {
        label = c;
        cout << "Construction de l'objet " << label << endl;
    }
    ~Obj()
    {
        cout << "Destruction de l'objet " << label << endl;
    }

  protected:
    char label;
};

void f1();
void f2();
int f3()
{
    Obj a('a');
    try
    {
        Obj b('b');
        f1();
    }
    catch(float f)
    {
        cout << "Interception de Float" << endl;
    }
    catch(int i)
    {
        cout << "Interception de Int" << endl;
    }
    catch(...)
    {
        cout << string("Interception générique") << endl;
    }
}
```

```
int main(int nNumberofArgs, char* pszArgs[])
{
    f3();

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}

void f1()
{
    try
    {
        Obj c('c');
        f2();
    }
    catch(string msg)
    {
        cout << "Interception de String" << endl;
    }
}
void f2()
{
    Obj d('d');
    throw 10;
}
```

La sortie de ce programme donne :

```
Construction de l'objet a
Construction de l'objet b
Construction de l'objet c
Construction de l'objet d
Destruction de l'objet d
Destruction de l'objet c
Destruction de l'objet b
Interception de Int
Destruction de l'objet a
Appuyez sur une touche pour continuer...
```

Vous constatez d'abord que les quatre objets `a`, `b`, `c` et `d` sont construits au fur et à mesure que le contrôle passe par chacune des déclarations avant que `f2()` ne lance l'entier `int` 10. Comme aucun bloc d'essai n'a été défini dans `f2()`, le C++ dépile `f2()`, provoquant la destruction de l'objet `d`. La fonction `f1()`

définit un bloc d'essai, mais sa seule phrase d'interception (`catch`) a été conçue pour gérer des chaînes, ce qui ne correspond pas du tout avec notre `int`, l'entier qui a été lancé. Par conséquent, le C++ continue sa recherche. Il dépile `f1()`, ce qui entraîne la destruction de l'objet `c`.

De retour dans `f3()`, le C++ découvre un autre bloc d'essai. La sortie de ce bloc fait que l'objet `b` se retrouve hors de portée. Il est donc détruit à son tour. La première phrase `catch` est censée intercepter les flottants qui ne correspondent pas aux entiers `int` ; elle est donc sautée. La prochaine phrase `catch` correspond exactement à `int` ; le contrôle s'y arrête donc. La dernière phrase `catch`, qui intercepte en principe tout objet lancé, est elle aussi sautée car une phrase `catch` correspondante a d'ores et déjà été trouvée.

Quelles sortes de choses puis-je lancer ?

Les éléments qui suivent le mot-clé `throw` forment en fait une expression qui crée un objet d'un certain genre. Dans les exemples précédents, nous avons lancé un entier et une chaîne, mais `throw` peut lancer n'importe quel type d'objet. C'est-à-dire que vous pouvez reprendre pratiquement autant d'informations que vous le désirez. Regardez cette version plus évoluée de notre calcul de factorielles :

```
//  CustomExceptionClass - exceptions et factorielles
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <sstream>
using namespace std;

// Exception - classe générique pour la gestion des exceptions
class Exception
{
  public:
    Exception(char* pMsg, int n, char* pFile, int nLine)
      : msg(pMsg), errorValue(n), file(pFile), lineNum(nLine)
    {}

    virtual string display()
    {
        ostringstream out;
        out << "Erreur <" << msg
            << " - la valeur est " << errorValue
            << ">\n";
```

```
        out << " @" << file << " - Ligne " << lineNum << endl;
        return out.str();
    }
  protected:
    // message d'erreur
    string msg;
    int    errorValue;

    // nom du fichier et numéro de la ligne
    // où l'erreur s'est produite
    string file;
    int lineNum;
};

// factorial - calcule une factorielle
int factorial(int n)
{
    // n ne peut pas être négatif;
    // il vaut mieux contrôler d'abord cette condition
    if (n < 0)
    {
        throw Exception("Argument de factorielle < 0",
                        n,  __FILE__, __LINE__);
    }

    // on continue le calcul de la factorielle
    int accum = 1;
    while(n > 0)
    {
        accum *= n;
        n--;
    }
    return accum;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    try
    {
        // cela fonctionne
        cout << "Factorielle de 3 vaut " << factorial(3) << endl;

        // ceci va générer une exception
        cout << "Factorielle de -1 vaut " << factorial(-1) << endl;
    }
    // le contrôle passe par là
    catch(Exception e)
```

```
    {
        cout << "ERREUR : \n" << e.display() << endl;
    }

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

La différence principale entre ce programme et celui qui a été proposé en début de chapitre, c'est la présence de la classe utilisateur `Exception` qui contient davantage d'informations sur la nature de l'erreur qu'une simple chaîne de caractères. Cette version est capable de lancer le message d'erreur, la valeur illégale et l'emplacement exact où l'erreur s'est produite.

`__FILE__` et `__LINE__` sont des déclarations `#define` intrinsèques qui correspondent respectivement au nom du fichier source et au numéro de ligne courant dans ce fichier.

L'interception `catch` s'empare de l'objet `Exception` puis utilise la fonction membre intégrée `display()` pour afficher le message d'erreur.

Voici la sortie produite par le programme :

```
Factorielle de 3 vaut 6
ERREUR :
Erreur <Argument de factorielle < 0 - la valeur est -1>
 @C:/Dev-Cpp/Nuls/Chap25/CustomExceptionClass.cpp - Ligne 45

Appuyez sur une touche pour continuer...
```

Bien entendu, le chemin d'accès au fichier peut être différent chez vous.

`Exception` représente une classe de rapport d'erreur générique. Vous pouvez toutefois la sous-classer afin de fournir davantage de détails sur un type d'erreur particulier. Par exemple, je peux définir une classe `InvalidArgumentException` qui mémorise la valeur d'un argument invalide en plus du message et de l'emplacement de l'erreur :

```
class InvalidArgumentException : public Exception
{
  public:
```

```
    InvalidArgumentException(int arg,
                             char* pFile,
                             int nLine)
          : Exception("Argument invalide", pFile, nLine)
    {
        invArg = arg;
    }

    virtual void display(ostream& out)
    {
        Exception::display(out);
        out << "Valeur de l'argument " << invArg << endl;
    }

  protected:
    int invArg;
};
```

La fonction appelante gère automatiquement la nouvelle classe `InvalidArgumentException` car une "exception d'argument invalide" est une `Exception` et parce que la fonction membre `display()` est polymorphe.

Chapitre 26

L'héritage multiple est-il une bonne affaire ?

Dans ce chapitre :

- Présentation de l'héritage multiple.
- Éviter les ambiguïtés de l'héritage multiple.
- Éviter les ambiguïtés de l'héritage virtuel.
- Règles d'ordre pour constructeurs multiples.
- Les problèmes de l'héritage multiple.

Dans les hiérarchies de classe que nous avions abordées par ailleurs dans ce livre, chaque classe héritait d'un seul parent. De tels héritages uniques sont suffisants pour décrire la plupart des relations du monde réel. Mais certaines classes représentent une union de deux classes en une seule (voilà qui semble très romantique).

Le canapé-lit est un bon exemple de ce genre de classe. Comme le suggère son nom, c'est à la fois un lit et un canapé (bien que, comme lit, ce ne soit pas toujours très confortable...). À ce titre, cette couche peut fort bien hériter des propriétés propres au lit. Pour ce faire, le C++ permet à une classe dérivée d'hériter de plus d'une classe de base. C'est ce que l'on appelle un *héritage multiple*.

Le mécanisme de l'héritage multiple

Pour voir comment l'héritage multiple fonctionne, je développerai l'exemple du canapé-lit. La Figure 26.1 montre le schéma de l'héritage de la classe `Sleeper-Sofa`. Remarquez comment cette classe hérite de la classe `Sofa` (canapé) et de la classe `Bed` (lit) ; en procédant ainsi, elle hérite donc des propriétés des deux.

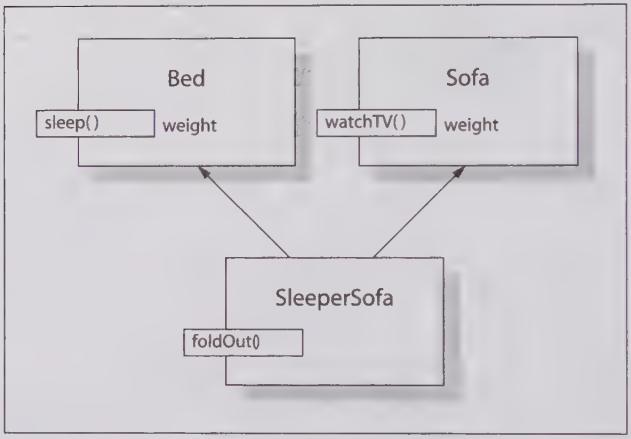


Figure 26.1
Hiérarchie de classe d'un canapé-lit.

Le code qui implémente la classe `SleeperSofa` se présente ainsi :

```
//  MultipleInheritance - une même classe peut hériter de
//                        plusieurs classes de base
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// voici le lit
class Bed
{
  public:
    Bed(){}
    void sleep(){ cout << "Dormir" << endl; }
    int weight;
};

// puis le canapé
```

```
class Sofa
{
  public:
    Sofa(){}
    void watchTV(){ cout << "Regarder la TV" << endl; }
    int weight;
};

// SleeperSofa - est à la fois un lit et un canapé
class SleeperSofa : public Bed, public Sofa
{
  public:
    SleeperSofa(){}
    void foldOut(){ cout << "Déplier" << endl; }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    SleeperSofa ss;

    // quand il est replié, on peut regarder la télé...
    ss.watchTV();     // Sofa::watchTV()

    //...puis on peut le déplier...
    ss.foldOut();     // SleeperSofa::foldOut()

    // ...et (essayer de) dormir
    ss.sleep();

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Ici, la classe `SleeperSofa` hérite à la fois de `Bed` et de `Sofa`. C'est évident rien qu'à voir l'apparence des deux classes dans la déclaration. `SleeperSofa` hérite de tous les membres des deux classes de base. De ce fait, les deux appels `ss.sleep()` et `ss.watchTV()` sont légaux. Vous pouvez utiliser un `SleeperSofa` (canapé-lit) comme `Bed` (lit) ou comme `Sofa` (canapé). De plus, la classe `SleeperSofa` peut avoir un membre qui lui est propre, comme `foldOut()` (déplier). C'est le grand confort, non ?

Lever les ambiguïtés de l'héritage

Bien que l'héritage multiple soit une fonctionnalité puissante, il n'en présente pas moins quelques problèmes potentiels. L'un est apparent dans l'exemple précédent. Remarquez que les deux classes `Bed` et `Sofa` contiennent un membre `weight` (poids). C'est logique puisque chacun a un poids mesurable. La question est la suivante : de quel poids (`weight`) hérite le canapé-lit (`SleeperSofa`) ?

La réponse est : les deux. `SleeperSofa` hérite d'un membre `Bed::weight` et d'un autre membre `Sofa::weight`. Comme ils ont le même nom, les références non qualifiées à `weight` sont maintenant ambiguës. C'est ce que démontre le bref morceau de programme que voici :

```
#include <iostream>

void fn()
{
  SleeperSofa ss;
  cout << "poids = "
       << ss.weight  // Illégal ! Quel poids ?
       << "\n";
}
```

Le programme doit donc spécifier explicitement l'un des deux poids en précisant la classe de base désirée. Le code suivant est correct :

```
#include <iostream>
void fn()
{
  SleeperSofa ss;
  cout << "poids = "
       << ss.Sofa::weight  // l'origine de weight est précisée
       << "\n";
}
```

Bien que cette solution corrige le problème, spécifier la classe de base dans la fonction d'application n'est pas souhaitable. En effet, cela force l'information de classe à se répandre hors de celle-ci, dans le code de l'application. Dans ce cas, `fn()` doit savoir que `SleeperSofa` hérite de `Sofa`. Ces types de collisions de noms ne seraient pas possibles avec un héritage simple, mais sont un danger constant des héritages multiples.

Ajouter un héritage virtuel

Dans le cas de `SleeperSofa`, la collision de noms portant sur `weight` est plus qu'un simple accident car le canapé-lit `SleeperSofa` n'a pas deux poids et deux mesures (pour le canapé d'une part, pour le lit d'autre part). Cette collision s'est produite parce que la hiérarchie de la classe ne décrit pas le monde réel avec exactitude. En fait, les classes n'ont pas été complètement dérivées.

Si on y réfléchit un peu, il devient clair que le lit et le canapé sont des cas particuliers d'un concept encore plus fondamental : les meubles (je suppose que je pourrais aller encore plus loin dans le concept et atteindre la notion générale d'objet pesant, mais les meubles conviennent très bien). Le poids est une propriété commune à tous les meubles. Cette relation est schématisée dans la Figure 26.2.

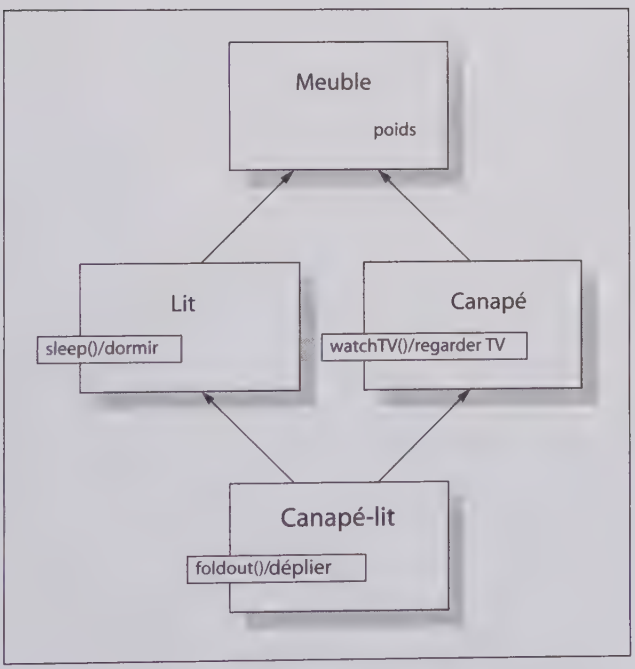


Figure 26.2
Une dérivation plus poussée du lit et du canapé (par poids).

La dérivation de la classe des meubles (`Furniture`) devrait éviter la collision de noms. Avec soulagement et en anticipant témérairement sur sa réussite, j'ai généré la hiérarchie de classe C++ suivante :

```
//  MultipleInheritanceFactoring - une même classe peut hériter
//                                 de plusieurs classes de base
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// Furniture - concept plus fondamental; cette classe
//             a une propriété "weight" (poids)
class Furniture
{
  public:
    Furniture(int w) : weight(w) {}
    int weight;
};

class Bed : public Furniture
{
  public:
    Bed(int weight) : Furniture(weight) {}
    void sleep(){ cout << "Dormir" << endl; }
};

class Sofa : public Furniture
{
  public:
    Sofa(int weight) : Furniture(weight) {}
    void watchTV(){ cout << "Regarder la TV" << endl; }
};

// SleeperSofa - est à la fois un lit et un canapé
class SleeperSofa : public Bed, public Sofa
{
  public:
    SleeperSofa(int weight) : Sofa(weight), Bed(weight) {}
    void foldOut(){ cout << "Déplier" << endl; }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    SleeperSofa ss(10);

    // Section 1 -
    // ce qui suit est ambigu ; est-ce un objet
    // Furniture::Sofa ou bien Furniture::Bed?
    /*
```

```
    cout << "Poids = "
         << ss.weight
         << endl;
     */

    // Section 2 -
    // on précise l'héritage sans ambiguïté
    // - le mystère s'évanouit
    SleeperSofa* pSS = &ss;
    Sofa* pSofa = (Sofa*)pSS;
    Furniture* pFurniture = (Furniture*)pSofa;
    cout << "Poids = "
         << pFurniture->weight
         << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Imaginez ma consternation quand je me suis rendu compte que cela ne nous avançait pas : `weight` reste ambigu (j'aimerais bien que mon poids soit lui aussi ambigu !). Je me suis dit que je pourrais essayer de convertir (*cast*) `ss` en `Furniture` :

```
#include <iostream>

void fn()
{
  SleeperSofa ss;
  Furniture* pF;
  pF = (Furniture*)&ss;  // Utilisation d'un pointeur Furniture...
  cout << "poids = "     // ... pour aller sur "weight", poids.
       << pF->weight
       << "\n";
};
```

La conversion de `ss` en `Furniture` n'est guère plus concluante. J'obtiens maintenant d'étranges messages lors de la compilation. Qu'est-ce qui se passe ?

L'explication est claire. `SleeperSofa` n'hérite pas directement de `Furniture`. En fait, `Bed` et `Sofa` héritent de `Furniture` et `SleeperSofa` hérite d'eux seulement ensuite. En mémoire, `SleeperSofa` se présente comme sur la Figure 26.3.

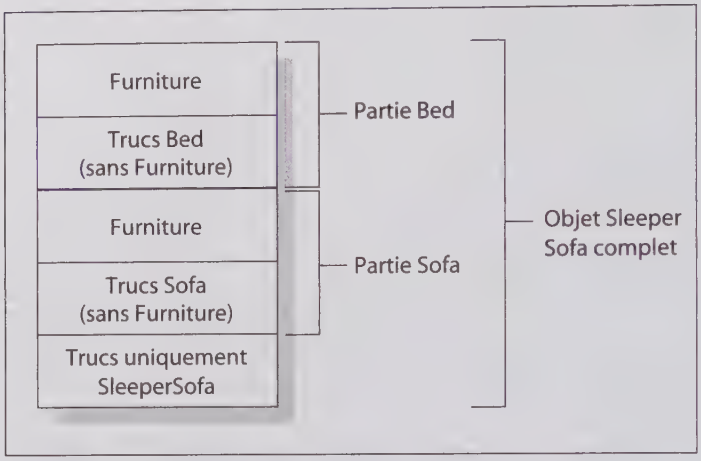


Figure 26.3
Cartographie de la mémoire de SleeperSofa.

Vous pouvez voir qu'un `SleeperSofa` (canapé-lit) est fait d'un `Bed` (lit) complet suivi d'un `Sofa` (canapé) complet et d'éléments propres uniquement à `SleeperSofa`. Chacun des sous-objets de `SleeperSofa` possède sa propre partie `Furniture` (mobilier) car ils héritent tous d'elle. De ce fait, le `SleeperSofa` contient deux objets `Furniture` !

En définitive, je n'ai pas créé la hiérarchie de la Figure 26.2. Celle que j'ai réalisée est visible sur la Figure 26.4.

Le programme `MultipleInheritanceFactoring` démontre cette duplication de la classe de base. La section 2 spécifie exactement l'objet poids (`weight`) voulu en redirigeant le pointeur `SleeperSofa` d'abord vers un `Sofa*`, puis vers un `Furniture*`.

Mais, le fait que `SleeperSofa` contienne deux objets `Furniture` est une aberration. `SleeperSofa` ne nécessite qu'une seule copie de `Furniture`. Je veux que `SleeperSofa` n'hérite que d'une seule copie de `Furniture` et que `Bed` et `Sofa` partagent cette copie. Le C++ appelle cela un *héritage virtuel* car il utilise le mot-clé `virtual`.

J'ai horreur que le terme "virtuel" soit ainsi galvaudé car un héritage virtuel n'a rien à voir avec les fonctions virtuelles.

Fort de ce savoir, je retourne dans la classe `SleeperSofa` et je l'implémente comme suit :

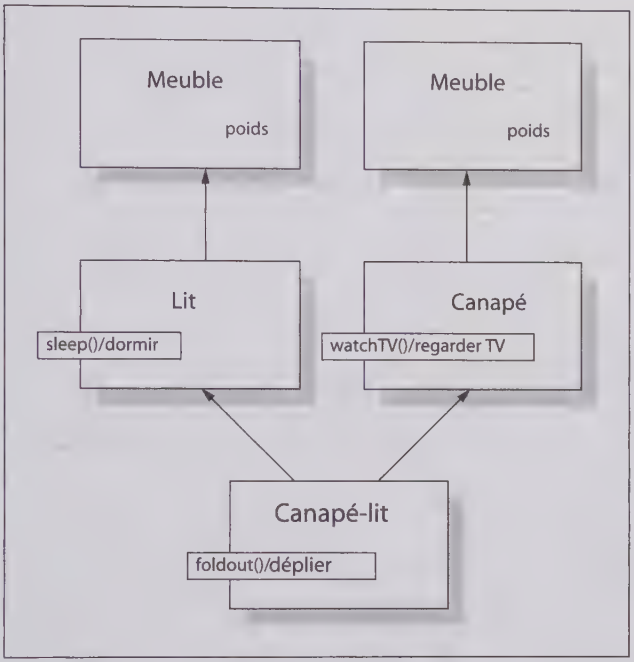


Figure 26.4
Le véritable résultat de ma première tentative.

```
//  VirtualInheritance - avec l'héritage virtuel, les classes
//                       Bed et Sofa peuvent partager
//                       une base commune
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// Furniture - concept plus fondamental; cette classe
//             a une propriété "weight" (poids)
class Furniture
{
  public:
    Furniture(int w = 0) : weight(w) {}
    int weight;
};

class Bed : virtual public Furniture
{
  public:
```

```
    Bed() {}
    void sleep(){ cout << "Dormir" << endl; }
};

class Sofa : virtual public Furniture
{
  public:
    Sofa(){}
    void watchTV(){ cout << "Regarder la TV" << endl; }
};

// SleeperSofa - est à la fois un lit et un canapé
class SleeperSofa : public Bed, public Sofa
{
  public:
    SleeperSofa(int weight) : Furniture(weight) {}
    void foldOut(){ cout << "Déplier" << endl; }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    SleeperSofa ss(10);

    // Section 1 -
    // ce qui suit n'est plus ambigu ;
    // un seul poids est partagé entre Sofa et Bed
    // Furniture::Sofa ou Furniture::Bed ? Tout devient clair !
    cout << "Poids = "
         << ss.weight
         << endl;

    // Section 2 -
    // on précise l'héritage sans ambiguïté
    // - le mystère s'évanouit
    SleeperSofa* pSS = &ss;
    Sofa* pSofa = (Sofa*)pSS;
    Furniture* pFurniture = (Furniture*)pSofa;
    cout << "Poids = "
         << pFurniture->weight
         << endl;

    // attend pour terminer le programme que l'utilisateur
    // lise le contenu de la fenêtre puis appuie sur une touche
    system("PAUSE");
    return 0;
}
```

Remarquez l'ajout du mot-clé `virtual` dans l'héritage de `Furniture`, pour `Bed` et `Sofa`. Cela peut se traduire par "Donne-moi une copie de `Furniture` à moins que tu n'en aies déjà reçu une d'une manière ou d'une autre, auquel cas je l'utiliserai". En mémoire, `SleeperSofa` finit par se présenter comme sur la Figure 26.5.

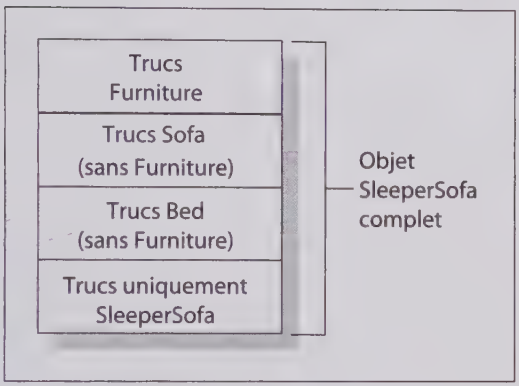


Figure 26.5
Cartographie de la mémoire de SleeperSofa avec héritage virtuel.

Vous voyez ici que `SleeperSofa` hérite de `Furniture`, puis de `Bed` moins la partie `Furniture`, suivi de `Sofa` moins la partie `Furniture`. On trouve tout en bas les membres spécifiques à `SleeperSofa` (notez que ce schéma ne représente pas forcément l'ordre des éléments dans la mémoire, mais cela n'a pas d'importance dans le cadre de cet exposé).

Maintenant, la référence à `weight` dans `fn()` n'est plus ambiguë parce que `SleeperSofa` ne contient qu'une seule copie de `Furniture`. En héritant virtuellement de `Furniture`, vous obtenez la relation désirée telle qu'elle est exprimée sur la Figure 26.2.

Si l'héritage virtuel résout aussi élégamment le problème, pourquoi n'est-il pas devenu la norme ? D'abord parce que les classes de base héritées virtuellement sont gérées en interne d'une façon très différente des classes qui sont héritées normalement, et ces différences entraînent une surcharge supplémentaire. La deuxième raison, c'est que vous pouvez parfois avoir besoin de deux copies de la classe de base (mais cela est assez inhabituel).

En ce qui concerne ce dernier point, imaginez un professeur-assistant, à la fois `professeur` et `étudiant`, qui sont tous deux des sous-classes de `Universitaire`. Si l'université donne au professeur-assistant deux numéros d'identification différents, l'un pour le professeur et l'autre pour l'étudiant, la classe `ProfesseurAssistant` devra contenir deux copies de la classe `Universitaire`.

Construction d'objets à héritage multiple

Les règles de construction des objets doivent être étendues afin de pouvoir gérer l'héritage multiple. Les constructeurs sont appelés dans l'ordre suivant :

1. **En premier lieu, le constructeur de chaque classe de base virtuelle est appelé en respectant l'ordre dans lequel les classes sont héritées.**

2. **Ensuite, le constructeur des classes de base non virtuelles est appelé dans l'ordre selon lequel les classes sont héritées.**

3. **Puis le constructeur de tous les objets membres est appelé dans l'ordre d'apparition de ces objets dans la classe.**

4. **Enfin, le constructeur de la classe elle-même est appelé.**

Remarquez que les classes de base sont construites en respectant l'ordre d'héritage, et non celui où elles apparaissent dans la ligne du constructeur.

Avis contraire

Je me dois de signaler qu'un certain nombre d'adeptes de la programmation orientée objet pensent que l'héritage multiple n'est pas une bonne chose. De plus, de nombreux langages orientés objet ne supportent pas cette technique.

Pour un langage informatique, l'héritage multiple n'est pas facile à implémenter. C'est pour l'essentiel le problème du compilateur (ou de celui qui a écrit le compilateur). Mais, comparé à l'héritage unique, l'héritage multiple surcharge le code, ce qui devient à ce moment-là le problème du programmeur.

Plus grave, l'héritage multiple ouvre la porte à des erreurs supplémentaires. D'abord, apparaissent des ambiguïtés comme celles que nous avons étudiées plus haut. Ensuite, en présence de multiples héritages, la conversion (*cast*) d'un pointeur d'une sous-classe vers une classe de base entraîne souvent de mystérieuses modifications de la valeur du pointeur. Laissons donc les détails de toute cette affaire aux "technogourous" de la programmation et aux auteurs de compilateurs.

Je vous recommande d'éviter les héritages multiples tant que vous ne serez pas parfaitement à l'aise avec le C++. L'héritage simple fournit largement de quoi travailler. Plus tard, vous pourrez vous plonger dans la littérature spécialisée et passer quelques nuits blanches à étudier le sujet, jusqu'à ce que vous soyez sûr de comprendre exactement comment fonctionnent les héritages

multiples. Une exception à cette règle est l'utilisation de librairies commerciales, dont les *Microsoft Foundation Classes (*MFC) qui ont plus ou moins recours aux héritages multiples. Ces classes ont fait leurs preuves et sont sûres.

Ne me faites pas dire ce que je n'ai pas dit. Je n'ai rien contre les héritages multiples. Le fait que Microsoft et d'autres les utilisent effectivement dans leurs librairies de classes prouve que cela marche. Autrement, ils ne les exploiteraient pas. Mais c'est néanmoins une fonctionnalité que vous aurez intérêt à n'utiliser que quand vous serez prêt à le faire.

Chapitre 27

C++ et les (top) modèles

Dans ce chapitre :

- Application de modèles aux fonctions.
- Combiner des fonctions dans un modèle.
- Définir un modèle de classe.
- Avantage des modèles sur l'approche plus générique "void".

La librairie standard du C++ fournit un ensemble complet de fonctions de base : mathématiques, date et heure, entrées/sorties, opérations DOS... Et ce n'est que la partie émergée de l'iceberg ! Nombre de programmes développés depuis le début de ce livre utilisent des fonctions de chaînes de caractères définies dans le fichier inclus `string`. Les types d'arguments de ces fonctions sont prédéfinis. Par exemple, les deux arguments de `strcpy(char*, char*)` doivent être un pointeur vers une chaîne terminée par le caractère nul. Toute autre situation serait invalide.

Certaines fonctions sont applicables à des types de données variés. Prenons l'exemple de la fonction `maximum()`, qui retourne le plus grand des deux arguments qui lui sont passés. Toutes les déclarations présentées dans le Tableau 27.1 sont licites.

Je voudrais implémenter la fonction `maximum()` pour ces quatre cas. Bien entendu, C++ peut convertir tous les types spécifiés en `double`. Vous pourriez alors en déduire que la version `maximum(double, double)` est la seule qui soit réellement nécessaire. Mais regardez attentivement le petit code qui suit :

Tableau 27.1 : Variantes possibles de la fonction maximum().

Nom de la fonction	Opération exécutée
`maximum(int, int)`	Retourne le plus grand de deux entiers.
`maximum(unsigned int, unsigned int)`	Retourne la plus grande de deux valeurs non signées. Comme il n'y a pas dans ce cas de nombres négatifs, l'expression maximum(0, -1) renvoie une valeur non signée extrêmement élevée.
`maximum(double, double)`	Retourne la plus grande de deux valeurs flottantes.
`maximum(char, char)`	Retourne le caractère dont le rang dans l'alphabet est le plus élevé (caractères spéciaux y compris).

```
// prototype d'une fonction maximum()
double maximum(double, double);

// fonction utilisateur
void fn(int nArg1, int nArg2)
{
  int nLarger = (int)maximum((double)nArg1, (double)nArg2);

  // ...on continue...
}
```

Ici, `nArg1` comme `nArg2` doivent être convertis en double, avec la perte de précision que cela implique. Cette version retourne un `double`. La valeur obtenue doit être convertie vers un `int` avant de pouvoir être affectée à `nLarger`. Même si cette fonction ne provoque pas une erreur d'appréciation, elle mobilise au moins bien trop de temps de calcul du processeur, du moins bien plus qu'une variante toute bête. En tout cas, son comportement n'est pas celui que l'utilisateur attend ou espère.

Vous pourriez évidemment surcharger `maximum()` avec toutes les variantes possibles :

```
double maximum(double d1, double d2)
{
    if (d1 > d2)
    {
```

```
        return d1;
    }
    return d2;
}
int maximum(int n1, int n2)
{
    if (n1 > n2)
    {
        return n1;
    }
    return n2;
}
char maximum(char c1, char c2)
{
    if (c1 > c2)
    {
        return c1;
    }
    return c2;
}

// ...répéter pour tous les autres types numériques...
```

Cette approche marche. Maintenant, C++ sélectionne la correspondance la mieux adaptée, par exemple `maximum(int, int)` pour l'expression `maximum(1, 2)`. Cependant, créer une variante pour chaque type de variable et de situation représente un gaspillage de temps considérable.

Le code source de toutes ces fonctions suit le même motif : `maximum(T,T)` où `T` est l'un des types numériques possibles. Il serait très pratique de pouvoir écrire une seule fois la fonction et de laisser C++ fournir le type approprié lors de l'appel. Réjouissez-vous : c'est possible !

Généraliser une fonction en un modèle

Un modèle (*template*) vous permet d'écrire le comportement d'une fonction, mais en utilisant un ou plusieurs "conteneurs" que C++ convertira en un type réel au moment de la compilation.

Le programme qui suit définit un modèle pour une fonction générique `maximum()` :

```
// MaxTemplate - crée un modèle de fonction max()
//               retournant le plus grand de deux types
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

// modèle de classe simpliste
template <class T>
T maximum(T t1, T t2)
{
    if (t1 > t2)
    {
        return t1;
    }
    return t2;
};

int main(int argc, char* pArgs[])
{
    // trouve le maximum de deux entiers
    cout << "Le maximum de 1 et de 2 est "
         << maximum(1, 2)
         << endl;

    // repeat for two doubles
    cout << "Le maximum de 1.5 et de 2.5 est "
         << maximum(1.5, 2.5)
         << endl;

    system("PAUSE");
    return 0;
}
```

Le mot-clé `template` est suivi de crochets contenant un ou plusieurs conteneurs de type, chacun étant précédé du mot réservé `class`, d'une constante ou des deux. Ici, le modèle de fonction `T maximum<T>(T t1, T t2)` retourne le plus grand des objets `t1` et `t2`, possédant tous deux le type `T`, où `T` est une classe qui sera définie plus tard.

Un modèle de fonction ne devient utile que lorsqu'il est converti en une fonction réelle. C++ transforme alors `T` en un type adapté. Dans notre exemple, `main()` provoque implicitement la création de deux version différentes de `maximum()`.

Créer une fonction à partir d'un modèle se dit *instancier* le modèle.

Le premier appel, `maximum(1, 2)`, demande à C++ de créer une version de la fonction dans laquelle `T` est remplacé par `int`. Le second appel génère une seconde version qui aura le format `maximum(double, double)`. La sortie produite par ce programme se présente donc ainsi :

```
Le maximum de 1 et de 2 est 2
Le maximum de 1.5 et de 2.5 est 2.5
Appuyez sur une touche pour continuer...
```

Faites bien attention à la terminologie. Par exemple, tondre un chien au ras du poil n'est pas la même chose que tondre un chien à poil ras. Autre exemple : un modèle de fonction n'est pas une fonction. Lorsque `T` est de type `int`, le programme crée la *fonction* (et non le modèle de fonction) `maximum(int, int)` à partir du *modèle*. Employer le bon vocabulaire peut simplifier la vie.

L'instruction ci-dessous ne marchera pas :

```
double d = max(1, 2.0);
```

En effet, les deux arguments ne possèdent pas le même type. Pour que le modèle `maximum<T>(T, T)` puisse être transformé en fonction, il faut que les deux arguments soient exactement du même type. Si le modèle de fonction était défini ainsi :

```
maximum<T1, T2>(T1, T2)
```

le compilateur C++ pourrait remplacer le type `T1` par `int` et `T2` par `double`.

Vous pouvez forcer l'instanciation d'un modèle en fournissant une déclaration de prototype. Il s'agit d'ailleurs en général d'une démarche plus sûre :

```
float maximum(float, float);   // crée une instance de
                               // maximum<T>(T, T) où T = float
```

Le C++ ne peut pas compiler un modèle de fonction tant qu'il n'a pas été instancié en une vraie fonction. Si votre modèle contient des erreurs, vous ne le découvrirez probablement qu'au moment de la compilation (voire plus tard), au moment de l'instanciation.

Modèles de classes

En C++, vous pouvez aussi définir des modèles de classes. Un tel modèle suit les règles générales de définition des classes, si ce n'est qu'on trouve un "conteneur" destiné à recevoir certains éléments encore inconnus. À titre d'exemple, le programme qui suit crée un vecteur pour toute classe fournie par l'utilisateur (un *vecteur* est un type de conteneur dans lequel les objets sont rangés en ligne ; le vecteur sans doute le plus connu est le *tableau*).

```
// TemplateVector - implémente un vecteur utilisant un modèle
//
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

// TemplateVector - un tableau modélisé simple
template <class T>
class TemplateVector
{
 public:
    TemplateVector(int nArraySize)
    {
        // nombre d'éléments
        nSize = nArraySize;
        array = new T[nArraySize];
               reset();
    }
    int size() { return nWriteIndex; }
    void reset() { nWriteIndex = 0; nReadIndex = 0; }
    void add(T object)
    {
        if (nWriteIndex < nSize)
        {
            array[nWriteIndex++] = object;
        }
    }
    T get()
    {
        return array[nReadIndex++];
    }

  protected:
    int nSize;
```

```
    int nWriteIndex;
    int nReadIndex;
    T* array;
};

// exemple pour deux vecteurs, un d'entiers et l'autre de noms
void intFn();
void nameFn();

int main(int argc, char* pArgs[])
{
    intFn();
    nameFn();

    system("PAUSE");
    return 0;
}

// Entiers - manipule une collection d'entiers
void intFn()
{
    // create a vector
    TemplateVector<int> integers(10);

    // add values to the vector
    cout << "Entrez une suite d'entiers pour les ajouter au vecteur\n"
         << "(terminez par un nombre plus petit que 0) :" << endl;
    for(;;)
    {
        int n;
        cin  >> n;

        if (n < 0) { break; }
        integers.add(n);
    }

    cout << "\nVoici les nombres que vous avez saisi..." << endl;
    for(int i = 0; i < integers.size(); i++)
    {
        cout << i << " : " << integers.get() << endl;
    }
    cout << endl;
}

// Noms - crée et manipule un tableau de noms
class Name
{
```

```
  public:
    Name(char* n = "") : name(n) {}
    string display() { return name; }
  protected:
    string name;
};

void nameFn()
{
    // crée un vecteur
    TemplateVector<Name> names(10);

    // ajoute les valeurs au vecteur
    cout << "Entrez les noms (tapez un 'x' pour quitter) :"
         << endl;
    for(;;)
    {
        char buffer[80];
        do
        {
            cin.getline(buffer, 80);
        } while(strlen(buffer) == 0);
        if (stricmp(buffer, "x") == 0)
        {
            break;
        }
        names.add(Name(buffer));
    }

    cout << "\nVoici les noms que vous avez saisis..." << endl;
    for(int i = 0; i < names.size(); i++)
    {
        Name name = names.get();
        cout << i << " : " << name.display() << endl;
    }
}
```

Le modèle de classe `TemplateVector<T>` contient un tableau d'objets appartenant à la classe `T`. Il propose deux fonctions membres : `add()` et `get()`. La première ajoute un objet de classe `T` dans l'emplacement vide qui suit dans le tableau. De même, `get()` renvoie l'objet suivant dans le tableau.

Le programme `TemplateVector` instancie deux fois cette classe vectorielle : d'abord pour une série de simples `int`, puis pour la classe utilisateur `Name`.

La fonction `intFn()` crée un vecteur formé d'au plus dix entiers. Le programme lit les nombres saisis sur le clavier, les sauvegarde dans le tableau, puis les affiche, le tout en utilisant les fonctions fournies par `TemplateVector`.

La fonction `nameFn()` crée un tableau d'objets `Name`. Là encore, les noms sont lus au clavier et affichés à la fin de la saisie.

Vous remarquerez que le programme `TemplateVector` est aussi à l'aise avec les noms qu'avec les entiers. Vous noterez également que les fonctions `intFn()` et `nameFn()` sont semblables, alors même que les entiers et les noms ont une nature tout à fait différente.

Voici un exemple de sortie possible :

```
Entrez une suite d'entiers pour les ajouter au vecteur
(terminez par un nombre plus petit que 0) :
5
10
15
-1

Voici les nombres que vous avez saisis...
0 : 5
1 : 10
2 : 15

Entrez les noms (tapez un 'x' pour quitter) :
Paul
André
Louis
x

Voici les noms que vous avez saisis...
0 : Paul
1 : André
2 : Louis
Appuyez sur une touche pour continuer...
```

Ai-je vraiment besoin des modèles de classe ?

Vous vous dites peut-être : "Est-ce qu'il ne serait pas plus simple de créer une classe `Array` ? Pourquoi s'embêter avec des modèles ?".

Vous pouvez évidemment procéder de cette façon, du moins si vous connaissez *a priori* le types des objets dont vous avez besoin. Par exemple, si

vous devez uniquement manipuler des tableaux d'entiers, il vous suffit de créer la classe `intArray` et le tour est joué. Pas besoin de modèle dans ce cas !

La seule autre alternative consiste à utiliser `void*`, qui peut pointer vers n'importe quel type d'objet. Le programme qui suit, `VoidVector`, est basé sur l'emploi de tels pointeurs :

```
// VoidVector - implémente un vecteur basé sur
//              un élément de stockage void*
#include <cstdlib>
#include <cstdio>
#include <iostream>
using namespace std;

typedef void* VoidPtr;

class VoidVector
{
 public:
    VoidVector(int nArraySize)
    {
        // nombre d'éléments
        nSize = nArraySize;
        ptr = new VoidPtr[nArraySize];
        reset();
    }
    int size() { return nWriteIndex; }
    void reset() { nWriteIndex = 0; nReadIndex = 0; }
    void add(void* pValue)
    {
        if (nWriteIndex < nSize)
        {
            ptr[nWriteIndex++] = pValue;
        }
    }
    VoidPtr get(){ return ptr[nReadIndex++]; }

  protected:
    int nSize;
    int nWriteIndex;
    int nReadIndex;
    VoidPtr* ptr;
};

int main(int argc, char* pArgs[])
{
    // crée un vecteur
```

```
    VoidVector vv(10);

    // ajoute des valeurs au vecteur
    cout << "Entrez une suite d'entiers pour les ajouter au vecteur\n"
         << "(terminez par un nombre plus petit que 0) :" << endl;
    for(;;)
    {
        int* p = new int;
        cin  >> *p;

        if (*p < 0)
        {
            delete p;
            break;
        }

        vv.add((void*)p);
    }

    cout << "\nVoici les nombres que vous avez saisis..." << endl;
    for(int i = 0; i < vv.size(); i++)
    {
        int* p = (int*)vv.get();
        cout << i << ":" << *p << endl;
    }

    system("PAUSE");
    return 0;
}
```

Ce programme définit un type `VoidPtr` qui est équivalent à un `void*`.

Le mot-clé `typedef` ne fait que définir un nouveau nom pour une classe existante. Vous pouvez insérer mentalement `void*` partout où vous voyez écrit `VoidPtr`. Cette technique peut non seulement faciliter la lecture d'une fonction, mais aussi améliorer la syntaxe d'une instruction. Parfois, il n'est pas possible de faire fonctionner correctement un modèle de classe lorsque le type requis est un pointeur. Habiller un type complexe, tel qu'un pointeur, à l'aide d'une instruction `typedef` résout ce problème.

`VoidVector` fournit les mêmes méthodes `add()` et `get()` que le modèle de classe `TemplateVector` du programme précédent.

Cette solution pose (au moins) trois problèmes. Tout d'abord, son utilisation n'est pas si évidente que cela, comme le démontre la fonction `main()`. Il n'est pas possible d'enregistrer directement une valeur. Vous ne pouvez que passer

l'adresse d'un objet. Vous devez donc allouer de la mémoire sur le tas via un pointeur `int*` pour y placer l'entier qui va être lu au clavier.

Le deuxième problème, c'est un sérieux risque d'erreur. Supposons que vous envisagiez d'ajouter simplement un `int` à la collection de la façon suivante :

```
int n;
cin >> n;
vv.add((void*)&n);
```

Cela ne marche pas ! La variable `n` a une portée locale. Son adresse disparaîtra dès que la boucle `for` n'aura plus le contrôle. Et l'adresse dans le vecteur n'aura donc plus aucun sens.

En fait, la chose est même plus vicieuse : l'adresse de `n` reste la même lors de chaque itération dans la boucle `for`.

Le troisième problème est encore plus grave. Pour retrouver une valeur à partir d'un `VoidVector`, vous devez connaître le type d'objet qui y est entreposé. C++ ne peut pas contrôler les types pour vérifier que votre supposition est correcte. Imagions par exemple que vous pensiez que `vv` contiennent des valeurs `double` et non `int`. Le code qui suit enregistrerait dans `dValue` quelque chose de totalement imprévisible :

```
double dValue = *(double*)get();
```

Le programme perdrait sans aucun doute la boussole. La conversion de et vers un `void*` mettrait en échec la gestion (pourtant solide) des types intégrés à C++.

Conseils pour l'utilisation des modèles

Revoyons les points qu'il ne faut jamais oublier lorsque l'on utilise des modèles.

Tout d'abord, aucun code n'est généré en soi pour un modèle. Cette construction n'intervient que lorsque le modèle est *instancié*, converti en une vraie classe ou fonction. Ceci implique qu'un fichier .CPP n'est pratiquement jamais associé à un modèle de classe. La totalité de la définition du modèle, y compris toutes ses fonctions membres, est placée dans un fichier inclus de façon à ce que le compilateur n'ait plus qu'à en réaliser l'implémentation le moment venu.

D'autre part, un modèle ne consomme aucune mémoire. Créer des modèles de classes qui ne seront jamais instanciés n'engendre aucune pénalité (quoi qu'on se demande alors quel est leur intérêt). À l'inverse, toute instanciation d'un modèle de classe consomme une fraction supplémentaire de la mémoire. Le code de `Array<Name>` occupe son propre espace, même si `Array<int>` existe déjà.

Enfin, un modèle de classe (ou de fonction) ne peut être ni compilé, ni analysé pour y détecter d'éventuelles erreurs tant qu'il n'est pas converti en une entité réelle. Un programme qui fait référence au modèle de classe `Array<T>` peut parfaitement être compilé, alors même que `Array<T>` contient d'évidentes fautes de syntaxe. Ces erreurs ne pourront apparaître que si et quand une classe `Array<int>` ou encore `Array<Name>` est créée.

Chapitre 28

La librairie de modèles standard

Dans ce chapitre :

- Utiliser la classe `string`.
- Utiliser le conteneur `list`.
- Accéder aux éléments d'un conteneur à l'aide d'un itérateur.
- Utiliser un conteneur `map`.

Certains programmes peuvent traiter immédiatement les données qu'ils reçoivent et en faire quelque chose. Mais la plupart du temps, il est nécessaire de stocker ces données pour les utiliser plus tard. Une structure servant à mémoriser des informations est appelée de façon générique un *conteneur* ou encore une *collection* (les technogourous ne jurent que par le premier nom, mais le second n'est pas mal non plus). Dans ce livre, nous avons largement fait appel à un conteneur pratiquement universel : le *tableau* (*array*). Il possède en effet quelques propriétés bien agréables. Il sait ranger et retrouver diverses choses très rapidement. Il peut être construit pour recevoir en toute sécurité des objets de types variés. Et ainsi de suite. En revanche, le tableau présente deux défauts majeurs.

Tout d'abord, vous devez connaître sa taille au moment où vous le créez. Cela n'est généralement pas possible, même s'il arrive parfois que vous soyez capable d'affirmer que le nombre d'éléments ne dépassera pas une certaine "valeur élevée". Les virus exploitent allègrement de telles certitudes fausses et trompeuses. Il n'existe pas vraiment de méthode pour faire "grossir" un tableau, si ce n'est en déclarer un nouveau, plus grand, et y copier le contenu de l'ancien.

En second lieu, l'insertion d'éléments à l'intérieur d'un tableau implique toute une procédure de copie, qui est coûteuse en termes de mémoire et de temps de calcul. Et ne parlons même pas du tri, encore plus onéreux.

Heureusement pour nous, C++ est fourni avec une librairie (une bibliothèque, si vous préférez) qui comprend de nombreux types de conteneurs (chacun ayant ses avantages et ses inconvénients). J'ai parlé de la STL : *Standard Template Library*, la librairie de modèles standard.

Cette STL est une très vaste librairie proposant un ensemble de conteneurs parfois très complexes. Nous ne ferons qu'effleurer ici la puissance de STL !

Le conteneur string

Le format de tableau le plus courant est la chaîne terminée par le caractère "nul" qui sert à afficher du texte. Soit dit en passant, c'est un excellent cas de figure pour comprendre les qualités et les défauts des tableaux. L'instruction suivante semble d'une simplicité biblique :

```
cout << "Ceci est une chaîne";
```

Mais la situation empire rapidement si vous tentez de réaliser une opération aussi courante que la concaténation de deux chaînes :

```
char* concatCharString(char* s1, char* s2)
{
    int length = strlen(s1) + strlen(s2) + 1;
    char* s = new char[length];
    strcpy(s, s1);
    strcat(s, s2);
    return s;
}
```

La STL fournit un conteneur `string` pour gérer les chaînes de caractères. La classe `string` offre de nombreuses opérations (dont la surcharge des opérateurs) destinées à faciliter la manipulation des chaînes. La concaténation précédente peut être réalisée en deux temps et trois mouvements si vous vous servez d'objets de type `string` :

```
string concat(string s1, string s2)
{
    return s1 + s2;
}
```

J'ai évité d'utiliser la classe `string` dans les autres chapitres car nous ne l'abordons que maintenant. Notons donc en passant que la plupart des programmeurs utilisent cette classe de préférence aux tableaux de caractères classiques.

Le programme qui suit vous montre quelques-unes des fonctionnalités de la classe `string` :

```
// STLString - démonstration de quelques fonctionnalités
//             de la classe string (qui fait partie de la
//             Librairie de Modèles Standard)
#include <string>
#include <cstdlib>
#include <iostream>
using namespace std;

// concat - concaténation de deux chaînes
string concat(string s1, string s2)
{
    return s1 + s2;
}

// removeSpaces - supprime tous les espaces dans une chaîne
string removeSpaces(string s)
{
    // trouve l'emplacement du premier espace;
    // continue la recherche jusqu'à ce qu'il n'y ait plus d'espace.
    size_t offset;
    while((offset = s.find(" ")) != -1)
    {
        // supprime l'espace courant
        s.erase(offset, 1);
    }
    return s;
}

// insertPhrase - insère une phrase à la position
//                du point d'insertion <ip>
string insertPhrase(string source)
{
```

```
    size_t offset = source.find("<ip>");
    if (offset != -1)
    {
        source.erase(offset, 4);
        source.insert(offset, "Randall");
    }
    return source;
}


int main(int argc, char* pArgs[])
{
    // crée une nouvelle chaîne à partir de deux autres chaînes
    cout << "string1 + string2 = "
         << concat("string1 ", "string2")
         << endl;

    // supprime tous les espace dans une chaîne de test
    string s2("Voici la phrase");
    cout << "<" << s2 << "> moins les espaces = <"
         << removeSpaces(s2) << ">" << endl;

    // insère un texte à l'intérieur d'une phrase
    // (à la position signalée par "<ip>")
    string s3 = "Stephen <ip> Davis";
    cout << s3 + " -> " + insertPhrase(s3) << endl;

    system("PAUSE");
    return 0;
}
```

L'opérateur `operator+()` effectue la concaténation que nous avions réalisée auparavant en utilisant la méthode `concatCharacterString()`.

La méthode `removeSpaces()` supprime tous les espaces trouvés dans la chaîne fournie en argument. Pour cela, elle fait appel à l'opération `find()` qui renvoie la position du premier caractère " " trouvé. Elle se sert alors de la méthode `erase()` afin de supprimer l'espace. La méthode `find()` retourne un décalage de -1 lorsque le caractère recherché est introuvable.

Le type `size_t` est défini dans la STL comme étant une variante d'entier capable de contenir le plus grand indice de tableau possible sur votre machine. Il s'agit généralement d'un entier long non signé. Ce type a pour vocation de faciliter le portage du code source d'un système à un autre. Visual Studio .NET générera une erreur si vous utilisez à la place un `int`.

La fonction `insertPhrase()` se sert de la méthode `find()` pour trouver le point d'insertion. Elle appelle ensuite `erase()` pour retirer le marqueur `<ip>` puis `insert()` pour insérer un morceau choisi à l'intérieur d'une chaîne existante.

Voici ce que donne notre petit programme :

```
string1 + string2 = string1 string2
<Voici la phrase> moins les espaces = <Voicilaphrase>
Stephen <ip> Davis -> Stephen Randall Davis
Appuyez sur une touche pour continuer...
```

Les conteneurs list

La librairie de modèles standard offre un grand nombre de conteneurs. Bien plus en tout cas qu'il n'est possible d'en présenter ici. Nous allons nous concentrer sur deux familles parmi les plus utiles.

Le conteneur `list` de la STL gère des objets en les chaînant les uns à la suite des autres, un peu comme un jeu de Lego. Sa nature est donc *séquentielle*. Les objets peuvent être extraits et replacés dans n'importe quel ordre. Ceci rend le conteneur `list` idéal pour l'insertion, le tri, la fusion et toutes sortes d'autres arrangements.

Le programme qui suit utilise ce conteneur `list` pour trier une série de noms :

```
// STLList - utilise le conteneur list de la STL
//           pour entrer et trier un série de noms
#include <list>
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>

// déclare une liste d'objets de type string
using namespace std;
list<string> names;

int main(int argc, char* pArgs[])
{
    // saisie d'une série de noms
    cout << "Entrez un nom (tapez un 'x' pour terminer)"
         << endl;
    while(true)
    {
```

```
        string name;
        cin >> name;
        if ((name.compare("x") == 0) ||
            (name.compare("X") == 0))
        {
            break;
        }
        names.push_back(name);
    }

    // trie la liste (ceci fonctionne car String
    // implémente un opérateur de comparaison)
    names.sort();

    // affiche la liste triée en la vidant
    // au fur et à mesure
    cout << "\nLa liste quand le tri est fait : " << endl;
    while(!names.empty())
    {
        // trouve le premier nom de la liste
        string name = names.front();
        cout << name << endl;

        // supprime le nom courant de la liste
        names.pop_front();
    }

    system("PAUSE");
    return 0;
}
```

Cet exemple définit la variable `names` comme étant une liste d'objets `string`. Le programme commence par lire les noms au clavier. Chaque nom est ajouté à la fin de la liste en utilisant la méthode `push_back()`. La boucle de saisie se termine lorsque l'utilisateur tape un caractère `"x"`. La liste des noms est alors affichée dans l'ordre alphabétique en invoquant la méthode `sort()` du conteneur `list`.

Pour afficher la liste triée, le programme récupère le premier élément, l'envoie vers la sortie standard, puis le supprime. Et ainsi de suite jusqu'à ce que la liste soit vide.

Voici un exemple de ce que fait notre programme :

```
Entrez un nom (tapez un 'x' pour terminer)
Adams
Davis
Valentine
Smith
Wesson
x

La liste quand le tri est fait :
Adams
Davis
Smith
Valentine
Wesson
Appuyez sur une touche pour continuer...
```

Le conteneur `list` propose un grand nombre d'opérateurs. On y trouve des opérations simples pour l'insertion (`insert`), l'échange (`swap`) ou encore la suppression (`erase`) d'objets. Ce conteneur permet également de parcourir automatiquement une liste en appliquant une certaine fonction utilisateur à tous les objets rencontrés.

L'opération que les listes n'autorisent pas est l'accès aléatoire. Du fait que les objets peuvent être associés dans un ordre quelconque, il n'existe aucun procédé simple et direct permettant de renvoyer l'objet *n* d'une classe `list`.

Les itérateurs

Le programme précédent emploie une méthode destructive pour parcourir la liste de noms. La méthode `pop_front()` effectue ce déplacement en supprimant systématiquement le premier objet rencontré.

Pour parcourir un tableau classique, il suffit de fournir l'indice de chaque élément. Mais cette technique ne marche pas avec des conteneurs du style `list` qui ne disposent pas d'un accès aléatoire. On pourrait imaginer une solution fondée sur des méthodes telles que `getFirst()` et `getNext()`. Cependant, les développeurs de la STL ont voulu fournir une méthode commune permettant de traverser n'importe quel type de conteneur. La réponse s'appelle *itérateur.*

Un itérateur est un objet qui pointe vers les membres d'un conteneur. En règle générale, il supporte les fonctions suivantes :

- Une classe peut retourner un itérateur qui pointe vers le premier membre de la collection.
- L'itérateur peut être déplacé d'un membre vers le suivant.
- Le programme peut retrouver l'élément pointé par l'itérateur.

Le code nécessaire au parcours d'une liste est différent de celui qui réalise le même genre d'opération sur un vecteur (pour ne prendre que ces deux exemples). Cependant, l'itérateur masque ces détails.

Le programme qui suit fait appel à un itérateur pour parcourir une liste sans la détruire :

```
// STLListUserClass - utilise une liste pour mémoriser et trier
//                    une classe définie par l'utilisateur
#include <list>
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

// Student - exemple quelconque de classe utilisateur
class Student
{
  public:
    Student(char* pszName, int id)
    {
        name = new string(pszName);
        ssID = id;
    }
    string* name;
    int ssID;
};

// la fonction suivante est nécessaire pour
// réaliser l'opération de tri
bool operator<(Student& s1, Student& s2)
{
    return s1.ssID < s2.ssID;
}

// définit la collection d'objets students
list<Student> students;
```

```
int main(int argc, char* pArgs[])
{
    // ajoute trois étudiants à la liste
    students.push_back(*new Student("Marion Haste", 10));
    students.push_back(*new Student("Dewie Cheatum", 5));
    students.push_back(*new Student("Stew Dent, Sr.", 15));

    // on trie maintenant la liste
    students.sort();

    // itération dans la liste :
    // 1) allouer un itérateur pointant vers
    //    le premier élément de la liste
    list<Student>::iterator iter = students.begin();

    // 2) boucler jusqu'à ce que l'itérateur
    //    atteigne la fin de la liste
    while(iter != students.end())
    {
        // 3) retrouve l'étudiant pointé par l'itérateur
        Student& s = *iter;
        cout << s.ssID << " - " << *s.name << endl;

        // 4) déplace maintenant l'itérateur vers
        //    l'élément suivant de la liste
        iter++;
    }

    system("PAUSE");
    return 0;
}
```

Ce programme définit une liste d'objets définis par l'utilisateur appartenant à la classe `Student` (avec une structure plus complexe qu'une simple série de noms). Trois appels à `push_back()` ajoutent ces éléments à la liste (on gagne un peu de temps en insérant directement ces données dans l'application). L'appel à `sort()` est identique à ce que nous avons vu plus haut.

La fonction `sort()` de la librairie de modèles standard impose la surcharge de l'opérateur "plus petit que". C'est l'une des quelques situations où il faut redéfinir un autre opérateur que l'affectation. L'expression `s1 < s2` invoquera l'opérateur `operator<(Student&, Student&)` si `s1` et `s2` sont de type `Student`.

Le programme alloue un itérateur `iter` pour circuler dans la liste. Observez attentivement la déclaration : `list<Student>::iterator` est un itérateur vers un conteneur `list` formé d'objets `Student`. Ce typage apparaît clairement dans

l'affectation de l'étape 3 du programme : `*iter` y renvoie une référence à un objet `Student`.

La sortie produite par le programme se présente ainsi :

```
5 - Dewie Cheatum
10 - Marion Haste
15 - Stew Dent, Sr.
Appuyez sur une touche pour continuer...
```

Comment un tri trie

Voici une question intéressante : comment la méthode `sort()` peut-elle savoir quel est le "plus grand" de deux éléments de la liste ? En d'autres termes, qu'est-ce qui détermine l'ordre de tri ? C++ définit sa propre méthode de tri pour certains types de données. Par exemple, il sait facilement reconnaître qu'un entier est plus grand qu'un autre. Nous avons également vu plus haut que la collection de chaînes ASCII contenue dans la liste `names` était triée par la STL selon les mêmes règles qu'un dictionnaire.

Le programme `STLList` n'a besoin de prendre aucune mesure particulière pour trier des noms. Le C++ ne sait pas quel est le "plus grand" de deux étudiants. C'est précisément à cela que sert la fonction globale `::operator<(Student&, Student&)`. La méthode `sort()` invoque cette fonction lorsqu'elle parcourt la liste pour déterminer l'ordre de tri correct.

Essayez par exemple d'inverser le sens de la fonction `operator<()` :

```
return s1.ssID > s2.ssID;
```

Le résultat sera une liste triée dans le sens inverse.

Les conteneurs map

Les conteneurs map constituent une autre classe de collections (dite cette fois *associative*). Le concept n'est plus ici celui d'une série d'objets rangés l'un après l'autre, chaînés, mais d'un groupement d'entités, d'un ensemble ou encore d'un dictionnaire. Il en existe différents types, mais tous ont une propriété en commun : celle de pouvoir ranger et retrouver rapidement des éléments grâce à un indice (ou *clé*). Précisons cela à l'aide d'un exemple.

Une université peut cataloguer les étudiants en leur associant un identificateur numérique unique (ou ID). Cet identificateur sert dans tous les domaines de la vie scolaire : pour retrouver des informations sur les étudiants, accéder à la bibliothèque, s'inscrire aux cours, manger au restaurant de la fac (pour les plus résistants d'entre eux), et ainsi de suite. Il est essentiel qu'un programme quelconque soit à même de retrouver un(e) étudiant(e) à partir de son identificateur avec rapidité et efficacité.

Le programme qui suit crée et utilise une collection d'objets `Student` possédant une telle clé unique (`ID`) :

```
// STLMap - utilise un conteneur map pour gérer une collection
//          d'objets ordonnés selon une clé
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <string>
#include <map>

using namespace std;

// SC - fonction de comparaison entre objets Student;
//      sert à déterminer l'ordre de tri des étudiant(e)s
struct SC
{
    bool operator()(const int id1, const int id2) const
    {
        return id1 < id2;
    }
};

// le conteneur map contient un couple d'éléments (Pair);
// celui de gauche est la clé tandis que celui de droite
// est la donnée (dans notre cas un objet Student)
class Student;
typedef Student* SP;
typedef pair<const int, Student*>  Pair;
typedef map<int, SP, SC> Map;
typedef map<int, SP, SC>::iterator MapIterator;

// collection de Students
Map students;

// Student - définit les propriétés importantes d'un(e) étudiant(e)
//           notamment la clé utilisée pour la/le rechercher
//           à partir du fichier des étudiants (id)
```

```
class Student
{
  public:
    Student(char* pszName, int id)
              : studentIDKey(id), name(pszName) {}

    // getKey - la clé sert d'indice dans le conteneur map
    const int getKey() { return studentIDKey; }

    // display - crée une sortie adaptée pour
    //           un objet Student
    string display()
    {
        ostringstream out;
        out << studentIDKey << " - " << name;
        return out.str();
    }

  protected:
    // clé servant à identifier l'étudiant(e)
    const int studentIDKey;

    // nom de l'étudiant(e) (plus éventuellement d'autres données)
    string name;
};

int main(int argc, char* pArgs[])
{
    // ajoute quelques étudiant(e)s à la collection -
    // le conteneur map enregistre les objets sous forme
    // de "paires" dont le membre gauche est la clé tandis
    // que le droit est l'objet proprement dit
    Student* pS;
    pS = new Student("Sean Yours", 3456);
    Pair* ptr = new Pair(pS->getKey(), pS);
    students.insert(*ptr);

    // l'opérateur d'index est surchargé pour créer
    // la paire et l'insérer dans la collection map
    students[1234] = new Student("Fresch Man", 1234);
    students[5678] = new Student("Student, Jr.", 5678);

    // itération dans la collection des étudiant(e)s;
    // une collection map se trouve toujours dans l'ordre
    // déterminé par la classe SC
    cout << "Liste des inscrits dans l'ordre :" << endl;
    MapIterator iter = students.begin();
```

```
    while(iter != students.end())
    {
        Pair p = *iter;
        Student* s = p.second;
        cout << s->display() << endl;
        iter++;
    }

    // les opérateurs d'incrémentation et de décrémentation
    // peuvent aussi servir à trouver le prédécesseur
    // ou le successeur
    cout << "\nRecherche de l'ID 3456" << endl;
    MapIterator p = students.find(3456);
    cout << "Le nom est " << p->second->display() << endl;

    MapIterator p1 = p;
    MapIterator prior = --p1;     // <- prédécesseur
    cout << "Prédécesseur = "
         << prior->second->display() << endl;

    MapIterator p2 = p;
    MapIterator successor = ++p2; // <- successeur
    cout << "Successeur = "
         << successor->second->display() << endl;

    // find() retourne l'itérateur end s'il ne trouve plus
    // l'objet en question; operator[] renvoie un NULL
    if (students.find(0123) == students.end())
    {
        cout << "L'appel students.find(0123) retourne\n"
             << "students.end() car l'ID 0123 n'existe pas"
             << endl;
    }

    // sortie utilisant l'index
    cout << "Test de : students[3456] = "
         << students[3456]->display() << endl;

    if (students[0123] == NULL)
    {
        cout << "mais students[0123] retourne un NULL"
             << endl;
    }

    system("PAUSE");
    return 0;
}
```

La clé du programme (heureuse formule !) se trouve dans les trois `typedef` du début. Une collection map contient un ensemble d'objets `Pair`, chacun étant à son tour formé d'un couple d'éléments. Le premier élément est l'identificateur de l'étudiant(e). Le second est l'objet `Student`. La classe `Map` ajoute un objet de type `SC`. Cette dernière classe contient une seule méthode qui compare deux objets `Student` pour déterminer celui qui est le plus "grand" (c'est légèrement plus compliqué que la fonction globale utilisée dans la collection `list`, mais l'effet est le même).

Le programme `STLMap` commence par créer trois objets `Student Pair` et les ajoute à la collection. L'itération dans le conteneur affiche ces objets classés dans l'ordre des clés (ID). Il est inutile d'invoquer une quelconque fonction `sort()` car les classes map enregistrent déjà les objets dans l'ordre des clés.

La seconde section de `STLMap` recherche un(e) étudiant(e) à partir de son identificateur personnel en se servant de la méthode `find()`. Il vous montre également la facilité avec laquelle on peut retrouver l'objet précédent et l'objet suivant dans la liste grâce aux opérateurs d'incrémentation et de décrémentation.

Voici pour terminer la sortie produite par ce programme :

```
Liste des inscrits dans l'ordre :
1234 - Fresch Man
3456 - Sean Yours
5678 - Student, Jr.

Recherche de l'ID 3456
Le nom est 3456 - Sean Yours
Prédécesseur = 1234 - Fresch Man
Successeur = 5678 - Student, Jr.
L'appel students.find(0123) retourne
students.end() car l'ID 0123 n'existe pas
Test de : students[3456] = 3456 - Sean Yours
mais students[0123] retourne un NULL
Appuyez sur une touche pour continuer...
```

Sixième partie

Les dix commandements

"On vient nettoyer le code !"

Dans cette partie...

Que serait un ouvrage *pour les Nuls* sans les dix commandements ? Dans le Chapitre 29, je vous indique dix moyens d'éviter des bogues dans vos programmes (la plupart sont aussi valables pour le langage C, et sans supplément de prix). Le Chapitre 30 présente les dix principales options du compilateur Dev-C++, cet environnement gratuit que vous avez téléchargé pour étudier tous les exemples de ce livre.

Chapitre 29

Dix façons d'éviter les bogues

Dans ce chapitre :

- Activer tous les messages d'alerte et d'erreur.
- Faire des compilations en toute connaissance de cause.
- Adopter un style de codage clair et cohérent.
- Limiter la visibilité.
- Ajouter des commentaires en cours d'écriture.
- Effectuer au moins une exécution pas à pas.
- Éviter les opérateurs surchargés.
- Apprendre à gérer le tas.
- Gérer les erreurs à l'aide d'exceptions.
- Éviter les héritages multiples.

Dans ce chapitre, nous allons voir plusieurs moyens pour réduire le risque d'erreur, de même que pour pouvoir les localiser et les corriger plus facilement.

Activez tous les messages d'alerte et d'erreur

La syntaxe du C++ permet la détection des erreurs. Lorsque le compilateur rencontre une construction qu'il n'arrive pas à déchiffrer, il n'a pas d'autre choix que de générer un message d'erreur. Il tente certes de passer à la prochaine instruction, mais il n'essaie plus de générer un programme exécutable.

Désactiver les messages d'alerte et d'erreur, ce serait un peu comme débrancher les voyants rouges du tableau de bord de votre voiture sous prétexte qu'ils vous ennuient. Ignorer les problèmes ne vous avancera à rien. Si votre compilateur est équipé d'un vérificateur de syntaxe, activez-le. Microsoft Visual Studio .Net et Dev-C++ ont une option du genre Activer tous les messages. Vous finirez par gagner beaucoup de temps.

Pendant que vous tricotez votre code source, un bon compilateur C++ recherche les syntaxes sujettes à caution, comme celle de ce bout de programme :

```
#include "student"
#include "MyClass"

Student* addNewStudent(MyClass MyObject,
                       char *pName, SSNumber ss)
{
  Student pS;
 if (pName != 0)
  {
  pS = new Student(pName, ss);
  MyObject.addStudent(pS);
  }
  return pS;
}
```

Vous voyez ici que la fonction crée d'abord un nouvel objet `Student` qui est ensuite ajouté à l'objet `MyClass` fourni (on peut penser que `addStudent()` est une fonction membre de `MyClass`).

Si un nom est fourni (c'est-à-dire si `pName` n'est pas 0), un nouvel objet `Student` est créé et ajouté à la classe `MyClass`. Ceci étant fait, la fonction retourne le `Student` ainsi créé à l'appelant. Le problème, c'est que si `pName` est 0, `pS` ne sera jamais initialisé à quoi que ce soit. Un bon compilateur C++ est capable de détecter ce trajet et de générer une alerte indiquant qu'il est possible que `pS` ne soit jamais initialisé s'il est retourné à l'appelant, que vous devriez vous occuper du problème, ou tout autre message comparable.

Faites des compilations en toute connaissance de cause

Ne commencez pas le débogage de votre code tant que des messages d'alerte apparaissent pendant la compilation et tant que vous n'avez pas bien compris ce qu'ils signifient. Activer l'affichage de tous les messages et ne pas en tenir compte ne vous fera aucun bien. Ce que vous ignorez se retournera forcément contre vous.

Adoptez un style de codage clair et cohérent

Un codage clair et cohérent améliore non seulement la lisibilité du programme, mais réduit également le risque d'erreurs. Rappelez-vous toujours que, moins vous devrez faire d'efforts pour déchiffrer la syntaxe du C++, plus vous pourrez accorder d'attention à la logique du programme. Un codage bien fait facilite notamment les tâches suivantes :

- La différenciation des noms de classes, d'objets et de fonctions.
- La détermination de la raison d'être des objets d'après leur nom.
- La différenciation des symboles du préprocesseur de ceux du C++ (mise en évidence des objets `#define`).
- L'identification des blocs de code C++ de même niveau (grâce à une indentation cohérente).

De plus, vous devez prévoir un en-tête de module standard fournissant des informations au sujet des fonctions et des classes du module, de l'auteur (vous, en principe), de la date, de la version du compilateur utilisé. Fournissez aussi un historique des modifications.

Enfin, tous les programmeurs impliqués dans un même projet devraient adopter le même style. Car rien n'est plus compliqué que de déchiffrer un programme fait d'un patchwork de différents styles d'écriture.

Limitez la visibilité

Limiter la visibilité des éléments internes d'une classe depuis le monde extérieur est une des clés de voûte de la programmation orientée objet. Chaque classe est responsable de ses propres éléments internes, c'est-à-dire qu'ils

doivent être signalés comme privés ou protégés. De plus, les fonctions membres que le logiciel d'application n'a pas besoin de connaître doivent être protégées. N'exposez que ce qui est strictement nécessaire. En retour, l'utilisation de la classe relève de la responsabilité du programme.

À ce sujet, une règle veut que les fonctions membres publiques doivent se fier aussi peu que possible au code d'application. Tout argument passé à une fonction membre publique doit être considéré *a priori* comme une source potentielle de bogue, et cela jusqu'à ce que sa fiabilité ait été établie. Une fonction comme celle-ci est une épée de Damoclès :

```
class Array
{
   public:
    Array(int s)
    {
       size = 0
       pData = new int[s];
       if (pData)
       {
         size = s;
       }
    }
    ~Array()
    {
       delete pData;
       size = 0
       pData = 0;
    }
    // Retourne ou définit les données du tableau.
    int data(int index)
    {
       return pData[index];
    }
    int data(int index, int newValue)
    {
       int oldValue = pData[index];
       pData[index] = newValue;
       return oldValue;
    }
   protected:
    int size;
    int *pData;
};
```

La fonction `data(int)` permet au logiciel d'application de lire les données du tableau `Array`. Cette fonction est trop confiante ; elle présume en effet que l'`index` fourni se trouve dans la plage de données. Mais que se passerait-il s'il n'y était pas ? La fonction `data(int, int)` fait même pire car elle écrit à un emplacement inconnu.

Ce qu'il faut, c'est une vérification qui confirme que l'`index` se trouve dans la plage. Dans le listing ci-dessous, pour rester concis, seule la fonction `data(int)` est montrée :

```
int data(unsigned int index)
{
    if (index >= size)
    {
        throw Exception("Indice du tableau en dehors de la plage !");
    }
    return pData[index];
}
```

Désormais, un indice en dehors de la plage de données sera détecté par le contrôle (un `index` non signé – `unsigned` – évite d'avoir à vérifier si sa valeur est bien positive).

Ajoutez des commentaires en cours d'écriture

Je suis persuadé que vous éviterez des erreurs en commentant le code pendant que vous l'écrivez, au lieu d'attendre que tout soit fini pour ajouter les commentaires après coup.

Je veux bien admettre que vous n'avez pas toujours le temps d'écrire immédiatement de volumineux en-têtes et descriptions de fonctions, mais vous aurez toujours un petit moment, en cours d'écriture du programme, pour placer une brève ligne de commentaire.

Les commentaires courts doivent être informatifs. Sinon, ce n'est pas la peine d'en mettre. Ils doivent être aussi descriptifs que possible. Quand vous reviendrez sur un programme quelques jours après, des commentaires clairs, précis et à jour vous aideront grandement à retrouver rapidement et sans effort ce que vous vouliez faire.

De plus, une indentation cohérente du code et des conventions de noms claires faciliteront considérablement sa lecture et sa compréhension. C'est bien beau d'avoir un programme facile à déchiffrer après coup, mais c'est aussi bien de

pouvoir le relire tout aussi facilement en cours d'écriture. C'est peut-être à ce moment-là que vous avez le plus besoin d'aide.

Effectuez au moins une exécution pas à pas

Cela va sans dire, mais c'est encore mieux en le précisant. En tant que programmeur, il est important de comprendre tout ce que fait votre programme. Rien ne sera plus révélateur que de l'exécuter pas à pas avec un bon débogueur (celui de Dev-C++ comme celui de Visual Studio .NET sont très bons).

En outre, au fur et à mesure que vous écrivez un programme, vous aurez besoin d'informations brutes pour comprendre ce qui se passe lorsqu'il se comporte bizarrement. Là encore, une exécution pas à pas des nouvelles fonctions est irremplaçable.

Enfin, lorsqu'une fonction est terminée et prête à être ajoutée au programme, tous les chemins logiques doivent être parcourus au moins une fois. Les bogues sont plus facilement détectables lorsque la fonction est examinée de façon totalement indépendante que lorsqu'elle a été intégrée dans le programme avec tout le reste de la famille. À ce moment-là, vous aurez déjà la tête ailleurs, le cerveau en ébullition, prêt à affronter de nouveaux défis.

Évitez les opérateurs surchargés

Hormis l'opérateur d'affectation `operator=()`, vous avez intérêt à éviter la surcharge des opérateurs tant que vous ne serez pas suffisamment à l'aise avec le C++. La surcharge d'un opérateur autre que l'affectation n'est quasiment jamais nécessaire et complique le débogage. Il vaut mieux définir et utiliser judicieusement des fonctions membres publiques.

Après quelques mois d'entraînement au C++, vous pourrez revenir sur cette question et vous en donner à cœur joie.

Gérez le tas

En règle générale, un programmeur devrait allouer et libérer la mémoire heap (le célèbre *tas*) au même "niveau". Si une fonction membre `MyClass::create()` alloue un bloc de mémoire du heap et la retourne à l'appelant, il doit exister une fonction `MyClass::release()` qui rend cette mémoire au tas. En particulier, ce n'est pas à la fonction parent de `MyClass::create()` de libérer la mémoire

elle-même. Cela n'évite certes pas tous les problèmes de mémoire (la fonction parent peut "oublier" d'appeler `MyClass::release`) mais les réduit quelque peu.

Gérez les erreurs à l'aide d'exceptions

Le mécanisme des exceptions du C++ a été conçu pour gérer facilement et efficacement les erreurs. En règle générale, il est recommandé de lancer un indicateur d'erreur plutôt que de renvoyer un drapeau quelconque. Il en résultera un code plus facile à écrire, à lire et à maintenir. Bien d'autres programmeurs s'y sont mis. Vous ne voudriez quand même pas les décevoir ?

Précisons un peu les choses. Il n'est pas nécessaire de lancer une exception dans le cas d'une fonction qui renvoie un indicateur "je ne marche pas" si c'est ainsi qu'elle vit au quotidien. Supposez une fonction prenant un argument entier positif et appelée `diviseurs_autre_que_1_et_moi()` (un peu long comme nom, mais assez parlant). Vous l'appelez en lui passant un nombre premier. Ce n'est pas une erreur. Simplement, cette fonction n'a rien à dire sur ce nombre...

Évitez l'héritage multiple

À l'instar de la surcharge des opérateurs, l'héritage multiple apporte un niveau de complexité supplémentaire que vous n'êtes pas obligé de supporter, surtout si vous débutez en C++. Fort heureusement, la plupart des relations du monde réel peuvent être décrites par des héritages uniques (certains programmeurs vont jusqu'à affirmer que l'héritage multiple n'est pas du tout nécessaire, mais je serai prudent sur ce point).

Ne vous privez pas d'utiliser les classes à héritage multiple des bibliothèques proposées dans le commerce, comme les classes MFC de Microsoft. Bill et ses amis ont passé énormément de temps à les élaborer, et ils savent ce qu'ils font.

Quand vous aurez atteint un bon niveau dans l'apprentissage du C++, expérimentez diverses hiérarchies à base d'héritage multiple. Vous serez ainsi mieux préparé à faire face à des situations inhabituelles qui exigent le recours aux hiérarchies multiples pour les décrire avec précision.

Chapitre 30

Les dix fonctionnalités (facultatives) les plus importantes de Dev-C++

Dans ce chapitre :

- Personnaliser les options de l'éditeur.
- Faire correspondre accolades et parenthèses.
- Activer la gestion des exceptions.
- Inclure (parfois) des informations de débogage.
- Créer un fichier de projet.
- Personnaliser le menu d'aide.
- Réinitialiser les points d'arrêt.
- Éviter les noms de fichiers illégaux.
- Inclure des fichiers dans vos projets.
- Profilage de code.

Ce chapitre revient sur quelques-uns des paramètres de l'environnement Dev-C++ susceptibles de transformer un jour de programmation quelconque en rayon de soleil. Nous aborderons également la notion de profilage des programmes.

Personnaliser les options de l'éditeur

Programmer devrait être une expérience agréable (voire amusante). Il y a suffisamment de choses déplaisantes à gérer en C++. Inutile en plus de vous heurter à un éditeur qui ne pense pas comme vous. Heureusement, Dev-C++ sait s'adapter à vos besoins. Choisissez simplement la commande Options de l'éditeur, dans le menu Outils, et c'est parti. La Figure 30.1 montre mes réglages favoris.

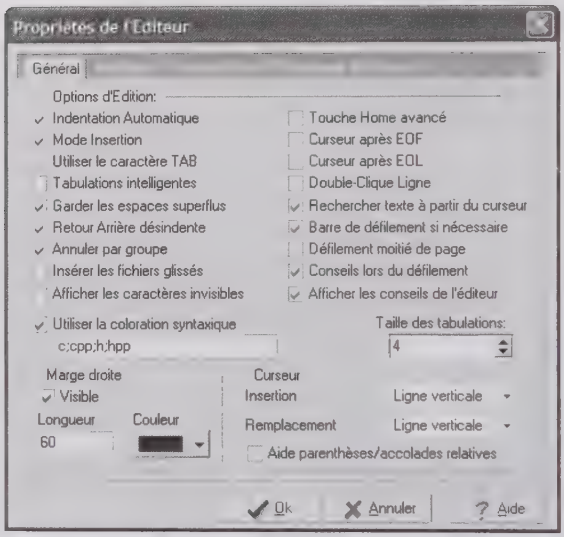


Figure 30.1
Ma configuration personnelle de l'éditeur est la meilleure.

Commençons par quelques options sans importance décisive. Par exemple, je préfère définir quatre espaces par tabulation, mais vous pouvez parfaitement préférer un autre choix. De même, je place une ligne verticale en colonne 60 pour créer une marge à droite et m'interdire ainsi de taper des lignes de code trop longues, ce qui peut gêner la lecture du code.

La coloration syntaxique demande à l'éditeur de mettre en couleur les mots de votre programme en fonction de leur nature. Par défaut, les commentaires apparaîtront en bleu clair, les mots-clés en noir gras, les chaînes entre guillemets en rouge, et ainsi de suite. Cette myriade de couleurs peut sembler un peu nauséeuse au début, mais c'est très utile dès que l'on s'y est habitué. Vous pouvez personnaliser les couleurs par défaut, mais c'est d'un intérêt très relatif.

L'indentation automatique permet de gagner du temps. L'éditeur place le curseur sur la colonne "appropriée" lorsque vous appuyez sur la touche Entrée. Normalement, il s'agit de la même colonne que le point de départ de la ligne précédente, sauf si c'est un commentaire ou une ligne vide. L'indentation automatique s'applique également lorsque vous ouvrez une accolade. Malheureusement, il n'y a pas de retour à la colonne antérieure quand vous refermez votre accolade (nul n'est parfait). L'option Retour arrière désindente (au-delà d'un français approximatif) fait exactement l'inverse.

J'ai désélectionné l'option Utiliser le caractère TAB. Ceci force l'utilisateur à utiliser des espaces, et uniquement des espaces, pour placer le curseur. Explication : cela simplifie le copier/coller des programmes vers mon traitement de texte (c'est comme cela que je fais apparaître le code source dans le livre !).

Enfin, l'option Aide parenthèses/accolades relatives vaut à elle seule un titre dans le Top 10.

Faire correspondre accolades et parenthèses

Lorsque l'option qui vient d'être signalée, Aide parenthèses/accolades relatives, est activée, Dev-C++ vérifie qu'il existe une accolade ouvrante chaque fois que vous tapez une accolade fermante. De plus, lorsque vous sélectionnez une accolade, Dev-C++ l'affiche en gras, de même que celle qui lui correspond. La même règle s'applique aux parenthèses.

Cette fonctionnalité vous aide donc à bien fermer les portes que vous ouvrez et à éviter des erreurs qui pourraient être difficiles à détecter par la suite.

En revanche, il peut y avoir une contrepartie ennuyeuse avec certaines versions de Dev-C++ (mais le problème semblerait maintenant résolu). Le problème viendrait du fait que l'éditeur contrôle la parité entre accolades ouvrantes et fermantes lorsque vous ouvrez le fichier .CPP et qu'il se bloque s'il n'en trouve pas la bonne quantité.

Dans ce cas, appuyez sur Ctrl+Alt+Suppr pour ouvrir le Gestionnaire de tâches. Sélectionnez Dev-C++ dans la liste des programmes actifs et cliquez sur Fin de tâche. Relancez Dev-C++ sans ouvrir de fichier. Il vous suffit maintenant de désactiver le contrôle des parenthèses et des accolades, puis d'ouvrir à nouveau votre fichier source.

Et si cela ne suffit pas, allez voir si le site `www.bloodshed.net` ne propose pas par hasard une version plus récente de Dev-C++.

Activer la gestion des exceptions

Nous avons abordé le mécanisme de gestion des exceptions dans le Chapitre 25. Dans le menu Outils, choisissez la commande Options du compilateur. Cliquez sur l'onglet Options. Dans le panneau de gauche, activez la ligne Génération du code. Dans le panneau de droite, cliquez si nécessaire au bout de l'option Gestion des exceptions. Choisissez alors la valeur Yes et validez.

Le mode de gestion des exceptions peut rendre votre programme légèrement plus gros, et surtout un peu plus lent. Mais c'est un prix léger à payer pour disposer du mécanisme de gestion des exceptions. Relisez le Chapitre 25 si vous ne me croyez pas.

Inclure (parfois) des informations de débogage

La génération d'informations de débogage fait également partie des options du compilateur. Restez sous l'onglet Options. Cliquez dans le panneau de gauche sur Éditeur de liens. À droite, donnez la valeur Yes à l'option Générer des informations de débogage. Le débogueur ne fonctionnera pas si ce drapeau n'est pas activé. De plus, Dev-C++ ne vous donnera dans ce cas que peu d'informations si votre programme plante.

Lorsque le mode débogage est actif, Dev-C++ mémorise l'emplacement de chaque étiquette et de chaque ligne de code dans le programme (c'est comme cela qu'il sait où définir des points d'arrêt). Même les lignes de code provenant des librairies externes que vous n'avez pas écrites vous-même sont incluses. Évidemment, toutes ces informations prennent de la place et viennent grossir le fichier exécutable.

À titre d'exemple, j'ai compilé deux fois un morceau de programme quelconque. Résultat : la version avec informations de débogage prenait 1,2 Mo, celle sans ces renseignements 440 Ko. Un vrai régime minceur.

Voici la morale de l'histoire. Activez la génération des informations de débogage pendant toute la phase de développement et de mise au point, puis désactivez ce mode pour la version définitive du programme.

Créer un fichier de projet

Vous pouvez générer un programme en partant d'un unique fichier .CPP sans passer par un projet. C'est très bien pour de petits programmes. En revanche,

les applications plus conséquentes devraient être décomposées en modules réduits, plus faciles à gérer et à comprendre. Pour lier tous ces modules, il vous faut définir un fichier de projet. J'ai décrit la façon de procéder dans le Chapitre 22.

Personnaliser le menu d'aide

Par défaut, l'aide de Dev-C++ est limitée au compilateur. Elle ne propose rien sur le langage C++ et/ou ses librairies. Heureusement, Dev-C++ vous permet de personnaliser les options de l'aide. Vous pouvez ainsi ajouter des fichiers au format .HLP ou .CHM (il s'agit de HTML compilé). Évidemment, il vous faudra pour cela trouver vous-même ces fichiers sur le Web (ne comptez pas sur le site de BloodShed pour vous aider).

Si vous avez trouvé une source intéressante, il vous suffira de choisir dans le menu d'aide l'option Éditer Menu d'Aide. Vous voyez alors l'éditeur de menu illustré sur la Figure 30.2. Choisissez le fichier voulu à l'aide du bouton Ajouter, décorez un peu grâce aux options proposées en bas de la fenêtre et vous aurez gagné le gros lot.

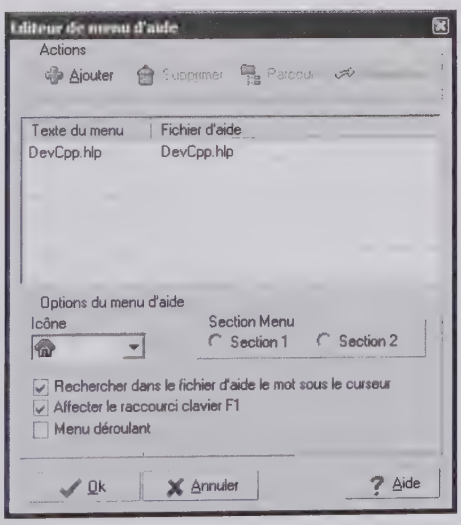


Figure 30.2 Dev-C++ vous permet de personnaliser le menu d'aide.

Vous pouvez ajouter autant de fichiers d'aide que vous le souhaitez, chacun disposant de ses propres options.

Réinitialiser les points d'arrêt

Dev-C++ pose des points d'arrêt en se basant sur la numérotation des lignes de code. En revanche, il n'actualise pas la position lorsque vous insérez ou supprimez une ligne du fichier source.

Supposons par exemple que j'aie placé un point d'arrêt sur la ligne 10 de mon programme (avec la commande Basculer Breakpoint du menu Debug ou en appuyant sur Ctrl+F5). Si j'ajoute maintenant un commentaire entre les lignes 9 et 10, c'est sur lui que va rester le point d'arrêt. Comme les commentaires ne sont pas exécutés, tout cela n'a plus de sens.

Rappelez-vous qu'il faut vérifier tous les points d'arrêt lorsque vous éditez vos fichiers sources .CPP.

Éviter les noms de fichiers illégaux

Dev-C+ n'est pas très efficace dans la détection de noms de fichiers illégaux. Au lieu de générer un avertissement clair (du genre "ce nom de fichier est illégal"), le compilateur affiche une série de messages d'erreur assez incompréhensibles.

Si votre version de Dev-C++ date un peu, il est possible que le compilateur ait du mal à gérer les noms de fichiers qui contiennent des espaces (dans le genre `Mon programme .cpp` ou pire encore `C:\Program Files\Mes sources\Mon fichier.cpp`). Mêmes causes, même remède : au moindre problème, allez voir sur le site `www.bloodshed.net` la dernière version disponible de Dev-C++.

Dev-C++ sait gérer des fichiers sur un réseau, mais la console risque d'être moins conciliante. La compilation fonctionnera sans doute, mais peut-être pas le débogage. Sans parler des obscurs messages que vous pouvez voir s'afficher ou des plantages intempestifs du système. Prudence donc.

Inclure des fichiers dans vos projets

C++ vous permet de rassembler des instructions dans des fichiers séparés que vous pouvez inclure dans différentes sources (remercions une fois de plus la directive `#include`). C++ ne pose aucune restriction sur le genre de choses que vous pouvez placer dans un fichier inclus. Cependant, vous ne devriez stocker dans ce type de fichier que les éléments suivants :

- Les prototypes de fonction.
- Les définitions de classe.
- Les modèles de n'importe quel type.
- La définition de toutes les variables globales.

En revanche, les instructions exécutables (mis à part les fonctions contenues dans les définitions des classes) sont à prohiber dans les fichiers inclus. N'oubliez pas d'ajouter leur nom dans vos listes de projets (même si elles ne contiennent pas directement du code exécutable). Procéder ainsi revient à demander à Dev-C++ de reconstruire le code source chaque fois qu'un fichier inclus est modifié.

Profilage de code

Lorsque vous écrivez votre code source, la rapidité du programme qui va en résulter n'est sans doute pas votre principal sujet d'inquiétude (ce qui ne veut pas dire que votre objectif est de faire n'importe quoi). Il est déjà assez difficile comme cela d'écrire un programme qui fonctionne correctement sans qu'il faille en plus se soucier de l'efficacité des instructions ! C'est d'ailleurs un fait avéré : si vous demandez à un développeur (ou une développeuse) ce qui lui prend le plus de temps, la réponse sera toujours à côté de la plaque (et je ne parle pas des circuits imprimés) !

Mais comment faire si votre programme est vraiment lent et que vous cherchez à l'optimiser ? Dev-C++ (comme d'ailleurs la plupart des environnements de développement) offre un outil généralement appelé *profiler* (si, exactement comme dans les polars). Certains parlent aussi de *profilage* ou encore de *profiling*, enfin ce genre de choses.

Ce sympathique petit outil analyse votre programme pour déterminer où il passe le plus clair de son temps. Quand vous avez cette information, vous pouvez réviser votre emploi du temps pour mieux vous concentrer sur les parties du programme qui méritent d'être revisitées.

Pour activer ce profilage, ouvrez le menu Outils et choisissez la commande Options du compilateur. Activez l'onglet Options. Dans le panneau de gauche, cliquez sur la ligne Profilage du code. Dans l'unique ligne de droite, placez l'indicateur Yes en face de la ligne Générer des infos de profilage.

Pour tester cette fonctionnalité, j'ai simplement modifié le programme `DeepCopy` proposé dans le Chapitre 18 :

```
//  DeepCopyProf - exemple de programme pour le profilage
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strings.h>

#include <profile.h>
using namespace std;

class Person
{
  public:
    Person(char *pN)
    {
        pName = new char[strlen(pN) + 1];
        if (pName != 0)
        {
            strcpy(pName, pN);
        }
    }

    Person(Person& p)
    {
        pName = new char[strlen(p.pName) + 1];
        if (pName != 0)
        {
            strcpy(pName, p.pName);
        }
    }

    ~Person()
    {
        if (pName != 0)
        {
            delete pName;
            pName = 0;
        }
    }

    char *pName;
};


void fn1(Person& p)
{
    // crée un nouvel objet
```

```
    // Person* p1 = new Person(p.pName);
    Person p1(p);
}
void fn2(Person p)
{
    // crée un nouvel objet
    Person* p1 = new Person(p);
    delete p1;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Person p("Ceci_est_un_nom_vraiment_long");

    for(int i = 0; i < 1000000; i++)
    {
        fn1(p);
        fn2(p);
    }
    system("PAUSE");
    return 0;
}
```

Cet exemple ne fait rien d'autre qu'appeler `fn1()` et `fn2()` quelques millions de fois. De toute façon, vous ne pouvez avoir aucune idée précise sur le déroulement d'un programme qui s'exécute en moins d'une ou deux secondes. D'ailleurs, comment feriez-vous pour l'optimiser ? Notre boucle permet d'allonger le déroulement du programme pour pouvoir récupérer quelques informations significatives.

J'ai également supprimé les instructions de sortie. L'une des premières choses que l'on découvre quand on s'intéresse à cette question, c'est que l'affichage est un processus très lent. Le temps passé à envoyer des messages à l'écran prendrait le pas sur tout le reste.

Lorsqu'on l'exécute, le programme ouvre une fenêtre de console pendant quelques secondes (sur un ordinateur moderne) avant d'afficher le message fatidique :

```
Appuyez sur une touche pour continuer...
```

Rien de bien excitant pour l'instant. J'ai ensuite sélectionné dans le menu Exécuter la commande Analyse du profil. Le résultat est illustré sur la Figure 30.3.

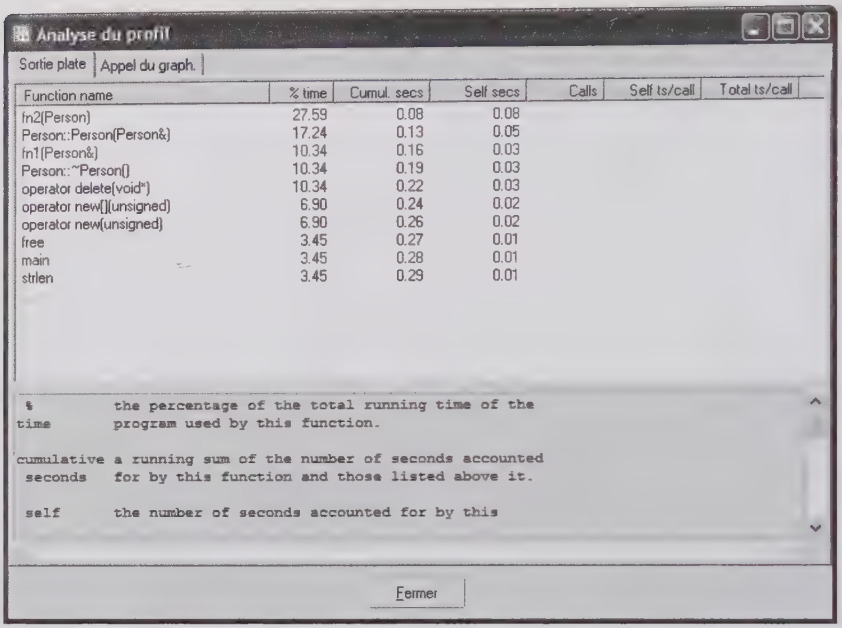


Figure 30.3
Une analyse de profil vous montre à quoi le programme passe son temps.

Interpréter un profil demande une certaine expérience. Cette fenêtre montre les fonctions invoquées au cours de l'exécution du programme (il se peut que d'autres fonctions soient présentes mais dans ce cas, elles ne sont pas appelées). La première liste les noms de ces fonctions, suivies du pourcentage de temps passé pour leur exécution. Ici, on peut remarquer que plus de 27 % ont été consommés par `fn2()`.

La quatrième colonne, Self secs, se réfère au temps total passé à l'intérieur de la fonction, soit un peu moins d'un centième de seconde pour `fn2()`. Quelle horreur !

Est-ce que cela signifie que `fn2()` est la fonction la plus lente du programme ? Pas nécessairement. Il faut aussi tenir compte d'autres paramètres, comme le nombre d'appels (par exemple, le constructeur de copie de la classe `Person`, `Person::Person(Person&)`, est appelé aussi bien par `fn1()` que par `fn2()`).

L'information la plus significative ici est que `fn2()` prend plus du double de temps qu'il n'en faut à `fn1()` pour faire son travail. Voyons cela de plus près.

`fn1()` crée une nouvelle copie de l'objet `Person` qui lui est passé par référence depuis la fonction `main()`.

En revanche, `main()` passe l'objet `Person` à `fn2()` par valeur. Ceci oblige C++ à appeler le constructeur de copie. La fonction fn2() fait alors une copie de la copie. Finalement, fn2() crée la copie sur le tas en utilisant le mot-clé `new`. Cette façon de procéder provoque un surcroît de charge.

Annexe

Glossaire

Abstraction : Concept consistant à simplifier la vision du monde réel en éléments essentiels. L'abstraction permet aux classes logicielles de représenter le monde réel d'une manière qui, autrement, aurait été difficile à exprimer.

Ami : Fonction ou classe qui, sans être un membre de la classe, a néanmoins un accès aux membres privés et protégés de cette classe.

Champ de signature : Membre de données non statique ayant reçu une valeur particulière. Cette valeur peut être vérifiée dans les fonctions membres pour déterminer si elle pointe vers un objet valide. C'est une technique de débogage très efficace.

Classe abstraite : Classe contenant une ou plusieurs fonctions virtuelles pures. Une telle classe ne peut pas être instanciée avec un objet.

Classe de base : Classe dont une autre hérite.

Classe dérivée : Classe qui hérite d'une autre.

Classification : Groupement d'objets similaires. Par exemple, les mammifères peuvent entrer dans un mode de classification comportant les animaux à sang chaud, vivipares et qui nourrissent leurs petits à la mamelle.

Constructeur : Fonction membre spéciale appelée automatiquement lorsqu'un objet est créé.

Constructeur de copie : Constructeur dont l'argument fait référence à un objet de la même classe. Par exemple, le constructeur de copie de la classe `Z` est déclaré sous la forme `Z::Z(Z&)`.

Constructeur par défaut : Constructeur possédant une liste d'arguments vides (`void`).

Copie en profondeur : Copie obtenue par réplication de l'objet et de tous les éléments que possède cet objet, y compris ceux sur lesquels pointent les membres de données de l'objet en cours de copie.

Copie superficielle : Copie binaire, bit à bit.

Déclaration de fonction : Description d'une fonction comportant son nom, le nom de la classe à laquelle elle est éventuellement associée, le nombre et le type des arguments, ainsi que le type de toute valeur retournée par la fonction.

Déclaration de prototype de fonction : Déclaration de fonction qui ne contient aucun code.

Désambiguïté : Procédé par lequel un appel décide à quelle fonction surchargée il doit se référer en comparant son usage avec les prototypes des fonctions surchargées.

Évaluation en court-circuit : Technique par laquelle la sous-expression de droite d'une expression binaire n'est pas évaluée si la valeur n'affectait pas la valeur de l'expression dans son ensemble. Ceci se produit avec les deux opérateurs `&&` et `||`. Par exemple, dans l'expression `a && b`, si l'argument placé à gauche est évalué à 0 (false, faux), il n'est pas nécessaire d'évaluer l'argument de droite car le résultat sera obligatoirement 0.

Expression : Séquence de sous-expressions et d'opérateurs. En C et en C++, une expression a toujours un type et une valeur.

Extensibilité : Capacité à pouvoir ajouter de nouvelles fonctionnalités à une classe sans modifier le code existant qui utilise cette classe.

Flux E/S : Entrées/sorties C++ basées sur les opérateurs de surcharge `operator<<` et `operator>>`. Les prototypes de ces fonctions se trouvent dans le fichier d'inclusion `iostream.h`.

Fonction de membre statique : Fonction membre n'ayant pas de pointeur `this`.

Fonction de rappel : Fonction appelée par le système d'exploitation lorsqu'un événement particulier se produit.

Fonction inline : Fonction étendue au point où elle est appelée, un peu à la manière d'une définition de macro.

Fonction outline : Fonction conventionnelle étendue au point où elle est déclarée. Toute référence subséquente à cette fonction génère un appel à l'emplacement en mémoire où elle est étendue. Voir *Fonction inline*.

Fonction membre : Fonction définie comme partie d'une classe, de la même manière qu'est défini un membre de données.

Fonction membre virtuelle : Fonction membre appelée polymorphiquement. Voir *Polymorphisme.*

Fonction virtuelle pure : Fonction membre virtuelle sans aucune implémentation.

Heap : Mémoire allouée par le programme par un appel à `malloc()`. Ce type de mémoire doit être retourné au heap (tas) par un appel à `free()`.

Héritage : Capacité, pour une classe C++, à recevoir les propriétés d'une classe existante.

Inline : Voir *Fonction inline.*

Instance d'une classe : Objet déclaré du type spécifié. Par exemple, dans la déclaration `int i`, `i` est une instance de la classe `int`.

IS_A : Relation de type "est un" entre une sous-classe et sa classe de base. Par exemple, un `colvert` IS_A `canard` car un objet de la classe `colvert` est aussi un `canard`.

Liaison dynamique : Processus par lequel le polymorphisme est mis en œuvre dans le C++.

Liaison statique : Méthode d'appel normale, non polymorphe. En langage C, tous les appels sont statiques.

Membre de classe : Synonyme de membre statique.

Membre de données statique : Membre de données non associé aux instances individuelles d'une classe. Il existe pour chaque classe une instance de chaque membre de données statique, indépendamment du nombre d'objets qui ont été créés pour cette classe.

Membre d'instance : Synonyme de membre non statique, normal.

Méthode : Synonyme de fonction membre.

Outline : Voir *Fonction outline.*

Paradigme : Manière de concevoir une approche lors d'une programmation : paradigme de la programmation orientée objet, paradigme de la programmation fonctionnelle.

Phase d'analyse : Phase de développement au cours de laquelle un problème est analysé afin d'en extraire les éléments essentiels.

Phase de codage : Étape pendant laquelle les résultats issus de la phase de conception sont transformés en code de programmation.

Phase de conception : Phase de développement au cours de laquelle la solution à un problème est formulée. La mise en œuvre de cette phase fait suite à la phase d'analyse.

Pointeur : Voir *Variable de pointeur.*

Polymorphisme : Capacité à décider quelle fonction membre surchargée doit être appelée, selon le type en temps réel de l'objet et non en fonction du type déclaré de l'objet.

Privé : Se dit d'un membre de classe accessible seulement aux autres membres de la même classe.

Programmation orientée objet : Technique de programmation reposant sur le masquage des données, l'abstraction, l'héritage et le polymorphisme.

Protégé : Membre de classe accessible aux autres membres de la même classe et aux membres de n'importe quelle sous-classe. Les membres protégés ne sont pas accessibles publiquement.

Public : Membre de classe accessible depuis l'extérieur de la classe.

Redéfinition : Le fait de fournir une fonction à une sous-classe ayant les mêmes noms et arguments que la fonction de la classe de base. Voir *Polymorphisme* et *Fonction membre virtuelle.*

Segment de code : Partie du programme qui contient les instructions exécutables.

Segment de données : Bloc de mémoire dans lequel les langages C et C++ conservent les variables globales et statiques. Voir *Segment de code* et *Segment de pile.*

Segment de pile : Partie d'un programme en mémoire qui contient les variables locales, non statiques.

Signature de fonction : Synonyme de nom de fonction complet, comprenant les types des arguments et le type de retour.

Sous-classe : Classe héritant publiquement d'une classe de base. Si `Lycéen` est une sous-classe de `Étudiant`, dans ce cas `Lycéen` IS_A `Étudiant`.

Surcharge de l'opérateur : Définition d'une signification pour les opérateurs intrinsèques lorsqu'ils sont appliqués à une classe définie par l'utilisateur.

Surcharge : Attribution d'un même nom à deux fonctions différentes. De telles fonctions doivent pouvoir être différenciées par le nombre ou par le type de leurs arguments.

this : Pointeur vers l'objet courant. `this` est un argument premier, implicite et caché vers toutes les fonctions membres non statiques. `this` est toujours du type "pointeur vers la classe courante".

Type de variable : Spécifie la taille ainsi que la structure interne de la variable. Les types de variables intégrés, ou intrinsèques, sont : `int`, `char`, `float` et `double`.

v_table : Tableau contenant les adresses des fonctions virtuelles d'une classe. Chaque classe comportant une ou plusieurs fonctions membres virtuelles doit avoir une v_table.

Variable de pointeur : Variable contenant une adresse.

Variable de référence : Variable qui sert d'alias à une autre variable.

Variable globale : Variable déclarée à l'extérieur d'une fonction et de ce fait, accessible par toutes les fonctions.

Variable locale : Variable déclarée dans une fonction et de ce fait, accessible uniquement par cette fonction.

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

" ENFIN DES LIVRES QUI VOUS RESSEMBLENT ! "

Egalement disponibles dans la même collection !

Titre	Auteur	ISBN	Code
3ds max 5 pour les Nuls	S. Mortier, phD	2-84427-454-4	653603 1
Access 2000 pour les Nuls	J. Kaufeld	2-84427-238-X	65 3077 8
Access 2002 pour les Nuls	J. Kaufeld	2-84427-969-4	65 3234 5
Access 2003 Pour les Nuls	J. Kaufield	2-84427-506-0	65 3642 9
Apprendre à programmer pour les Nuls 3e éd.	Wallace Wang	2-84427-577-6	65 3775 7
ASP.NET pour les Nuls	B. Hatfield	2-84427-331-9	65 3381 4
AutoCAD 2005 Pour les Nuls	M. Middlebrook	2-84427-643-1	65 4076 9
C# pour les Nuls	S. R. Davis	2-84427-259-2	65 3303 8
C++ pour les Nuls 2e éd.	S. R. Davis	2-84427-642-3	65 4075 1
Combattre les hackers Pour les Nuls	K. Beaver	2-84427-640-7	65 4074 4
Créer des pages Web pour les Nuls 6e éd.	B. Smith, A. Bebak	2-84427-261-4	65 3305 3
Créer un réseau domestique Pour les Nuls	K. Ivens	2-84427-624-5	65 4072 8
Créer un réseau sans fil pour les Nuls	Danny Briere, Walter R. Bruce III, Pat Hurley	2-84427-457-9	65 3606 4
Créer un site Web pour les Nuls 5e éd.	D. & R. Crowder	2-84427-644-X	65 4077 7
Dépanner et optimiser Windows pour les Nuls	D. Gookin	2-84427-452-8	65 3601 5
Design web pour les Nuls	L. Lopuck	2-84427-932-5	65 3168 5
Devenir Webmaster pour les Nuls	B. Kienan, D. A. Tauber	2-84427-930-9	65 3166 9
Dreamweaver MX 2004 pour les Nuls	J. Warner, S. Gardner	2-84427-553-2	65 3769 0
eCommerce pour les Nuls	G. Holden	2-84427-240-1	65 3079 4
Excel 2000 pour les Nuls	G. Harvey	2-84427-237-1	65 3076 0
Excel 2002 pour les Nuls	G. Harvey	2-84427-947-3	65 3194 1
Excel 2003 pour les Nuls	G. Harvey	2-84427-456-0	65 3605 6
Easy Media Creator 7 Pour les Nuls	G. Harvey	2-84427-646-6	65 4079 3
Final Cut Express 2 Pour les Nuls	H. Kobler	2-84427-625-3	65 4073 6
Fireworks 4 pour les Nuls	D. Sahlin	2-84427-888-4	65 3151 1
Flash MX 2004 pour les Nuls	G. Leete, E. Finkelstein	2-84427-555-9	65 3771 6
FrontPage 2002 pour les Nuls	A. Dornfest	2-84427-973-2	65 3227 9
GoLive 5 pour les Nuls	B. Sanders	2-84427-885-X	65 3113 1
Gravure des CD et DVD pour les Nuls 3e éd.	M. L. Chambers	2-84427-549-4	65 3764 8
HTML 4 pour les Nuls 3e éd.	E. Tittel, N. Pitts, C. Valentine	2-84427-890-6	65 3153 7
Illustrator 9 pour les Nuls	T. Alspach, M. LeClair	2-84427-935-X	65 3184 2
iMac pour les Nuls 3e éd.	D. Pogue	2-84427-258-4	65 3302 0
Internet Explorer 5.5 pour les Nuls	D. Lowe	2-84427-244-4	65 3083 6
Internet Explorer 6 pour les Nuls	D. Lowe	2-84427-251-6	65 3295 6
Internet pour les Nuls 10e éd.	J. R. Levine, C. Baroudi, M. Levine Young	2-84427-550-8	65 3766 6
Java 2 pour les Nuls	B. Burd	2-84427-940-6	65 3192 5
JavaScript pour les Nuls	E. Van der Veer	2-84427-887-6	65 3115 6
Le Mac pour les Nuls 8e éd.	D. Pogue	2-84427-260-6	65 3304 6
Les réseaux pour les Nuls	D. Lowe	2-84427-886-8	65 3114 9
Linux pour les Nuls 5e éd.	Dee-Ann LeBlanc	2-84427-589-3	65 3786 4
Mac et iMac		2-84427-557-5	65 3772 4
Mac OS/X Pour les Nuls	B. Levitus	2-84427-892-2	65 3155 2
Mac OSX v 10.2 pour les Nuls	B. Levitus	2-84427-390-4	65 3499 4
Mac OS X Panther Pour les Nuls	B. LeVitus	2-84427-552-4	65 3768 2
Money 2003 pour les Nuls	P. Weverka	2-84427-388-2	65 3497 8
Office 2000 pour les Nuls	W. Wang, R. C. Parker	2-84427-235-5	65 3074 5
Office 2003 pour les Nuls	Wallace Wang	2-84427-505-2	65 3641 1
Office 2004 Mac Pour les Nuls	T. Negrino	2-84427-645-8	65 4078 5
Office v. X Mac pour les Nuls	T. Negrino	2-84427-311-4	65 3340 0
Office XP pour les Nuls	W. Wang	2-84427-967-8	65 3232 9
Outlook 2002 pour les Nuls	B. Dyszel	2-84427-974-0	65 3228 7
Outlook 2003 Pour les Nuls	B. Dyszel	2-84427-509-5	65 3645 2
PC pour les Nuls 10e éd.	D. Gookin	2-84427-551-6	65 3767 4

" ENFIN DES LIVRES QUI VOUS RESSEMBLENT ! "

Egalement disponibles dans la même collection !

Titre	Auteur	ISBN	Code
Photographie numérique pour les Nuls 5e éd.	Julie Adair King	284427-579-6	65 3777 3
Photoshop 7 pour les Nuls	D. McClelland	2-84427-344-0	65 3404 4
Photoshop CS Pour les Nuls	D. McClelland, Ph. Davis	2-84427-511-7	65 3647 8
Photoshop Elements 2 pour les Nuls	D. McClelland, G. Fott	2-84427-387-4	65 3496 0
PHP 5 Pour les Nuls	Janet Valade	2-84427-607-5	65 4055 3
PHP et MySQL pour les Nuls 2e éd.	Janet Valade	2-84427-590-7	65 3787 2
PowerPoint 2003 Pour les Nuls	Doug Lowe	2-84427-510-9	65 3646 0
Les réseaux pour les Nuls 6e éd.	D. Lowe	2-84427-389-0	65 3498 6
Retouche Photo pour les Nuls	Julie Adair King	2-84427-353-X	65 3413 5
Sécurité des réseaux pour les Nuls	Chey Cobb	2-84427-420-x	65 3546 2
Sécurité sur Internet pour les Nuls	J.R. levine, R. Everett-Church, G. Stebben	2-84427-352-1	65 3412 7
SQL pour les Nuls	A. G. Taylor	2-84427-931-7	65 3167 7
TCP/IP pour les Nuls	C. Leiden, M. Wilensky	2-84427-241-X	65 3080 2
UNIX pour les Nuls 4e éd.	J.R. Levine, M. Levine Young	2-84427-250-9	65 3294 9
VBA pour les Nuls 4e éd.	John Paul Mueller	2-84427-534-6	65 3719 5
Vidéo numérique pour les Nuls 4e éd.	M. Doucette	2-84427-580-X	65 3778 1
Visual Basic .NET pour les Nuls	W. Wang	2-84427-970-8	65 3235 2
Windows 2000 Professionnel pour les Nuls	A. Rathbone	2-84427-882-5	65 3110 7
Windows 2000 Server pour les Nuls	E. Tittel	2-84427-881-4	65 3109 9
Windows 98 pour les Nuls	A. Rathbone	2-84427-234-7	65 3073 7
Windows Me pour les Nuls	A. Rathbone	2-84427-239-8	65 3078 6
Windows XP pour les Nuls 3e éd.	A. Rathbone	2-84427-596-6	65 4053 8
Word 2000 pour les Nuls	D. Gookin	2-84427-236-3	65 3075 2
Word 2002 pour les Nuls	D. Gookin	2-84427-946-5	65 3193 3
Word 2003 pour les Nuls	D. Gookin	2-84427-455-2	65 3604 9

C++ pour les Nuls
2e édition

Mon opinion sur ce livre :

❒ Excellent ❒ Moyen

❒ Satisfaisant ❒ Insuffisant

Ce que j'aime dans ce livre :

_____

_____

_____

_____

_____

_____

_____

_____

Mes suggestions pour l'améliorer :

_____

_____

_____

_____

_____

_____

_____

_____

Mon équipement :

- Matériel : _____

- Système d'exploitation : _____

J'utilise mon ordinateur :

❒ Au bureau ❒ À l'école

❒ À la maison ❒ Autre : _____

Lieu d'achat du livre :

- Pays : _____

- Ville : _____

❒ Grande librairie ❒ Petite librairie

❒ Grande surface ❒ Hypermarché

❒ Magasin spécialisé ❒ Autre : _____

Mon adresse :

Nom : _____

Prénom : _____

Adresse : _____

_____

Code postal : _____

Ville : _____

Pays : _____

En informatique, je me considère comme :

❒ Débutant ❒ Expérimenté

❒ Initié ❒ Professionnel

J'ai vraiment adoré ce livre !

Vous pouvez citer mon témoignage dans vos documents promotionnels. Voici mon numéro de téléphone en journée :

Fiche lecteur à découper ou à photocopier, et à nous retourner à :

FIRST
Interactive
27, rue Cassette
F-75006 Paris - France
Tél. : 01 45 49 60 00 – Fax : 01 45 49 60 01

Découvrez en exclusivité nos prochaines parutions sur internet :
http://www.efirst.com

2-05

Avec les Nuls, apprenez à mieux vi au quotidien !

Pour comprendre enfin quelque chose à la micro-informatique !

Vous voici confronté à un micro-ordinateur – plus par nécessité que par goût, avouez-le –, sans savoir par quel bout prendre cet instrument barbare et capricieux. Oubliez toute appréhension, cette nouvelle collection est réellement faite pour vous !

Le C++ en toute simplicité !

C++ est le langage de programmation le plus populaire. Dans l'esprit de beaucoup de gens, qui dit langage dit compliqué. Contrairement à d'autres ouvrages sur la programmation, *C++ pour les Nuls* considère que le "pourquoi" est aussi important que le "comment". Les fonctionnalités du langage C++ sont comme un puzzle. Et, plutôt que de présenter bêtement des fonctions, il nous a paru plus important de faire en sorte que vous compreniez comment celles-ci s'agencent entre elles. Avec *C++ pour les Nuls*, vous apprendrez simplement à structurer un programme et à utiliser toute la puissance du langage !

Découvrez

Variables, opérateurs arithmétiques et opérations logiques.

Créer des fonctions, stocker des données, travailler avec les pointeurs.

Mise au point d'un programme, ou partir à la chasse aux bugs.

La programmation orientée objet.

Les constructeurs et les destructeurs (il en faut pour tous les goûts).

L'ESPRIT DES NULS

L'informatique en français dans le texte.

Tout et seulement tout ce que vous devez savoir.

Un accès rapide à l'information grâce à un système d'icônes d'aide à la navigation.

Les dix commandements.

Une bonne dose d'humour.

Suivez le guide

À la vue de ce symbole, si vous êtes allergique à la technique, passez votre chemin.

Cette icône signale une manipulation qui va vous simplifier la vie.

Désolé, il faut quand même retenir ceci.

65 4075 1 VIII-04 21,90 €
ISBN-2-84427-642-3
9 782844 276421

FIRST Interactive
www.efirst.com

Rayon librairie : Informatique / C++

Public : Débutant / Initié