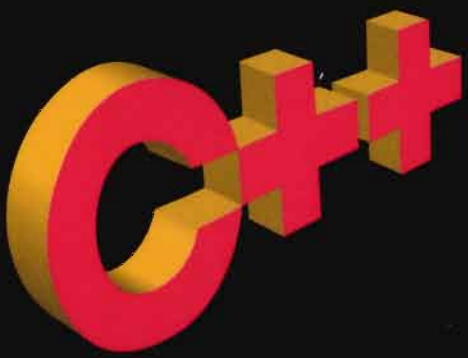
comprendre et utiliser C++ pour programmer objets

G. CLAVEL I. TRILLAUD L. VEILLON



MASSON

Pour la liste complète, voir catalogue

MÉTHODES DE PROGRAMMATION ET ALGORITHMIQUE

ANALYSE INFORMATIQUE POUR LES IUT ET BTS. P.-A. Goupille et J.-M. Rousse.
CONCEPTION OBJET DES STRUCTURES DE DONNÉES. B. Quément.
ÉLÉMENTS D'ALGORITHMIQUE. D. Beauquier, J. Berstel et Ph. Chrétienne.
CONCEPTION ET PROGRAMMATION PAR OBJETS. Techniques, outils, et applications. J.-P. Aubert et P. Dixneuf.
- ALGORITHMIQUE. Conception et analyse. G. Brassard et P. Bratley.

PROGRAMMATION IMPÉRATIVE ET PROGRAMMATION DÉCLARATIVE. Ph. Collard.
- INTRODUCTION À LA PROGRAMMATION
  1. Algorithmique et langages. J. Biondi et G. Clavel.
  2. Structures de données. G. Clavel et J. Biondi.
  3. Exercices corrigés. G. Clavel et F.B. Jorgensen.
- SCHÉMAS ALGORITHMIQUES FONDAMENTAUX. P.-C. Scholl et J.-P. Peyrin.

ASSEMBLAGE, MODÉLISATION, PROGRAMMATION (80×86). M. Margenstern.
LA COMPRESSION DES DONNÉES. Méthodes et applications. G. Held.
- ALGORITHMIQUE ET PRÉSENTATION DES DONNÉES
  1. Files, automates d'états finis. M. Lucas, J.-P. Peyrin et P.-C. Scholl.
  2. Évaluations, arbres, graphes, analyse de texte. M. Lucas.
  3. Récursivité et arbres. P.-C. Scholl.

PROCESSUS CONCURRENTS. Introduction à la programmotion parallèle. M. Ben Ari.
PROCESSUS SÉQUENTIELS COMMUNICANTS. C.A.R. Hoare.
CONSTRUCTION ET VÉRIFICATION DE PROGRAMMES. R. Backhouse.
- COMPRENDRE ET UTILISER C++ POUR PROGRAMMER OBJETS. G. Clavel, I. Trillaud et L. Veillon.

LES LANGAGES ET LEUR TRAITEMENT

LE LANGAGE C. B.W. Kernighan et D.M. Ritchie.
LE LANGAGE C. Solutions. C.L. Tondo et S.E. Gimpel.
CONSTRUCTION LOGIQUE DE PROGRAMMES COBOL. Mise à jour COBOL 85. M. Koutchouk.
- LANGAGE C norme ANSI. Vers une opproche orientée objet. Ph. Drix.

TURBO INITIATION À LA PROGRAMMATION EN PASCAL, pour Turbo-Poscal 4.0, 5.0, 5.5, 6.0. J. Thiel, C. Léger et G. Jacquet.
- MÉTHODOLOGIE DE LA PROGRAMMATION EN LANGAGE C. Principes et applications. J.-P. Braquelaire.
- (COMMON) LISP. Une introduction à la programmation. H. Wertz.

COBOL. Perfectionnement et pratique. M. Koutchouk.
PROGRAMMER EN C++. S.C. Dewhurst et K.T. Stark.
LE GÉNÉRATEUR AUTOMATIQUE DE PROGRAMME RPG. M. Rémy.
LANGAGE C : MANUEL DE RÉFÉRENCE. S.H. Harbison et G.L. Steele.
LANGAGE C. PROBLÈMES ET EXERCICES. A.R. Feuer.
- LANGAGE C, norme ANSI. Variations sur des thèmes Pascal. Ph. Drix.
- LES LANGAGES DE PROGRAMMATION. Concepts essentiels, évolution et classification. J. Lonchamp.

INTRODUCTION AU LANGAGE ADA. D. Price.
- TRAITEMENT DES LANGAGES ÉVOLUÉS. Compilation. Interprétation. Support d'exécution. Y. Noyelle.
- APPRENDRE PASCAL ET LA RÉCURSIVITÉ. Avec exemples en Turbo-Pascal. R. Romanetti.

LE LANGAGE PASCAL. J.-M. Crozet et D. Serain.
MANUEL ADA. LANGAGE NORMALISÉ COMPLET. M. Thorin.

INFORMATIQUE THÉORIQUE

THÉORIE DES LANGAGES ET DES AUTOMATES. J.-M. Autebert.
CALCULABILITÉ ET DÉCIDABILITÉ. J.-M. Autebert.

- Cours rédigé et enseigné par un professeur francophone.

(Suite page 3 de couverture)

comprendre et utiliser C++ pour programmer objets

CHEZ LE MÊME ÉDITEUR

Des mêmes auteurs

DÉCOUVRIR LA PROGRAMMATION ORIENTÉE OBJETS avec SMALLTALK V, par G. CLAVEL et L.VEILLON. 1991, 242 pages.

INTRODUCTION À LA PROGRAMMATION, par G. CLAVEL et J. BIONDI. *Collection MIM-Programmation-Algorithmique.*

Tome 1. — Algorithmique et langages. Préface de O. LECARME. 1987, 3e édition révisée et complétée, 280 pages.

Tome 2. — Structures des données. 1989, 2e tirage, 272 pages.

Tome 3. — Exercices corrigés, par G. CLAVEL et F.B. JØRGENSEN. 1985, 176 pages.

Dans la collection MIM

LE DÉVELOPPEMENT DE LOGICIEL EN C++, par D. WINDER. 1994, 568 pages.

PROGRAMMER EN C++, par S.C. DEWHURST et K.T. STARK. Traduit de l'anglais par J.-F. GROFF. 1990, 208 pages.

LE LANGAGE C, par B.W. KERNIGHAN et D.M. RITCHIE. Traduit de l'anglais par J.-F. GROFF et E. MOTTIER. 1992, 2e édition, 3e tirage, 296 pages.

LE LANGAGE C, SOLUTIONS AUX EXERCICES DE L'OUVRAGE DE B.W. KERNIGHAN ET D.M. RITCHIE, par C.L. TONDO et S.E. GIMPEL. Traduit de l'anglais par A. BERTIER. 1992, 2e édition, 2e tirage, 168 pages.

LANGAGE C, NORME ANSI. VERS UNE APPROCHE ORIENTÉE OBJET, par P. DRIX. 1990, 2e tirage, 376 pages.

VARIATIONS *C ANSI* SUR DES THÈMES PASCAL, par P. DRIX. 1991, 216 pages.

MÉTHODOLOGIE DE LA PROGRAMMATION EN LANGAGE C. PRINCIPES ET APPLICATIONS, par J.-P. BRAQUELAIRE. 1994, 2e édition, 528 pages.

CONCEPTION OBJET DES STRUCTURES DE DONNÉES. RÉALISATION EN LANGAGE C, par B. QUÉMENT. 1992, 252 pages.

CONCEPTION ET PROGRAMMATION PAR OBJET. TECHNIQUES, OUTILS ET APPLICATIONS, par J.-P. AUBERT et P. DIXNEUF. 1991, 192 pages.

Dans la collection MIPS

INGÉNIERIE DES OBJETS. APPROCHE CLASSE-RELATION, APPLICATION À C++, par P. DESFRAY. 1992, 244 pages.

ANALYSE ORIENTÉE OBJETS, par P. COAD et E. YOURDON. Traduit de l'anglais par A. BOUGHLAM. Préface de M. GALINIER. 1991, 216 pages.

CONCEPTION ORIENTÉE OBJET, par P. COAD et E. YOURDON. Traduit de l'anglais par A.-B. FONTAINE. 1993, 200 pages.

Autres ouvrages

C++, par B. BEAUDOING et D. EDELSON. *Collection Objectif.* 1994, 200 pages.

C, C++ ET UNIX. INITIATION AUX LANGAGES ET ENVIRONNEMENT, par G. KHALIL. *Collection Techniques de l'Informatique.* 1991, 176 pages.

MCO. MÉTHODOLOGIE GÉNÉRALE D'ANALYSE ET DE CONCEPTION DES SYSTÈMES D'OBJETS, par X. CASTELLANI.

Tome 1. — L'ingénierie des besoins. 1993, 420 pages.

Tome 2. — L'ingénierie de l'implantation. 320 pages. À paraître.

TECHNOLOGIE DES SYSTÈMES D'INFORMATION : AU CŒUR DES NOUVELLES STRATÉGIES D'ENTREPRISE, par T. GUNTON. *Collection Stratégies et Systèmes d'Information.* 1993, 344 pages.

MANUELS INFORMATIQUES MASSON

comprendre et utiliser C++ pour programmer objets

Gilles CLAVEL

Directeur consultant de la société IMA-Informatique
Professeur à l'Institut National Agronomique

Isabelle TRILLAUD

Responsable études et projets de la société IMA-Informatique

Luc VEILLON

Ingénieur de recherche
Directeur du centre de calcul de l'Institut National Agronomique

MASSON
Paris Milan Barcelone
1994

Ce logo a pour objet d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, tout particulièrement dans le domaine universitaire, le développement massif du « photocopillage ».
Cette pratique qui s'est généralisée, notamment dans les établissements d'enseignement, provoque une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que la reproduction et la vente sans autorisation, ainsi que le recel, sont passibles de poursuites. Les demandes d'autorisation de photocopier doivent être adressées à l'éditeur ou au Centre français d'exploitation du droit de copie : 3, rue Hautefeuille, 75006 Paris. Tél. : 43 26 95 35.

ISBN : 2-225-**84527**-1
ISSN : 0249-6992

MASSON S.A. 120, bd Saint-Germain, 75280 Paris Cedex 06
MASSON S.p.A. Via Statuto 2/4, 20121 Milano
MASSON S.A. Avenida Principe de Asturias 20, 08012 Barcelona

Table des matières

Avant-propos IX
Avertissement au lecteur XI
1 Des techniques de base aux mécanismes objets 1
1.1 - Simulation d'un distributeur 1
1.2 - Définir des variables, affecter des valeurs 2
1.3 - Schémas itératifs et conditionnels 3
1.3.1 - *L'énoncé while* 3
1.3.2 - *Structure de bloc, énoncés for et do ... while* 4
1.3.3 - *Expression d'un choix* 6
1.4 - Utilisation de fonctions, structure d'un programme 7
1.4.1 - *La fonction main* 7
1.4.2 - *Définition et déclaration de fonctions* 8
1.4.3 - *Le corps de la fonction main* 9
1.4.4 - *Plusieurs fonctions pour un programme* 10
1.4.5 - *Utilisation de la valeur renvoyée par une fonction* 11
1.4.6 - *Programme composé d'un seul module* 12
1.4.7 - *Programme composé de plusieurs modules* 13
1.5 - Objets et messages 18
1.6 - Notion de classe 20
1.7 - Données membres et fonctions membres 22
1.8 - Accès public ou privé 24
1.9 - Utilisation d'un constructeur 26
1.10 - Allocation dynamique 27
1.11 - Une simulation élémentaire 29

2 Techniques de base en C++ 33
2.1 - Une classe pour enregistrer des entiers 33
2.2 - Créer des objets : choix d'un constructeur 34
2.2.1 - *Constructeur par défaut* 34
2.2.2 - *Définir son constructeur* 35
2.2.3 - *Allocation dynamique* 38
2.2.4 - *Rétablir le constructeur par défaut* 39
2.2.5 - *Plusieurs constructeurs dans une même classe* 40
2.2.6 - *Signature d'une fonction, arguments par défaut* 43
2.2.7 - *Fonctions à nombre variable d'arguments* 44

2.3 - Macros et fonctions en ligne 47
2.4 - Destructeur par défaut et destructeur explicite 50
2.4.1 - *Le mécanisme implicite de destruction* 50
2.4.2 - *Destructeur explicite* 51
2.4.3 - *Les opérateurs new et delete* 52
2.4.4 - *Portée et durée de vie d'une variable* 53
2.4.5 - *Durée de vie d'une variable dynamique* 54
2.5 - Fonctions : transmission de paramètres 55
2.5.1 - *Echanger les valeurs de deux paramètres dans une fonction* 56
2.5.2 - *Modes de transmission de paramètres en C++* 57
2.5.3 - *&, la spécification de référence* 58
2.5.4 - *Valeurs renvoyées par une fonction (ou un opérateur)* 58
2.6 - Et si nous reparlions des constructeurs ? 61
2.6.1 - *Constructeur et conversion de type* 61
2.6.2 - *Constructeur-copie* 63
2.6.3 - *Importance du constructeur par défaut* 65
2.7 - Le modificateur const 66
2.7.1 - *Déclarer une constante dans un programme* 67
2.7.2 - *Partager une constante entre plusieurs fichiers sources* 67
2.7.3 - *Transmettre une constante en paramètre de fonction* 67
2.7.4 - *Prévenir toute modification d'un paramètre dans une fonction* 68
2.7.5 - *Pointeurs et constantes* 68

3 Créer sa première classe fonctionnelle 71
3.1 - Une classe pour gérer les chaînes de caractères 71
3.2 - Première implémentation de la classe String 73
3.2.1 - *Les données membres* 73
3.2.2 - *Définition de la classe String* 74
3.2.3 - *Définition d'accesseurs pour la classe String* 76
3.3 - Surcharge des opérateurs 78
3.3.1 - *Désigner l'objet qui reçoit le message* 80
3.3.2 - *Surcharge de l'opérateur d'indexation* 80
3.3.3 - *Surcharge de l'opérateur d'affectation* 82
3.3.4 - *Surcharge de l'opérateur +* 84
3.3.5 - *Surcharge des opérateurs de comparaison* 86
3.4 - Opérateurs d'entrées/sorties 88
3.4.1 - *Surcharge de l'opérateur de sortie* 88
3.4.2 - *Surcharge de l'opérateur d'entrée* 90
3.5 - Compter le nombre d'instances d'une classe 91
3.5.1 - *Variables de classe* 91
3.5.2 - *Notion de méthode de classe* 93
3.5.3 - *Utilisation des variables de classe pour la mise au point de la classe* 94
3.5.4 - *Une classe String fonctionnelle* 95
3.6 - Annexe : la classe String 96
3.6.1 - *Fichier d'en-tête entete.h* 96
3.6.2 - *Fichier d'en-tête c_String.h* 96
3.6.3 - *Fichier c_String.cpp* 98

4 Utiliser une classe existante : l'héritage 103
4.1 - Héritage : construire et utiliser la classe de base 103
4.1.1 - *Une classe de base : Produit* 104
4.1.2 - *Un premier exemple d'héritage : ProduitPerissable* 105
4.1.3 - *Problèmes d'accès aux membres de la classe de base* 108
4.1.4 - *Surcharge d'une donnée membre* 114
4.2 - Héritage des fonctions membres et des opérateurs 115
4.2.1 - *Constructeurs* 115
4.2.2 - *Les fonctions virtuelles* 118
4.2.3 - *Héritage des opérateurs* 123
4.2.4 - *Transtypage et héritage* 128
4.3 - Héritage multiple : ProduitFugace 130
4.3.1 - *Créer une classe héritant de deux classes dérivées* 131
4.3.2 - *Constructeur d'une classe à héritage multiple* 132
4.3.3 - *Transtypage et classe de base virtuelle* 134
4.3.4 - *Destructeurs virtuels* 136

5 Construire et organiser une librairie de classes 139
5.1 - Une classe Tableau ? 139
5.1.1 - *Des tableaux d'entiers* 139
5.1.2 - *Un tableau de pointeurs* 141
5.2 - Une classe Object et sa descendance 142
5.2.1 - *Deux sous-classes pour la classe Object* 144
5.2.2 - *Un Integer peut-il être un int ?* 146
5.2.3 - *Comparer des objets comparables* 146
5.2.4 - *Conversion par constructeur ou par cast ?* 149
5.2.5 - *Le transtypage vers une référence* 150
5.2.6 - *Comparer des objets non comparables ?* 150
5.3 - Une classe Array pour répertorier des descendants de Object 154
5.3.1 - *Répertorier un objet dans un tableau* 155
5.3.2 - *Un objet pour représenter l'absence d'objet* 157
5.3.3 - *Récapitulons les accesseurs de Array* 160
5.3.4 - *Itérer sur le contenu d'une instance de Array* 162
5.3.5 - *Une classe d'itérateurs* 163
5.4 - Une hiérarchie de classes-conteneurs 166
5.4.1 - *Une classe abstraite : Collection* 166
5.4.2 - *La fonction membre NewIterator* 169
5.4.3 - *Le polymorphisme de la fonction membre Includes* 171
5.4.4 - *Pour terminer, une implémentation rapide* 172

6 Développer une application en C++ 177
6.1 - Description du distributeur 177
6.2 - Les pièces et les produits 179
6.2.1 - *Les pièces* 179
6.2.2 - *Les produits* 181
6.2.3 - *La classe Date : une classe annexe* 185
6.2.4 - *Une amélioration de l'implémentation de la classe Object* 186
6.3 - Le distributeur 187

6.3.1 - *Une classe Sac* 187
6.3.2 - *Structure interne du distributeur* 189
6.3.3 - *Traitements disponibles* 189
6.4 - Les clients 192
6.4.1 - *Représentation du client* 192
6.4.2 - *Les actions du client* 193
6.5 - Exemple d'exécution : la fonction main 194
6.6 - Annexe : les classes de l'application 198
6.6.1 - *La hiérarchie de classes* 198
6.6.2 - *La classe Piece* 198
6.6.3 - *La classe Produit* 200
6.6.4 - *La classe Date* 202
6.6.5 - *La classe Sac* 204
6.6.6 - *La classe Distributeur* 206
6.6.7 - *La classe Client* 212

Annexe A : compléments C/C++ 217
A1 - Séquences d'échappement 217
A2 - Mot-clé static 218
A3 - Arguments de la fonction main 220
A4 - Priorités des opérateurs 221

Annexe B : les templates 223
B1 - Pourquoi les templates ? 223
B2 - Une classe Bag en C++ 223
B3 - Définir une classe paramétrée 226
B4 - Modèles de fonctions 227

Index 229

Avant-propos

Voici un ouvrage « pratique », sur un langage difficile. En reprenant, à deux mots près[1], la première phrase de l'avant-propos de notre *Introduction à la programmation orientée objets*, nous ne cédons pas à la facilité d'une pirouette de style : nous affirmons une continuité. En 1991, nous pressentions le mouvement de fond de la programmation orientée objets. Aujourd'hui, en 1994, nous constatons le succès industriel d'un langage.

C++ est certainement un langage difficile, sans doute le plus complexe de tous ceux que nous utilisons et avons utilisés dans notre équipe. Mais c'est surtout un langage dont le succès se confirme de mois en mois. Les chiffres de ventes des éditeurs de compilateurs sont là pour en témoigner.

Pour un langage difficile, mais paradoxalement de plus en plus utilisé, nous avons jugé qu'il manquait un ouvrage pratique. Nous avons donc écrit un livre qui devrait permettre au lecteur de démarrer sans être rapidement noyé dans un océan de complexités. Notre démarche a constamment respecté trois objectifs :

N'exiger aucun prérequis du lecteur (si ce n'est une pratique préalable de la programmation). On peut aborder ce livre sans connaître le langage C ni les concepts objets.

Conserver l'approche pédagogique progressive qui a fait le succès de nos précédents manuels. La plupart des ouvrages existants sur C++ sont articulés autour des fonctionnalités du langage, qu'ils présentent les unes après les autres. Nous proposons au lecteur une progression pédagogique dans la découverte des concepts objets. Cette progression introduit chaque mécanisme de C++ au moment opportun, pour répondre à un besoin bien ressenti par le lecteur.

Maîtriser la complexité, en limitant la dimension du livre. Les ouvrages sur C++ dépassent très souvent les six cents pages. Un tel volume ne convient pas à une initiation. Nous avons donc restreint notre pagination en conservant cependant l'étude de toutes les caractéristiques de C++ indispensables à un bon démarrage.

---

1 *Voici un ouvrage « pratique »*, sur un thème à la mode (G. Clavel & L. Veillon, février 1991)

Notre livre s'articule en trois parties :

- La première partie, constituée des deux premiers chapitres, permet au lecteur qui ne connaîtrait pas le langage C de disposer des éléments nécessaires. Elle introduit aussi les concepts objets élémentaires, illustrés par les mécanismes correspondants en C++ (classes, objets, messages, instanciation).
- La seconde partie, qui recouvre les chapitres 3 et 4, initie le lecteur à la conception et la définition d'une classe puis à l'utilisation de l'héritage.
- La dernière partie s'intéresse à la conception et à l'utilisation d'un ensemble de classes. On étudie d'abord les techniques de construction d'une librairie de classes (chapitre 5) puis on présente un exemple complet d'application (chapitre 6).

Notre livre s'adresse à tous ceux qui doivent programmer en C++, quelles que soient leurs préoccupations. C++ est un langage objets qui se pratique à deux niveaux. Le niveau applicatif est celui des programmeurs concernés par les particularités de l'application qu'ils construisent. Ceux-ci utilisent la plupart du temps des classes qu'ils n'ont pas à implémenter eux-mêmes. Le niveau conceptuel objet est celui des programmeurs qui conçoivent et implémentent des classes qui seront utilisées pour une ou plusieurs applications. Les deux catégories de programmeurs devraient utiliser avec profit notre ouvrage. A tous, il apportera les connaissances de base indispensables. Aux programmeurs applicatifs il fournira les connaissances nécessaires pour comprendre et utiliser une librairie de classes. Pour les concepteurs de classes il expose de manière progressive, au chapitre 5, les mécanismes à mettre en oeuvre.

Nous voulons remercier ici tous ceux qui nous ont aidé pour la rédaction de cet ouvrage. Le département de mathématique et informatique de l'Institut National Agronomique a encouragé notre travail. La compagnie CGI-informatique participe depuis 1990 au financement de notre laboratoire IGLOO (interfaces graphiques et langages orientés objets). Enfin, la société IMA-informatique nous a fourni un support logistique et technique fort précieux. A ces trois organismes nous exprimons notre reconnaissance.

Notre gratitude va aussi à Frédérique Darcy-Moreau, qui a bien voulu relire notre manuscrit : nous avons beaucoup apprécié, à cette occasion, sa grande compétence et son expérience professionnelle du langage C++. Olivier Clavel a préparé, avec une grande efficacité, de nombreux éléments techniques pour le chapitre 5 ; Philippe Clavel a patiemment confectionné bon nombre de figures : nous les remercions tous les deux. Enfin, nous ne saurions oublier Frédérique Lauque qui a assuré la composition de nos textes avec une compétence, une patience et une disponibilité que nous avons beaucoup appréciées.

Avertissement au lecteur

Le texte de ce livre a été composé avec une police Times dont le corps et le style varient suivant les parties du texte (titres, paragraphes, notes). Pour la présentation des exemples de code C++, nous avons fait un certain nombre de choix typographiques destinés à faciliter la lecture. Nous les explicitons ci-dessous.

Séquence de code

Une séquence de code est composée en `Courier gras 11`, sauf pour les commentaires et les chaînes de caractères, qui apparaissent en *Times italique 11* :

```
void AfficheToi()
{   // affiche la description de la confiserie
    cout << "Produit : "   << Nom
         << ", de prix " << Prix << " franc";
    cout << ((Prix > 1) ? "s\n" : "\n");
    } // void AfficheToi()
```

Citation C++ dans du texte

Lorsque le texte cite un extrait de code C++, nous avons, pour cette citation, utilisé la police Tekton au lieu de la police `Courier`, pour des raisons d'esthétique et de lisibilité :

> L'attribut static laisse la variable globale mais limite strictement sa portée au fichier dans lequel elle est définie.
>
> Le constructeur de la classe ProduitPerissable attend trois arguments (le nom, le délai et le prix). Sur ces trois arguments, deux servent à initialiser des données membres héritées (Nom et Prix).

1 Des techniques de base aux mécanismes objets

Ce chapitre commence par une présentation des principales caractéristiques de base du langage C++ (types élémentaires, schémas itératifs et conditionnels, fonctions et structure d'un programme élémentaire). Il aborde ensuite, avec un exemple simple, les concepts objets : objets, classes d'objets, messages et méthodes. Ces notions sont systématiquement illustrées avec les mécanismes correspondants disponibles en C++ : classes et fonctions membres, construction et suppression d'objets. La présentation reste cependant, à ce niveau, aussi peu technique que possible, en attendant les exposés détaillés qui viendront dans les chapitres suivants.

1.1 - Simulation d'un distributeur

Le but de ce premier chapitre est de familiariser le lecteur avec les concepts de base de la programmation par objets. Pour illustrer ces notions, nous utiliserons les mécanismes de C++ que nous mettrons en œuvre à partir d'un exemple de simulation. Pour cela, nous supposerons qu'un fabricant de distributeurs automatiques de confiseries a conçu un nouveau modèle de machine. Avant d'en lancer la fabrication, il désire valider la conception à l'aide d'un programme de simulation. Ce programme devra montrer comment le distributeur réagira en fonction des achats et des réapprovisionnements.

Dans un tel programme, il faudra représenter les confiseries, le distributeur, les clients éventuels de ce distributeur ainsi que les opérations de réapprovisionnement. Mais avant d'analyser en détail les opérations à réaliser, il convient, pour le lecteur qui ne connaîtrait pas le langage C, de découvrir les éléments de C++ qui sont empruntés à ce premier langage et qui ne sont pas, à proprement parler, liés aux mécanismes objets. Nous allons, dans les prochains paragraphes, présenter brièvement les connaissances nécessaires. Le lecteur qui a déjà pratiqué le langage C pourra se reporter directement au paragraphe 1.5.

1.2 - Définir des variables, affecter des valeurs

Si nous souhaitons décrire, en C++, le premier article de confiserie que proposera le distributeur, nous pouvons écrire les définitions suivantes :

```
char Nom1[25] = "des pastilles chocolat";
int  Prix1    = 3;   // 3 francs pour un paquet de pastilles
```

Nous définissons ainsi deux variables Nom1 et Prix1, qui indiquent respectivement la dénomination et le coût du premier article. Nom1 est un tableau de 25 caractères que la définition initialise avec la chaîne de caractères "*des pastilles chocolat*". Prix1 est une variable de type entier (int), initialisée avec la valeur 3. En C++, un des moyens utilisés pour définir une variable est de se conformer au modèle :

```
NomDeType NomDeVariable [Dimension] = ValeurInitiale ;
```

Dans un tel modèle, la spécification [Dimension] est facultative et sert à définir un tableau, indexé de 0 à Dimension -1 et dont tous les éléments sont du type NomDeType. Pour en terminer avec l'exemple précédent, on notera l'écriture :

```
// 3 francs pour un paquet de pastilles
```

Cette notation introduit, avec deux barres obliques (//), un commentaire qui s'arrête en fin de ligne[1].

Les types élémentaires que l'on peut utiliser en C++ sont les suivants :

char	valeur d'un caractère ou entier représenté dans l'espace-mémoire occupé par un caractère,
int	entier représenté dans un mot de la machine utilisée,
short	entier *court* qui, selon la machine, occupe un demi-mot ou la même place qu'une valeur de type int (int et short ne sont pas distingués dans les implémentations sous système MS-DOS),
long	entier *long* occupant en général deux mots de la machine,
float	valeur réelle représentée en virgule flottante,
double	valeur réelle représentée en virgule flottante double précision,
long double	valeur réelle représentée en virgule flottante avec la meilleure précision fournie par l'implémentation.

Une variante du modèle de définition donné plus haut permet, en omettant la spécification ValeurInitiale, de définir des variables dont la première valeur est indéterminée. On peut aussi dans une même instruction définir plusieurs variables du même type :

```
int Total, NbPaquets = 5, Solde, Montant;
```

L'instruction précédente définit quatre variables dont une seule, NbPaquets, a une valeur initiale déterminée. On notera le point-virgule qui, en C++, est le signe de terminaison *obligatoire* de toute instruction[2].

---

1 On peut également insérer un commentaire sur une ou plusieurs lignes, en l'encadrant par les signes /* et */, comme en C.

2 A la différence de langages comme Pascal, pour lequel le point-virgule est un *séparateur* d'instructions.

Pour attribuer une nouvelle valeur à une variable, on utilise une instruction d'affectation dont la forme générale est semblable à celle de la plupart des langages procéduraux :

DésignationDeVariable = *Expression* ;

On pourra ainsi écrire :

```
Total = NbPaquets * Prix1;
Solde = Montant = 0;
```

On remarquera que le signe d'affectation, noté =, est un opérateur C++ et qu'il renvoie la variable qui reçoit la valeur affectée. Cela permet d'écrire des affectations en cascade comme le montre la seconde instruction de l'exemple précédent.

Quand *Expression* décrit un calcul, les opérateurs utilisables sont les suivants :

- `+` addition et plus unaire,
- `-` soustraction et moins unaire,
- `*` multiplication,
- `/` division (quotient entier ou approché selon les types utilisés : `16/3` renverra 5 mais `16/3.0` renverra `5.33...` avec la précision permise par la machine),
- `%` reste de la division entière.

Dans une expression arithmétique faisant intervenir des types différents ou une affectation, C++ effectue les conversions attendues. En particulier, quand une valeur de type *char* intervient dans un calcul, elle est toujours considérée comme l'entier qui code le caractère représenté. Ainsi l'exécution de :

```
float X; int K = 10;
char C = 'A'; // le code ASCII de A est 65
X = (K + C) * 0.5;
```

affectera la valeur 37.5 à la variable *X*, si la machine utilise le code ASCII, car le compilateur utilisera l'entier 65 (code du caractère 'A') comme valeur de la variable *C*. On notera, dans l'exemple précédent, la forme à utiliser pour exprimer une constante de type caractère (entre apostrophes). Une telle constante est bien distincte d'une constante-chaîne (exprimée entre guillemets).

1.3 - Schémas itératifs et conditionnels

Etudions maintenant la manière d'exprimer, en C++, des répétitions d'opérations et des choix entre plusieurs possibilités.

1.3.1 - L'énoncé *while*

Ecrivons une séquence qui calcule le nombre de caractères de la chaîne contenue dans la variable *Nom1* définie au paragraphe précédent par :

```
char Nom1[25] = "des pastilles chocolat";
```

Bien entendu, ce calcul n'a d'intérêt que si on suppose que la valeur de la variable peut changer (sinon, la longueur est constante et égale à 22). Pour pouvoir déterminer cette longueur en C++, il faut savoir que, dans ce langage, les chaînes de caractères sont la plupart du temps délimitées par un caractère de fin de chaîne. Ce caractère correspond à la valeur nulle du code utilisé et on peut le noter par l'entier 0 ou la constante-caractère '\0'[3]. On remarquera aussi que, pour une chaîne, le caractère \0 de terminaison n'est qu'un délimiteur qui ne fait pas partie de la chaîne. Cela a deux conséquences :

- une chaîne ne peut contenir de caractère nul,
- la représentation interne d'une chaîne comprend toujours un caractère de plus que ses caractères significatifs, ce qui explique que la chaîne "*A*" (caractère A suivi de \0) soit distincte de la constante-caractère '*A*' (caractère A seul).

Dans l'initialisation de Nom1, le compilateur a respecté cette convention et a placé, en fin de chaîne, après le caractère t final, un caractère nul.

Revenons au problème posé en début de paragraphe. Pour calculer, dans une variable Lgr initialisée à 0, la longueur de la valeur de Nom1, il suffit d'examiner un à un les caractères de Nom1 jusqu'à ce que l'on trouve le caractère nul :

```
int Lgr = 0;
while (Nom1[Lgr] != '\0') Lgr = Lgr+1;
```

On remarquera ici, la notation utilisée pour désigner un élément de tableau : Nom1[Lgr] est l'élément de rang Lgr du tableau Nom1. L'énoncé while de l'exemple ci-dessus s'exprime conformément à la syntaxe :

```
while ( Condition ) UneInstruction
```

et répète l'exécution de UneInstruction tant que l'évaluation de Condition renvoie la valeur *vrai* (l'opérateur != exprimant, dans notre exemple, la différence).

Avant d'examiner les formes possibles pour l'expression de Condition, intéressons-nous à celles que peut prendre UneInstruction. Dans le modèle syntaxique précédent, UneInstruction représente une instruction élémentaire comme une affectation ou un appel de fonction (Cf. 1.4). Ce modèle peut être aussi un énoncé itératif *unique* (par exemple, un second while) ou encore un énoncé conditionnel *unique* (Cf. 1.3.3). Ainsi exprimée, cette syntaxe peut laisser supposer qu'il est impossible d'écrire des itérations dans lesquelles on répète plus d'une instruction. Il n'en est rien : UneInstruction peut aussi s'écrire sous la forme d'un *bloc d'instructions*, comme nous allons le voir dans le paragraphe suivant.

1.3.2 - Structure de bloc, énoncés *for* et *do ... while*

Le modèle UneInstruction de l'exemple précédent peut être un énoncé composé, encore appelé *bloc*, qui rassemble, entre deux accolades, { et }, plusieurs énoncés simples ou composés. On peut ainsi exprimer la répétition, dans un même énoncé itératif, d'une suite d'instructions, comme dans l'exemple suivant :

---

3 Dans l'expression d'un caractère sous la forme \x, la présence d'un signe \ indique la représentation d'un caractère spécial comme par exemple \n pour le passage à la ligne suivante en affichage ou impression (voir l'annexe A pour les différentes formes possibles).

```
int T[250];          // T est un tableau de 250 éléments indexés de 0 à 249
int K = 0;
while (K < 250)  {
    // on met à zéro chaque élément de T
    T[K] = 0;
    K = K + 1;
}
```

L'exécution de l'énoncé while précédent entraînera la répétition des deux instructions entre accolades, pour chacune des valeurs de K de 0 à 249.

La notion de bloc offre également la possibilité de définir des variables locales, dont la portée est limitée au bloc dans lequel elles sont définies. Etudions par exemple l'exécution de la séquence :

```
int X = 3, Y = 40, Z = 0;
{ int Aux = X; X = Y; Y = Aux; }
Z = Y;
```

La dernière instruction exécutée affectera à la variable Z la valeur 3. Dans l'exécution de cette séquence, la variable Aux sera créée à l'entrée dans le bloc et détruite à la sortie dès que l'instruction Y = Aux; aura été exécutée.

Un autre schéma itératif, l'énoncé for permet d'exprimer une répétition et il peut être utilisé pour exprimer le traitement de l'exemple précédent en écrivant :

```
int T[250];
// une autre version de la mise à zéro des éléments de T
for (int K=0; K < 250; K = K+1) T[K] = 0;
```

L'énoncé for a pour syntaxe le modèle :

```
for ( Expression1 ; Expression2 ; Expression3 ) UneInstruction
```

dans lequel Expression2 est interprétée comme une condition dont l'évaluation doit renvoyer *vrai* ou *faux*. L'exécution de cet énoncé est équivalente à celle de la séquence exprimée avec *while* :

```
évaluer Expression1
while (Expression2)  {
   UneInstruction
   évaluer Expression3
}
```

Enfin, un dernier schéma itératif est représenté par l'énoncé :

```
do UneInstruction while ( Condition ) ;
```

dans lequel le contrôle pour l'arrêt de l'itération se fait en fin de répétition : l'expression Condition représente ici le maintien des répétitions. Avec un tel énoncé, la mise à zéro des éléments de T s'exprimerait par :

```
int T[250];          // T est un tableau de 250 éléments indexés de 0 à 249
int K = 0;
do {
    T[K] = 0;
    K = K+1;
} while (K < 250);
```

On notera que, pour ce dernier énoncé, aussi bien que pour l'énoncé while, la syntaxe impose que la condition qui contrôle les répétitions soit écrite entre parenthèses. Nous allons, dans le paragraphe suivant, revenir sur l'expression de valeurs logiques.

1.3.3 - Expression d'un choix

Les séquences de choix en C++ exploitent l'évaluation d'une expression logique. Dans ce langage, il n'existe pas de type logique et les valeurs *vrai* et *faux* sont représentées par des entiers avec la convention :

- une valeur nulle représente toujours la valeur *faux*,
- une valeur entière non nulle représente toujours la valeur *vrai*.

Une expression logique élémentaire s'exprime souvent par une comparaison, pour laquelle les opérateurs utilisables sont :

`==`	égal à
`!=`	différent de
`<`	inférieur à
`>`	supérieur à
`<=`	inférieur ou égal à
`>=`	supérieur ou égal à

Ces opérateurs renvoient la valeur 0 si leur résultat est *faux*, la valeur 1 si ce résultat est *vrai*. Les opérateurs logiques :

`&&`	et
`||`	ou
`!`	non

permettent de combiner des expressions logiques élémentaires et renvoient 0 ou 1 avec les mêmes conventions.

L'énoncé if permet d'exprimer un choix entre deux possibilités et il peut prendre deux formes selon que l'alternative est ou non complète :

```
if ( Condition ) UneInstruction
if ( Condition ) UneInstruction  else UneInstruction
```

La séquence suivante qui détermine la plus grande et la plus petite valeur d'un tableau d'entiers illustre l'utilisation de ces deux formes.

```
int T[100];
... // ici, instructions qui affectent des valeurs aux éléments de T
int Min, Max;
Min = Max = T[0];
for (int K = 1; K < 100; K = K + 1)
    if (T[K] < Min)
        Min = T[K];
    else
        if (T[K] > Max)  Max = T[K];
```

1.4 - Utilisation de fonctions, structure d'un programme

Jusqu'ici, nous avons étudié quelques uns des mécanismes de base de C++, mais nous n'avons pas encore construit un programme complet. Nous allons maintenant écrire un programme qui nous permette de simuler le choix d'un produit du distributeur, en tirant un entier au hasard. Pour cela, une première approche est proposée avec le programme de la figure 1.1.

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
   int EntierHasard;
   EntierHasard = rand();
   printf ("Premier entier aléatoire : %d\n", EntierHasard);
   printf ("Second entier aléatoire : %d\n", rand());
   return 0;
} // fin de main()
```

Figure 1.1 : afficher deux entiers aléatoires

1.4.1 - La fonction *main*

Si nous laissons provisoirement de côté les deux premières lignes du programme de la figure 1.1, nous constatons que celui-ci se réduit à la définition d'une fonction principale : la fonction main(). Tout programme C++ comprend une ou plusieurs unités de code qui sont des fonctions. L'une d'entre elles doit toujours être la fonction main() : l'exécution du programme est un appel de cette fonction. Comme toute fonction C++, la fonction main() se définit en spécifiant un en-tête, suivi du corps de la fonction. L'en-tête est ici :

```
int main()
```

et il précise la manière d'utiliser la fonction. Le corps de la fonction est encadré par une paire d'accolades : il décrit la suite des instructions qui seront exécutées à chaque appel de la fonction. Avant d'examiner ces instructions, revenons à l'en-tête. L'écriture int main() fournit trois informations : le type de la valeur renvoyée, le nom de la fonction et les paramètres d'appel.

Le type de la valeur renvoyée

est toujours indiqué avant l'identificateur. Ici, on précise int : la fonction main renverra un entier en fin d'exécution. L'entité qui aura appelé le programme dont elle est la fonction principale (et qui est, la plupart du temps, le système d'exploitation) recevra cette information et pourra éventuellement l'exploiter. Dans l'exemple, la dernière instruction exécutée est return 0; qui renvoie une valeur nulle.

Le nom de la fonction
est ici main et il est imposé. Normalement le programmeur choisit librement les identificateurs des fonctions qu'il définit, sauf pour la fonction principale.

Les paramètres ou arguments d'appel
éventuels sont précisés, entre parenthèses, dans l'en-tête, après l'identificateur. Dans notre exemple, la fonction main n'a pas de paramètre[4] : la parenthèse fermante suit immédiatement la parenthèse ouvrante. Normalement, les paramètres se décrivent entre deux parenthèses, immédiatement après l'identificateur de la fonction.

1.4.2 - Définition et déclaration de fonctions

Avant d'examiner le corps de la fonction main de notre exemple, il est nécessaire de donner plus de détails sur l'utilisation de fonctions. En C++, le programmeur peut définir d'autres fonctions que la fonction main : elles seront appelées pendant l'exécution de la fonction main. Le programmeur peut aussi appeler des fonctions prédéfinies et dont le code compilé est disponible dans une bibliothèque. Par exemple, pour tirer un entier au hasard et affecter le résultat du tirage à la variable EntierHasard, il écrira :

```
EntierHasard = rand();
```

L'exécution de cette instruction s'effectue en deux temps :

1 La fonction rand, de la bibliothèque standard, est appelée sans paramètre. Elle s'exécute et renvoie un entier aléatoire.

2 Cet entier est affecté à la variable entière EntierHasard.

Avec la fonction rand, nous utilisons une fonction que nous n'avons pas définie nous-mêmes. Pour vérifier que nous l'utilisons correctement, le compilateur devra pouvoir consulter une *déclaration* de cette fonction. Pour cela, nous avons demandé au début du programme, l'inclusion du fichier d'en-tête stdlib.h avec la directive[5] :

```
#include <stdlib.h>
```

Cette directive provoque l'inclusion, dans notre programme, du fichier d'en-tête indiqué. Ce fichier contient en particulier la ligne :

```
int rand();
```

Cette écriture est une *déclaration* de la fonction rand : elle ne contient que l'en-tête de cette fonction, immédiatement suivi par un point-virgule. Avec cette seule déclaration, le compilateur peut valider notre appel de rand : pas de paramètre d'appel, utilisation correcte de la valeur renvoyée, par affectation à une variable de même type (ici, EntierHasard).

---

4 Nous verrons, à l'annexe A, que l'on peut définir une fonction main avec des paramètres d'appel.

5 Une directive n'est pas une instruction du langage. Elle est traitée avant la compilation, par le préprocesseur (Cf. 1.4.7).

De même, le compilateur vérifiera l'appel de la fonction standard printf, quand nous l'utiliserons dans le programme de la figure 1.1. Il trouvera la déclaration de printf dans le fichier stdio.h et il vérifiera qu'elle peut être appelée avec une liste de deux paramètres, dont le premier est obligatoirement une chaîne de caractères.

Pour résumer, le programme de la figure 1.1 utilise trois fonctions. La fonction main est ***définie*** avec son en-tête et son corps. Les fonctions rand et printf sont seulement ***déclarées*** : le compilateur en vérifie la bonne utilisation. Le code de ces deux dernières fonctions, qui résulte de la compilation de leurs définitions, est déjà disponible dans une bibliothèque. Il sera inclus, dans le programme, par l'éditeur de liens, après la compilation.

1.4.3 - Le corps de la fonction *main*

Nous pouvons maintenant examiner le corps de la fonction main de la figure 1.1 et commenter les instructions qui seront exécutées. Rappelons tout d'abord cette fonction :

```
int main()
{
    int EntierHasard;
    EntierHasard = rand();
    printf ("Premier entier aléatoire : %d\n", EntierHasard);
    printf ("Second entier aléatoire : %d\n", rand());
    return 0;
} // fin de main()
```

La première instruction définit la variable EntierHasard qui sera créée en début d'exécution. Comme nous l'avons déjà vu, la deuxième instruction appelle la fonction rand et affecte un entier aléatoire à la variable EntierHasard. L'instruction suivante appelle la fonction printf :

```
printf ("Premier entier aléatoire : %d\n", EntierHasard);
```

pour envoyer des informations dans le flux standard de sortie (stdout). Ce flux correspond en général à l'écran du poste de travail. La fonction printf est appelée ici avec deux arguments. Le premier d'entre eux est une chaîne de caractères qui va s'afficher telle quelle sauf pour deux combinaisons de caractères.

- La première des deux est le code de mise en forme %d. Ce code sera remplacé, à l'affichage, par la valeur du second argument EntierHasard. Il indique en effet que ce second argument est un entier à afficher en base 10 (%d pour décimal).
- De même la séquence d'échappement \n provoquera, à l'affichage, un passage à la ligne d'écran suivante (Cf. annexe A, pour les séquences d'échappement).

Le second appel de printf montre que l'argument d'une fonction peut être une expression quelconque, à condition que l'évaluation de cette expression fournisse le type attendu. Ici, le deuxième argument de printf est un appel de la fonction rand. Cet appel renvoie un entier qui sera mis en correspondance avec le code %d pour l'affichage (les arguments de type float correspondent au code %f).

Enfin, la dernière instruction termine l'exécution du programme en renvoyant une valeur nulle.

1.4.4 - Plusieurs fonctions pour un programme

Examinons maintenant un programme plus élaboré dans lequel la fonction main n'est pas la seule qui soit définie. Ce programme est présenté à la figure 1.2.

```
#include <stdlib.h>
#include <stdio.h>

int EntierAleatoireInf(int Max)
{   // renvoie un entier >=0 et < Max
    float R = rand();       // rend un entier aléatoire de 0 à RAND_MAX
    R = R / (RAND_MAX+1);   // R est un réel aléatoire >= 0 et < 1
    return (R * Max);       // on renvoie un entier >= 0 et < Max
} // int EntierAleatoireInf(int Max)

const int NbProduits = 4;

int PrixProduit[NbProduits] = {3, 2, 1, 8};
     // quatre produits de prix respectifs 3, 2, 1 et 8 francs

float PrixMoyen()
{   // calcule le prix moyen d'un produit
    float Total = 0;
    int K;
    for (K = 0; K < NbProduits; K++)
    Total = Total + PrixProduit[K];
    return (Total/NbProduits);
} // float PrixMoyen()

void main()
{
    int Rang;
    printf ("Prix moyen d'un produit : %5.2f\n", PrixMoyen());
    Rang = EntierAleatoireInf(NbProduits);
    printf ("Le produit tiré au hasard coûte %d francs\n",
             PrixProduit[Rang]);
} // fin de main()
```

Figure 1.2 : tirer au hasard le prix d'un produit

Par rapport à l'exemple précédent, nous constatons de nombreuses différences. La fonction main n'est plus la seule fonction définie ; il y a aussi les fonctions EntierAleatoireInf et PrixMoyen. Mais, avant de les étudier, commentons la définition de NbProduits et de PrixProduit.

NbProduits n'est pas une variable : le modificateur const et l'initialisation présente dans sa définition indiquent au contraire qu'il s'agit d'une constante que le code du programme ne pourra modifier. Cette constante indique le nombre de produits différents qui seront gérés par le distributeur.

Le second identificateur, PrixProduit, désigne un tableau qui est initialisé avec les prix des produits utilisés.

NbProduits et PrixProduit sont définis en dehors de toute fonction. Leur portée s'étend de leur définition jusqu'à la fin du programme. Ainsi, ils sont utilisés dans les fonctions PrixMoyen et main. Ils ne pourraient pas l'être dans la fonction EntierAleatoireInf[6].

Etudions maintenant cette fonction. Elle est définie ici, parce que la seule fonction rand ne permet pas de paramétrer correctement notre tirage au hasard. La valeur qu'elle renvoie est en effet un entier de l'intervalle 0..RAND_MAX, RAND_MAX étant une constante définie dans le fichier stdlib.h et dont la valeur dépend du compilateur utilisé. Or, pour désigner un élément du tableau PrixProduit, il nous faut tirer un entier au hasard de 0 à NbProduits-1. La fonction EntierAleatoireInf effectue un tel traitement et elle peut le faire de manière plus générale, puisqu'elle utilise son paramètre effectif Max, pour tirer un entier au hasard de 0 à Max-1. Pour comprendre le code de cette fonction, il faut noter que :

- L'affectation R = rand(); provoque la conversion de l'entier renvoyé par rand en une valeur v_1 de type float (type de R) telle que $0 \leq v_1 \leq$ RAND_MAX.
- Dans le calcul R / (RAND_MAX+1), la division est effectuée en virgule flottante et produit un résultat v_2 de type float car R est de ce type. Ce résultat est tel que $0 \leq v_1 < 1$).
- La valeur renvoyée par l'instruction return est construite en deux temps. On calcule d'abord v_3, de type float, en multipliant R par Max ($0 \leq v_3 <$ Max). Ensuite, la valeur renvoyée est celle qui est obtenue par conversion de v_3 en une valeur de type int, type de renvoi indiqué dans l'en-tête de la fonction.

Pour en terminer avec cette fonction, on notera que la variable R est locale à la fonction. D'une manière générale, seules sont globales les variables et constantes définies à l'extérieur de toute fonction, comme NbProduits et PrixProduit.

1.4.5 - Utilisation de la valeur renvoyée par une fonction

Dans l'exemple que nous venons d'examiner, les appels des fonctions printf et EntierAleatoireInf diffèrent par l'utilisation de la valeur renvoyée. En fait la fonction printf est déclarée dans stdio.h par

```
int printf(const char * Format, ...);
```

qui indique que chaque appel de cette fonction renvoie un entier[7]. Cet entier est égal au nombre d'octets envoyés à l'affichage. Or, pour les deux appels de printf

---

6 Sauf si leurs définitions étaient déplacées, pour se situer avant la définition de la fonction EntierAleatoireInf.

7 Dans cette définition, les trois points de suspension après le premier paramètre indiquent que la fonction printf accepte un nombre variable de paramètres (Cf. 2.2.7).

du programme précédent, nous n'avons pas utilisé cette information et nous avons appelé cette fonction sans recueillir sa valeur de renvoi. Il est tout à fait licite, en C++, d'appeler une fonction sans utiliser sa valeur de renvoi. On le fait en général quand seul importe l'effet du traitement effectué par la fonction. Il serait tout aussi acceptable, bien que sans intérêt, d'exécuter une instruction telle que :

```
EntierAleatoireInf(NbProduits);
```

On peut enfin définir des fonctions, en spécifiant qu'elles ne renvoient aucune valeur. Pour cela, on utilise le mot-clé *void* pour caractériser l'absence de renvoi. Ainsi, en environnement MS-DOS, la déclaration :

```
void clrscr();   // efface l'écran
```

est-elle celle d'une fonction qui permet d'obtenir un écran d'affichage vierge et que l'on appellera par :

```
clrscr();
```

Notons enfin que la fonction *main* peut aussi être définie sans indication de renvoi :

```
void main();
```

dans ce cas, elle se terminera par une instruction *return* simple :

```
return;
```

qui peut alors être omise si elle figure immédiatement avant l'accolade fermante du corps de la fonction.

1.4.6 - Programme composé d'un seul module

Avant d'étudier, dans le paragraphe suivant, les programmes construits à partir de plusieurs fichiers, récapitulons les mécanismes de définition de fonctions et de variables que nous venons d'étudier.

- Un même programme peut comprendre plusieurs définitions de fonctions. L'une d'entre elles doit être la fonction *main*.
- Les fonctions sont toutes définies au même niveau de construction du programme. Contrairement à d'autres langages, C++ n'admet pas l'emboîtement de fonctions : une définition de fonction ne peut être interne à celle d'une autre fonction.
- Dans un programme décrit en un seul fichier, la portée d'une fonction s'étend de sa déclaration (ou de sa définition) jusqu'à la fin du programme. De même, la portée d'une variable *V*, définie en dehors de toute fonction et dite globale, s'étend de la définition de *V* jusqu'à la fin du programme.
- Une variable définie à l'intérieur d'un bloc (énoncé composé ou corps de fonction) est toujours locale à ce bloc. Elle est créée en début d'exécution du bloc et détruite en fin d'exécution[8].

8 Nous parlons ici des variables automatiques. Les variables statiques sont traitées différemment (Cf. annexe A).

Ainsi, dans le programme de la figure 1.3, la variable *G* est globale. Elle peut être utilisée dans la fonction f2 et la fonction *main* mais pas dans la fonction f1 (car sa définition apparaît après celle de f1). Le programme utilise deux variables de nom X. La première est locale à f1, la seconde locale à f2. Quand on exécute l'appel de f1 qui apparaît dans f2, la variable X de f2 n'est plus accessible jusqu'au retour de cet appel. Toujours dans le même exemple, on remarquera que la portée de f2 ne couvre pas le domaine de définition de f1. La fonction f2 peut donc appeler f1 mais f1 ne peut appeler f2. Si on avait voulu implémenter une récursivité croisée entre f1 et f2, il aurait fallu ajouter, avant la définition de f1, une déclaration de f2 :

```
int f2 (int P);
```

pour que f1 puisse, elle aussi, appeler f2;

```
int f1 ()
{
    int X;
    ...
} // fin de f1

int G;

int f2 (int P)
{
    int X;
    ...
    X = f1 ()  // f2 appelle f1
    ...
} // fin de f2

int main()
{
    ...
} // fin de main
```

Figure 1.3 : plusieurs fonctions dans un même programme

1.4.7 - Programme composé de plusieurs modules

C++ permet la *compilation séparée*. On peut ainsi répartir le code d'un programme entre plusieurs fichiers-sources ou *modules*. Une telle pratique suppose que la production d'un programme utilise deux outils :

Le compilateur qui traduit chaque fichier-source en un fichier-objet. Un tel fichier-objet contient le code-machine résultant de la compilation, mais n'est pas directement exécutable.

L'éditeur de liens qui, à partir de l'ensemble des fichiers-objets, construit le programme exécutable en faisant, entre les modules objets, les liaisons nécessaires.

La figure 1.4 décrit les étapes de la fabrication d'un programme, à partir de deux modules-sources m1.cpp et m2.cpp. Chacun de ces modules est compilé et l'on obtient deux modules-objets m1.o et m2.o. Ces deux derniers fichiers sont repris par l'éditeur de liens qui combine les informations qu'ils contiennent pour produire le programme exécutable p.

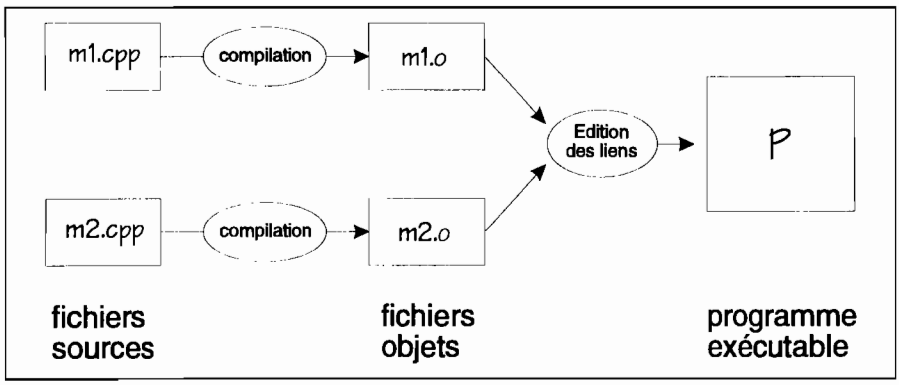


Figure 1.4 : compilation et édition des liens

L'intérêt de la structuration d'un programme en plusieurs unités est multiple :

- On peut ainsi mieux organiser le travail en répartissant les traitements du programme entre les différents fichiers-sources dont l'écriture peut être confiée à des membres distincts d'une équipe de programmeurs.
- On peut aussi réutiliser des composants logiciels déjà écrits et compilés, en utilisant directement des fichiers-objets, sans avoir besoin de disposer des fichiers-sources correspondants.

Bien entendu, les fichiers-sources peuvent partager des informations et des traitements. Examinons l'exemple suivant, dans lequel les fichiers A, B et C sont destinés à la composition d'un même programme.

fichier A.cpp

```
int X;
extern int Y;
int f1(int P);
void main()
{
  X = f1(...);
  Y = 10;
  ...
} // fin de main
```

fichier B.cpp

```
int XB;
int f1(int P)
{
   ...
} // fin de f1
... // etc.
```

fichier C.cpp

```
int Y;
... // etc.
```

Le module A utilise deux variables globales X et Y. L'une d'entre elles, X, est définie dans ce module. L'autre, Y, est définie dans le module C. Dans le module A, la spécification :

```
extern int Y;
```

indique au compilateur qu'il doit accepter des références non résolues pour la variable Y. Ainsi, la compilation de l'instruction du module A :

```
Y = 10;
```

produira-t-elle une instruction-machine *I* qui recopiera la constante 10 vers un emplacement-mémoire d'adresse encore inconnue mais qui sera identifiée comme celle de la variable externe Y. C'est l'éditeur de liens qui déterminera cette adresse en utilisant les modules-objets résultant de la compilation de A et C. Faisant la liaison entre la référence à Y dans A et la variable Y implantée dans le module C, il complétera l'instruction-machine *I* avec l'adresse finale de la variable Y.

On remarquera aussi dans l'exemple précédent que la fonction f1 est définie dans le module B et déclarée dans le module A. Cette déclaration permet au compilateur de vérifier la validité de l'appel de f1 dans A, sans inclure toutefois dans la traduction de l'appel, la rupture de séquence vers le code d'exécution de f1. Là aussi, c'est l'éditeur de liens qui, disposant à la fois de la séquence d'appel incomplète (module-objet de A) et du code complet de f1 (module-objet de B), fera la liaison.

Observons aussi que, dans un programme, les variables globales telles que X (définie dans A.cpp) et Y (définie dans C.cpp) doivent avoir des noms distincts. Si on avait, par exemple, changé le nom de la variable XB du fichier B.cpp, pour l'appeler elle aussi X, l'éditeur de liens aurait refusé de construire le programme en signalant une possible ambiguïté entre deux variables globales de même nom.

Si on avait souhaité conserver deux variables globales de même nom X, l'une, celle de A.cpp, exportable dans tous les autres fichiers, l'autre celle de B.cpp, dont la portée aurait été restreinte à son fichier de définition, il aurait fallu définir la seconde avec le modificateur static :

```
// une autre version de B.cpp
static int X;
int f1(int P)
{
...
} // fin de f1
... // etc.
```

L'attribut static[9] laisse la variable globale mais limite strictement sa portée au fichier dans lequel elle est définie.

Pour compléter notre étude de la modularité, donnons un dernier exemple en reprenant le programme de la figure 1.2 pour le restructurer en trois modules, conformément au schéma de la figure 1.5.

9 L'attribut static n'a pas la même signification pour une variable locale (Cf. Annexe A).

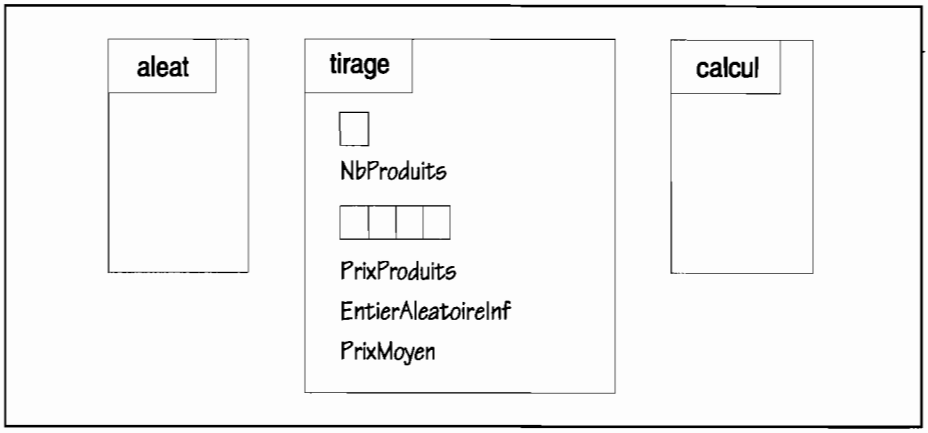


Figure 1.5 : un programme composé de trois modules

Pour cette restructuration, on a distingué trois modules :

- le module principal tirage, qui affiche le prix moyen d'un produit (appel de la fonction PrixMoyen) puis tire au hasard un produit (appel de la fonction EntierAleatoireInf),
- le module de tirage au hasard aleat, qui contient la définition de la fonction EntierAleatoireInf,
- le module calcul, qui contient la définition de la fonction PrixMoyen.

Tous les fichiers qui permettent de construire le programme sont représentés à la figure 1.6, dans laquelle, pour ne pas surcharger la présentation, on a supprimé les commentaires déjà présents figure 1.2. On notera que, dans le fichier tirage.cpp, les déclarations des fonctions PrixMoyen et EntierAleatoireInf n'apparaissent pas de la même manière :

- Pour la fonction PrixMoyen, la déclaration est explicite : elle reproduit l'en-tête de la fonction qui est définie dans le fichier calcul.cpp.
- Pour la fonction EntierAleatoireInf, en revanche, la déclaration sera obtenue avec l'inclusion du fichier d'en-tête aleat.h. On a en effet considéré ici que les outils de tirage au hasard du fichier aleat.cpp sont suffisamment généraux pour être utilisés par d'autres programmes. On a donc standardisé l'interface de ces outils en en faisant un fichier d'en-tête qui sera inclus automatiquement dans tout programme comportant la directive #include "aleat.h"[10].

Par rapport à la figure 1.2, nous avons modifié la fonction EntierAleatoireInf et son mode d'utilisation. Nous avons limité vers le haut le domaine des entiers utilisables, en définissant la constante ENTMAX avec :

```
#define ENTMAX 10000
```

10 Par rapport à la notation usuelle qui encadre le nom du fichier d'en-tête par < et >, l'utilisation des guillemets demande au préprocesseur d'aller chercher le fichier d'en-tête d'abord dans le répertoire courant du programmeur et seulement ensuite, en cas d'échec de la recherche, d'explorer la bibliothèque des en-têtes standard.

// tirage.cpp

```
#include <stdio.h>
#include "aleat.h"
extern const int NbProduits = 4;
int PrixProduit[NbProduits] = {3, 2, 1, 8};
float PrixMoyen();
void main()
{
  int Rang;
  printf ("Prix moyen d'un produit : %5.2f\n", PrixMoyen());
  Rang = EntierAleatoireInf(NbProduits);
  printf ("Le produit tiré au hasard coûte %d francs\n", PrixProduit[Rang]);
}
```

// aleat.h

```
#include <stdlib.h>
#define ENTMAX 10000
int EntierAleatoireInf(int Max);
```

// aleat.cpp

```
#include <stdio.h>
#include "aleat.h"
int EntierAleatoireInf(int Max)
{
  if (Max<=0 || Max>ENTMAX) {
        printf ("%d : argument incorrect "
                "EntierAleatoireInf\n", Max);
  exit(1);
  };
  float R = rand();
  R = R / (RAND_MAX+1);
  return (R * Max);
}
```

// calcul.cpp

```
extern const int NbProduits;
extern int PrixProduit[];
float PrixMoyen()
{
  float Total = 0;
  int K;
  for (K = 0; K < NbProduits; K++)  Total = Total + PrixProduit[K];
  return (Total/NbProduits);
}
```

Figure 1.6 : tirage, aleat *et* calcul *composent un programme exécutable*

Cette constante est utilisée par la fonction EntierAleatoireInf pour vérifier qu'aucune valeur d'appel correspondant au paramètre Max ne dépassera la limite ainsi fixée. La directive #define permet de définir des constantes symboliques qui sont traitées par le préprocesseur. La syntaxe d'une telle directive est ici :

#define Identificateur ValeurDeSubstitution

Avant la compilation proprement dite, le préprocesseur remplacera chaque occurrence de Identificateur par ValeurDeSubstitution dans le texte du module. Ce

mécanisme de définition de constante n'est donc pas équivalent à celui que nous avons déjà utilisé avec :

```
const int NbProduits = 4;
```

qui implante, dans le programme un emplacement nommé (ici NbProduits) et initialisé. Cet emplacement sera traité comme une variable entière, sauf pour les opérations qui pourraient modifier la valeur d'initialisation et dont la compilation sera refusée.

On remarquera aussi que la fonction EntierAleatoireInf rejettera désormais tout argument négatif ou nul, alors que, dans la version précédente de la figure 1.2, nous n'avions pas traité ce cas d'erreur. Cette modification va dans le bon sens : celui d'une généralisation de l'interface, avec le maximum de sécurité en vue d'une réutilisation dans de nombreuses applications.

Pour terminer, examinons la liaison entre les modules tirage et calcul. Le premier module appelle la fonction PrixMoyen dont le code est disponible dans le second. Mais le module calcul a, lui-même, besoin de deux informations du module tirage : PrixProduit et NbProduits.

Dans le module calcul, le tableau PrixProduit est référencé par la déclaration :

```
extern int PrixProduit[];
```

On n'y indique pas de dimension. En effet, le tableau n'est pas défini dans ce module : la fonction PrixMoyen utilisera l'adresse de ce tableau et elle devra en respecter l'intégrité en gérant elle-même la dimension. Pour cela, le module calcul établit aussi la liaison avec la constante NbProduits définie dans le module tirage. Dans le module calcul, la référence à NbProduits est déclarée avec :

```
extern const int NbProduits;
```

le modificateur const étant mis ici par précaution pour indiquer au compilateur que la référence externe est celle d'une constante. Dans le module tirage, la définition de NbProduits peut paraître surprenante :

```
extern const int NbProduits = 4;
```

En effet, la présence du spécificateur extern pourrait laisser supposer qu'il s'agit d'une déclaration et non d'une définition. Mais l'initialisation à la valeur 4 indique bien au compilateur qu'il traite la définition de NbProduits. Pour comprendre, il faut savoir que le modificateur const, quand il apparaît seul, limite la portée au module où la constante est définie. Pour qu'une constante soit exportable, il faut adjoindre au modificateur const le spécificateur extern, qui n'a donc pas ici son sens habituel. Sans ce modificateur dans la définition de NbProduits de tirage, l'éditeur de liens indiquera une référence externe non résolue pour NbProduits dans calcul.

1.5 - Objets et messages

Nous pouvons maintenant aborder les mécanismes « objets » du langage C++. Revenons pour cela à notre simulation d'un distributeur de confiseries. Pour représenter le choix d'un article au distributeur par la frappe d'un numéro de produit

au clavier, nous pouvons exécuter la séquence d'instructions suivante, dans laquelle nous supposons que le choix doit s'effectuer entre quatre articles.

```
const NbProduits = 4;
int Numero; // indiquera le numéro du produit choisi
cout << "Tape un numéro de produit de 1 à ";
cout << NbProduits;
cout << " : ";
cin >> Numero;
while (Numero < 1  ||  Numero > NbProduits) {
   cout << "numéro incorrect, recommence : ";
   cin >> Numero;
}
```

Après l'exécution de cette séquence, la variable Numero aura une valeur allant de 1 à 4. Examinons de plus près la séquence ci-dessus. Par rapport à ce que nous avons étudié dans les paragraphes précédents, un certain nombre de nouveautés apparaissent.

La première ligne définit la constante NbProduits, de valeur 4. Nous utilisons ici le modificateur const sans autre indication. En l'absence d'une spécification de type, const induit systématiquement le type int. La deuxième ligne de la séquence définit la variable entière Numero dans laquelle on va saisir le numéro de produit. La ligne suivante est l'instruction :

```
cout  << "Tape un numéro de produit de 1 à ";
```

Pour comprendre cette instruction, il faut savoir que cout est un *objet* prédéfini en C++ et qu'il représente le flux standard de sortie du programme, stdout, qui est normalement dirigé vers l'écran de la station de travail. Avec la programmation orientée objets, les traitements sont contrôlés par les objets : pour exécuter une opération, on enverra un *message* à un objet. A ce message doit correspondre un *protocole* qui décrit les instructions à exécuter. En C++, un message correspond toujours à un appel de fonction ou d'opérateur et le compilateur va vérifier que la fonction ou l'opérateur peuvent être associés à l'objet qui reçoit le message. Dans notre exemple, le message reçu par l'objet cout est un appel de l'opérateur << :

```
<< "Tape un numéro de produit de 1 à "
```

Cet opérateur qui correspond normalement à une opération de décalage sur une suite de bits est redéfini, en C++, pour l'objet cout. Son protocole correspond aux opérations d'écriture, dans le flux standard de sortie, de l'information qui l'accompagne. A travers cet opérateur, l'objet cout traite correctement les valeurs qui lui parviennent, en envoyant en sortie (ici, à l'écran) leur représentation sous la forme d'une suite de caractères. Dans l'exécution de :

```
cout  << Valeur
```

l'objet cout reconnaît le type qui correspond à Valeur pour choisir le mode de conversion approprié et afficher l'information de manière adéquate. On notera la simplification apportée avec l'approche objets, par rapport à l'utilisation de la fonction C printf : ici, c'est l'objet cout qui assure correctement la conversion. Alors qu'avec printf, le programmeur doit spécifier lui-même le type de mise en forme.

Dans la séquence examinée, les deux lignes suivantes adressent chacune un nouveau message à l'objet *cout* :

```
cout << NbProduits;
cout << " :";
```

et l'exécution produira pour la suite de ces messages, l'affichage :

```
Tape un numéro de produit de 1 à 4 :
```

On aurait pu obtenir le même résultat avec une seule instruction qui aurait enchaîné les trois messages :

```
cout << "Tape un numéro de produit de 1 à "
     << NbProduits  << " :";
```

La partie de la séquence que nous avons examinée jusqu'à présent pose une question à l'utilisateur. Pour saisir la réponse que celui-ci frappe au clavier, on exécute ensuite l'instruction :

```
cin  >> Numero;
```

Là aussi, un objet reçoit un message. L'objet *cin* représente le flux standard d'entrée, *stdin*, associé ici au clavier. Cet objet reçoit le message >> *Numero* qui fait intervenir l'opérateur >> redéfini de manière à correspondre à une entrée d'information. En réponse au message, l'objet *cin* va décoder les caractères frappés pour composer une valeur correspondant au type de l'argument *Numero* du message.

La suite de la séquence est une itération qui garantit que la valeur frappée par l'utilisateur est un numéro de produit correct.

1.6 - Notion de classe

Examinons maintenant un programme complet dans lequel nous reprendrons les opérations que nous venons d'étudier. Ce programme est représenté figure 1.7.

Dans ce programme, la séquence que nous venons de commenter est reprise, sous la forme de la fonction *ProduitChoisi*, qui renvoie une valeur entière de 0 à 3 (et non pas de 1 à 4). En effet, les produits que nous allons gérer sont conservés dans un tableau de quatre éléments (*Liste*), indexé, comme tous les tableaux en C++, à partir de 0. Avec la fonction *ProduitChoisi*, le choix du produit s'exprime désormais en début de la fonction *main* par :

```
int NumProd = ProduitChoisi();
```

En examinant le programme, on s'interroge sans doute pour savoir comment sont spécifiés les traitements contrôlés par les objets *cout* et *cin* et associés aux opérateurs << et >>. Les définitions nécessaires sont incorporées au programme avec l'inclusion du fichier *iostream.h*.

Les objets *cout* et *cin* sont prédéfinis dans le langage. Comment peut-on définir soi-même des objets et leur associer des propriétés ? Une réponse partielle est donnée dans le programme de la figure 1.7, avec la définition :

```
struct Confiserie {
   char Nom[25];
   int Prix;
}; // struct Confiserie
```

```
#include <iostream.h>
struct Confiserie {
   char Nom[25];
   int Prix;
}; // struct Confiserie
const NbProduits = 4;
Confiserie Liste[NbProduits] = {
   "des pastilles chocolat", 3,
   "des bonbons acidulés", 2,
   "du chewing gum",      1,
   "une barre-nougat",    8
};
int ProduitChoisi()
{// simule le choix d'un produit
   // et renvoie le rang du produit dans le tableau Confiserie
   int Numero;
   cout  << "Tape un numéro de produit de 1 à "
         << NbProduits  << " : ";
   cin >> Numero;
   while (Numero < 1   ||  Numero > NbProduits) {
      cout << "numéro incorrect, recommence : ";
      cin >> Numero;
   }
   return Numero-1;
} // int ProduitChoisi()
void main()
{
   int NumProd = ProduitChoisi();
   cout << "Produit choisi : "  << Liste[NumProd].Nom
        << ", de prix " << Liste[NumProd].Prix << " franc";
   cout << ((Liste[NumProd].Prix > 1) ? "s\n" : "\n");
} // void main()
```

Figure 1.7 : choix d'un produit

Cette définition indique que nous allons gérer des objets qui auront tous deux composantes, appelées habituellement données membres :

- la donnée membre Nom qui est un tableau pouvant contenir une chaîne allant jusqu'à 24 caractères,
- la donnée membre Prix, qui contiendra une valeur entière.

Chacun de ces objets décrira une confiserie par sa désignation (Nom) et son coût en francs (Prix). L'ensemble de ces objets, qui ont tous la même structure, forme une famille que l'on appelle une classe. L'identificateur (ici Confiserie), qui apparaît après le spécificateur struct dans la définition de la classe, est le nom de la classe. Ayant défini cette classe, on peut l'utiliser, par exemple avec :

```
Confiserie C1 = {"cacahuètes", 3}, C2;
C2.Prix = 5;
```

La première ligne de l'exemple précédent définit deux objets C1 et C2. Les données membres de C1 sont initialisées avec les valeurs indiquées entre accolades, celles de C2 sont indéterminées. La deuxième ligne de l'exemple fixe la donnée membre Prix de C2 à la valeur 5.

Dans l'exemple précédent, la définition des objets C1 et C2 est en fait une opération dynamique, qui créera deux nouvelles entités. La terminologie usuelle de la programmation par objets qualifie cette création d'**instanciation** et on dira aussi que C1 et C2 sont des **instances** de la classe Confiserie. Dans le programme de la figure 1.7, on instancie un tableau de quatre objets de cette classe : le tableau Liste, que l'on initialise avec une liste de valeurs spécifiées entre accolades.

L'exécution du même programme tire un entier au hasard pour désigner par son rang l'un des produits de ce tableau. La description du produit ainsi désigné est ensuite affichée à l'écran. Dans cette opération, l'affichage du prix traite correctement le substantif *franc* en ajoutant un *s* si le prix est de plus d'un franc. Pour distinguer ainsi entre singulier et pluriel, on exécute :

```
cout  << ((Liste[NumProd].Prix > 1) ? "s\n" : "\n");
```

en utilisant l'opérateur ternaire ? : dont la syntaxe est :

```
Condition ? ValeurSiVrai : ValeurSiFaux
```

Cet opérateur évalue Condition et renvoie ValeurSiVrai si l'évaluation calcule une valeur *vrai* (non nulle). Il renvoie ValeurSiFaux dans le cas contraire.

Pour notre exemple, l'opérateur ? : enverra la chaîne "*s*\n" (pluriel) ou "\n" (singulier) pour terminer la ligne d'affichage.

1.7 - Données membres et fonctions membres

Le traitement dans lequel interviennent les objets de la classe Confiserie n'a pas les caractéristiques de l'approche objets. En effet, l'affichage des informations d'un objet devrait être contrôlé par l'objet lui-même et être déclenché par un message adressé à l'objet.

Par rapport au programme de la figure 1.7, cette approche aurait l'avantage d'encapsuler le traitement dans la classe et d'assurer plus facilement son évolution. Si un objet sait s'afficher quand il reçoit le message AfficheToi, il suffira si on change la structure de cet objet, de modifier en conséquence l'implémentation du message pour que tous les programmes qui utilisent ce message soient à jour après une simple recompilation.

Pour qu'un objet sache traiter un message, il faut que ce message soit défini dans sa classe, sous la forme d'une *fonction membre*. Ajoutons donc la fonction membre AfficheToi à la classe Confiserie. La nouvelle définition de cette classe est présentée figure 1.8.

Cette nouvelle définition indique que chaque objet de la classe Confiserie a deux données membres (Nom et Prix) et qu'il peut effectuer un traitement, décrit par la fonction membre AfficheToi.

```
struct Confiserie {
   char Nom[25];
   int Prix;

   void AfficheToi()
   {
      // affiche la description de la confiserie
      cout << "Produit : "   << Nom
           << ", de prix " << Prix << " franc";
      cout << ((Prix > 1) ? "s\n" : "\n");
   } // void AfficheToi()

}; // struct Confiserie
```

Figure 1.8 : deuxième version de la classe Confiserie

Pour déclencher ce traitement, on devra envoyer un message à l'objet considéré, sous la forme d'un appel de la fonction membre :

```
Confiserie C1 = {"cacahuètes", 3};
C1.AfficheToi(); // L'objet C1 reçoit le message AfficheToi()
```

De même, on peut maintenant récrire le corps de la fonction main de la figure 1.7:

```
int NumProd = ProduitChoisi();
Liste[NumProd].AfficheToi();
```

Si on revient à la figure 1.8, on constate que le code de la fonction AfficheToi manipule des données membres Prix et Nom sans les rattacher explicitement à un objet. A chaque exécution, ce sont celles de l'objet qui reçoit le message (C1, puis Liste[NumProd] dans les deux exemples précédents).

Plus généralement, dans le traitement exprimé par Obj.Msg(...), l'objet Obj est toujours un argument implicite de la fonction membre Msg. Dans le code de Msg(...), les désignations de données membres qui ne sont pas préfixées par une indication d'objet se rapportent toutes à cet objet Obj. Le corollaire est que toute donnée membre qui appartient à un autre objet doit être complètement qualifiée. A titre d'exemple, ajoutons à la définition de la classe Confiserie la fonction membre MemePrixQue :

```
struct Confiserie {
   ...   // comme précédemment
   MemePrixQue(Confiserie C)
   {
      Prix = C.Prix;
   } // MemePrixQue (Confiserie C)
}; // struct Confiserie
```

Quand cette fonction est exécutée dans :

```
Confiserie C1 = {"cacahuètes", 3}, C2;
C2.MemePrixQue(C1);
```

l'instruction Prix = C.Prix est exécutée en considérant que la donnée membre Prix qui reçoit la valeur affectée est celle de C2 et que l'écriture C.Prix désigne la donnée membre correspondante de C1.

1.8 - Accès public ou privé

Nous désirons maintenant simuler l'arrivée de clients au distributeur. Chacun de ces clients veut acheter une confiserie à condition que son prix lui semble acceptable. Un tel client sera donc caractérisé par :

- la confiserie qu'il désire acheter (choisie par exemple parmi les produits décrits par le tableau Liste (Cf. 1.7),
- le montant qu'il est prêt à payer.

Nos clients peuvent être considérés comme les instances d'une nouvelle classe, que nous pouvons définir par :

```
struct Client {
   Confiserie achat;  // produit que le client désire acheter
   int depenseMax;    // montant maximum que le client est prêt à
                      // dépenser pour ce produit
}; // struct Client
```

On pourra par exemple instancier un client en exécutant :

```
Client UnClient;
UnClient.achat = Liste[EntierAleatoireInf(NbProduits)];
UnClient.depenseMax = 5;   // il veut bien dépenser jusqu'à 5 francs
```

On fixe ainsi les valeurs des données membres de ce client. Jusqu'à présent, nous avons manipulé librement les données membres. Pourtant des affectations non contrôlées sont souvent dangereuses. On pourrait par exemple écrire :

```
UnClient.achat = Liste[10];  // le tableau Liste n'a que 4 éléments !
```

En programmation par objets, un objet doit pouvoir contrôler l'usage de ses propres données membres. Il est facile, en C++, de limiter l'accès aux membres (données et fonctions). Supposons, par exemple que la définition de la classe Client devienne celle de la figure 1.9.

```
class Client {
   Confiserie achat;  // produit que le client désire acheter
   int depenseMax;    // montant maximum que le client est prêt à
                      // dépenser pour ce produit
}; // class Client
```

Figure 1.9 : des données membres privées pour la classe Client

Avec cette définition, si nous écrivons maintenant :

```
Client AutreClient;
AutreClient.depenseMax = 5;
```

la compilation de la deuxième ligne signalera une erreur en indiquant que le membre depenseMax n'est pas accessible. Par rapport à la première définition, la seule différence d'écriture est l'utilisation du spécificateur class au lieu de struct. Ce changement a pour effet d'indiquer que les membres de la classe Client sont *privés* par défaut (alors qu'avec struct, ils sont *publics* par défaut).

Un membre public n'a pas de restriction d'accès. Si X est un objet, les membres publics de X peuvent être manipulés (consultés, modifiés ou appelés) par n'importe quelle partie du programme relevant de la portée de X.

L'accès à un membre privé est, en principe[11], réservé aux fonctions membres de la classe.

Une classe peut comporter des membres publics et privés. Quand elle est définie avec struct, les membres privés doivent être introduits par le spécificateur private, quand elle est définie avec class, les membres publics doivent être introduits par le spécificateur public.

Revenons maintenant à la classe Client, telle qu'elle est définie figure 1.9. Si nous souhaitons pouvoir instancier un client en donnant des valeurs déterminées aux membres privés achat et depenseMax, il nous faut définir une fonction membre, par exemple la fonction membre Initialise (figure 1.10).

```
class Client {
 private:
   Confiserie achat;        // produit que le client désire acheter
   int        depenseMax;   // dépense maximum du client
 public:
   void Initialise(int MaxPrix)
   {  // fonction membre qui fixe les données membres d'un nouveau client
      achat = Liste[EntierAleatoireInf(NbProduits)];
      if (depenseMax > 0) depenseMax = MaxPrix;
      else {
         cerr << "Une intention de dépense doit être positive"
         exit (1);
      }
   } // Initialise(int MaxPrix)
}; // class Client
```

Figure 1.10 : membres publics et membres privés

Dans cette nouvelle définition de la classe, on distingue deux parties :

- celle des membres privés, introduite par le spécificateur private (spécificateur facultatif ici, puisque la définition de la classe est introduite par le mot-clé class),
- celle des membres publics, introduite par le spécificateur public et qui ne répertorie ici que la fonction membre Initialise.

On utilisera cette fonction membre de la manière suivante :

```
Client UnClient;
   // Le client UnClient est instancié, ses données membres sont indéterminées
UnClient.Initialise(5);
   // L'intention d'achat du client UnClient est précisée
```

11 Une fonction amie d'une classe CCC n'est pas membre de CCC mais elle a accès aux membres privés de CCC (Cf. chapitre 3).

On notera la sécurité apportée par le mécanisme. Le programmeur ne peut pas affecter des valeurs quelconques aux données membres *achat* et *depenseMax*. En revanche, il peut adresser un message Initialise à l'objet UnClient. L'exécution de ce message sera contrôlée par l'objet via la fonction membre correspondante :

- la donnée membre *achat* prendra obligatoirement l'une des valeurs admissibles, grâce au tirage aléatoire,
- l'affectation d'une valeur à la donnée membre *depenseMax* sera refusée si la valeur fournie par le programmeur est inacceptable. Dans ce cas, le programme se terminera (appel de la fonction *exit*[12] avec code-retour 1) après avoir envoyé un message d'affichage au flux standard des messages d'erreur, représenté par l'objet *cerr*. Cet objet traite un message << comme l'objet *cout*, mais en envoyant l'affichage vers la destination prévue pour les messages d'erreur[13].

1.9 - Utilisation d'un constructeur

Pour obtenir un objet *Client* correctement initialisé, le programmeur doit donc instancier un objet dont les données membres sont indéterminées, puis envoyer à cet objet un message Initialise. On peut souhaiter rassembler les deux opérations dans une écriture plus concise. On peut le faire en effet en spécifiant comment les données membres doivent être initialisées lors de l'instanciation, encore appelée *construction* de l'objet. Pour ce faire, on définit dans la classe, une fonction membre qui se substitue à la fonction membre Initialise et que l'on appellera un *constructeur*. Examinons la nouvelle définition de la classe, figure 1.11, qui spécifie un constructeur.

Un constructeur est une fonction membre publique qui a pour identificateur le nom de la classe et qui ne spécifie aucune valeur renvoyée (pas même *void*). En fait, chaque fois que l'on définit un objet X d'une classe *CCC* avec la notation[14] :

```
CCC X;
```

l'exécution du code correspondant à cette instanciation appelle un constructeur fourni par défaut qui construit un objet dont les données membres ont des valeurs indéterminées. Si le programmeur fournit lui même un constructeur de la classe *CCC*, il pourra appeler ce constructeur pour créer un objet X avec la notation :

```
CCC X( ... );
```

dans laquelle les points de suspension représentent le ou les valeurs des arguments d'appel du constructeur.

---

12 Cette fonction, utilisable en C et C++, provoque quand elle est appelée avec un argument entier E, un arrêt du programme après fermeture de tous les fichiers ouverts. Elle renvoie le code-retour E à l'appelant du programme qu'elle termine. En général cet appelant est le système d'exploitation qui peut éventuellement examiner la valeur entière renvoyée.

13 Sauf indication contraire avant le lancement du programme, cette destination (flux *stderr*) est souvent confondue avec le flux standard *stdout*. Elle correspond à l'écran du poste de travail.

14 Le programmeur débutant prendra garde à ne pas écrire cette définition *CCC* X() car celle-ci sera interprétée par le compilateur comme une déclaration de fonction et non comme une instanciation.

```
class Client {
 private:
   Confiserie achat;          // produit que le client désire acheter
   int        depenseMax;     // dépense maximum du client
 public:
   Client(int MaxPrix)
   {  // constructeur d'un client dont le produit souhaité est tiré au hasard
      // et le montant maximum est aléatoire de 1 à MaxPrix
      achat = Liste[EntierAleatoireInf(NbProduits)];
      depenseMax = MaxPrix;
   } // Client(int MaxPrix)
}; // class Client
```

Figure 1.11 : un constructeur pour la classe Client

Si nous revenons à la classe Client, l'instanciation de l'objet UnClient peut désormais se décrire avec :

```
Client UnClient(5);
```

Ajoutons à la classe Client la fonction membre publique PresenteToi, qui permettra à un objet de cette classe d'afficher à l'écran les informations qu'il représente :

```
class Client {
  // etc., Cf. figure 1.11
 public:
  // etc., définition du constructeur, Cf. figure 1.11
   void PresenteToi()
   {
      // le client-récepteur affiche ses intentions à l'écran
      cout << "Je désire " << achat.Nom
           << " de prix " << achat.Prix;
      cout << "\nJe suis prêt à payer " << depenseMax << "franc"
           << ((depenseMax > 1) ? "s\n" : "\n");
   } // void PresenteToi()
}; // class Client
```

Cette nouvelle fonction membre peut-être utilisée après chaque instanciation pour visualiser l'objet créé. Par exemple, on écrira :

```
Client UnClient(5);
UnClient.PresenteToi();
```

1.10 - Allocation dynamique

Dans l'instanciation du paragraphe précédent pour l'objet UnClient, la portée de l'objet instancié va de son instanciation jusqu'à la fin du bloc dans lequel est effectuée cette instanciation. On peut souhaiter créer un objet dans un bloc sans limiter sa « durée de vie » à l'exécution du bloc dans lequel cet objet est créé. On peut

aussi vouloir créer, à l'exécution d'un programme, des objets dont ni le nombre ni le nom ne sont connus au moment de la compilation. Pour cela, C++ fournit un mécanisme d'allocation dynamique dont un exemple est donné figure 1.12.

Dans cette séquence, deux objets sont construits sans allocation dynamique : la place-mémoire qu'ils occuperont est prévue dès la compilation. Ce sont :

- l'objet C1, dont la portée s'étendra au moins jusqu'à la fin de la séquence décrite par la figure (en l'absence d'indication supplémentaire sur le bloc dans lequel se trouve cette séquence),
- l'objet C2, dont la portée est limitée au bloc qui s'étend de l'accolade ouvrante à l'accolade fermante.

Dans ce bloc, l'expression *new* Client(8) provoquera la construction d'un troisième objet de la classe Client qui n'aura pas de nom et sera obtenu par allocation dynamique. Pour comprendre les mécanismes mis en jeu, revenons au début de la séquence :

```
Client * PtrClient;
```

L'instruction précédente définit la variable PtrClient comme un pointeur sur un objet de la classe Client. En effet, si UnType désigne un type prédéfini, ou une classe, la notation :

```
UnType * NomDePointeur;
```

définit la variable NomDePointeur comme un pointeur sur le type UnType, c'est-à-dire une variable dont la valeur sera l'adresse d'une valeur UnType. La variable PtrClient peut donc désigner un objet de la classe Client par son adresse. On pourrait par exemple écrire :

```
PtrClient = & C1;
```

pour que PtrClient pointe sur l'objet C1[15].

```
Client * PtrClient;
Client C1(5);
   {
    Client C2(6);
    PtrClient = new Client(8);
    C2.PresenteToi(); // allocation dynamique d'un client
   }
// ici, C2 n'existe plus
PtrClient->PresenteToi;
   // le client alloué dynamiquement affiche ses informations
```

Figure 1.12 : allocation dynamique d'un client

En fait, PtrClient est initialisé dans la figure 1.12 avec l'instruction :

```
PtrClient = new Client(8);
```

qui fait appel à l'opérateur C++ d'allocation dynamique, noté *new* et qui s'exécute en trois temps :

15 Nous utilisons ici l'opérateur & (*adresse de*) que nous reverrons en détail au chapitre 2.

- la place nécessaire pour l'implantation en mémoire d'un objet de la classe Client est réservée par le gestionnaire de la mémoire associé au programme,
- dans l'emplacement ainsi obtenu, un objet de la classe Client est implanté, par appel du constructeur avec l'argument 8,
- l'adresse de cet objet est alors renvoyée par l'opérateur new : cette adresse devient la valeur de la variable PtrClient.

La dernière ligne de la séquence de la figure 1.12 :

```
PtrClient->PresenteToi;
    // le client alloué dynamiquement affiche ses informations
```

montre comment on peut exprimer l'envoi d'un message à une variable désignée par un pointeur à l'aide de l'opérateur `->` de désignation d'un membre (donnée ou fonction) pointé.

On notera également que, pour l'allocation dynamique de la figure 1.12, nous n'avons pas envisagé la possibilité d'un échec. Dans un programme qui effectue beaucoup d'allocations, cette éventualité n'est pas à écarter car le gestionnaire de mémoire peut tomber à court de ressources. Dans un tel cas, l'opérateur new signalera cet incident en renvoyant une constante prédéfinie du langage, la constante NULL. Par convention, quand un pointeur a la valeur NULL, il ne désigne aucune adresse et ne doit pas être utilisé pour accéder à une variable. Pour cela, chaque appel de l'opérateur new est souvent suivi d'un test qui vérifie que l'adresse renvoyée n'est pas NULL.

Dans le même esprit, pour gérer correctement les ressources dynamiques, il convient de restituer au gestionnaire de mémoire tout espace qui n'est plus utilisé. On utilise pour cela l'opérateur *delete* auquel on fournit l'adresse de l'espace alloué. Si nous revenons à la séquence de la figure 1.12, il faudra, pour libérer l'espace alloué dynamiquement et pointé par PtrClient, exécuter l'instruction[16] :

```
delete PtrClient;
```

Cette instruction provoquera la suppression de l'objet pointé par PtrClient (Cf. l'utilisation d'un destructeur au chapitre 2), puis la libération de l'espace occupé par cet objet.

1.11 - Une simulation élémentaire

Pour conclure ce chapitre, le programme C++ présenté dans les deux dernières pages de ce chapitre rassemble les exemples que nous avons étudiés et effectue une simulation élémentaire du comportement des clients d'un distributeur.

16 Bien entendu, si cette allocation dynamique est la seule effectuée par le programme, il n'est pas nécessaire de l'annuler avec l'opérateur *delete*. En effet, la fin de l'exécution du programme provoque toujours la libération de la totalité de l'espace alloué au programme. Il vaut mieux cependant prendre de bonnes habitudes et toujours libérer les espaces alloués quand il ne sont plus utilisés.

Pour ce faire, deux classes sont utilisées :

- la classe Confiserie, qui correspond aux produits mis en vente dans le distributeur et dont tous les membres sont publics,
- la classe Client, dont les instances représentent les clients successifs.

Dans la classe Client, une fonction membre publique supplémentaire est définie. Elle permet, si Clt est un client, de représenter le choix du client, en fonction de sa capacité financière et du prix du produit qu'il désire. Pour cela, on exécute l'instruction Clt.Decide().

L'exécution du programme (Cf. fonction main, page 32), simule l'arrivée des clients avec l'exécution d'un schéma itératif *do*, dans lequel, à chaque répétition, on effectue les opérations suivantes :

- allocation dynamique et instanciation d'un client pointé par le pointeur P (P = new Client(...)),
- affichage du profil du client (fonction membre PresenteToi),
- décision du client (fonction membre Decide),
- suppression du client (*delete* P)

Enfin, la figure 1.13 présente un exemple d'affichage obtenu avec une exécution du programme.

```
Simulation de l'arrivée de clients

Je désire des pastilles chocolat de prix 3
Je suis prêt à payer 1 francs
Je renonce à l'achat

Encore un client (O/N) : o
Je désire des pastilles chocolat de prix 3
Je suis prêt à payer 4 francs
J'achète des pastilles chocolat

Encore un client (O/N) : o
Je désire une barre nougat de prix 8
Je suis prêt à payer 8 francs
J'achète une barre nougat

Encore un client (O/N) : o
Je désire des bonbons acidulés de prix 2
Je suis prêt à payer 3 francs
J'achète des bonbons acidulés

Encore un client (O/N) : n
```

Figure 1.13 : exemple d'exécution de la simulation

```
#include <stdlib.h>
#include <iostream.h>
#include "aleat.h"
#include <ctype.h>   // pour la fonction toupper (passage en majuscules)
struct Confiserie {
   char Nom[25];
   int Prix;
   void AfficheToi()
   {  // affiche la description de la confiserie
      cout << "Produit : "  << Nom
           << ", de prix " << Prix << " franc";
      cout << ((Prix > 1) ? "s\n" : "\n");
   } // void AfficheToi()
}; // struct Confiserie
const NbProduits = 4;
const Confiserie Liste[NbProduits] = {
   "des pastilles chocolat", 3,
   "des bonbons acidulés", 2,
   "du chewing gum", 1,
   "une barre-nougat", 8
};
class Client {
 private:
   Confiserie achat;          // produit que le client désire acheter
   int        depenseMax;     // dépense maximum du client
 public:
   Client(int MaxPrix)
   {  // constructeur d'un client dont le produit souhaité est tiré au hasard
      // et le montant maximum est aléatoire de 1 à MaxPrix
      achat = Liste[EntierAleatoireInf(NbProduits)];
      depenseMax = MaxPrix;
   } // Client(int MaxPrix)
   void PresenteToi()
   {  // le client-récepteur affiche ses intentions à l'écran
      cout << "Je désire " << achat.Nom
           << " de prix " << achat.Prix;
      cout << "\nJe suis prêt à payer " << depenseMax << "franc"
           << ((depenseMax > 1) ? "s\n" : "\n");
   } //  void PresenteToi()
   void Decide()
   {  // le client-récepteur indique, à l'écran, sa décision
      if (depenseMax < achat.Prix)
         cout << "Je renonce à l'achat\n";
      else
         cout << "J'achète " << achat.Nom << "\n";
   } //  void Decide()
}; // class Client
```

```
void main()
{
   Client * P;
   char Carac;
   cout << "Simulation de l'arrivée de clients\n\n";
   do {
      P = new Client (EntierAleatoireInf(10)+1);
      P -> PresenteToi();
      P -> Decide();
      delete P;
      cout << "\nEncore un client (O/N) : ";
      cin >> Carac;
   } while (toupper(Carac) == 'O');
} // void main()
```

2 Techniques de base en C++

Ce chapitre aborde les techniques d'instanciation, fondamentales pour la programmation orientée objets. En C++, la création d'un objet fait appel à un constructeur, sa suppression fait intervenir un destructeur. Ces deux mécanismes sont étudiés en détail dans ce chapitre. Au fil de cette étude, on aborde également un certain nombre de concepts fondamentaux : portée et durée de vie d'une variable, conversion de type. On approfondit aussi des techniques de base comme l'allocation dynamique de mémoire, la transmission de paramètres, la spécification de référence, la copie d'un objet et l'utilisation du modificateur const.

2.1 - Une classe pour enregistrer des entiers

Pour étudier les techniques de base de C++, nous utiliserons tout au long des exemples une classe TableEntiers. Une instance de cette classe sera un conteneur d'entiers dans lequel on pourra enregistrer des entiers. Chaque objet de cette classe sera représenté par trois données membres :

- taille représente la taille du conteneur (instance de la classe TableEntiers) ;
- nbElem représente le nombre d'entiers répertoriés à chaque instant dans le conteneur ;
- table est un tableau d'entiers. Il contiendra les entiers stockés dans le conteneur.

La figure 2.1 illustre cette représentation. La dimension du tableau est indiquée par la donnée membre taille qui limite la capacité de l'objet à cent entiers. Le tableau table a donc cent éléments indexés de 0 à 99[1]. La donnée membre nbElem indiquera le nombre d'entiers répertoriés. Sur la figure 2.1, elle a la valeur 4 : l'instance représentée contient donc quatre entiers : 14, -20, 30 et 40, enregistrés dans les éléments de table de rangs 0, 1, 2 et 3. Les instances de TableEntiers seront gérées comme des piles. Chaque nouvel entier sera ajouté à

1 En C++, les éléments d'un tableau sont toujours indexés à partir de 0.

l'emplacement de rang nbElem, puis nbElem augmentera d'une unité. De même, l'entier supprimé sera toujours le dernier ajouté : pour enlever un entier, on diminuera nbElem d'une unité.

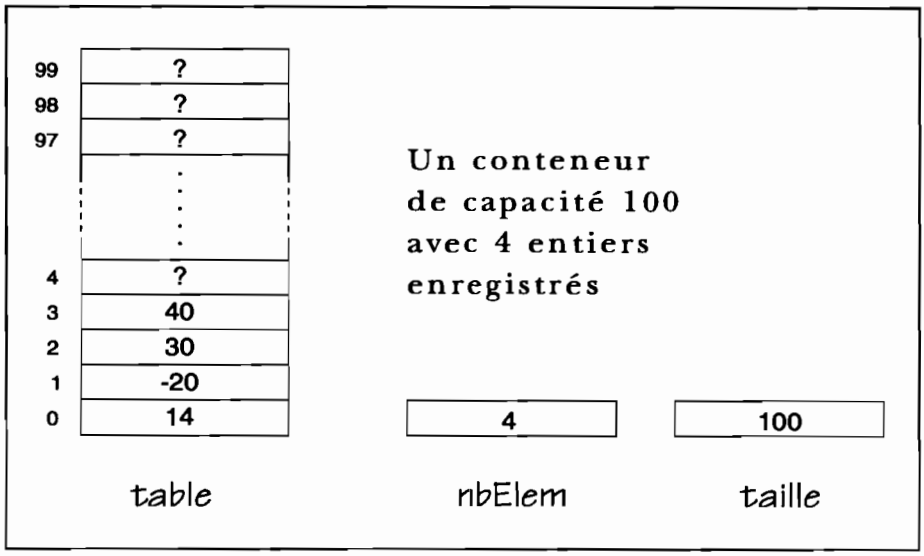


Figure 2.1 : trois données membres représentent un objet TableEntiers

La capacité maximale d'une instance est, pour le moment, fixée par une constante, que nous appellerons NbMax. Cette constante fixe la dimension de la donnée membre *table*. Nous pouvons donc écrire une première définition de la classe TableEntiers.

```
// première version de la classe TableEntiers
const int NbMax = 100;
class TableEntiers {
  private :
    int taille;
    int nbElem;
    int table[NbMax];
}; // class TableEntiers
```

2.2 - Créer des objets : choix d'un constructeur

2.2.1 - Constructeur par défaut

Nous avons vu au chapitre 1 que le compilateur C++ fournit, par défaut, un constructeur pour instancier des objets d'une classe. Avec la définition précédente de la classe TableEntiers, ce constructeur est simplement invoqué par :

```
TableEntiers T;
```

Les données membres sont créées en mémoire par le constructeur, mais leur valeur reste indéterminée. Si l'on souhaite par exemple qu'un objet tel que T soit considéré comme un conteneur vide de capacité 100, il faut que les données membres nbElem et taille soient initialisées respectivement à 0 et 100. Il faut donc leur donner une première valeur en utilisant une fonction membre spécifique, Initialise. Cette fonction doit être publique, afin d'être utilisée à l'extérieur de la définition de la classe. Etant fonction membre de TableEntiers, elle peut accéder aux données membres des instances de TableEntiers. La figure 2.2 propose une seconde définition de la classe, avec la fonction Initialise.

```
const int NbMax = 100;
class TableEntiers {
  private :
   int taille;
   int nbElem;
   int table[NbMax];
  public:
   void Initialise() { taille = NbMax; nbElem = 0; }
} // class TableEntiers
```

Figure 2.2 : Initialise, fonction membre publique de TableEntiers

La création d'un conteneur TableEntiers devra être systématiquement suivie d'un appel à la fonction d'initialisation.

```
TableEntiers T;
T.Initialise();
```

Oublier cette initialisation peut provoquer des résultats aberrants : le nombre réel d'éléments stockés étant incontrôlable, les fonctions d'ajout, de retrait et d'affichage du contenu d'un objet TableEntiers ne seront plus fiables.

D'autre part, en fournissant une fonction publique telle que Initialise, le programmeur fournit un outil dangereux, puisque rien n'empêchera l'utilisateur de la classe d'envoyer un message Initialise() à un objet TableEntiers en cours d'utilisation et d'effacer ainsi le nombre d'entiers stockés.

Il est préférable de pouvoir initialiser les données membres au moment même de la création de l'objet TableEntiers. Le langage C++ permet au programmeur de définir lui-même un constructeur pour y incorporer ses propres opérations d'initialisation ou de contrôle.

2.2.2 - Définir son constructeur

Un constructeur en C++ est une fonction membre dont l'identificateur est toujours le nom de la classe. Cette fonction membre ne doit spécifier aucun type renvoyé. Sa définition peut indiquer zéro, un ou autant d'arguments de n'importe quel type, du moment que le programmeur a l'usage de ces arguments. Un constructeur est défini dans la section public (on ne pourrait créer d'instances de la

classe s'il était défini comme fonction dans la section public de la classe[2]). La figure 2.3 présente un constructeur qui reprend les opérations de la fonction membre Initialise. Cette fonction n'est donc plus nécessaire dans la classe.

```
const int NbMax = 100;
class TableEntiers {
  ... // etc. Cf. figure 2.2
  public:
   TableEntiers()
   {
      taille = NbMax;
      nbElem = 0;
   }
}; // class TableEntiers
```

Figure 2.3 : un constructeur explicite pour la classe TableEntiers

Ce constructeur utilise les mécanismes implicites de C++ pour réserver un espace mémoire aux données membres taille, nbElem et table. Il effectue ensuite les deux instructions d'initialisation précisées par le programmeur.

La classe TableEntiers que nous venons de décrire est utilisable en l'état. Nous pourrions maintenant définir des fonctions membres d'ajout et de suppression d'entiers.

Cette classe présente néanmoins un défaut important dans l'optique d'une utilisation généralisée : la taille du conteneur est définitivement fixée par la constante NbMax. On ne peut, dans un même programme, manipuler des objets TableEntiers de taille différente. Il est donc nécessaire de modifier la définition de la classe pour y introduire le paramétrage de la taille du conteneur.

Plutôt que de définir la donnée membre table comme un tableau dont la dimension est fixée à la compilation, nous définirons cette donnée membre comme un pointeur sur des valeurs entières :

```
int * table;
```

Ce pointeur sera initialisé à l'exécution du constructeur pour désigner un espace alloué dynamiquement dans lequel on enregistrera les entiers contenus par l'objet construit. D'un objet à l'autre, la taille de cet espace pourra changer et, pour un même objet, elle pourra même varier selon les besoins de stockage de l'objet au fil du temps. Bien sûr, cette variation supposera que l'on dispose des fonctions membres adéquates pour modifier l'allocation initiale effectuée par le constructeur.

Examinons la figure 2.4 qui présente la nouvelle version de la classe TableEntiers.

Avec cette nouvelle définition, le constructeur utilise la fonction membre privée alloueTable. Nous étudierons plus particulièrement cette fonction au paragraphe suivant. Dans un premier temps, considérons-la comme une boîte noire

2 On peut aussi définir certains constructeurs comme fonctions membres privées, pour interdire leur emploi (Cf. chap. 5).

dont le rôle est d'effectuer l'allocation dynamique de la donnée membre table et examinons la définition du constructeur.

```
// seconde version de la classe TableEntiers
class TableEntiers {
  private:
   int taille;
   int nbElem;
   int * table;
   void alloueTable(int Dimension)
   {
      table = new int [Dimension];
      if (table == NULL) {
         cerr << "TableEntiers : allocation impossible\n";
         exit(1);
      }
      // ici, l'allocation a réussi
      taille = Dimension;
   }
  public:
   TableEntiers (int Dim)
   {  // constructeur pour une capacité initiale de Dim entiers
      alloueTable(Dim);
      nbElem = 0;
   }
}; // class TableEntiers
```

Figure 2.4 : deuxième définition de la classe TableEntiers

Lorsque la taille du contenu était prédéfinie par la constante NbMax, il n'était pas nécessaire d'effectuer à la construction une opération particulière pour la donnée membre table. Avec la nouvelle définition, cette donnée membre doit être initialisée et cette initialisation dépend d'une information donnée par le programmeur : le constructeur attend un argument entier qui précise la capacité initiale de l'objet-conteneur. Cet argument, Dim, est utilisé par le constructeur pour appeler la fonction membre alloueTable. Cette fonction réservera l'emplacement-mémoire nécessaire. Elle initialisera ensuite le pointeur table avec l'adresse de cet emplacement. Elle fixera enfin la valeur de la donnée membre taille.

Avec le constructeur de la figure 2.4, nous pouvons maintenant définir des objets TableEntiers de capacités différentes :

```
TableEntiers T1(50), T2(320);
```

L'exécution de cette instruction instancie les objets T1 et T2 de capacités respectives 50 et 320. On notera la syntaxe particulière de l'appel d'un constructeur avec argument : alors qu'un appel de fonction ordinaire place les paramètres d'appels immédiatement après l'identificateur de la fonction, l'appel d'un constructeur intercale le nom de l'objet construit entre l'identificateur (qui est alors le nom de la classe) et le ou les paramètres.

Remarquons enfin qu'avec la classe de la figure 2.4, la définition de :

```
TableEntiers T;
```

provoque une erreur dès la compilation, avec un message expliquant qu'il est impossible de trouver un modèle pour le constructeur TableEntiers() sans argument. Cette instruction, qui se compilait correctement avec la première définition de la classe TableEntiers, n'est plus valide. En effet, il n'existe plus de constructeur défini sans aucun argument. On constate ainsi que la définition explicite d'un constructeur dans la classe TableEntiers inhibe l'utilisation du constructeur par défaut (qui, lui, n'attend pas d'arguments). Nous verrons en 2.2.4 comment on peut rétablir ce constructeur par défaut.

2.2.3 - Allocation dynamique

Revenons à la fonction membre privée *alloueTable*, dont nous rappelons ci-dessous la définition :

```
void alloueTable(int Dimension)
{
   table = new int [Dimension];
   if (table == NULL) {
      cerr << "TableEntiers : allocation impossible\n";
      exit(1);
   }
   // ici, l'allocation a réussi
   taille = Dimension;
}
```

La première instruction de cette fonction effectue l'allocation-mémoire nécessaire pour la donnée membre table. Pour ce faire, l'opérateur new est utilisé. Cet opérateur réserve l'espace-mémoire correspondant à l'argument qui lui est fourni (ici int [Dimension]) puis renvoie l'adresse de cet espace-mémoire. Avec la fonction membre *alloueTable* cette adresse devient la valeur de la donnée membre table.

Dans notre exemple, la syntaxe utilisée pour l'opérateur new est la suivante :

```
new UnType [UnEntier]
```

Dans cette écriture, UnType est un type prédéfini[3] et l'opérateur new alloue un espace-mémoire correspondant à UnEntier valeurs contiguës de ce type. A l'issue de l'allocation, l'opérateur new renvoie l'adresse du premier octet de l'espace alloué. Dans la forme syntaxique utilisée, la spécification [UnEntier] peut être omise : en son absence, l'exécution de new UnType alloue un espace-mémoire pour une seule valeur du type UnType.

Si la mémoire disponible est insuffisante pour l'allocation, l'opérateur new renvoie la constante prédéfinie NULL, qui représente l'absence de pointage.

Dans la fonction *alloueTable*, l'allocation mémoire est contrôlée par un test sur la valeur du pointeur table. Si la place est insuffisante, la fonction new a renvoyé

3 Nous verrons en 2.4.3 que new peut aussi être utilisé lorsque UnType représente le nom d'une classe.

le pointeur NULL. La sortie du programme est alors immédiate, avec un message à l'objet *cerr*, puis un appel à la fonction *exit*. L'appel de la fonction *exit* termine l'exécution. Dans notre exemple, cet appel est fait avec l'argument 1. Cette valeur sera renvoyée à l'entité qui a déclenché l'exécution du programmeur et qui est, la plupart du temps, le système d'exploitation. Cette valeur de retour avec *exit* caractérise en général le type d'erreur qui a provoqué l'arrêt du programme.

L'objet *cerr* représente le flux de sortie réservé aux erreurs : par défaut, il s'agit de l'écran. Ce flux peut aussi être redirigé vers un fichier. Il nécessite pour son utilisation la même librairie que les objets *cin* et *cout* (fichier d'en-tête iostream.h).

On peut se demander pourquoi nous avons, avec *allouеTable*, défini une fonction membre particulière pour l'allocation. Sachant qu'elle n'est utilisée que par le constructeur nous aurions en effet pu inclure directement son code dans ce constructeur. Nous ne l'avons pas fait, parce que nous prévoyons d'utiliser ce code dans plusieurs constructeurs différents. Il pourrait aussi être appelé pour l'extension de la capacité d'un objet existant. Nous avons donc choisi de « factoriser » cette allocation qui sera commune à plusieurs fonctions membres et d'en faire une fonction membre séparée. Cette fonction membre est privée, car elle n'est pas à mettre à la disposition des utilisateurs de la classe : son utilisation est réservée aux implémenteurs de la classe.

2.2.4 - Rétablir le constructeur par défaut

Avec le seul constructeur défini à la figure 2.4 :

```
TableEntiers (int Dim)
{   // constructeur pour une capacité initiale de Dim entiers
    alloueTable(Dim);
    nbElem = 0;
}
```

nous avons constaté qu'une définition telle que

```
TableEntiers T;
```

était refusée par le compilateur. Dès qu'un constructeur est explicitement défini, la possibilité d'utiliser le constructeur par défaut disparaît.
On peut cependant obtenir un constructeur par défaut en définissant un constructeur supplémentaire sans aucun arguments. On peut aussi modifier la définition du constructeur en spécifiant une valeur par défaut pour son argument :

```
TableEntiers (int Dim = 100)
{   // constructeur pour une capacité initiale de Dim entiers
    alloueTable(Dim);
    nbElem = 0;
}
```

Cette définition permet d'appeler le constructeur de deux manières différentes :

- soit en spécifiant un argument d'appel entier pour le paramètre formel Dim,
- soit sans spécifier d'argument d'appel : dans ce cas, le paramètre Dim prendra la valeur prévue par défaut.

Avec la définition ainsi modifiée, nous pouvons compiler avec succès les définitions suivantes :

```
TableauEntiers T1 (500); // conteneur de capacité 500
TableauEntiers T2; // conteneur de capacité 100
```

Nous reviendrons en 2.2.6 sur la spécification de valeurs par défaut pour les arguments d'une fonction C++.

```
class TableEntiers {
  private:
   int taille;
   int nbElem;
   int * table;
   void alloueTable(int Dimension)
   {
      table = new int [Dimension];
      if (table == NULL) {
         cerr << "TableEntiers : allocation impossible\n";
         exit(1);
      }
      // ici, l'allocation a réussi
      taille = Dimension;
   }
  public:
   TableEntiers (int Dim = 100)
   {// constructeur pour une capacité initiale de Dim entiers
      alloueTable(Dim);
      nbElem = 0;
   }
   TableEntiers(int Dim, int ValInit)
   {
      alloueTable(Dim);
      for (nbElem = 0; nbElem < Dim; nbElem++)
          table[nbElem] = ValInit;
   }
}; // class TableEntiers
```

Figure 2.5 : un deuxième constructeur pour TableEntiers

2.2.5 - Plusieurs constructeurs dans une même classe

L'utilisation d'un seul constructeur peut paraître limitante. On peut en effet souhaiter que l'instanciation puisse engendrer des objets avec des caractéristiques variées. Le langage C++ autorise la définition de plusieurs constructeurs différents dans une même classe. Supposons que l'on souhaite pouvoir créer des objets TableEntiers de capacité n et qui contiennent, dès l'instanciation, n entiers tous égaux. Nous pouvons, pour cela, définir un nouveau constructeur attendant deux

arguments entiers, l'un pour la taille souhaitée, l'autre pour l'entier répété. Ce deuxième constructeur est défini dans la figure 2.5.

Dans le corps de ce constructeur, l'itération définie avec for, effectue l'initialisation de l'espace pointé par la donnée membre table en traitant celle-ci comme un identifiant de tableau. Il y a donc une analogie entre pointeur et tableau : nous allons l'étudier d'un peu plus près.

En C++, les pointeurs et les tableaux sont gérés avec les mêmes mécanismes. Quand on définit un tableau Tab de n valeurs de type UnType en écrivant UnType Tab[n], on dispose :

- d'un espace-mémoire pouvant accueillir les n valeurs, débutant à une adresse A,
- du pointeur invariable Tab sur le type UnType dont la valeur est A.

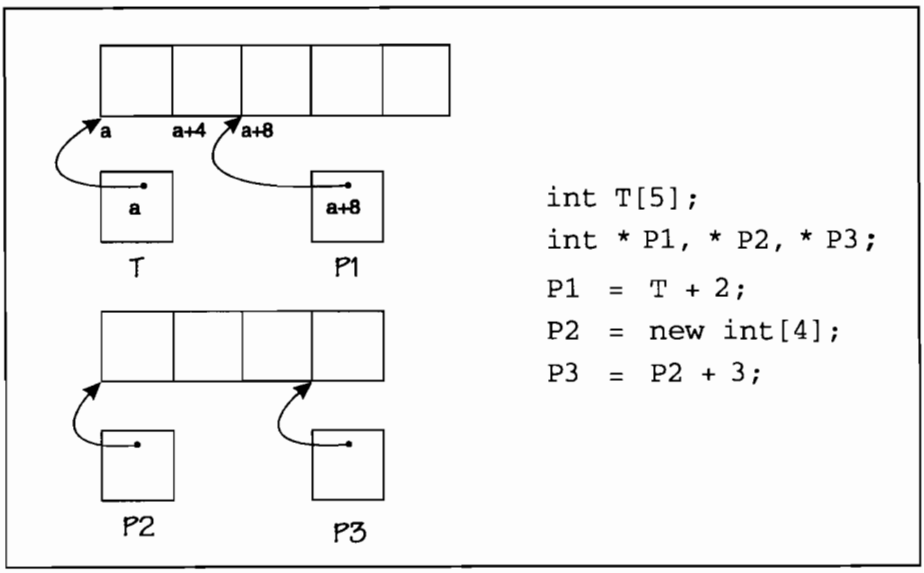


Figure 2.6 : pointeurs et tableaux

La figure 2.6 illustre les analogies existant entre pointeurs et tableaux. Parmi les quatre pointeurs qui y sont définis (T, P1, P2, P3), un seul est invariable : c'est T, l'identificateur du tableau de cinq entiers. L'initialisation de P1 avec P1 = T+2, montre que les pointeurs peuvent intervenir dans des opérations arithmétiques à valeurs entières, dans lesquelles un entier n ne représente pas n positions-mémoire, mais le nombre d'octets occupés par n valeurs du type pointé. Si nous supposons que le compilateur utilisé représente une valeur de type int sur quatre octets, la valeur affectée à P1 avec P1 = T+2; est l'adresse *a+8* (*a* étant la valeur de T, adresse du premier élément de T). En d'autres termes P1 pointe désormais sur l'élément de T de rang 2. Toujours sur la même figure 2.6, l'initialisation de P2 avec le résultat d'une allocation dynamique (new int [4]) montre encore l'analogie entre pointeur et tableau. La variable P2 désigne alors un tableau de dimension 4,

alloué dynamiquement et on pourrait par exemple exécuter une instruction telle que :

```
for (int K = 0; K < 4; K++) P2[K] = 0;
```

pour initialiser à zéro les éléments de ce tableau. Enfin, la dernière instruction présentée à la figure 2.6 :

```
P3 = P2 + 3; // P3 désigne le quatrième élément du tableau pointé par P2
```

montre bien que, par rapport à T, un pointeur tel que P3 peut voir sa valeur varier. En effet, au moment de sa définition P3 avait pris une valeur indéterminée. Avec P3 = P2+3, cette valeur indéterminée est remplacée par l'adresse du quatrième élément du tableau pointé par P2.
Revenons maintenant au nouveau constructeur de la classe TableEntiers :

```
TableEntiers(int Dim, int ValInit)
{
   alloueTable(Dim);
   for (nbElem = 0; nbElem < Dim; nbElem++)
       table[nbElem] = ValInit;
}
```

Nous pourrions aussi utiliser un pointeur variable pour initialiser le tableau pointé par table :

```
for (int * Ptr = table; Ptr < table+dim; Ptr++)
     *Ptr = ValInit;
```

Cette séquence présente des notations nouvelles pour le lecteur. Examinons-les. L'itération va faire varier Ptr en l'amenant à désigner successivement chacun des éléments du tableau pointé par table :

- l'initialisation de Ptr lui fait désigner le premier élément : Ptr = table;
- l'itération passe d'un élément au suivant en augmentant Ptr d'une unité (Ptr++;),
- chaque élément désigné par Ptr prend la valeur ValInit : *Ptr = ValInit;

Cette dernière opération montre l'utilisation de l'opérateur unaire *[4] qui, appliqué à un pointeur, renvoie la variable pointée. L'opération inverse qui permet de passer d'une valeur à son adresse est effectuée avec l'opérateur unaire &[5] :

```
Ptr = & table[2] // Ptr prend pour valeur l'adresse de table[2]
```

Après cette initiation à l'utilisation de pointeurs, revenons à la classe TableEntiers de la figure 2.5. Ses deux constructeurs sont définis par :

```
TableEntiers (int Dim = 100) {....}
TableEntiers (int Dim, int ValInit) {....}
```

et on peut, par exemple, les utiliser pour les instanciations suivantes :

```
TableEntiers T1; // premier constructeur avec argument par défaut
TableEntiers T2 (50); // premier constructeur
TableEntiers T3 (200, 0); // second constructeur
```

---

4 A ne pas confondre avec l'opérateur binaire de multiplication, représenté par le même caractère * !

5 A ne pas confondre avec l'opérateur binaire de conjonction logique bit à bit ni avec la spécification de référence, notés tous deux par le même caractère !

2.2.6 - Signature d'une fonction, arguments par défaut

On pourrait multiplier les exemples et ajouter un troisième constructeur, TableEntiers (int Dim, int Nbre, int ValInit) chargé de créer un objet TableEntiers de capacité Dim et contenant au départ Nbre entiers tous égaux à ValInit :

```
TableEntiers(int Dim, int Nbre, int ValInit)
{
   alloueTable(Dim);
   if (Nbre < 0 || Nbre > Dim) {
      cerr "trop d'entiers pour cet objet \n"
      exit(2);
   }
   for (nbElem = 0; nbElem < Nbre; nbElem++)
       table[nbElem] = ValInit;
}
```

On peut définir autant de constructeurs que l'on veut dans une classe donnée, pour autant que toutes les signatures de ces constructeurs soient distinctes. On appelle *signature d'une fonction* la combinaison de sa classe (si elle est membre d'une classe), de son identificateur et de la suite des types de ses paramètres. La signature d'un constructeur est constituée du nom de sa classe et de la suite des types d'arguments qu'il utilise.

Illustrons le traitement de ces signatures par un exemple. Supposons que notre classe TableEntiers dispose maintenant des trois constructeurs :

```
TableEntiers (int Dim = 100) {....}
TableEntiers (int Dim, int ValInit) {....}
TableEntiers (int Dim, int Nbre, int ValInit) {....}
```

Si nous décidions de fixer pour le second constructeur, une valeur par défaut pour l'argument ValInit avec, par exemple,

```
TableEntiers (int Dim, int ValInit = 0) {....}
```

nous donnerions à ce constructeur deux signatures possibles :

- TableEntiers (int, int),
- TableEntiers (int) (valeur par défaut pour le second paramètre).

La deuxième de ces signatures serait en conflit avec celle du premier constructeur quand celui-ci est employé sans valeur par défaut. Le compilateur ne pourrait donc plus compiler correctement une définition telle que :

```
TableEntiers T2(50);
```

De même, si nous supposons maintenant que la classe ne dispose que des deux constructeurs :

```
TableEntiers (int Dim = 100, int ValInit = 0);
TableEntiers (int Dim, int Nbre, int ValInit = 0);
```

la signature TableEntiers::TableEntiers (int, int) est ambiguë, car elle peut correspondre soit à un appel du premier constructeur, soit à un appel du second avec le dernier argument fourni par défaut.

Les arguments par défaut sont donc à manipuler avec précaution.

L'utilisation de valeurs par défaut pour un argument n'est pas réservée aux constructeurs. En C++, on peut fournir des valeurs par défaut pour un ou plusieurs arguments de toute fonction. On pourrait, par exemple, décider de remplacer les trois constructeurs de notre classe TableEntiers par un seul constructeur de signature :

```
TableEntiers(int Dim = 100, int Nbre = 100, int ValInit = 0)
```

Avec ce constructeur, l'appel TableEntiers T(1000) instanciera un objet TableEntiers de capacité 1000 et contenant au départ 100 entiers de valeur nulle.

Les valeurs par défaut sont toujours données en partant du dernier argument de la fonction et, pour éviter des ambiguïtés, on ne peut intercaler un argument sans valeur par défaut entre deux arguments ayant une telle valeur. On ne peut pas, par exemple, déclarer le constructeur

```
TableEntiers (int Dim = 100, int Nbre, int ValInit = 0);
```

2.2.7 - Fonctions à nombre variable d'arguments

Supposons maintenant que le programmeur souhaite disposer d'un constructeur qui permette d'obtenir, dès l'instanciation, des objets TableEntiers contenant quelques entiers non forcément identiques. Aucun des constructeurs précédents ne peut convenir.

```
class TableEntiers {
 private:
... // etc. Cf. figure 2.5
 public:
... // etc. Cf. figure 2.5
   TableEntiers(int Dim, int NbArgs, ... )
   { // l'objet est créé avec NbArgs entiers, fournis par la liste ...
      if (NbArgs < 1 || NbArgs > Dim) {
           cerr << "nombre d'arguments incorrect\n";
           exit(3);
      }
      va_list PtrArg;
      alloueTable(Dim);
      va_start (PtrArg, NbArgs);
      for (nbElem = 0; nbElem < NbArgs; nbElem++)
           table[nbElem] = va_arg(PtrArg, int);
      va_end(PtrArg);
   }
} // class TableEntiers
```

Figure 2.7 : constructeur à nombre variable d'arguments

On pourrait bien sûr envisager de définir des constructeurs supplémentaires tels que :

```
TableEntiers (int Dim, int Val1);
TableEntiers (int Dim, int Val1, int Val2);
TableEntiers (int Dim, int Val1, int Val2, int Val3);
```

etc. Mais, pour gérer ce genre de situation, il vaut mieux définir un constructeur avec un nombre variable d'arguments. On utilise pour cela des macro-instructions du langage, qui sont obtenues avec l'inclusion du fichier d'en-tête stdarg.h. Ces macros permettent la gestion d'un nombre variable d'arguments. Ce constructeur particulier est défini figure 2.7.

La définition de ce constructeur introduit un certain nombre de nouveautés qu'il est nécessaire d'approfondir :

- La syntaxe de C++ impose que la suite variable d'arguments soit matérialisée par trois points (...) successifs. On place évidemment les arguments obligatoires en début de liste (Dim et NbArgs). NbArgs est ici le dernier argument obligatoire : dans notre cas, il indique le nombre d'arguments de la liste variable.

- Un nouveau type (va_list) et trois macros (va_start, va_arg et va_end) sont ensuite utilisées.

- PtrArg est un pointeur de type va_list. Il pointe sur la liste des arguments. Son initialisation se fait avec la macro va_start, avec l'appel va_start (PtrArg, NbArgs). Dans cet appel, le second paramètre de va_start doit être le nom du dernier argument obligatoire passé à la fonction (dans notre cas : NbArgs). La macro va_start attend donc deux arguments : le premier est le pointeur qui désignera successivement chacun des arguments, le second est le nom du dernier argument obligatoire du constructeur.

- Chaque appel de la macro va_arg renvoie l'argument pointé dans la liste et déplace le pointeur PtrArg d'un argument. On précise à va_arg le type d'argument attendu à ce niveau. Il est en effet nécessaire d'indiquer ce type car, pour faire avancer PtrArg d'un argument, la macro doit augmenter ce pointeur du nombre d'octets occupé par l'argument. Ce nombre d'octets peut être obtenu si l'on connaît le type : la macro utilisera l'opérateur sizeof : sizeof(int) renvoie le nombre d'octets occupés par une valeur de type int. Ainsi, l'appel de sizeof(TableEntiers) donnera le nombre d'octets occupés par les données membres d'un objet TableEntiers[6]. La macro va_arg attend donc deux arguments : le premier est le pointeur sur la liste des arguments de la fonction, le second est le type de l'argument désigné.

- On clôt le parcours de la liste avec une indication convenue lors de l'établissement du constructeur : dans notre cas, le nombre d'arguments sera annoncé par l'utilisateur à l'appel du constructeur (variable NbArgs). On peut également décider que le dernier argument est 0 (ou

6 Pour une classe donnée, CCC, sizeof(CCC) ne comprend jamais la dimension des allocations dynamiques effectuées par l'objet : elles ne sont connues qu'à l'exécution.

NULL si les arguments variables sont de type pointeur) et interrompre le parcours une fois cette valeur rencontrée, etc. Il est de toutes manières impératif de savoir quand arrêter le déplacement du pointeur va_list : l'information ne peut être fournie par C++.

- Pour restaurer un état correct de la pile des arguments, après les manipulations effectuées avec *va_start* et *va_arg*, il est nécessaire, en fin de traitement, d'appeler la macro *va_end*. Cette macro attend en argument le nom du pointeur utilisé pour les arguments. En négligeant l'appel à *va_end*, on s'expose par la suite à un comportement erratique du programme.

Pour vérifier le fonctionnement de nos différents constructeurs, définissons une fonction membre publique qui permettra à un objet d'envoyer sa représentation dans le flux standard de sortie. Cette fonction (AfficheToi) est présentée figure 2.8.

```
void AfficheToi()
{ // fonction membre publique : affiche les informations de l'objet dans stdout
   cout  << "\nTableEntiers : "  << "\n : "  << taille
         << ", contient "  << nbElem  << " entiers";
   for (int K = 0; K < nbElem ; K++ )
       cout << "\n " << K << " : " << table[K];
   cout << '\n';
}
```

Figure 2.8 : fonction membre d'affichage d'un objet TableEntiers

Si nous supposons que notre classe dispose maintenant des deux constructeurs :

```
TableEntiers(int Dim = 100)
TableEntiers(int Dim, int NbArgs, ... )
```

on peut maintenant compiler et exécuter une séquence telle que :

```
TableEntiers T0; T0.AfficheToi();
TableEntiers T1(4,4,10,20,30,40); T1.AfficheToi();
TableEntiers T2 (3,4,1,2,3,4);
    // erreur d'exécution à l'instanciation de T2, capacité insuffisante.
```

Dans cet exemple, T0 est instancié avec le constructeur TableEntiers (int Dim = 100). Il représente un conteneur de capacité 100, vide au départ. T1 est instancié avec le second constructeur. Il a une capacité de 4 entiers et contient les entiers 10, 20, 30 et 40. L'instanciation de T2 arrêtera le programme avec le message *nombre d'arguments incorrects*.

Pour conclure, remarquons que chaque nouvelle définition d'un constructeur doit tenir compte de la forme des définitions des constructeurs déjà existants.

Le fait d'attendre un nombre indéterminé d'arguments augmente ainsi les risques d'ambiguïté. Dans notre cas, le constructeur à nombre indéterminé

d'arguments introduit une ambiguïté d'interprétation face à d'éventuels constructeurs ayant deux, trois, quatre (ou plus) arguments de type int.

Telles qu'elles sont conçues, les macros va_xx permettent d'entrer un nombre quelconque d'arguments de types différents. La seule contrainte est de connaître, à chaque déplacement du pointeur, le type d'argument attendu. La fonction printf que nous avons étudiée au chapitre 1 utilise une liste variable d'arguments qui lui indique les valeurs à afficher. Le type de chaque valeur est fourni dans le premier argument par un code de mise en forme (%d, %s, etc..).

2.3 - Macros et fonctions en ligne

Nous avons rapidement évoqué l'utilisation de macros dans l'exemple du constructeur de la figure 2.8. Il est possible, en C++, d'utiliser des macros afin de simplifier l'écriture d'un programme. Par exemple, la directive :

```
#define chiffre(carac) (carac) - '0'
```

définit une macro permettant de remplacer une valeur de type caractère, qui représente un chiffre, par l'entier correspondant. Le remplacement est effectué, avant la compilation proprement dite, par le préprocesseur associé au compilateur C++. Par exemple, une instruction telle que :

```
X = X * chiffre (C)
```

sera compilée comme :

```
X = X * (C - '0')
```

De même, la définition de :

```
#define carre(x) x*x
```

introduit une macro qui permet de remplacer dans le corps du programme toutes les occurrences de *carre*(x) par x * x. Ce remplacement s'effectue avant la compilation. Il n'y a donc aucun contrôle préalable. Dans notre exemple, rien n'empêche le programmeur d'écrire *carre*("A"). Le précompilateur remplacera cet appel par "A" * "A", ce qui n'a aucun sens. De même, l'écriture de *carre*(z+1) se traduit après la compilation en z+1 * z+1, ce qui n'a numériquement rien à voir avec le carré de (z+1). Bien sûr, on pourrait définir cette macro par

```
#define carre(x) (x)*(x)
```

ce qui permettrait un traitement correct de l'écriture *carre*(z+1). Mais on ne pourrait pas empêcher que l'appel *carre*(z++) ait un effet de bord qui augmente deux fois de 1 la valeur de z.

La définition de *carre* aurait pu être codée sous forme de fonction en ligne :

```
inline int carre(int x) { return x*x; }
```

L'identificateur *carre* ne désigne alors plus une macro mais une véritable fonction qui est traitée par le compilateur. Celui-ci effectue le contrôle de type : l'appel *carre*("A") est rejeté explicitement par un message signalant que l'argument de la fonction *carre* ne convient pas. De même, la compilation de *carre*(z+1) passera correctement la valeur de l'expression z+1 comme argument de la fonction. Egalement, la compilation de *carre*(z++) n'augmentera que d'une unité la valeur de z.

Le mot clé inline précise au compilateur que tout appel de la fonction doit être explicitement remplacé par la séquence d'instructions de cette fonction. L'exécution sera donc plus rapide qu'avec un appel normal de fonction. En effet, on économise les opérations d'appel et de retour (sauvegarde de l'environnement d'appel, création de l'environnement de la fonction et suppression de cet environnement en fin d'exécution de la fonction). En revanche, le code de la fonction est inséré à chaque appel : si ce code est important on augmente l'encombrement du programme.

Par rapport à une macro, la fonction en ligne est plus performante, car elle permet de contrôler le type de ses arguments.

Les fonctions définies dans le corps d'une classe sont automatiquement considérées comme des fonctions en ligne, sauf si elles comportent dans leur séquence d'instructions des itérations ou des énoncés switch qui sont incompatibles avec le développement en ligne. Dans ce cas, le compilateur avertit en général le programmeur de l'impossibilité de traiter cette fonction en ligne.

Les fonctions déclarées dans le corps de la classe, mais définies à l'extérieur sont traitées comme des fonctions normales dont le code ne sera présent qu'une seule fois dans le programme. Chaque appel d'une telle fonction empilera un environnement dans la pile d'exécution, déclenchera ensuite l'exécution du code de la fonction et se terminera par une restauration de l'environnement empilé.

Quand on définit une classe, on se contente en général de déclarer les fonctions membres dans le corps de la classe, pour les définir à l'extérieur. La nouvelle définition de notre classe TableEntiers répondant à ce principe est décrite figure 2.9.

Quand on définit une fonction membre FFF à l'extérieur de sa classe CCC, on doit indiquer explicitement la classe avec l'opérateur C++ de portée, noté `::`. La forme d'une telle définition est :

```
TypeRenvoyé CCC::FFF (..) {..}
```

La qualification par le nom de la classe est en effet indispensable pour permettre au compilateur de distinguer deux fonctions homonymes qui ont le même type renvoyé et les mêmes arguments, mais qui ne sont pas membres de la même classe.

Sur la figure 2.9, on peut remarquer :

- que la définition de la classe est plus compacte, et donc mieux lisible, quand les fonctions y sont seulement déclarées,
- que l'on peut cependant obtenir qu'une fonction membre déclarée à l'extérieur de la classe soit traitée en ligne : il suffit que la définition soit explicitement introduite par le spécificateur inline. Nous l'avons fait pour le premier constructeur,
- que, si une fonction est déclarée dans la classe avec une ou plusieurs valeurs par défaut pour les arguments, la définition de cette même fonction, extérieure à la classe, ne doit pas reprendre la ou les valeurs par défaut (Cf. premier constructeur).

```
class TableEntiers {
  private:
    int taille;
    int nbElem;
    int * table;
    void alloueTable(int Dimension);
 public:
    TableEntiers(int Dim = 100);
    TableEntiers(int Dim, int NbArgs, ...);
    void AfficheToi();
}; // fin class TableEntiers

void TableEntiers::alloueTable(int Dimension)
{
    table = new int [Dimension];
    if (table == NULL) {
       cerr << "TableEntiers : allocation impossible\n"; exit(1);
    }
    // ici, l'allocation a réussi
    taille = Dimension;
}
inline TableEntiers::TableEntiers (int Dim)
{// constructeur pour une capacité initiale de Dim entiers
    alloueTable(Dim); nbElem = 0;
}
TableEntiers::TableEntiers(int Dim, int NbArgs, ... )
{ // objet construit avec NbArgs entiers, fournis par la liste ...
    if (NbArgs < 1 || NbArgs > Dim) {
        cerr << "nombre d'arguments incorrect\n"; exit(3);
    }
    alloueTable(Dim);
    va_list PtrArg;   va_start (PtrArg, NbArgs);
    for (nbElem = 0; nbElem < NbArgs; nbElem++)
        table[nbElem] = va_arg(PtrArg, int);
    va_end(PtrArg);
}
void TableEntiers::AfficheToi()
{ // fonction membre publique : affiche les informations de l'objet dans stdout
    cout  << "\nTableEntiers : "   << "\n : "   << taille
          << " , contient "   << nbElem  << " entiers";
    for (int K = 0; K < nbElem ; K++ )
        cout << "\n " << K << " : " << table[K];
    cout << '\n';
}
```

Figure 2.9 : troisième définition de la classe TableEntiers

2.4 - Destructeur par défaut et destructeur explicite

L'instanciation fait appel à des opérations de construction que nous venons d'étudier. Quels mécanismes sont mis en jeu pour l'opération inverse, c'est-à-dire la suppression d'un objet ? Avant de répondre en détail à cette question, examinons un exemple qui montrera les limites des opérations effectuées par défaut avec C++.

2.4.1 - Le mécanisme implicite de destruction

Utilisons la dernière version de notre classe TableEntiers présentée à la figure 2.9. Avec cette définition de classe, exécutons la séquence de la figure 2.10.

```
int NbRepetitions;
cout <<"\nTest de création itérative d'objets TableEntiers de taille 5000";
cout <<"\nEntrez le nombre d'itérations : ";
cin >> NbRepetitions;
for (int K = 0; K < NbRepetitions; K++) {
    cout << "\nCréation de la table numéro " << K;
    TableEntiers T(5000);
}
```

Figure 2.10 : création itérative d'objets TableEntiers

L'entier NbRepetitions indique le nombre de répétitions de l'itération définie par l'énoncé for. Il est initialisé avec une valeur fournie au clavier (cin >> NbRepetitions). Le programme crée ensuite NbRepetitions fois un objet TableEntiers. Cet objet T n'a d'existence qu'entre les accolades marquant le début et la fin du bloc d'instructions de la boucle. A chaque répétition, T est instancié puis il est supprimé. On peut donc s'attendre à ce que la mémoire nécessaire à l'exécution de la séquence ne dépende pas du nombre de répétitions de l'itération.

L'exécution de la séquence ne pose pas de problèmes pour les petites valeurs de NbRepetitions. Passé un seuil dépendant de la mémoire disponible sur l'ordinateur utilisé, la création d'un nouveau TableEntiers est impossible. Le message émis à l'interruption du programme vient de la fonction allouerTable : il n'y a plus de mémoire disponible pour réserver un espace à la variable d'instance table.

Si la variable de bloc, T, est bien créée puis détruite NbRepetitions fois, il n'en va pas de même pour l'espace alloué à chaque instanciation et pointé par la donnée membre table. Ces allocations successives, qui ne sont jamais annulées, finissent par accaparer la totalité de l'espace-mémoire disponible.

En fait, C++ fournit pour chaque classe un destructeur par défaut. Ce destructeur est une fonction membre chargée de défaire ce qu'a fait le constructeur. Elle est appelée automatiquement chaque fois qu'un objet sort de sa portée (en particulier, quand l'exécution atteint la fin du bloc où cet objet est défini). Le destructeur par défaut se contente de libérer en mémoire les espaces réservés aux données membres de l'objet détruit. Mais si ces données membres sont elles-mêmes des pointeurs comme table, seule la donnée est supprimée (i.e. l'adresse-mémoire

stockée dans table). Les valeurs pointées par ces données restent allouées en mémoire.

Il est donc nécessaire de remplacer le destructeur proposé par C++ pour y adjoindre des opérations plus spécifiques, comme la libération de l'espace alloué et pointé par table.

2.4.2 - Destructeur explicite

Un destructeur peut être défini ou seulement déclaré dans le corps de la classe. Il ne peut en exister qu'un seul : les objets d'une classe correspondent à un seul moule (représenté par les données membres de la classe), et la destruction de ce moule ne nécessite aucune variation. Pour chaque objet, un destructeur est automatiquement appelé par C++ pendant le déroulement du programme quand l'objet sort de sa portée. Pratiquement, il est nécessaire de définir un destructeur explicite dès que les constructeurs de la classe effectuent une allocation dynamique.

Pour la classe TableEntiers, nous définirons par exemple le destructeur avec :

```
class TableEntiers {
    ... // etc., Cf. figure 2.9
    ~TableEntiers()
    { // destructeur
      delete table;
    }
    // etc. Cf. figure 2.9
};
```

Comme le montre cette définition, un destructeur est une fonction membre qui se définit sans type renvoyé (comme un constructeur) et dont l'identificateur est le nom de la classe, préfixé par un tilde (~). Un destructeur se définit toujours sans argument.

L'exemple que nous venons de donner fait intervenir un nouvel opérateur C++, l'opérateur *delete*. Avec lui, on peut restituer au gestionnaire des allocations dynamiques un espace alloué avec l'opérateur *new*. La syntaxe utilisée ici est la suivante :

```
delete AdresseObtenueAvecNew
```

L'opérateur *delete* invoqué dans le destructeur ne doit être utilisé que sur des pointeurs créés par *new*[7]. Il existe sous deux formes, *delete* et *delete* [ ]. Nous allons les étudier dans le paragraphe suivant.

---

7 On pourrait en effet utiliser en C++ la fonction d'allocation usuelle en C, malloc. Mais, dans ce cas, il faut appeler, pour la libération, la fonction free.

2.4.3 - Les opérateurs *new* et *delete*

Examinons la séquence de la figure 2.11.

```
class Client {.....};
Client * PC1, * PC2;
PC1 = new Client ("chocolats");
PC2 = new Client [100];
```

Figure 2.11 : allocations dynamiques avec new

PC1 est un pointeur sur la classe Client. Il est initialisé avec le résultat de l'opération new Client ("chocolats") qui s'effectue en trois temps :

- réservation de l'espace mémoire à un objet de la classe Client,
- initialisation de cet espace avec l'appel du constructeur Client("chocolats"),
- renvoi de l'adresse de l'espace-mémoire ainsi initialisé.

La syntaxe utilisée pour l'opérateur new est ici la suivante :

new AppelDeConstructeur

La réservation de mémoire effectuée est alors faite pour un seul objet. En revanche l'initialisation de PC2 utilise la syntaxe que nous avions définie en 2.2.3 pour réserver de la place pour un tableau de 100 objets de la classe Client :

```
PC2 = new Client [100];
```

L'allocation dynamique s'effectue alors de la manière suivante :

- réservation de l'espace nécessaire à un tableau de 100 objets de la classe Client,
- dans chaque élément de ce tableau, construction d'un objet Client avec le constructeur par défaut,
- renvoi de l'adresse du début du tableau.

On notera que l'allocation dynamique d'un tableau d'objets d'une classe C, fait appel, pour chaque élément, à un constructeur sans argument de C : la syntaxe de new ne permet pas de spécifier d'arguments pour l'appel de ce constructeur. Quand on définit une classe dont les instances pourront être éléments de tableau, il faut que cette classe ait un constructeur sans arguments[8] (appelé également constructeur par défaut).

Pour restituer un espace-mémoire obtenu avec new, on utilise l'opérateur delete. Selon que l'espace alloué représente ou non un tableau, la syntaxe de delete est différente :

- **delete** AdresseObtenueAvecNew
  est utilisé si l'espace alloué contient une valeur unique (objet ou valeur

8 ou un constructeur dont tous les arguments ont une valeur par défaut.

d'un type prédéfini). Ainsi, pour libérer l'espace pointé par PC1 (Cf. figure 2.11), on écrira *delete PC1*.

- **delete []** *AdresseObtenueAvecNew*
  doit être utilisé si l'espace alloué représente un tableau. Dans ce cas, si ce tableau est un tableau d'objets d'une classe C, le destructeur de C est exécuté sur chaque élément, avant la libération de l'espace alloué. Ainsi, pour libérer l'espace pointé par PC2 (Cf. figure 2.11), on écrira *delete [ ] PC2*.

Si l'espace à libérer est un tableau de valeurs d'un type prédéfini, aucun destructeur n'est exécuté et les deux formes de *delete* sont donc équivalentes. Par analogie avec un tableau d'objets, on utilise cependant aussi la seconde forme. Le destructeur de la classe TableEntiers présenté au début de ce paragraphe doit donc s'écrire :

```
~TableEntiers::TableEntiers() { delete [] table; }
```

2.4.4 - Portée et durée de vie d'une variable

On appelle portée d'un nom la plage du programme à l'intérieur de laquelle l'objet ou la variable désigné par ce nom est utilisable. La portée s'étend de la déclaration de l'objet ou de la variable (premier moment dans le programme où le nom est cité) jusqu'à la fin du bloc à l'intérieur duquel est faite la déclaration :

- variable définie dans un bloc : portée se terminant à la fin du bloc ;
- variable définie dans une fonction : portée limitée à l'accolade finale de la fonction ;
- variable globale définie en dehors de toute fonction : portée se terminant avec la dernière instruction du fichier-programme. Une variable globale est donc accessible à toutes les fonctions du programme principal qui sont définies après elles.

Pour expliciter ces notions, examinons la figure 2.12. Elle présente un fichier-programme qui rassemble trois fonctions : f1, f2 et main.

La variable G est globale et donc accessible dans toutes les fonctions qui apparaissent après sa définition. Elle est utilisée par f1 et main. De même, la variable X est globale, elle est utilisée par f1 et f2.

La fonction f1 utilise une variable locale Y, dont la portée est limitée au corps de f1. Le corps de la fonction f2 mérite un examen attentif :

- On y définit une variable locale X qui porte le même nom que la variable globale X. Pour les distinguer dans la suite de l'exposé, notons X_l la variable locale et X_g la variable globale.
- La variable X_l est ensuite initialisée avec l'expression 2 * (:: X). On utilise ici l'opérateur de portée sous la forme d'un opérateur unaire, pour désigner explicitement la variable globale X_g. En effet, une variable locale de même nom N qu'une variable globale cache toujours la variable globale. Dans une pareille situation la notation ::N permet d'accéder à la variable globale. Si nous revenons à l'expression

2 * (:: X) notons que les parenthèses ne sont utilisées ici que pour faciliter la lecture. L'opérateur :: étant plus prioritaire que l'opérateur * de multiplication, on aurait pu écrire 2 * :: X.

- La fonction main définit une variable locale A et utilise les variables globales G et X.

```
#include <iostream.h>
int G = 100;
int X = 3;
int f1() { int Y = X + G; return Y; }
int f2() { int X = 2*(::X); return X; }
void main()
{
   int A = 2 * G + X;
   cout << "A="  << A  << "\n";
   cout << "f1():"  << f1() << "\n";
   cout << "f2():"  << f2() << "\n";
}
```

Figure 2.12 : variables globales, variables locales

Si on exécute le programme de la figure 2.12, on obtiendra l'affichage :

```
A=203
f1():103
f2():6
```

Toutes les variables que nous avons examinées dans ce paragraphe sont des variables nommées, qui sont définies à la compilation. Leur durée de « vie » va de leur définition ou déclaration jusqu'à la fin de leur portée. Une variable locale sera détruite lorsque l'exécution terminera le bloc dans lequel elle est définie. Une variable globale sera détruite à la terminaison du programme.

Il n'en va pas de même d'une variable obtenue par une allocation dynamique de mémoire. Intéressons-nous à cette dernière catégorie.

2.4.5 - Durée de vie d'une variable dynamique

Avec l'instruction :

```
{ int *Ptr = new int ; ...}
```

nous créons deux variables :

- La variable Ptr est une variable nommée dont la portée et la durée de vie se limitent au bloc dans lequel elle est définie.
- L'emplacement de type int obtenu par allocation dynamique est une seconde variable qui n'a pas de nom mais qui est désignée par le pointeur Ptr. On la désignera par la notation *Ptr.

La portée et la durée de vie de cette variable ne sont pas limitées au bloc dans lequel elle est créée. Par exemple, avec

```
int E;
int * PtrG;
{ // bloc A
  int * Ptr = new int;
  * Ptr = 2;
  PtrG = Ptr;
}
E = * PtrG
   //etc.
delete PtrG; // la variable pointée par PtrG est détruite
```

l'exécution du bloc A créera la variable locale Ptr, puis effectuera l'allocation d'une variable dynamique entière qui sera initialisée avec la valeur 2. L'adresse de cette variable sera affectée au pointeur PtrG avant la fin d'exécution du bloc. Après l'exécution du bloc A, la variable Ptr n'existe plus, mais la variable dynamique est toujours accessible via PtrG et on recopie sa valeur dans la variable E avec E = * PtrG. C'est l'instruction *delete* PtrG qui met fin à la « vie » de cette variable dynamique.

La portée d'une variable dynamique est liée au(x) pointeur(s) qui la désigne(nt). Elle est accessible chaque fois qu'on dispose d'au moins un pointeur permettant d'obtenir son adresse. Dans l'exemple précédent, si nous supprimons l'instruction PtrG = Ptr du bloc A, la variable dynamique n'est plus accessible après le bloc A. Elle n'en est pas pour autant détruite.

La durée de vie d'une variable dynamique va de son allocation (avec *new*) jusqu'à sa destruction (avec *delete*).

Les environnements de programmation qui font beaucoup appel à l'allocation dynamique doivent prendre en compte les situations où les variables dynamiques ne sont plus désignées par aucun pointeur mais ne sont pas détruites. Ces environnements mettent alors en jeu des mécanismes complexes de récupération de la mémoire que l'on appelle des ramasse-miettes (*garbage collector* en Anglais). Des environnements tels que Lisp et Smalltalk fournissent ce genre de mécanisme, car le programmeur ne manipule pas lui-même les pointeurs. C++ est un langage de bas niveau qui laisse cette opération à la responsabilité du programmeur.

2.5 - Fonctions : transmission de paramètres

Lorsqu'on passe des arguments à une fonction, une question se pose dans chaque langage de programmation : la fonction va-t-elle manipuler les arguments eux-mêmes ou leur copie ? Dans le premier cas (la fonction manipule directement les arguments), les modifications éventuelles que la fonction apporte aux arguments sont répercutées dans l'environnement d'appel. Dans le second cas (manipulation d'une copie), une place mémoire plus importante est nécessaire (puisqu'il y a duplication des arguments) au profit de la sécurité (les originaux ne sont pas affectés).

Dans tout programme, il faut pouvoir utiliser les deux modes. Quand un argument est un paramètre-donnée, on peut le transmettre par copie puisqu'il ne doit pas être modifié par l'exécution de l'appel. En revanche, si l'argument est un paramètre-résultat ou donnée/résultat, il faut que la fonction appelée travaille directement avec cet argument. Certains langages, comme Pascal, ont défini des conventions d'écriture précisant au compilateur si le passage des arguments a lieu par valeur (copie) ou par référence (originaux). D'autres, plus proches de la machine comme C, travaillent sur des copies et obligent le programmeur à une manipulation supplémentaire pour les passages par référence (utilisation de pointeurs).

Nous allons voir dans l'exemple suivant, traitant de l'échange de deux paramètres dans une fonction, que C++ permet d'utiliser les mécanismes du C, mais introduit une convention d'écriture permettant un passage des arguments par référence.

2.5.1 - Echanger les valeurs de deux paramètres dans une fonction

Lorsqu'on manipule des paramètres dans une fonction, il s'agit par défaut d'une copie des originaux qui ont été transmis à l'appel de la fonction. Examinons la fonction suivante :

```
void Echange1 (int E1, int E2)
{   // échange des valeurs de E1 et E2
    int aux = E1;
    E1 = E2;
    E2 = aux;
}
```

Cette fonction échange bien, quand elle est exécutée, les valeurs de ses paramètres E1 et E2. Mais, si on l'appelle avec la séquence :

```
int A = 11, B = 99;
Echange1 (A, B);
cout << "A=" << A << "\n B=" << B << "\n";
```

on constate que les valeurs de A et B n'ont pas été échangées.

Telle qu'elle est définie, Echange1 manipule deux paramètres formels, E1 et E2. Ces paramètres sont des variables locales de la fonction. Les valeurs des paramètres effectifs A et B (c'est à dire 11 et 99) sont copiées dans les variables locales E1 et E2. L'échange, dans la mesure où le code a été bien écrit, a lieu correctement : la variable locale E1 a pris la valeur 99 en fin de fonction, E2 a pris la valeur 11. Par contre, les paramètres effectifs A et B n'ont pas été modifiés.

En C, quand on veut qu'une fonction appelée modifie une variable V de l'environnement appelant, on transmet à la fonction, non pas la variable V elle-même, mais un pointeur P sur cette variable. La fonction travaillera avec une copie de P, mais la valeur de cette copie étant l'adresse de V, la fonction pourra accéder à V et donc la modifier. En C, le programmeur gère lui-même la transmission de référence. La figure 2.13 montre la fonction Echange2 qui permet un échange correct.

```
void Echange2(int * ptrE1, int * ptrE2)
{   // échange les valeurs des variables pointées par ptrE1 et ptrE2
    // fonction utilisable en C ou C++
    int aux = * ptrE1;
    *ptrE1 = *ptrE2;
    *ptrE2 = aux;
}

// exemple d'utilisation
int A = 11, B = 99;
Echange2(&A, &B);
```

Figure 2.13 : gérer des paramètres-résultats en C ou C++

Cette technique force le programmeur à gérer lui-même le mécanisme de la transmission de référence. Elle est utilisable en C++ mais, dans ce dernier langage, on peut aussi utiliser un mécanisme prédéfini en utilisant la spécification de référence. Pour cela, on définira la fonction Echange3 avec l'en-tête :

```
Echange3(int & E1, int & E2)
```

Cette fonction est représentée figure 2.14. Dans son corps, l'utilisation des paramètres se fait simplement en mentionnant E1 ou E2. Les paramètres formels E1 et E2 sont considérés dans ce cas comme synonymes des paramètres effectifs : toute modification de ces synonymes entraîne de facto la modification des paramètres eux-mêmes, puisqu'il s'agit des mêmes objets.

```
void Echange3 (int & E1, int & E2)
{   // échange les valeurs de E1 et E2
    // E1 et E2 sont des références aux arguments d'appel
    int aux = E1;
    E1 = E2;
    E2 = aux;
}
```

Figure 2.14 : transmission par référence en C++

2.5.2 - Modes de transmission de paramètres en C++

C++ accepte la syntaxe utilisée par le langage C. Il existe donc trois manières de passer des paramètres à une fonction :

1 – Comme en C, en travaillant sur une copie des paramètres : void Echange (int E1, int E2). Les paramètres formels de Echange sont les entiers E1 et E2. Les paramètres effectifs que l'on transmettra lors de l'appel de cette fonction ne seront pas modifiés par Echange.

2 – Comme en C, en travaillant sur une copie des adresses : void Echange (int * ptrE1, int * ptrE2). Les paramètres formels de Echange sont

maintenant des pointeurs sur entiers ptrE1 et ptrE2. Les paramètres effectifs transmis à l'appel de la fonction devront correspondre à des adresses d'entiers. Les paramètres effectifs ne seront pas modifiés, mais on peut – dans le corps de la fonction Echange – travailler sur les valeurs désignées par ces paramètres effectifs et les modifier.

3 – Exclusivement en C++, en travaillant sur un synonyme des paramètres : void Echange (int & E1, int & E2). Les paramètres formels de Echange sont les entiers E1 et E2 et sont déclarés comme référence aux paramètres effectifs qui pourront être transmis par la suite à l'appel de la fonction. Si Echange modifie E1 et E2, elle modifie de ce fait les paramètres effectifs.

2.5.3 - &, la spécification de référence

La spécification de référence, définie avec la notation & n'est pas exclusivement réservée au mécanisme de transmission de paramètres. On peut ainsi écrire la séquence suivante :

```
int E = 5;
int & RefE = E;     // RefE est un autre nom pour E
RefE ++;            // c'est E qui est augmenté de 1
```

dans laquelle la définition de RefE n'introduit pas une nouvelle variable mais spécifie un deuxième nom pour la variable E.

Une définition de référence suit la forme générale :

Untype & NomRef = NomDéjaDéfini

La spécification de référence peut aussi être employée pour une transmission de paramètre. Ainsi, avec la fonction void f (int & P) {.....} un appel tel que f(A) entraîne une exécution de f dans laquelle P est un autre nom pour A.

2.5.4 - Valeurs renvoyées par une fonction (ou un opérateur)

Un problème analogue se pose pour les valeurs renvoyées par une fonction : quand celle-ci se termine avec return X ; l'entité renvoyée est-elle X ou une copie de X ?

En C++, la valeur renvoyée par défaut est une copie du paramètre effectif. Mais une convention d'écriture autorise le renvoi d'une référence. La classe TableEntiers va nous permettre d'illustrer les deux manières de procéder. Revenons à la définition de cette classe, que nous complétons comme indiqué sur la figure 2.15.

Par rapport à la version de la figure 2.9, nous avons défini le destructeur étudié en 2.4.3 et nous avons ajouté deux nouvelles fonctions.

– La fonction ValeurCase permet d'obtenir la valeur d'un entier enregistré dans un objet TableEntiers. Elle reçoit en paramètre le rang de l'élément dans lequel figure l'entier désiré. Elle renvoie cet entier, s'il existe. Pour exister, cet entier doit en effet être situé dans un élément dont le rang va de 0 à nbElem -1. Cette vérification de validité est effectuée par appel d'une autre fonction membre, valideRang.

– La fonction valideRang joue le rôle d'un filtre. Elle reçoit une valeur entière en argument et elle renvoie la même valeur, si cette valeur est valide sinon elle bloque l'exécution, avec un code de retour égal à 2. Cette fonction est privée, car elle n'est pas destinée aux utilisateurs de la classe, mais réservée aux implémenteurs qui pourront l'utiliser dans l'écriture des fonctions membres.

```
class TableEntiers {
  private:
    ... // etc. Cf. figure 2.9
    int valideRang(int Position);
        // renvoie Position si elle indique le rang d'un entier de la table
 public:
    ... // etc. Cf. figure 2.9
    ~TableEntiers() { delete [] table; }
    int ValeurCase(int Rang);
        // renvoie l'entier à la position Rang, s'il existe
}; // fin class TableEntiers

int TableEntiers::valideRang(int Position)
{    // privée, renvoie Position si elle indique le rang d'un entier de la table
     if (Position < 0  ||  Position >= nbElem) {
        cerr << "TableEntiers : index incohérent\n"; exit(2);
     }
    return Position;
}

int TableEntiers::ValeurCase(int Rang)
{    // renvoie l'entier à la position Rang, s'il existe
     return table[valideRang(Rang)];
}
```

Figure 2.15 : classe TableEntiers, *accès à un élément*

Avec la définition de la classe ainsi complétée, on peut maintenant compiler et exécuter une séquence telle que

```
TableEntiers T2 (10, 3, 11, 22, 33); // 3 entiers au départ
cout << T2.ValeurCase(3);
```

qui affichera 33 à l'écran. Le programmeur pourrait aussi être tenté d'écrire la séquence suivante :

```
TableEntiers T(4,2,0,0); // deux entiers nuls au départ
T.ValeurCase(1) = 10;
```

La compilation bloque sur l'utilisation de la fonction ValeurCase en précisant qu'il faudrait avoir une *lvalue* à gauche de l'opérateur =.

Une *lvalue* (left value) est une expression qui peut figurer à gauche d'un opérateur d'affectation. En C++, ce peut être un nom de variable (par exemple X),

un emplacement désigné par un pointeur (par exemple *P) ou une référence à une variable (par exemple R avec int & R = X;). Or, en écrivant

```
T.ValeurCase(1) = 100;
```

nous plaçons à gauche de l'affectation le résultat de l'exécution de la fonction *ValeurCase*. Ce résultat provient de l'instruction :

```
return table[valideRang(Rang)];
```

On pourrait penser que ce que la fonction renvoie ainsi est bien un élément du tableau *table* et s'étonner alors de la réaction du compilateur. Cependant, quand une fonction f est définie par

```
int f (...) {....return X;}
```

la valeur qu'elle renvoie est toujours une copie de X, que X soit ou non une variable locale à f.

L'exécution de l'appel *T.ValeurCase*(1) se termine donc par la création d'un entier temporaire dans lequel la valeur de *T.table*[1] est recopiée. Bien évidemment, le compilateur ne peut accepter une affectation dans cette variable temporaire. Pour qu'une instruction telle que

```
T.ValeurCase(1) = 100;
```

soit acceptée, il faudrait que la fonction *ValeurCase* renvoie non pas une copie de l'élément de *table* mais cet élément lui-même. C'est possible en C++, à condition d'indiquer que la fonction ne renvoie pas une valeur de type int, mais une référence à un entier. Ainsi la fonction

```
int & f (....) {...return X;}
```

renverra-t-elle effectivement la variable X qui devra alors impérativement ne pas être locale à f.

On peut donc modifier la définition de la fonction *ValeurCase* pour qu'elle soit utilisable aussi bien pour consulter que pour obtenir un élément valide d'un objet *TableEntiers*. Cette nouvelle définition est présentée figure 2.16.

```
int & TableEntiers::Case(int Rang)
{    // renvoie la case de position Rang, si elle contient un entier
   // peut être utilisée aussi bien pour consulter que pour modifier une case
   return table[valideRang(Rang)];
}
```

Figure 2.16 : la fonction Case renvoie une référence à un emplacement de table

Par rapport à la fonction membre *ValeurCase*, nous avons apporté deux changements :

- Le premier, facultatif, relève de la terminologie. Puisque la fonction ne fournit pas seulement la valeur mais la case elle-même, nous l'appelons *Case* et non plus *ValeurCase*.
- Le second, obligatoire, modifie le type renvoyé qui n'est plus int mais int&. Maintenant, on peut exécuter une instruction telle que : *T.Case*(1) = 100;.

On peut se demander si on ne peut pas aussi simplifier l'écriture et écrire, pour la même opération, T[1] = 100. On utiliserait ainsi la notation usuelle qui fait intervenir l'opérateur d'indexation. Nous verrons au chapitre 3 que cela est possible, en définissant, pour la classe considérée, le fonctionnement de l'opérateur [int &] (Cf. 3.3.2).

2.6 - Et si nous reparlions des constructeurs ?

Nous avons abordé longuement le sujet des constructeurs. Et pourtant, tout n'a pas été dit. Il nous faut en particulier étudier les mécanismes de la copie et de la conversion de type.

2.6.1 - Constructeur et conversion de type

On effectue une conversion de type lorsqu'une valeur, comme 2, ne peut plus être traitée comme un entier (ce qui apparaît naturel à la lecture) mais doit être représentée sous une autre forme (comme un réel, un entier long, etc.). Par exemple, avec les définitions

```
void f(float R) {....}
int X;
```

un appel tel que f(X) provoquera de la part du compilateur la conversion de la valeur entière de X en un réel qui deviendra la valeur de l'argument R.

La conversion de type est couramment pratiquée par C++ dans les opérations arithmétiques. C'est cette conversion qui permet d'effectuer indifféremment des opérations portant à la fois sur des entiers longs, courts ou normaux ou des réels.

La notion de constructeur est puissante : elle introduit indirectement un outil de conversion entre types. S'il existe un constructeur de la classe TableEntiers qui peut être appelé avec un seul argument de type TTT, alors ce constructeur définit la règle de conversion entre le type TTT et le type TableEntiers.

Par exemple avec le constructeur TableEntiers (int Dim = 100), on peut écrire la séquence suivante :

```
TableEntiers Tab(100);
... // on utilise Tab
Tab = 50;
```

L'exécution de l'instruction Tab = 50 sera interprétée par le compilateur comme une demande de conversion du type int vers le type TableEntiers. Le compilateur acceptera cette conversion et générera le code des opérations suivantes :

- construction d'un objet TableEntiers temporaire avec l'appel TableEntiers (50),
- affectation de cet objet temporaire à l'objet Tab. Le mécanisme d'affectation par défaut[9] est utilisé : les valeurs des données membres

9 Pour une étude détaillée du mécanisme d'affectation, Cf. 3.3.3

de l'objet temporaire sont recopiées dans les membres correspondants de *Tab*,

- l'objet temporaire est détruit.

En l'état actuel de la définition de notre classe, cette conversion de type introduira une incohérence dans la représentation de l'objet *Tab*. Comprenons-le en examinant la figure 2.17 qui représente l'objet *Tab* avant et après l'opération.

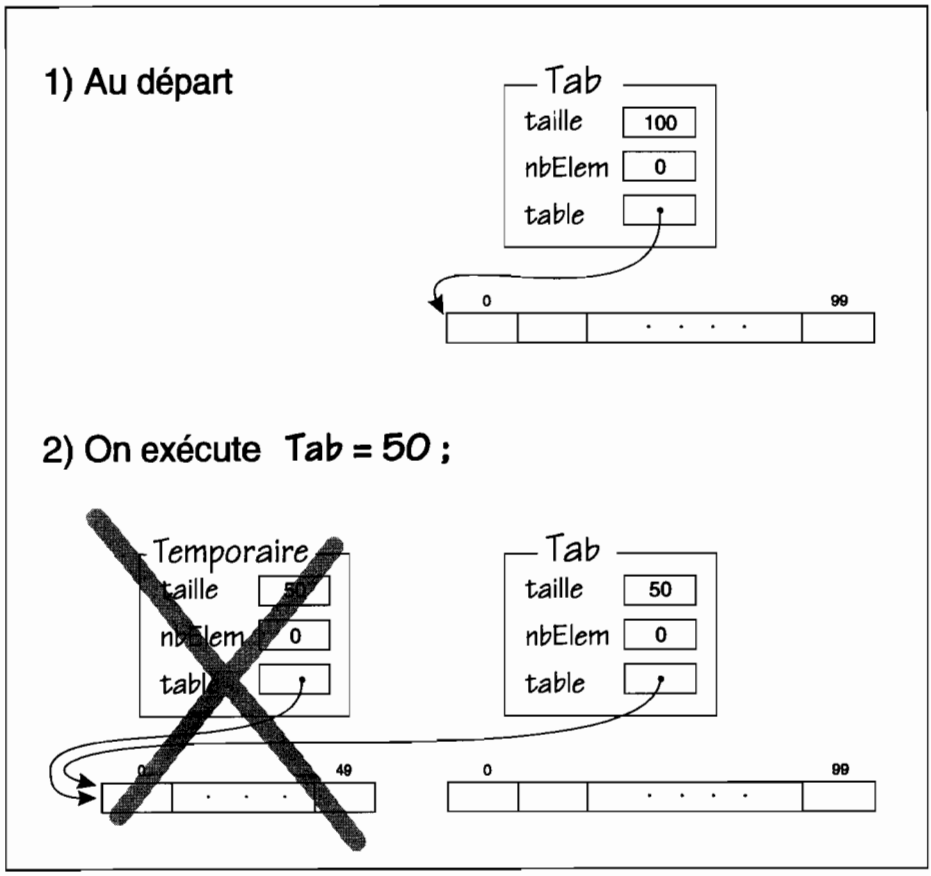


Figure 2.17 : une conversion malheureuse

A l'issue de l'affectation, l'objet temporaire est détruit, ce que la figure matérialise par la croix superposée à cet objet. Cette destruction se fait par appel du destructeur que nous avons défini en 2.4.3. Elle libère également l'espace alloué et pointé par le membre *table* de l'objet temporaire. Malheureusement, à cause de la copie membre à membre effectuée par l'affectation, cet espace est aussi désigné par le membre *table* de *Tab*. L'objet *Tab* n'a donc plus de tableau pour enregistrer ses entiers !

Un autre effet négatif de l'affectation ainsi effectuée est la perte de l'espace alloué à Tab à la construction de cet objet. Cet espace n'est *plus* désigné par la donnée membre table de Tab mais il n'a pas été libéré. Un tel incident n'a pas, en général, de conséquence immédiate aussi importante que celui décrit au paragraphe précédent, mais sa répétition qui épuise l'espace alloué peut avoir des conséquences tout aussi néfastes.

Que faut-il retenir de ce que nous venons d'étudier? D'abord que tout constructeur dont la signature est :

```
CCC :: CCC (TTT);
```

définit une règle de conversion du type TTT vers le type CCC et que cette conversion entraîne toujours la construction d'un objet temporaire.

Un second point important est que le mécanisme fourni par défaut par C++ pour l'affectation est dangereux dès que les objets auxquels on l'applique font appel à l'allocation dynamique. Pour remédier à ce problème pour notre classe TableEntiers, il faudrait redéfinir l'opérateur d'affectation pour cette classe. Nous ne le ferons pas ici : la redéfinition d'opérateur sera étudiée au chapitre 3. Signalons cependant que pour empêcher l'utilisateur de la classe TableEntiers d'exécuter des instructions telles que Tab = 50; il suffira de redéfinir l'opérateur d'affectation comme une fonction membre privée :

- l'utilisateur ne pourra plus programmer d'affectation dont un objet TableEntiers serait l'opérande de gauche,
- le concepteur de la classe pourra, s'il en a besoin, utiliser une telle affectation dans les fonctions membres de cette même classe.

2.6.2 - Constructeur-copie

Abordons maintenant le cas d'un constructeur un peu particulier, le constructeur par copie. Ce constructeur utilise un objet existant pour en construire une copie. On peut imaginer bien des cas où disposer d'un tel constructeur peut être intéressant. Supposons, par exemple, qu'un utilisateur de la classe TableEntiers définisse la fonction

```
int Somme (TableEntiers T) {....}
```

qui calcule et renvoie la somme des entiers contenus dans son argument T. Pour traiter un appel tel que :

```
TableEntiers Tab;
...// ici, on enregistre des entiers dans Tab
int X = Somme (Tab);
```

le compilateur réalisera la transmission de paramètre en créant l'objet T (paramètre formel de Somme) par copie de Tab. Pour cela, un constructeur-copie sera utilisé.

En fait, comme pour le destructeur ou le constructeur, C++ fournit un constructeur-copie par défaut. Les opérations effectuées par ce constructeur sont une simple copie membre à membre des données privées et publiques.

Lorsqu'on met au point une classe, il est indispensable de regarder si le constructeur par copie proposé par défaut est suffisant. L'exemple suivant illustre la nécessité, pour la classe TableEntiers, de redéfinir le constructeur-copie.

```
TableEntiers T(4,4,0,0,0,0);
TableEntiers Tbis(T);   // construction de Tbis par copie de T
cout <<"Tbis après sa construction :\n"; Tbis.AfficheToi();
T.Case(0) = T.Case(1) = 111;
cout <<"Tbis après modification de T :\n"; Tbis.AfficheToi()
```

Dans cette séquence, on instancie un objet T de capacité 4, contenant quatre valeurs nulles. Puis on construit l'objet Tbis par copie de T. On modifie ensuite les deux premières valeurs rangées dans T qui deviennent toutes deux égales à 111. On affiche enfin les valeurs du tableau Tbis. Si on exécute cette séquence, on obtient l'affichage.

```
Tbis après sa construction :
TableEntiers de capacité 4, contient 4 entiers :
   0 : 0
   1 : 0
   2 : 0
   3 : 0
Tbis après modification de T :
TableEntiers de capacité 4, contient 4 entiers :
   0 : 111
   1 : 111
   2 : 0
   3 : 0
```

On constate que l'objet Tbis a été modifié par l'exécution de l'instruction T.*Case* (*0*) = T.*Case* (1) = 111;. En effet, le constructeur-copie s'appuie sur le même mécanisme que l'affectation par défaut et nous nous retrouvons dans une situation comparable à celle du paragraphe précédent. Après la copie, les membres *table* de T et Tbis désignent le même emplacement et toute modification d'un entier contenu dans un objet est aussi effectuée sur l'autre objet !

D'une manière générale, dès qu'une classe définit une ou plusieurs données membres de type pointeur, il faut se demander si on ne doit pas redéfinir le constructeur-copie (et aussi l'affectation, Cf. chapitre 3). La figure 2.18 propose, pour la classe TableEntiers, un constructeur-copie correct.

Sur cette figure on remarquera que :

- Le passage de l'argument du constructeur doit se faire par référence, pour éviter une boucle infinie (le passage par valeur nécessiterait la création d'une copie de *Source*, qui s'effectuerait par un nouvel appel au constructeur-copie, etc.).
- Pour l'itération de recopie des entiers présents dans l'argument *Source*, nous aurions pu utiliser la fonction membre *Case* et écrire :
  Case (nbElem) = *Source.Case* (nbElem) ;
  Nous ne l'avons pas fait parce que l'écriture du constructeur-copie relève de la responsabilité du concepteur de la classe. Celui-ci peut « prendre des risques » (en n'utilisant pas la fonction membre *Case*) pour améliorer la performance des composants logiciels de la classe. Bien sûr, il doit garantir que le risque est nul et ici, l'itération ne

copiant que les valeurs des éléments de rang 0 à Source.nbElem, nous sommes sûrs que la copie sera correcte.

```
TableEntiers::TableEntiers (TableEntiers & Source)
{// constructeur-copie
   alloueTable(Source.taille);
   for (nbElem = 0; nbElem < Source.nbElem; nbElem++)
       table[nbElem] = Source.table[nbElem];
}
```

Figure 2.18 : un constructeur-copie pour la classe TableEntiers

2.6.3 - Importance du constructeur par défaut

Le constructeur par défaut est celui qui est fourni par C++ lorsque le programmeur n'en définit aucun. C'est un constructeur sans arguments. Nous avons déjà étudié son rôle en 2.2.1 et 2.2.4.

Ce constructeur est supprimé dès que le programmeur en définit un dans la classe. Revenons ici sur la fréquente nécessité de le rétablir.

L'exemple suivant va nous montrer l'utilisation implicite qui peut être faite du constructeur par défaut de TableEntiers. Dans le cadre d'un cours sur les matériels informatiques, nous voulons simuler le fonctionnement de la mémoire vive. Nous limiterons notre objet mémoire à trois segments (données, code et pile) et nous définirons la classe Mémoire comme suit.

```
class Memoire {
 private:
   TableEntiers donnees(32768);
   TableEntiers code(32768);
   TableEntiers pile(32768);
 // etc.
};
```

Cette définition n'est pas acceptée par le compilateur. En effet, la définition des données membres *données*, *code* et *pile* se fait par appel d'un constructeur qui effectue une initialisation. Or la définition d'une classe ne peut que spécifier le "moule" qui servira à instancier les objets et ce moule ne peut enregistrer aucune valeur initiale. Si une donnée membre est un objet, elle ne peut être décrite que par un appel au constructeur par défaut. On doit donc décrire notre classe Mémoire par :

```
class Memoire {
 private:
   TableEntiers donnees;
   TableEntiers code;
   TableEntiers pile;
   ... // etc.
};
```

Dans ces conditions, pour construire un objet de la classe Mémoire, C++ fera appel au constructeur par défaut de la classe TableEntiers. Si nous nous reportons à la figure 2.9, ce constructeur est obtenu avec le constructeur :

```
TableEntiers (Dim = 100);
```

appelé sans argument. On voit donc que les données membres *données*, *code* et *pile* seront instanciées comme des objets TableEntiers de capacité 100, ce qui ne correspond pas à notre souhait. En effet, nous voulions que chacun de ces objets ait une capacité de 32768. Pour remédier à ce problème, nous avons trois solutions :

- modifier la valeur par défaut du constructeur TableEntiers utilisé pour la fixer à 32768. Cette solution a l'inconvénient de rendre la classe TableEntiers dépendante d'une de ses utilisations et elle ne correspond pas à l'esprit de l'approche objets. Le concepteur de la classe TableEntiers ne doit pas la spécialiser pour une utilisation particulière. Il doit lui conserver le spectre d'utilisations le plus large possible.

- définir un constructeur par défaut pour la classe Mémoire avec :

```
Memoire()
{ données = TableEntiers (32768);
  code = TableEntiers (32768);
  pile = TableEntiers (32768);
}
```

Cette solution a l'inconvénient d'effectuer deux instanciations successives d'un objet TableEntiers pour chaque donnée membre de Mémoire. Par exemple, *données* sera instanciée d'abord avec le constructeur par défaut de TableEntiers puis, dans le corps du constructeur de Mémoire, on créera un second objet TableEntiers que l'on recopiera dans *données*. De plus, comme nous l'avons vu en 2.6.1, il faut gérer correctement l'affectation pour TableEntiers.

- la troisième solution est la bonne : il faut spécifier, pour le constructeur de Mémoire, une liste d'initialisation. Nous étudierons cette solution au chapitre 4 (Cf. 4.1.2).

2.7 - Le modificateur *const*

Le modificateur const peut être utilisé dans tous les cas où le programmeur veut interdire un changement de valeur :

- définir une constante dans un programme ;
- prévenir toute modification de l'argument d'une fonction ;
- définir un pointeur sur une valeur constante ;
- définir un pointeur constant sur une variable ;
- définir un pointeur constant sur une valeur constante ;
- garantir l'intégrité d'un membre pointé ;
- définir un objet constant.

2.7.1 - Déclarer une constante dans un programme

La déclaration suivante :

```
const float Pi = 3.1415;
```

définit et initialise un réel à une valeur non modifiable par la suite. Il est nécessaire d'initialiser les constantes lors de leur déclaration : le compilateur n'admet pas de définition telle que :

```
const int Pi;
Pi = 3.1415;
```

Cette écriture est dans son principe contradictoire avec la notion de constante (puisque l'on pourrait toucher à la valeur de Pi après sa définition).

Le modificateur *const* employé ici permet d'indiquer que Pi est du type *const* int et, pour le compilateur, ce type est bien distinct de int. D'une manière générale, si TTT est un type (prédéfini ou classe), le compilateur acceptera une affectation de *const* TTT vers TTT, mais refusera l'opération inverse.

2.7.2 - Partager une constante entre plusieurs fichiers sources

Le modificateur *const* inhibe l'exportation implicite d'une variable globale. Par exemple si nous définissons dans le fichier M1.*c* la variable globale :

```
int X = 0; // déclaration de M1.c
```

un autre fichier M2.*c* pourra utiliser cette variable X en la déclarant externe :

```
extern int X; // déclaration de M2.c
```

Cependant, si la définition de X dans M1.*c* spécifie

```
const int X = 0; // définition dans M1.c
```

le module M2 n'aura plus accès à X (même s'il spécifie la référence externe). Dans ce cas l'éditeur de liens qui traitera les modules objets résultant de la compilation de M1.*c* et M2.*c* signalera que X de M2.*c* est une référence externe non résolue.

Si l'on veut définir la constante dans M1.*c* et l'utiliser aussi dans M2.*c*, il faut définir dans M1 :

```
extern const int X = 0; // définition dans M1.c
```

et la déclarer dans M2 :

```
extern const int X; // déclaration dans M2
```

Le compilateur reconnaîtra que X n'est pas dans M2 car sa déclaration comporte la spécification *extern* avec le modificateur *const* sans initialisation. Dans ce cas, on peut employer *const* sans initialisation.

2.7.3 - Transmettre une constante en paramètre de fonction

On peut transmettre une constante comme paramètre effectif d'une fonction. Cela ne pose pas de problème au compilateur qui transmet une copie du paramètre effectif à l'appel de la fonction.

Dans le cas d'un paramètre passé par référence, le compilateur ne peut laisser la fonction travailler directement sur la constante (qui pourrait être modifiée) : il

utilise, malgré l'opérateur de référence, une copie de la constante. Un avertissement signale à la compilation le non respect du passage par référence.

```
int PlusUn(int & A) { return ++A }
const int X = 10;
PlusUn(X);
// le compilateur avertit le programmeur qu'il passe un temporaire à la place de X
```

De la même manière, le compilateur utilise un objet temporaire pour une référence initialisée par une constante :

```
const int Y = 10;
int & A = Y
// le compilateur crée un entier temporaire pour initialiser la référence A
```

On peut par contre signaler au compilateur que la référence est elle-même constante : dans ce cas, il n'y a pas génération d'objet temporaire :

```
const int Y = 10;
const int & A = Y // passe la compilation sans avertissement
```

2.7.4 - Prévenir toute modification d'un paramètre dans une fonction

Pour prévenir toute modification des paramètres transmis dans une fonction, on peut utiliser le spécificateur `const` à la définition de cette fonction :

```
int PlusUn(const int & A) { return ++A; }
// erreur de compilation - impossible de modifier un objet constant
```

2.7.5 - Pointeurs et constantes

On peut définir un pointeur sur une valeur constante. Il est nécessaire d'initialiser l'objet pointé dès la déclaration :

```
const char * Mot = "Bonjour";
    // la chaîne "Bonjour" n'est pas modifiable
char * Ptr;
Ptr = Mot;  // affectation refusée à la compilation : on pourrait sinon modifier
            // la chaîne pointée en utilisant Ptr
Mot = "Ay revoir";  // accepté car le pointeur Mot lui-même n'est pas constant
Mot[1] = 'u'  // affectation refusée car la chaîne pointée par Mot est définie
              // comme une constante
```

Un pointeur variable sur une valeur constante de type `TTT` se définit donc avec la syntaxe :

```
const TTT * NomDuPointeur;
```

On peut définir un pointeur constant sur un objet variable :

```
char * const Mot = "Ay revoir";
Mot[1] = 'u';   // accepté, car la chaîne pointée par mot n'est pas constante
Mot = "Bonjour";  // refusé : on ne peut modifier la valeur de
                  // Mot, pointeur constant
```

Un pointeur constant sur une valeur variable de type TTT se définit donc avec la syntaxe :

```
TTT * const NomDuPointeur;
```

On peut, enfin, définir un pointeur constant sur un objet constant :

```
const char * const Mot = "Bonjour";
Mot = "Au revoir"; // refusé car le pointeur Mot est constant
Mot[0] = 'a';   // refusé car la chaîne pointée par Mot est constante
```

3 Créer sa première classe fonctionnelle

Ce chapitre utilise les connaissances acquises dans les deux chapitres précédents pour implémenter une classe fonctionnelle. Après s'être familiarisé avec les concepts de base de la programmation objet en C++, le lecteur va approfondir ses connaissances à travers un exemple complet et concret de conception de classes. Il apprendra à concevoir, créer et utiliser sa propre classe tout en découvrant des mécanismes C++ complémentaires. En particulier, il abordera les notions de fonctions amies, de classes amies, de variables et méthodes de classe ou encore d'itérateurs.

3.1 - Une classe pour gérer les chaînes de caractères

Une classe, avec ses données et fonctions membres fournit la représentation concrète d'un concept. Ainsi, elle constitue un nouveau type, qui viendra ensuite compléter les types prédéfinis.

Dans le langage C++, il manque des classes de base pour représenter les chaînes, les matrices, les tableaux d'objets, etc. Si nous les implémentons, nous pourrons ensuite les utiliser comme des types prédéfinis. Pour combler une de ces lacunes, nous allons étudier la mise en oeuvre d'une classe capable de gérer les chaînes de caractères.

Avec le langage C, une chaîne de caractères est représentée sous la forme d'un tableau de caractères, dont le dernier élément est supposé contenir le caractère nul '\0', qui indique la fin de chaîne. Ainsi, la longueur réelle de la chaîne "ABCDE" est de six caractères, soit un caractère de plus que le nombre de caractères entre guillemets. Cette implémentation est la source de nombreux problèmes pour le programmeur. Ainsi, la déclaration et l'initialisation de la chaîne :

```
const int dim = 5;
char Chaine[dim] = "ABCDE";
```

sont correctes du point de vue du compilateur. Dans ce cas, il considère que cette écriture est équivalente à :

```
char Chaine[dim] = {'A','B','C','D','E'};
```

Mais le caractère nul est absent de la fin de cette chaîne, ce qui engendre des incohérences lors de l'exécution de traitements élémentaires qui présupposent sa présence. Par exemple, lors de l'appel de la fonction standard strcpy de copie de chaîne pour la chaîne définie plus haut :

```
char * ChaineDest;
ChaineDest = new char[dim];
strcpy(ChaineDest, Chaine);
```

les caractères sont copiés dans *ChaineDest* à partir de l'adresse de *Chaine* jusqu'au premier caractère nul rencontré. Après le caractère '*E*', il reste au moins un caractère à traiter, il sera copié dans un octet-mémoire qui ne fait pas partie de l'espace-mémoire alloué à *ChaineDest*. Des données en mémoire peuvent alors être altérées et *ChaineDest* ne contient pas les informations escomptées. Pour définir correctement la chaîne de caractères précédente, il suffit d'écrire :

```
char Chaine[] = "ABCDE";
```

Le compilateur allouera alors en mémoire un espace suffisant pour stocker la chaîne avec son caractère nul.

Une autre imperfection peut entraîner des résultats fantaisistes lors des manipulations de chaînes. En effet, pour accéder à un élément, nous pouvons utiliser l'opérateur d'indexation qui ne contrôle pas la valeur de l'index et écrire par exemple :

```
char C = Chaine[dim*4];
```

Chaine ayant été définie précédemment comme un tableau de six caractères, et *dim* valant 5, le caractère retourné n'appartient pas à la chaîne.

Pour remédier à ces problèmes, nous allons élaborer la classe String, qui garantira la présence systématique du caractère nul de fin de chaîne et contrôlera les accès à la mémoire de la machine. Ainsi, elle fiabilisera les manipulations des chaînes de caractères et évitera les débordements de zone mémoire.

Cette classe deviendra une boîte noire pour le programmeur qui ne la manipulera qu'à travers un certain nombre de messages que nous aurons à implémenter. L'introduction de ce nouveau type permettra de produire un code plus robuste et plus concis que celui obtenu avec une représentation classique des chaînes de caractères, dont la maintenance sera facilitée.

De plus, la programmation par objets permet de tester une application de façon progressive en vérifiant le fonctionnement de chacune des classes au fur à mesure de leur élaboration. Nous veillerons donc à ne pas négliger la recherche d'éventuels dysfonctionnements lors de la conception de notre objet élémentaire. Une fois ce nouveau type créé, il nous fournira une aide importante dans nos développements futurs en prenant en charge le contrôle des erreurs de manipulation des objets de type String.

3.2 - Première implémentation de la classe *String*

Avant de commencer à implémenter notre classe, il nous faut réfléchir à la description d'une chaîne de caractères, c'est à dire aux éléments essentiels qui nous permettront de la caractériser.

3.2.1 - Les données membres

Pour décrire, en C++, l'information contenue dans la chaîne de caractères, nous utiliserons un pointeur sur caractère, `chaine`. Avec cette représentation nous allouerons dynamiquement une zone mémoire pour la chaîne, plutôt que d'utiliser un tableau de caractères qui nous obligerait à donner une taille maximum arbitraire à la chaîne. Ce choix d'implémentation nous permet d'optimiser la mémoire utilisée en fonction de la taille réelle de la chaîne.

Afin de pouvoir manipuler les chaînes de caractères tout en limitant les allocations et les libérations de mémoire, il est intéressant de connaître la taille maximale de la chaîne pour chacune des instances de la classe `String`. Celle-ci nous permet de savoir si une chaîne peut-être copiée dans l'espace alloué à une autre chaîne. La donnée membre `tailleMem` indiquera donc le nombre d'octets alloués.

Il est également intéressant de maintenir en permanence, pour chaque objet, le nombre de caractères effectifs de la chaîne. Nous utiliserons, pour cela, la donnée membre `nbCarac`. On pourra ainsi accélérer les traitements en évitant un appel systématique de la fonction standard `strlen` pour connaître la longueur de la chaîne. Mais pour chaque chaîne de caractères, nous aurons un espace-mémoire occupé plus important. La figure 3.1 illustre la représentation choisie.

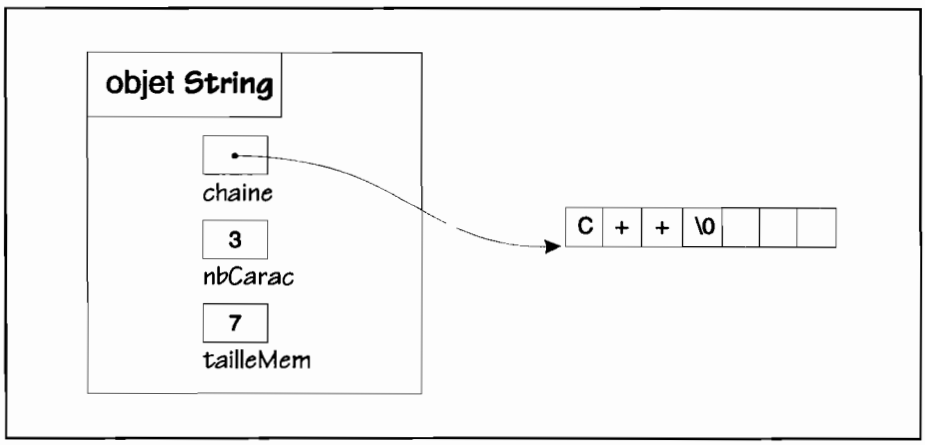


Figure 3.1 : trois données membres pour un objet `String`

Nous déclarerons privées les données membres de cette classe afin que seules les fonctions membres puissent y avoir accès. En empêchant leurs manipulations

directes, en particulier pour l'adresse de la chaîne, nous garantirons le fonctionnement correct des objets et le maintien de l'intégrité de leurs membres.

3.2.2 - Définition de la classe *String*

Pour créer une classe élémentaire fonctionnelle, il nous faut mettre en oeuvre des constructeurs et un destructeur. Nous pouvons envisager quatre types de constructeurs (Cf. figure 3.2) :

- à partir d'une chaîne de caractères du type char *,
- à partir d'une chaîne de caractères du type String,
- à partir d'un caractère,
- à partir de la longueur de la chaîne,

```
class String {
   private:
      char * chaine;
      int tailleMem;
      int nbCarac;
      char * alloue (int Longueur);
   public:
      String(const char * Chn = "");
      String(const String & Source);
      String(char Carac);
      String(int LgrChn);
      ~String();
};
```

Figure 3.2 : première définition de la classe String

Tous ces constructeurs faisant appel à l'allocation dynamique, nous définissons une fonction membre réservée au concepteur de la classe, c'est à dire privée, pour gérer les allocations dynamiques. Cette fonction, alloue(int Longueur), prend en paramètre le nombre de caractères qui composent la chaîne.

```
char * String::alloue(int Longueur)
{  // fonction privée d'allocation, utilisée par les constructeurs
   if (Longueur < 0) {
      cerr << "Allocation négative impossible\n"; exit(1)
   }
   tailleMem = Longueur+1;
                  // +1, pour enregistrer le \0 de fin de chaîne
   char * P = new char[tailleMem];
   if (! P) {
      cerr << "Echec de l'allocation\n"; exit(2);
   }
   return P;
}
```

Cette fonction arrête l'exécution avec une indication d'erreur si Longueur est négative. Sinon, elle affecte la valeur adéquate à tailleMem. Ensuite, elle alloue un espace mémoire de tailleMem caractères avec l'opérateur *new*. Si l'allocation réussit, l'adresse de l'espace alloué est retournée (elle sera valide jusqu'à la destruction explicite de la zone pointée par appel de l'opérateur *delete*). Dans le cas contraire, le traitement s'arrête avec une indication d'erreur.

Nous pouvons maintenant définir un premier constructeur qui, à partir de l'adresse d'une chaîne de caractères, construit un objet de type String :

```
String::String(const char * Chn)
{
   nbCarac = strlen(Chn);
   chaine = alloue(nbCarac);
   strcpy(chaine, Chn);
}
```

Après un appel à la fonction d'allocation, avec comme paramètre la longueur de la chaîne, Chn est copié avec la fonction standard strcpy dans le membre *chaine*. Lors de l'appel de cette fonction, toute la chaîne est copiée, y compris son caractère terminal, ce qui est conforme à notre cahier des charges initial.

Comme nous l'avons vu au chapitre 2, un constructeur par défaut (sans argument) est souvent nécessaire dans une classe. Nous l'implémentons en donnant une valeur par défaut au paramètre du constructeur précédent lors de sa déclaration dans la classe String :

```
String(const char * Chn = "");
```

La présence du membre *chaine* alloué dynamiquement nous oblige à définir un destructeur :

```
String::~String() { delete [] chaine ;}
```

Si un objet String peut être créé à partir d'une chaîne de caractères, il peut également être créé vide avec une capacité donnée :

```
String::String(int LgrChn)
{
   nbCarac = 0;
   chaine = alloue(LgrChn);
   chaine[0] = '\0';
}
```

ou à partir d'un autre objet String, par appel au constructeur copie :

```
String::String(const String & Source)
{
   nbCarac = Source.nbCarac;
   chaine = alloue(Source.nbCarac);
   strcpy(chaine, Source.chaine);
}
```

et enfin à partir d'un caractère :

```
String::String (char Carac)
{
   nbCarac = 1;
   chaine = alloue(nbCarac);
   *chaine = Carac;
   *(chaine+1) = '\0';
}
```

Nous pouvons maintenant instancier des objets de la classe String :

```
String S0;
String S1(15);
const String S2("abc");
char Chn[] = "ijklm";
String S3(Chn);
String S4(S3);
```

où :

- S0 est une chaîne vide,
- S1 est une chaîne pouvant contenir jusqu'à 15 caractères,
- S2 est une chaîne constante initialisée à "*abc*",
- S3 vaut "*ijklm*" et
- S4 est une copie de S3.

3.2.3 - Définition d'accesseurs pour la classe *String*

Quand une classe définit des données membres privées, le concepteur de la classe doit préciser les règles d'accès à ces données membres. Pour cela il peut définir des fonctions membres publiques que l'on appelle des accesseurs. Il y a deux catégories d'accesseurs :

- des accesseurs en consultation qui renvoient la valeur d'une donnée membre privée,
- des accesseurs en modification, qui valident les modifications effectuées sur les données membres privées d'un objet.

Pour la classe String, nous implémenterons des accesseurs en consultation, Chaine et NbCarac, qui permettent de consulter les données membres de même nom, ainsi que la fonction membre Capacite qui retourne le membre tailleMem décrémenté de un (c'est à dire le nombre maximum de caractères du membre chaine de l'objet). Ces fonctions sont définies par :

```
class String {
 private:    // ... etc. Cf. figure 3.2
 public:     // ... etc. Cf. figure 3.2
   const char * Chaine() { return chaine; }
   int NbCarac() { return nbCarac; }
   int Capacite() { return tailleMem-1; }
};
```

Ces fonctions sont en ligne afin de ne pas pénaliser les traitements lors de l'accès aux données membres.

On peut se demander si l'accesseur Chaine n'est pas dangereux. Ne peut-il pas permettre à un programmeur de modifier directement la chaîne pointée ? Pour prévenir ce genre d'incident, nous avons spécifié un type renvoyé const char * pour cette fonction membre. Le pointeur renvoyé est ainsi un pointeur sur une chaîne constante, que le programmeur ne peut pas modifier. En particulier, si S1 et Source sont deux objets String, il sera impossible d'écrire :

```
strcpy(S1.Chaine(), Source.Chaine());
```

En effet, comme le prototype de strcpy est :

```
char * strcpy(char *, const char *);
```

et qu'il est impossible de transtyper une variable constante d'un type quelconque vers une variable du même type non constante, le compilateur refusera la conversion du type const char * renvoyé par S1.Chaine() en char *. On remarquera cependant que l'accesseur Chaine ne nous protège pas contre un programmeur décidé à n'en faire qu'à sa tête et qui pourrait écrire

```
char * P = (char *) S1.Chaine();
strcpy (P, Source.Chaine());
```

Le modificateur const peut aussi être utilisé pour définir des objets constants. Si par exemple, nous écrivons la séquence :

```
const String S1("abc");
cout << S1.NbCarac();
```

le compilateur nous signale qu'une fonction non constante est appelée pour un objet constant. La chaîne S1 a été définie comme une chaîne constante. On tente de lui appliquer une fonction, ici un accesseur, qui ne garantit pas que l'objet récepteur du message ne soit pas modifié. En C++, une fonction membre constante est une fonction autorisée à consulter les données membres de l'objet sur lequel elle est appelée, mais qui ne peut le modifier. Pour spécifier qu'une fonction membre est constante, il suffit d'ajouter le mot-clé const derrière le nom de la fonction, juste après la liste des arguments. Ainsi,

```
void Maclasse::MaFonction() const {...};
```

définit la fonction MaFonction comme fonction membre constante ce qui empêche de modifier la valeur de l'objet. Il est important de signaler que, si une fonction non constante ne peut être appelée sur un objet constant, une fonction constante peut être appliquée à un objet non constant. Ceci évite de définir des fonctions membres constantes et non constantes, spécifiant les mêmes traitements, dans le cas où les fonctions membres ne modifient pas l'objet récepteur du message.

A ce stade, la définition de la classe String de la figure 3.2 a été complétée par les éléments que nous venons de décrire et nous la remplaçons par celle de la figure 3.3.

```
class String {
   private:
      char * chaine;
      int tailleMem;
      int nbCarac;
      char * alloue (int Longueur);
   public:
      // Constructeurs
      String(const char * Chn = "");
      String(const String & Chn);
      String(char Carac);
      String(int tailleChn);
      // Destructeur
      ~String();
      // Accesseurs
      const char * Chaine() const { return chaine; }
      int NbCarac() const { return nbCarac; }
      int Capacite() const { return tailleMem-1; }
};
```

Figure 3.3 : nouvelle définition de la classe String

3.3 - Surcharge des opérateurs

En surchargeant un opérateur, nous le redéfinissons et lui conférons un sens en fonction de la classe à laquelle il s'applique et nous fournissons une syntaxe plus intuitive pour la manipulation des objets. Ainsi, à la place d'un appel à une fonction membre ayant un comportement proche de celui de la fonction standard strcpy, il est plus simple d'écrire :

```
ChaineDest = ChaineSource;
```

où ChaineDest et ChaineSource sont de type String. De même, la surcharge de l'opérateur + permettrait de remplacer l'appel d'une fonction membre de concaténation équivalente à strcat par une notation plus concise et plus parlante.

Toutefois, lors de la surcharge des opérateurs, il faut garder présent à l'esprit que les règles de priorité et la syntaxe des opérateurs sont immuables. Ainsi, un opérateur unaire surchargé restera un opérateur unaire. Tous les opérateurs, sauf :

```
.   .*    ::    ?:    sizeof
```

peuvent être surchargés, ces derniers opérateurs ayant une signification par défaut pour les objets instanciés d'une classe. Il est également impossible de créer de nouveaux opérateurs.

A chaque opérateur, il est associé une fonction opérateur qui s'écrit sous la forme du mot-clé operator suivi de l'identification de l'opérateur, puis des paramètres de cette fonction.

Ainsi, un opérateur unaire noté <U>, qui s'applique à une classe CCC, peut-être déclaré comme une fonction non membre de la classe CCC :

```
TypeRenvoyé operator <U> (CCC ObjetCourant);
```

Cet opérateur prend comme unique paramètre ObjetCourant de type CCC. Nous pouvons également déclarer la fonction associée à l'opérateur <U> comme membre de la classe CCC :

```
class CCC {
    // etc.
    TypeRenvoyé operator <U> ();
    // etc.
};
```

L'objet récepteur du message est alors le paramètre implicite de la fonction membre opérateur.

Pour un opérateur binaire <B> comme l'addition, les opérateurs de comparaison, nous pouvons choisir d'écrire l'une des deux formes de déclaration :

```
// fonction non membre
TypeRenvoyé operator <B> (CCC Objet1 , CCC Objet2);
// ou bien
class CCC {
    // etc.
    TypeRenvoyé operator <B> (CCC Objet2);
    // l'opérateur <B> est une fonction membre de la classe CCC
    // etc.
};
```

Ces deux déclarations ne peuvent coexister, car pour tout appel de B le compilateur détecterait une ambiguïté.

Si nous choisissons de définir un opérateur binaire, par exemple +, comme une fonction membre de la classe String, alors les écritures suivantes :

```
String S1 = "ABC", S2 = "DEF";
String S3, S4;
S3 = S1 + S2; // appel court
S4 = S1.operator +(S2); // appel explicite à la fonction membre
```

sont équivalentes.

Pour garder un code lisible, il faudra veiller à garder une signification proche du sens usuel de l'opérateur initial et à conserver les liens entre les différents opérateurs. Par exemple, si nous définissons l'opérateur +=, pour la classe String l'opération :

```
ChaineDest += ChaineSource;
```

doit rester équivalente à[1] :

```
ChaineDest = ChaineDest + ChaineSource;
```

1 Comme le langage C, C++ fournit pour les types prédéfinis des opérateurs d'affectation avec calcul, qui sont += -= *= /= %= >>= <<= &= ^= |= et dont la seule utilité est de permettre une concision d'écriture : X += 4 est équivalent à X = X + 4.

3.3.1 - Désigner l'objet qui reçoit le message

Les fonctions membres ont toutes comme paramètre implicite l'objet récepteur du message. Par exemple, dans :

```
objet.message(arg1, arg2, ...)
```

objet est un paramètre implicite de la fonction message, alors que *arg1* et *arg2* sont des paramètres effectifs de l'appel. Cela signifie que l'on peut manipuler toutes les données et fonctions membres de *objet* dans le corps de *message*, sans préciser explicitement que l'on fait référence aux données ou fonctions membres de l'objet qui a reçu le message.

Il est quand même possible de désigner explicitement l'objet récepteur du message à l'intérieur du corps même de la fonction associée à ce message, en utilisant le mot-clé this, qui est un pointeur sur cet objet. Ainsi, *this représente, dans une fonction membre, l'objet qui recevra le message. Pour vérifier l'identité de deux objets appartenant à une classe *CCC*, nous avons implémenté, dans l'exemple suivant, la fonction membre EstIdentiqueA. Cette fonction renvoie le résultat de la comparaison entre le pointeur sur l'objet récepteur du message (this) et le pointeur sur l'objet argument (&Objet) :

```
class CCC {
   ... // etc.
 public:
   int EstIdentiqueA (CCC & Objet)
   { // indique si le récepteur du message et Objet sont le même objet
      return (this == &Objet);
   }
   ... // etc.
};
```

Grâce à cet identifiant, on peut renvoyer à la fin d'une fonction, l'objet lui-même avec l'instruction :

```
return *this;
```

Nous utiliserons cette notation pour implémenter l'opérateur d'affectation (Cf. 3.3.3).

3.3.2 - Surcharge de l'opérateur d'indexation

Dans un souci de simplification des notations, il est légitime de vouloir accéder à un élément de la chaîne de caractères en utilisant l'opérateur [ ]. On définit cet opérateur pour notre classe en surchargeant l'opérateur d'indexation du langage. Dans T[K], l'opérateur [ ] est binaire, il prend comme premier paramètre l'objet T et comme second paramètre l'indice K. Nous pouvons donc remplacer cette syntaxe par un appel à la fonction opérateur associée :

```
operator [] (T, K);
```

ou encore par un appel à la fonction membre opérateur associée :

```
T.operator [] (K);
```

Nous choisissons de mettre en oeuvre l'opérateur d'indexation comme une fonction membre, et nous ajoutons à la déclaration de la classe de la figure 3.3, la ligne suivante :

```
char & operator[] (int Index);
```

puis nous définissons cette fonction :

```
char & String::operator[] (int Index)
{
   if (Index < 0 || Index >= nbCarac) {
      cerr << "Index hors limite \n";
      exit(3);
   }
   return chaine[Index];
}
```

Cet opérateur prend comme paramètre implicite l'instance de la classe String qui reçoit le message et comme unique paramètre explicite, la position du caractère dans la chaîne. Lors de l'appel de cette fonction, nous vérifions si l'index passé en paramètre correspond bien à un index valide. S'il est incorrect, nous abandonnons le traitement ; dans le cas contraire, nous renvoyons une référence sur le caractère de rang Index. En renvoyant une référence, nous fournissons un caractère utilisable comme une *lvalue* et qui est exactement chaine[Index]. Nous pouvons alors écrire :

```
String T(6);
T[1] = 'a';
```

et l'exécution de la séquence suivante :

```
String S1("glace");
printf("Chaine initiale : %s\n",S1.Chaine());
S1[0] = 'p';
printf("Modification d'un caractère : %s\n",S1.Chaine());
S1[6] = 'b';
printf("Fin du programme : %s\n",S1.Chaine());
```

affiche sur la sortie standard :

```
Chaine initiale : glace
Modification d'un caractère : place
Index hors limite
```

L'opérateur ainsi surchargé gère donc l'accès aux caractères en vérifiant que l'indice donné en paramètre est un indice valide. Toutefois, nous pouvons écrire :

```
S1[1] = '\0';
```

ce qui invalide la valeur de nbCarac. Pour remédier à cet inconvénient, nous pourrions supprimer la donnée membre nbCarac et appeler la fonction standard strlen quand nous avons besoin de connaître le nombre de caractères de la chaîne. Une autre solution consisterait à interdire la modification du caractère retourné par l'appel de l'opérateur d'indexation en le définissant :

```
const char & String::operator [] (int Index) {...}
```

Nous choisissons de ne pas modifier notre implémentation, mais il faut garder à l'esprit ce défaut.

Afin de pouvoir manipuler une chaîne constante, nous définissons une seconde fonction opérateur [ ] constante dans la section publique de la classe String :

```
char String::operator[] (int i) const
{
   if (i < 0 || i >= nbCarac) {
      cerr << "Index hors limite\n"; exit(3);
   }
   return chaine[i];
}
```

Si nous supprimions la fonction opérateur d'indexation non constante, en conservant la fonction constante, nous empêcherions toute modification d'une chaîne non constante, la valeur retournée étant constante.

3.3.3 - Surcharge de l'opérateur d'affectation

L'affectation est définie par défaut pour une classe comme une copie membre à membre. Dans de nombreux cas, cette définition suffit. Cependant elle devient dangereuse pour une classe dont les objets effectuent des allocations dynamiques. Ainsi, dans le cas de la classe String, l'exécution de :

```
String S1("abc"), S2("defg");
S1 = S2;
```

pose un problème qui est explicité par la figure 3.4.

En effet, le mécanisme fourni par défaut pour l'affectation, copie les valeurs des membres de S2 dans les membres correspondants de S1. A l'issue de l'opération, nous constatons :

- que l'espace alloué initialement pour le membre Chaine de S1 est inaccessible mais n'a pas été libéré ;
- que les membres Chaine de S1 et S2 désignent la même chaîne. Ainsi, toute modification de l'un des deux objets aura une incidence cachée sur l'autre objet !

Un autre effet pervers de l'affectation par défaut se manifesterait avec l'exécution de :

```
S1 = "xyzt";
```

qui déclencherait les opérations suivantes :

1 – Conversion de "xyzt" en un objet temporaire String que, pour les besoins de l'explication, nous nommerons Temp. Cette conversion s'effectue par appel du constructeur String (const char *).

2 – Exécution de l'affectation par défaut : copie membre à membre de Temp vers S1.

3 – Destruction de l'instance temporaire Temp de String, qui libère la zone-mémoire pointée par Temp.chaine. Comme S1.chaine pointe désormais vers la même zone, l'objet S1 est endommagé.

Il faut donc que nous surchargions l'affectation pour la classe String. Pour ce faire, nous supposons que la chaîne résultant d'une affectation est en tout point équivalente à la chaîne copiée et nous ne chercherons pas à économiser la place mémoire utilisée.

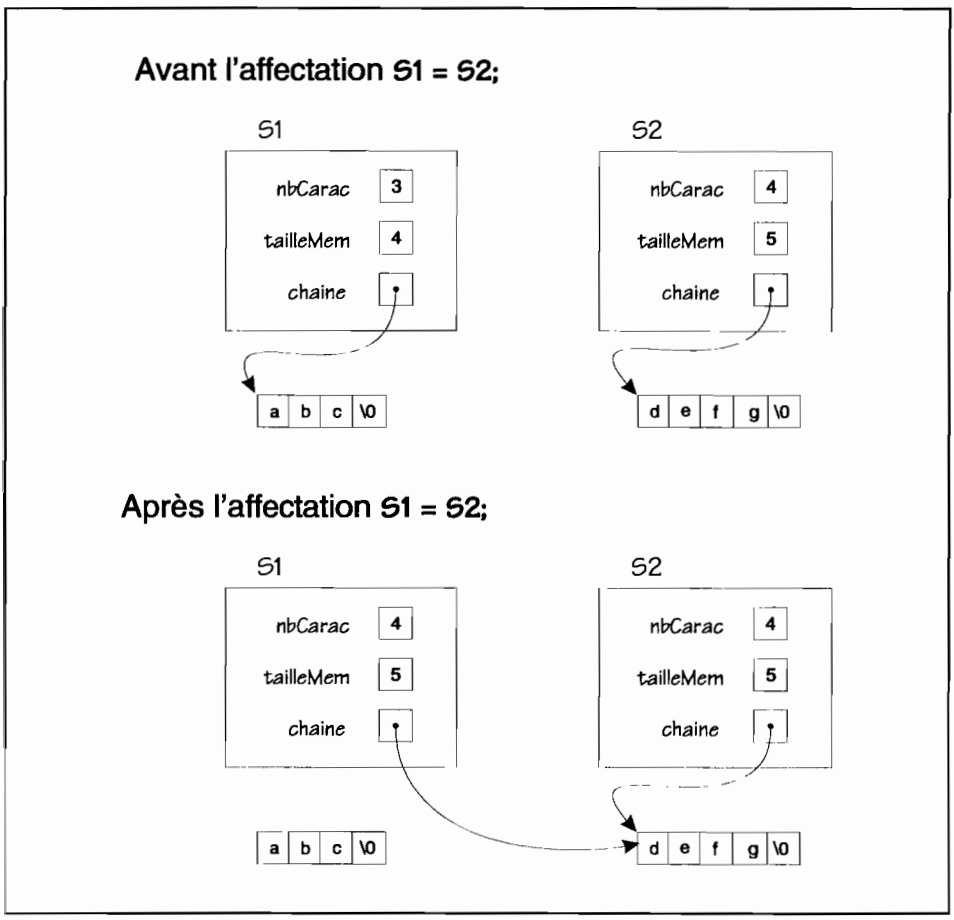


Figure 3.4 : effet de l'affectation par défaut pour S1 = S2

Nous ajoutons à la définition de la classe de la figure 3.3, la déclaration de l'opérateur d'affectation en surchargeant la fonction opérateur associée :

```
String & operator = (const String & Source);
```

qui prend comme argument une référence sur une chaîne de caractères et retourne une référence à une chaîne de caractères.

Nous définissons cette fonction :

```
String & String::operator = (const String & Source)
{
   if (this != &Source) { // ce n'est pas X = X
      if (Capacite() < Source.nbCarac) {
         delete [] chaine;
         chaine = alloue(Source.nbCarac);
      }
      strcpy(chaine, Source.chaine);
      nbCarac = Source.nbCarac;
   }
   return *this;
}
```

L'affectation proprement dite n'est exécutée que si les deux objets qu'elle concerne sont distincts. Si c'est le cas, on vérifie que la mémoire allouée est suffisante pour l'objet récepteur : si elle n'est pas suffisante, on la libère puis on réalloue une zone de dimension suffisante. Ensuite, nous copions l'information pointée par *chaine* et nous affectons à *nbCarac* la valeur de *Source.nbCarac*.

Enfin, nous retournons l'objet qui a reçu le message. En effet, en C++, une affectation telle que X = Y est une opération qui renvoie l'objet qui reçoit la valeur affectée. C'est grâce à ce renvoi et à l'associativité de droite à gauche et que l'on peut écrire (si toutes les variables de l'exemple sont entières) :

```
A = B = C = D = 0;
```

Si, comme il est souhaitable, nous voulons conserver à l'affectation de la classe *String* son sens usuel, il faut donc renvoyer l'objet *String* récepteur du message en fin d'exécution de l'opérateur. Nous le faisons avec *return *this*.

3.3.4 - Surcharge de l'opérateur +

Au lieu de redéfinir une fonction du type *strcat*[2] pour notre classe, nous choisissons de surcharger l'opérateur d'addition pour concaténer deux chaînes.

L'opérateur + étant un opérateur binaire, il peut être défini comme une fonction membre constante (car les données membres de l'objet récepteur du message ne sont pas modifiées par la concaténation) à un argument :

```
String String::operator + (const String & Source) const
{ // opérateur de concaténation
   String Temp(nbCarac + Source.nbCarac);
   strcat(strcpy(Temp.chaine, chaine) ,
          Source.chaine);
   Temp.nbCarac = strlen(Temp.chaine);
   return Temp;
}
```

2 *strcat* est une fonction standard de concaténation de chaînes qui est déclarée par :
*char * strcat (char * dest, const char * source)*
et qui concatène la chaîne pointée par *source* à la chaîne pointée par *dest*.

Pour construire le résultat de l'opération, nous instancions un objet de type String, d'une capacité égale à la somme des chaînes à concaténer. Puis le membre chaine de l'objet recevant le message est copié dans Temp.chaine. Ensuite nous concaténons le résultat de cette opération[3] avec le membre Source.chaine. Enfin, nous retournons l'objet temporaire ainsi créé. Nous pouvons maintenant écrire le code de la figure 3.5 :

```
String S1("ABCD"), S2("EFGH"), S3;
S3 = S1 + S2;
S3 = S1 + "EFGHI";
S3 = "ABCD" + S2; // refusée par le compilateur
```

Figure 3.5 : exemple d'utilisation de l'opérateur + surchargé

Si nous voulons compiler le code de la figure 3.5, le compilateur nous indique pour la dernière ligne que l'opération est illégale. En effet, si nous analysons les lignes de code précédentes :

- la première concaténation appelle directement l'opérateur d'addition avec comme argument implicite S2,
- la seconde appelle le même opérateur, puis cherche une règle de conversion entre le type de "EFGHI" et celui de l'argument. En utilisant le constructeur String(const char*), elle trouve une règle de conversion pour effectuer l'opération,
- dans le troisième cas, l'opération est interprétée comme l'envoi du message + avec comme argument un objet de type String, à un pointeur sur caractère. Or le type prédéfini char* n'accepte l'opérateur + que pour l'incrémentation des pointeurs. La compilation échoue.

Pour que notre opérateur fonctionne également dans le cas où le récepteur du message est un pointeur sur caractère, il faut que ce pointeur puisse être un paramètre explicite de la fonction opérateur. Il pourra alors être transtypé vers le type String. Il suffit que nous définissions la fonction opérateur + comme une fonction non membre :

```
String operator + (const String & S1, const String & S2)
{   // renvoie la concaténation de S1 et S2
    String Temp(S1.NbCarac()+S2.NbCarac());
    strcat(strcpy(Temp.Chaine(), S1.Chaine()),
           S2.Chaine());
    Temp.nbCarac = strlen(Temp.Chaine());
    return Temp;
}
```

Dans cette définition, nous sommes obligés d'utiliser les accesseurs aux membres privés nbCarac et chaine des arguments S1 et S2. En effet, notre opérateur n'est plus une fonction membre.

---

3 Le résultat de l'appel de strcpy est utilisé comme premier argument de strcat. En effet strcpy (Dest, Source) renvoie Dest.

Pour simplifier l'écriture, on peut déclarer la fonction opérateur comme une fonction *amie* de la classe String :

```
class String {
...// etc.
   friend String operator + (const String & S1,
                             const String & S2);
...// etc.
}
```

Le spécificateur friend autorise, pour une fonction non membre, l'accès aux membres privés d'une classe. Une fonction amie d'une classe MaClasse doit être explicitement déclarée dans la définition de la classe MaClasse. Ainsi, une fonction amie de la classe MaClasse n'est pas une fonction membre de MaClasse, mais elle a le droit d'accéder aux membres privés[4] de MaClasse.

Ayant déclaré la fonction opérateur + comme une fonction amie de la classe String, nous la définissons maintenant plus simplement :

```
String operator + (const String & S1, const String & S2)
{
   String Temp(S1.nbCarac + S2.nbCarac);
   strcat(strcpy(Temp.chaine , S1.chaine),
          S2.chaine);
   Temp.nbCarac = strlen(Temp.chaine);
   return Temp;
}
```

Nous pouvons maintenant compiler et exécuter le code de la figure 3.5.

3.3.5 - Surcharge des opérateurs de comparaison

Forts de notre expérience précédente, nous allons définir les opérateurs de comparaison comme des fonctions amies de la classe String. En effet, si l'on veut pouvoir comparer un objet String avec, par exemple, une chaîne au sens du langage C :

```
String S1(...);
char * Chn = "...";
```

et écrire aussi bien Chn < S1 que S1 < Chn, il faut que l'opérateur < soit une fonction non membre de String.

Pour comparer deux chaînes de caractères, pointées par des pointeurs P1 et P2, nous allons utiliser la fonction standard de comparaison strcmp, dont la déclaration est la suivante

```
int strcmp (const char * P1, const char * P2);
```

Cette fonction retourne :

- une valeur négative si la chaîne P1 est plus petite que la chaîne P2,
- zéro si la chaîne P1 est égale à la chaîne P2,

4 Les membres privés sont définis avec les spécificateurs private ou protected (Cf. chapitre 4).

– une valeur positive si la chaîne P1 est plus grande que la chaîne P2.

La famille d'opérateurs de comparaison que nous allons implémenter retournera un entier égal à zéro si le résultat de la comparaison est faux, une valeur non nulle dans le cas contraire. Tous ces opérateurs de comparaison étant des opérateurs binaires, leur déclaration sera de la forme :

```
int operator <B> (const String & S1, const String & S2);
```

<B> étant l'opérateur (==, !=, <, etc.)

Egalité et différence

En utilisant ce qui a été énoncé précédemment, nous définissons les deux fonctions de la figure 3.6.

```
int operator != (const String & S1, const String & S2)
{
   return (strcmp(S1.chaine, S2.chaine));
}

int operator == (const String & S1, const String & S2)
{
   return (! (S1 != S2));
}
```

Figure 3.6 : définition de la différence et de l'égalité

Si les deux chaînes sont différentes, `strcmp` renvoie une valeur non nulle, correspondant à la valeur que doit retourner l'opérateur de différence. Il nous suffit donc de retourner le résultat de `strcmp`.
Nous définissons l'égalité comme la négation de l'inégalité, ce qui simplifie la maintenance ultérieure de la classe : si l'inégalité fonctionne, l'égalité aussi.

Autres opérateurs de comparaison

```
int operator > (const String & S1, const String & S2)
{   return (strcmp(S1.chaine,  S2.chaine) > 0); }

int operator < (const String & S1, const String & S2)
{   return (strcmp(S1.chaine, S2.chaine) < 0); }

int operator >= (const String & S1, const String & S2)
{   return (! (S1 < S2)); }

int operator <= (const String & S1, const String & S2)
{   return (! (S1 > S2)); }
```

Figure 3.7 : définition des inégalités

Comme pour les deux opérateurs précédents, les autres opérateurs de comparaison fonctionnent par couple, ainsi >= est la négation de < et <= est la négation de >. Il nous suffit donc de définir deux des opérateurs à l'aide de la fonction standard de comparaison et les deux autres par négation des premiers (figure 3.7).

Nous déclarons évidemment ces fonctions en tant qu'amies à l'intérieur de la définition de la classe String

3.4 Opérateurs d'entrées/sorties

Comme nous l'avons vu dans les deux chapitres précédents, le langage C++ fournit des librairies de gestion des entrées/sorties par flots. Ces mécanismes d'entrées et sorties ont été définis en standard pour les types élémentaires du langage. Cette librairie est simple à utiliser ; elle s'appuie sur la conversion des données vers le type chaîne de caractères.

Pour noter l'envoi d'une information dans un flux de sortie, les concepteurs de C++ ont choisi de surcharger l'opérateur <<[5], utilisant ainsi une notation suggestive pour l'envoi d'information dans un flux :

```
ObjetFlot << Information
```

De même, pour les flots d'entrée, l'opérateur >> a été surchargé :

```
char Nom[80];
cout << "Tapez votre nom :";
cin >> Nom;
```

Dans cette séquence, cin et cout sont les deux objets, des classes respectives istream et ostream, qui représentent les flux standard stdin et stdout. Les classes istream et ostream sont définies dans le fichier d'en-tête iostream.h.

Afin de garder une certaine cohérence avec le mécanisme établi pour les types de base, on surcharge souvent les opérateurs << et >> pour qu'ils acceptent aussi des objets des classes que l'on définit soi-même.

3.4.1 - Surcharge de l'opérateur de sortie

Deux choix s'offrent à nous pour surcharger l'opérateur << pour la classe String. Nous pouvons le définir de manière à avoir le même résultat que pour une chaîne standard ou afficher des informations complémentaires pour le programmeur. Dans le cadre de la mise en place de notre classe, nous choisissons la seconde option qui nous permettra de vérifier la cohérence de nos objets, en visualisant les valeurs des données membres.

Cet opérateur s'applique à un objet de la classe ostream et doit pouvoir accéder aux données membres privées de notre classe. Nous le déclarons donc comme ami de notre classe :

---

5 Cet opérateur, tout comme l'opérateur >> pour les flots d'entrée, perd ainsi, pour les flots, sa signification usuelle (décalage de bits).

```
class String {
   // etc.
   friend ostream & operator <<
         (ostream & Flot, const String & Source);
   // etc.
};
```

La fonction opérateur prend comme premier argument (écrit à gauche de l'opérateur <<) une référence au flux de sortie et comme second argument une référence à l'objet de type String à afficher. Elle retourne une référence sur le flux passé en paramètre, ce qui permet d'enchaîner des écritures de sortie sur un des flux standard. En effet, dans une écriture telle que :

```
cout << X << Y;
```

l'opérateur << est associatif de gauche à droite et le premier message traité est << X. Ce message s'exécute avec l'appel

```
operator <<(cout, X)
```

qui affiche la valeur de X et renvoie l'objet *cout*. Cet objet renvoyé reçoit à son tour le second message << Y.

Nous surchargeons l'opérateur << de la manière suivante :

```
ostream & operator << (ostream & Flot,
                       const String & Source)
{
   return Flot << "Chaine \"" << Source.chaine
               << "\" de " << Source.nbCarac
               << "caractères et pouvant en contenir "
               << Source.Capacite();
}
```

Pour afficher sur le flux standard de sortie chacun des membres, nous utilisons tout simplement << qui est prédéfini pour tous les types élémentaires. Nous pouvons alors écrire :

```
String C1 = "ABCDEF";
String C2 = "IJK";
cout << C1 << endl
     << C2 << endl;
C1 = C2;
cout << C1 << endl;
```

ce qui affiche à l'écran

```
Chaine "ABCDEF" de 6 caractères et pouvant en contenir 6
Chaine "IJK" de 3 caractères et pouvant en contenir 3
Chaine "IJK" de 3 caractères et pouvant en contenir 6
```

A la fin de l'affichage, on note la présence du mot-clé *endl*, qui est un manipulateur de sortie : il ajoute à la fin du flux un '\n' et vide le flux.

Pour afficher uniquement la chaîne de caractères dans le flux standard de sortie, il suffit d'utiliser l'accesseur Chaine :

```
String S("ABCDE");
cout << S.Chaine();
```

3.4.2 Surcharge de l'opérateur d'entrée

Les entrées, symétriques des sorties, sont définies dans la classe istream. Pour redéfinir >> pour la classe String, nous allons utiliser la fonction membre *get* de la classe de flots istream. Cette fonction a deux signatures :

```
istream & istream::get(char & c);
istream & istream::get(char * p, int n, char Fin = '\n');
```

Si l'on appelle cette fonction avec un paramètre de type char &, elle lit un caractère dans le flux d'entrée standard. Elle retourne une référence sur le flux standard et stocke le caractère lu à l'adresse du caractère passé par référence.

Lors de l'appel avec un pointeur sur caractère p, les caractères sont lus dans le flux, puis copiés dans le tampon pointé par p. Un caractère nul est mis à la fin de la chaîne lue. La lecture s'arrête au premier caractère Fin trouvé et au plus tard au n-ième caractère. Si le caractère Fin est rencontré, *get* le laisse dans le flux. Elle retourne une référence sur le flux standard.
Nous utiliserons aussi la fonction :

```
istream & istream::putback (char c);
```

qui remet le caractère c dans le flux d'entrée recevant le message.

Après avoir déclaré dans la définition de la classe String, une fonction opérateur >> amie, nous la définissons :

```
istream & operator >> (istream & Flot, String & Source)
{   // lit par morceaux de TailleMax caractères
    const int TailleMax = 5;
    char c;
    Source = "";
    String Temp(TailleMax);
    Flot.get(c);
    while ( c != '\n' ) {
        Flot.putback(c);
        Flot.get(Temp.chaine, Temp.Capacite(), '\n');
        Temp.nbCarac = strlen(Temp.chaine);
        Source = Source + Temp;
        Flot.get(c);
    }
    return Flot;
}
```

Nous initialisons à vide la chaîne de caractères passée en paramètre et nous construisons un tampon de lecture Temp d'une capacité TailleMax.
Si le premier caractère du flux d'entrée est différent du caractère de fin de lecture, nous le replaçons dans le flux avec la fonction membre putback. Lors de l'appel

de la fonction membre *get* à trois paramètres de la classe istream, nous lisons au plus TailleMax caractères depuis le flux d'entrée vers Temp, nous comptons le nombre de caractères effectifs de la chaîne, puis nous concaténons Temp à Source. Si le caractère terminal a été rencontré, il se trouve en tête du flux. On poursuit ensuite la lecture jusqu'à rencontrer le caractère terminal. On termine l'exécution en renvoyant une référence au flux que nous venons de manipuler afin de pouvoir enchaîner une autre entrée.

Nous pouvons utiliser cet opérateur de la façon suivante :

```
String S1, S2;
cout << endl << "Votre chaine ? ";
cin >> S1;
cout << endl;
cout << "Vos deux chaînes ? "
     << "(tapez <return> entre chaque chaine)";
cin >> S1 >> S2
cout << endl;
```

Nous voyons à travers ces deux exemples que les mécanismes d'entrées/sorties ont été implémentés de façon à ce que la définition des classes ostream et istream ne soient pas à modifier lors de la surcharge des opérateurs d'entrées/sorties.

3.5 Compter le nombre d'instances d'une classe

S'il est indispensable de gérer correctement l'allocation mémoire en vérifiant que la place demandée existe, il peut être intéressant de connaître le nombre d'instances d'une classe ainsi que la place mémoire effectivement occupée par ces instances. Cette connaissance permet d'avoir une bonne adéquation entre le matériel et les besoins du programme.

Pour cela, il suffit de définir deux variables globales dont les valeurs seront augmentées ou diminuées au fur à mesure des appels aux constructeurs et au destructeur de la classe. Cette solution présente de nombreux inconvénients.

- Les données qui se rapportent à la classe ne sont pas encapsulées dans la classe, mais sont déclarées et manipulées indépendamment de la classe. Il est alors impossible de contrôler l'accès à ses variables et donc de garantir le bon fonctionnement de la classe.
- Si l'on implémente ce mécanisme pour chacune des classes, le nombre de variables globales augmente de façon importante ce qui réduit la lisibilité du code et engendre à terme des problèmes de maintenance.

3.5.1 - Variables de classe

L'information que représente le nombre d'instances d'une classe ne peut être associée à aucune instance particulière de la classe. C'est une information partagée entre toutes les instances, et qui doit exister en l'absence de toute instance. C'est une information que nous associons à la classe.

Le langage C++ fournit un mécanisme qui permet d'associer des données à une classe. Pour déclarer une donnée membre partagée lors de la déclaration de la classe `CCC` , il suffit d'écrire :

```
class CCC {
...// etc.
    static int VarClasse;
};
```

Cette nouvelle utilisation du mot-clé `static` ne doit pas être confondue avec la définition d'une variable statique dans une fonction (Cf. Annexe A). Ici, elle indique au compilateur qu'une donnée existe en un seul exemplaire partageable entre toutes les instances de la classe `CCC` et que la portée de sa définition est externe. Cette donnée, que nous nommerons variable de classe (par analogie avec Smalltalk), est accessible depuis les autres modules. Toutefois cette variable n'est pas définie dans la classe, elle y est seulement déclarée[6]. Il nous donc faut la définir à l'extérieur de la classe où elle est déclarée, en qualifiant la définition par le nom de cette classe.

Nous pouvons ainsi associer à la classe `String` le nombre d'instances et la taille de la mémoire qu'elles occupent. La figure 3.8 montre la déclaration et la définition des deux variables de classe correspondante.

```
class String {
 private:
   static int nbInstances;   // compteur
   static int memInstances; // mémoire allouée
... // etc.
};
// ici on définit les variables de classe et on les initialise.
int String::nbInstances = 0;
int String::memInstances = 0;
```

Figure 3.8 : deux variables partagées pour les instances

L'initialisation d'une variable de classe à zéro n'est pas indispensable, car le compilateur initialise par défaut les variables de classe à zéro si elles ne sont pas définies. Il n'est donc pas utile pour notre classe d'initialiser `nbInstances` et `memInstances` sauf pour améliorer la lisibilité du programme.

On notera que la définition de `nbInstances` et de `memInstances` est la seule opération qui permette d'accéder directement à ces deux variables, qui sont privées. En général, ces définitions ne sont pas faites par l'utilisateur, mais incluses par le concepteur de la classe dans le module de code de cette classe.

6 En effet, la définition d'une classe ne définit aucune variable : elle précise seulement le moule qui servira à la construction de chaque objet et les données membres ne sont pas à proprement parler définies comme des variables ordinaires : elles sont véritablement créées à chaque instanciation. Dans le même esprit, les variables de classes ne peuvent être que déclarées à l'intérieur de la classe.

3.5.2 - Notion de méthode de classe

Ces variables de classe étant déclarées comme privées, il nous faut également définir des accesseurs pour ces données. Les accesseurs doivent pouvoir accéder aux variables de classe, mais ils n'ont pas besoin d'accéder aux variables d'instances.

Nous pouvons les définir comme des méthodes de classes, c'est à dire des fonctions qualifiées par le mot-clé *static*. Associées à la classe, elles accèdent aux membres statiques de la classe, mais ne peuvent pas manipuler les données membres non statiques, ni le pointeur *this*. Nous noterons qu'une fonction membre statique ne peut avoir le même nom qu'une fonction membre non statique, même si leurs signatures sont différentes.

Nous ajoutons à la section publique de *String*, les accesseurs suivants :

```
static int NbInstances()  { return nbInstances; }
static int MemInstances() { return memInstances; }
```

qui retourne les valeurs des deux membres statiques.

Toute fonction membre de la classe *CCC* peut accéder directement aux variables de classes et aux méthodes de classe de *CCC*. Une fonction non membre de *CCC* a accès aux membres statiques publics de *CCC* en les qualifiant par *CCC::*. En dehors de toute fonction, on peut accéder à un membre statique public de *CCC* en qualifiant l'accès soit par *CCC::*, soit par l'intermédiaire d'un objet de *CCC*. La séquence suivante illustre ces caractéristiques :

```
cout << String::nbInstances(); // aucun objet CCC n'existe
String S1("abcd");
cout << S1.NbInstances(); // équivalent à String::nbInstances()
```

Définissons maintenant la fonction qui met à jour *memInstances* et *nbInstances*. Cette fonction doit incrémenter ses valeurs pour chaque allocation et les décrémenter à chaque destruction.

```
enum TypActMem { New = 1, Delete = -1 };
void gereVarClasse (TypActMem Action);
```

La fonction *gereVarClasse* modifiera les valeurs des variables de classe en fonction de la valeur de *Action*.

```
void String::gereVarClasse (TypActMem Action)
{   //appelée par alloue et par le destructeur
    if (Action != New && Action != Delete)
       {cerr << "Action mémoire incorrecte" << endl; exit(4); }
    nbInstances  += Action;
    memInstances += Action * (sizeof(String)+
                              tailleMem * sizeof(char));
}
```

Si *Action* est incorrect, le traitement est abandonné. Sinon nous incrémentons *nbInstances* de *Action*, qui vaut 1 lors d'une allocation et -1 lors d'une destruction. Il nous faut également calculer la taille totale d'un objet de type *String*. Elle est égale à la taille de l'objet lui-même à laquelle s'ajoute celle de la zone mémoire pointée par *chaine*.

Nous multiplions ce résultat par Action afin de l'ajouter ou de le retrancher de la taille de l'espace-mémoire déjà utilisé.

Il faut maintenant modifier le code de la fonction d'allocation dynamique et du destructeur afin qu'il appelle cette fonction (figure 3.9).

```
char * String::alloue(int Taille)
{
   if (Taille < 0) cerr << "Allocation négative impossible\n";
   tailleMem = Taille + 1;
   char * P = new char[tailleMem];
   if (! P)
      { cerr << "Echec de l'allocation\n"; exit(1); }
   gereVarClasse(New);
   return P;
}

String::~String()
{
   delete [] chaine;
   gereVarClasse (Delete);
}
```

Figure 3.9 : nouvelle définition de l'allocation dynamique et du destructeur.

3.5.3 - Utilisation des variables de classe pour la mise au point de la classe

Les variables et méthodes de classe associent des données et des fonctions à la classe. Elles limitent ainsi l'occupation mémoire des données communes à une classe et permettent leur encapsulation.

Si leur utilisation est limitée à la phase de test du code, il ne faut pas compiler leurs déclarations et définitions, ainsi que les fonctions qui les manipulent dans la version finale. Pour cela, il suffit d'utiliser des directives de compilation conditionnelle :

```
#define DEBUG_CLASS_STRING
class String {
   // etc.
   #ifdef DEBUG_CLASS_STRING
   static int memInstances;
   void gereVarClasse(...) {...}
   #endif
};
#ifdef DEBUG_CLASS_STRING
int String::memInstances = 0;
#endif
```

Ces directives sont traitées par le préprocesseur et permettent de transmettre ou de ne pas transmettre au compilateur une séquence de code. Ainsi, dans l'exemple ci-dessous, la directive

```
#define DEBUG_CLASS_STRING
```

définit la constante DEBUG_CLASS_STRING . Dans la suite du code, la directive #ifdef DEBUG_CLASS_STRING provoque l'inclusion dans le source à compiler des instructions qui suivent jusqu'au *#endif*, seulement si cette constante est définie. Dans notre exemple, il suffit de retirer la directive

```
#define DEBUG_CLASS_STRING
```

pour que les séquences encadrées par *#ifdef* et *#endif* disparaissent de la compilation.

3.5.4 - Une classe *String* fonctionnelle

Arrivé à cette phase de notre développement, nous allons répartir notre code en trois modules afin de pouvoir fournir à un développeur un fichier d'en-tête (Cf. chapitre 1) et un module compilé. Les modules *c_String.h* et *c_String.cpp*, qui constituent l'implémentation de la classe, sont présentés en annexe du présent chapitre (3.6.2 et 3.6.3). Le module *c_String.h* contient la définition de la classe. L'utilisation de la directive de compilation *#ifndef* permet d'éviter un message d'erreur indiquant qu'il y a plusieurs déclarations de la classe lors de la compilation de modules incluant plusieurs fois *c_String.h*. Le module *c_String.cpp* contient la définition des fonctions membres privées et publiques, ainsi que la définition des variables et méthodes de classe.

Nous avons également ajouté, afin d'avoir une classe complète, quelques fonctions membres qui n'ont pas été étudiées ici :

```
String SubString(int Debut, int NbCarac) const ;
```

extrait une sous-chaîne de type *String* de la chaîne de type *String* qui reçoit le message, à partir du caractère de position *Debut* (qui vaut 0 pour le premier caractère) sur une longueur de *NbCarac* (*NbCarac* étant au plus égal à la longueur de la chaîne à partir de *Debut*). La fonction renvoie la sous-chaîne extraite.

```
int StringString(const String & Cherche) const;
```

renvoie la position du début de la chaîne *Cherche* dans l'objet récepteur, ou une valeur négative si *Cherche* n'est pas trouvé.

```
int StringChar(char Carac) const;
```

retourne la position du caractère recherché *Carac* dans la chaîne de type *String* recevant le message, ou une valeur négative si ce caractère n'est pas trouvé.

```
void ToUpper();
```

met la chaîne de caractères pointée par le membre *chaine* en majuscules.

3.6 - Annexe : la classe *String*

3.6.1 - Fichier d'en-tête *entete.h*

```
#include <string.h>
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>

#ifndef _MIN
#define _MIN
#ifdef __cplusplus
int max(int value1, int value2);
int min(int value1, int value2);
#endif
#endif

#ifndef _MONBOOL
#define _MONBOOL
typedef boolean;
#endif
```

3.6.2 - Fichier d'en-tête *c_String.h*

```
#ifndef _STRING
#define _STRING
#include "entete.h"
class String {
 private:
   enum TypActMem { New = 1, Delete = -1 };
   char * chaine;
   int nbCarac;
   int tailleMem;
   static int nbInstances;
   static int memInstances;
   char * alloue(int Longueur);
   void gereVarClasse ( TypActMem Action );
 public:
   // Constructeurs
   String ( const char * Chn = "" );
   String ( int tailleChn );
   String ( const String & Chn );
   String ( const char Carac );
   // Destructeur
   ~String();
```

```
    // Accesseurs
    int NbCarac() const;
    int Capacite() const ;
    const char * Chaine();
    static int NbInstances();
    static int MemInstances();
    // Fonctions membres opérateurs d'indexation
    char & operator[] (int i);
    char operator[] (int i) const;
    // Fonction membre opérateur d'affectation
    String & operator = ( const String & Source );
    // Fonction amie opérateur de concaténation de chaînes
    friend String operator + (const String & Source1,
                              const String & Source2 );
    // Fonctions amies opérateurs de comparaison
    friend int operator != (const String & Source1,
                            const String & Source2 );
    friend int operator == (const String & Source1,
                            const String & Source2 );
    friend int operator >  (const String & Source1,
                            const String & Source2 );
    friend int operator <  (const String & Source1,
                            const String & Source2 );
    friend int operator >= (const String & Source1,
                            const String & Source2 );
    friend int operator <= (const String & Source1,
                            const String & Source2 );
    // Fonctions opérateurs d'entrées/sorties amies
    friend ostream & operator << (ostream & stream,
                                  const String & Source);
    friend istream & operator >> (istream & stream,
                                  String & Source);
    // Affiche les valeurs des variables de classe
    static void AfficheVarClasses(ostream & Stream);
    // Fonctions de manipulation des chaînes de caractères
    String SubString(int Debut, int NbCarac) const ;
    int StringString(const String & Cherche) const;
    int StringChar(char Carac) const;
    void ToUpper();
};
#endif
```

3.6.3 - Fichier *c_String.cpp*

```
#include "c_String.h"
// fonctions de comparaison numérique utilisée max et min utilisées dans les
// définitions
int max(int value1, int value2)
{
   return ( (value1 > value2) ? value1 : value2);
}
int min(int value1, int value2)
{
   return ( (value2 > value1) ? value1 : value2);
}
// définition/initialisation des variables de classes
int String::nbInstances = 0;
int String::memInstances = 0;

// Fonctions private
char * String::alloue(int Longueur)
{ /* fonction privée d'allocation d'espace-mémoire pour la chaîne stockée. Renvoie
     un pointeur sur char ou interrompt le programme si l'allocation échoue */
   if (Longueur < 0) cerr << "Allocation négative impossible\n";
   tailleMem = Longueur+1;
   char * P = new char[tailleMem];
   if ( ! P ) {
      cerr << "Echec de l'allocation\n";
      exit(1);
   }
   return P;
}

void String::gereVarClasse ( TypActMem Action )
{  /* fonction privée de mise à jour des variables de classe. Permet de suivre le
      nombre d'instances créées, et la mémoire qu'elles occupent. */
   if ( Action != New && Action != Delete )
      cerr << "Valeur du type de l'action mémoire incorrect" << endl;
   nbInstances += Action;
   memInstances += Action * (
sizeof(String)+(tailleMem)*sizeof(char));
}

// Fonctions public
String::String(const char * Chn)
{  // constructeur à partir d'un pointeur sur char
   nbCarac = strlen(Chn);
   chaine = alloue(nbCarac);
   strcpy(chaine,Chn);
   gereVarClasse (New);
}
```

```
String::String (int LgChn)
{   // constructeur à partir d'une longueur de chaîne
    nbCarac = 0;
    chaine = alloue(LgChn);
    chaine[0] = '\0';
    gereVarClasse (New);
}

String::String(const String & Chn)
{   // constructeur copie
    nbCarac = Chn.nbCarac;
    chaine = alloue(Chn.Capacite());
    strcpy(chaine,Chn.chaine);
    gereVarClasse (New);
}

String::String ( const char Carac )
{   // constructeur à partir d'un caractère
    nbCarac = 1;
    chaine = alloue(nbCarac);
    chaine[0] = Carac;
    chaine[1] = '\0';
    gereVarClasse (New);
}

String::~String()
{   // destructeur
    delete [] chaine;
    gereVarClasse (Delete);
}

int String:: NbCarac() const
{   // accesseur en consultation de nbcarac
    return nbCarac;
}

// accesseur en consultation de la taille mémoire de la chaîne
int String:: Capacite() const { return tailleMem-1; }

// accesseur en consultation de la chaîne stockée
const char * String::Chaine() { return chaine; }

// accesseur en consultation de la variable de classe nbInstances
int String::NbInstances() { return nbInstances; }

// accesseur en consultation de la variable de classe memInstances
int String::MemInstances() { return memInstances; }
```

```
char & String::operator[] (int i)
{  // opérateur d'indexation - Permet la modification du caractère renvoyé
   if ( i < 0 || i >= nbCarac ) {
      cerr << "Index hors limite \n";
      exit(1);
   }
   return chaine[i];
}

char String::operator[] (int i) const
{ /* opérateur d'indexation - Ne permet pas la modification du caractère renvoyé.
     Utilisé sur les objets constants */
   if ( i < 0 || i >= nbCarac ) {
      cerr << "Index hors limite \n";
      exit(1);
   }
   return chaine[i];
}

String & String :: operator = ( const String & Source )
{  // opérateur d'affectation
   if ( chaine != Source.chaine ) {
      if ( Capacite() < Source.nbCarac ) {
         String::memInstances -= tailleMem;
         delete [] chaine;
         chaine = alloue(Source.nbCarac);
         String::memInstances += tailleMem;
      }
      nbCarac = Source.nbCarac;
      strcpy(chaine,Source.chaine);
   }
   return * this;
}

String operator + ( const String & Source1,
                    const String & Source2 )
{  // opérateur de concaténation
   String Temp(Source1.Capacite() + Source2.Capacite());
   strcat( strcpy(Temp.chaine,Source1.chaine) ,
           Source2.chaine );
   Temp.nbCarac = strlen(Temp.chaine);
   return Temp;
}

int operator != (const  String & Source1,
                 const String & Source2 )
{  // opérateur différent de
   return ( strcmp(Source1.chaine,Source2.chaine));
}
```

```
int operator == (const String & Source1,
                 const String & Source2 )
{  // opérateur égal à
   return (!(Source1 != Source2));
}

int operator > (const  String & Source1,
                const String & Source2 )
{  // opérateur supérieur à
   return ( strcmp(Source1.chaine , Source2.chaine)
            > 0 );
}

int operator < (const  String & Source1,
                const String & Source2 )
{  // opérateur inférieur à
   return ( strcmp(Source1.chaine , Source2.chaine)
            < 0 );
}

int operator >= (const  String & Source1,
                 const String & Source2 )
{  // opérateur supérieur ou égal à
   return ( ! ( Source1 < Source2) );
}

int operator <= (const  String & Source1,
                 const String & Source2 )
{  // opérateur inférieur ou égal à
   return ( ! ( Source1 > Source2) );
}

istream & operator >> (istream & Flot,
                       String & Source)
{  // opérateur de saisie dans un flux
   int TailleMax = 80;
   char c;
   String Temp(TailleMax);
   Source = "";
   while ( Flot.get(c) && c != '\n' ) {
      Flot.putback(c);
      Flot.get(Temp.chaine,Temp.tailleMem,'\n');
      Temp.nbCarac = strlen(Temp.chaine);
      Source = Source + Temp;
   }
   return Flot;
}
```

```
ostream & operator << (ostream & Flot,
                       const String & Source)
{  // opérateur de sortie dans un flux
   return Flot << " Chaine :\"" << Source.chaine << "\" de "
               << Source.nbCarac << " et pouvant en contenir "
               << Source.Capacite();
}

void String::AfficheVarClasses(ostream & Flot)
{  // fonction d'affichage des variables de classe
   Flot << String::nbInstances << " instances occupant "
        << String::memInstances << " octets" << endl;
}

String String::SubString(int Debut, int NbCarac=0) const
{  // extrait une sous-chaîne de NbCarac depuis la position Debut.
   NbCarac = ( NbCarac > 0
                  ? min(NbCarac,nbCarac)
                  : nbCarac);
   String Temp(NbCarac);
   strncpy(Temp.chaine, chaine+Debut, NbCarac);
   Temp.chaine[NbCarac] = '\0';
   Temp.nbCarac = NbCarac;
   return Temp;
}

int String::StringString(const String & Cherche) const
{ /* renvoie la position du début de la chaîne Cherche dans l'objet récepteur, ou
     une valeur négative si Cherche n'est pas trouvé. */
   return (strstr(chaine, Cherche.chaine)-chaine);
}

int String::StringChar(char Carac) const
{ /* retourne la position du caractère recherché Carac ou une valeur négative si
     ce caractère n'est pas trouvé. */
   return (strchr(chaine,Carac)-chaine);
}

void String::ToUpper()
{  // met la chaîne de caractère pointée par le membre chaine en majuscules
   if ( nbCarac == 0 ) return;
   for ( char * Ptr = chaine; * Ptr != '\0'; Ptr ++ ) {
      if ( isalpha(*Ptr) ) *Ptr = toupper(*Ptr);
      else
         if (! isdigit(*Ptr)) *Ptr = ' ';
   }
   Ptr--;
   if ( * Ptr == ' ') { * Ptr = '\0'; nbCarac--; }
}
```

4 Utiliser une classe existante : l'héritage

Après avoir décrit les mécanismes de base de la programmation orientée objet en C++ et les avoir utilisés pour implémenter une classe, nous allons voir comment réutiliser une classe existante pour l'adapter à nos besoins. Nous utiliserons pour cela un mécanisme de base des langages orientés objets : l'héritage.

Nous prendrons pour exemple la gestion de produits chez un distributeur. Nous nous limiterons aux caractéristiques essentielles des produits mis en vente. Nous décrirons d'abord la classe de base, Produit, dont les objets seront des produits commercialisables, décrits par leur libellé et leur prix. Nous verrons ensuite comment réutiliser le code de cette classe pour implémenter de nouvelles classes d'objet peu différentes de Produit, comme celles traitant de produits périssables (nous ajouterons la notion de délai de consommation) ou de produits saisonniers (saison de vente).
En fin de chapitre, nous détaillerons les mécanismes permettant un héritage multiple, avec la classe ProduitFugace.

4.1 - Héritage : construire et utiliser la classe de base

La classe Produit doit nous permettre de manipuler des objets produits, dont les caractéristiques seront pour l'instant le nom et le prix. S'il s'avère nécessaire de manipuler par la suite une catégorie particulière de produits, comme les produits périssables, il sera toujours possible de redéfinir une nouvelle classe, ProduitPerissable, dont la structure et les méthodes s'inspireront largement de Produit, auxquelles s'ajouteront une donnée caractérisant le délai de consommation et des fonctions utilisant cette donnée supplémentaire. Lorsqu'il faudra traiter des produits mis en vente à des périodes bien ciblées (moufles, maillot de bain, parasol...), il faudra de nouveau créer une nouvelle classe, ProduitSaisonnier, proche de Produit, que l'on complétera simplement avec la saison de vente.

Ainsi, à partir d'une classe initiale, ou classe de base, nous réutilisons une grande partie d'un code existant, en le complétant par quelques fonctions, pour

construire de nouvelles classes proches de la première et s'en distinguant par l'ajout d'une ou de quelques propriétés (données ou fonctions membres).

Les langages orientés objets, comme C++, offrent au programmeur un mécanisme permettant de réutiliser le code de la classe de base, en le complétant (ou le modifiant légèrement) pour obtenir de nouvelles classes, qu'on appelle classes dérivées. Ce mécanisme est l'héritage.

L'héritage évite la redondance de code, puisqu'on réutilise directement le code de la classe de base pour toutes les fonctions qui ne nécessitent pas d'être modifiées. Il évite également de surcharger inutilement la classe de base en y définissant toutes les données et fonctions imaginables dans la famille d'objets dont elle pourrait être la source, mais laisse aux classes dérivées le soin d'implémenter et d'approfondir telle ou telle particularité. Il facilite enfin la maintenance du code, puisque les fonctions communes à toutes les classes (de base et dérivées) sont rassemblées dans la classe de base, et toute modification du corps d'une de ces fonctions (du moment que l'interface de la fonction reste identique) se répercute automatiquement dans la hiérarchie des classes dérivées.

Nous verrons dans un premier temps la définition de la classe de base utilisée (Produit), avant de construire une classe dérivée (ProduitPerissable) pour illustrer trois mécanismes de l'héritage :

- l'accès aux données définies dans la classe de base ;
- l'ajout de nouveaux membres ;
- la modification des fonctions membres héritées.

4.1.1 - Une classe de base : *Produit*

La définition de la classe Produit reste sommaire (figure 4.1). Nous y trouvons deux données membres privées : nom et prix.

La fonction membre privée fixeNom permet l'initialisation de la donnée membre nom, en tronquant les chaînes de caractères qui dépassent les NbMaxCarac autorisés. Sa définition est donnée à l'extérieur de celle de la classe Produit.

Le seul constructeur dont nous disposons attend en arguments un nom de produit et son prix. Il fait appel à la fonction fixeNom de manière à sécuriser l'initialisation de la donnée membre nom. Il n'y a pas de redéfinition du destructeur, des constructeurs par copie ni de l'opérateur d'affectation. Le choix d'un tableau de dimension fixe pour coder le nom du produit, préféré à un pointeur sur char, nous permet d'éviter ces redéfinitions qui ne sont pas l'objet de ce chapitre.

Nous avons défini deux accesseurs permettant de consulter les valeurs des données membres nom et prix : ce sont les fonctions Nom et Prix.

Il nous reste à citer une dernière fonction, AfficheToi simplement chargée d'afficher le descriptif du produit (nom et prix).

Ainsi construite, la classe Produit nous permet juste de créer des produits, puis de leur demander d'afficher leurs caractéristiques principales, comme dans l'exemple ci-dessous :

```
Produit P1("Chocolat", 5.45);
P1.AfficheToi();
```

```
const NbMaxCarac = 25 ;

class Produit {
 private:
   char nom[NbMaxCarac+1]; // décrit le produit
   float prix; // indique le prix du produit
   void fixeNom(const char * texte);
 public:
   Produit(const char * Nom, float Prix)
   {
      fixeNom(Nom);
      prix = Prix;
   }
   float Prix() const { return prix; } // accesseur de prix
   const char * Nom() const { return nom; } // accesseur de nom
   void AfficheToi()
   {
      cout << "Produit " << nom << "\n\tprix: " << prix
           << endl;
   }
}; // fin définition classe Produit

void Produit::fixeNom(const char * Texte)
{  // copie au maximum les NbMaxCarac premiers caractères de Texte dans le
   // membre nom du produit
   strncpy(nom, Texte, NbMaxCarac);
   nom[NbMaxCarac] = '\0'; // si la chaîne copiée était trop longue
} // void fixeNom
```

Figure 4.1 : définition de la classe de base Produit

4.1.2 - Un premier exemple d'héritage : *ProduitPerissable*

Etudions maintenant l'implémentation de la classe dérivée, ProduitPerissable, concernant toujours des produits vendus dans le magasin, mais nécessitant l'entrée d'une information supplémentaire : la durée de conservation (delai). Nous allons utiliser le mécanisme de l'héritage pour simplifier la construction de cette nouvelle classe.

Lien d'héritage entre deux classes

A partir de la classe de base Produit, nous créons une classe dérivée qui hérite des membres (données et fonctions) de la classe de base. Comme nous le voyons dans l'exemple donné figure 4.2, il n'est pas nécessaire de mentionner à nouveau les données membres héritées.

Le lien d'héritage se fait en précisant dans l'en-tête de définition de la classe dérivée le nom de la classe de base :

```
class ProduitPerissable : public Produit
```

```
class ProduitPerissable : public Produit {
 private:
   int delai; // durée de conservation
 public:
   ProduitPerissable (const char * Nom,
                      int Delai,
                      float Prix)
     : Produit (Nom, Prix)
       { delai = Delai; }
   int Delai() const { return delai; } // accesseur de delai
   void AfficheToi()
   {
      Produit::AfficheToi();
      cout << "\tvalidité : " << delai << " jours\n";
   }
}; // fin classe ProduitPerissable
```

Figure 4.2 : une première classe héritée de Produit *:* ProduitPerissable

Plus généralement, on dérive une classe DDD d'une classe de base BBB en mentionnant le lien entre les deux classes dans l'en-tête de définition de la classe dérivée :

```
class DDD : <spécificateur> BBB
```

Le terme <spécificateur> précise l'étendue des droits d'accès aux membres hérités de la classe de base. Ce spécificateur est soit public, soit private. Avec le spécificateur public :

- les membres private de la classe de base (Produit) sont inaccessibles depuis la classe dérivée ;
- les membres public de la classe de base restent public dans la classe dérivée.

Nous décrirons plus loin les effets du spécificateur private (Cf. 4.1.3).

Constructeur de la classe dérivée

Le constructeur de Produit ne convient pas dans ProduitPerissable. La donnée membre delai n'y est pas initialisée, puisqu'elle n'existe pas au niveau de la classe de base. Il faut donc définir un constructeur adapté, qui réutilisera quand même le constructeur de la classe de base pour initialiser les données membres héritées.

L'appel au constructeur de la classe de base se fait en précisant, dans l'en-tête du constructeur de la classe dérivée, le constructeur que l'on désire utiliser. Il faut évidemment transmettre au constructeur de la classe de base les paramètres appropriés :

```
ProduitPerissable (const char * Nom, int Delai, float Prix)
    : Produit (Nom, Prix) // liste d'initialisation
    { delai = Delai; }
```

Le constructeur de ProduitPerissable attend trois arguments (le nom, le délai et le prix). Sur ces trois arguments, deux servent à initialiser des données membres héritées (Nom et Prix). Ces deux arguments sont donc immédiatement transmis au constructeur de la classe de base, dont *l'appel précède le corps du constructeur*. Le troisième argument (Delai) est utilisé dans le corps du constructeur pour initialiser la nouvelle donnée membre.

Le constructeur hérité (Produit(Nom, Prix)) est le seul élément de la liste d'initialisation du constructeur de la classe dérivée. On appelle *liste d'initialisation* la suite d'expressions permettant d'affecter des valeurs aux données membres ou de construire les parties héritées, avant d'exécuter le corps de la fonction. La liste d'initialisation est utilisée par le constructeur lorsqu'il prépare les emplacements mémoires correspondant aux données membres ou aux parties héritées. Ces emplacements sont immédiatement occupés par les valeurs initiales. La liste d'initialisation est requise dans certains cas :

- appel explicite d'un constructeur hérité ;
- initialisation d'une donnée membre déclarée comme constante (Cf. 4.2.1) ;
- initialisation de données membres déclarées comme objets de classes définies sans constructeur par défaut.

Dans les deux derniers cas, l'absence de liste d'initialisation obligerait le constructeur à créer des données membres indéterminées, puis à les initialiser dans le corps du constructeur : la compilation échouerait d'abord sur l'impossibilité de définir des objets indéterminés (cas de données membres appartenant à une classe qui n'aurait pas de constructeur par défaut) puis sur une tentative de modification de constante (cas des données membres constantes).

La donnée *delai* n'est pas une constante, mais nous pourrions modifier la définition du constructeur de manière à l'introduire dans une liste d'initialisation :

```
ProduitPerissable (const char * Nom,
                   int Delai,
                   float Prix)
    : Produit(Nom, Prix), delai(Delai)
    {}
```

Dans ce cas précis, le corps du constructeur est vide, car il n'y a plus aucune opération d'initialisation à effectuer.

Pour terminer nos commentaires sur la liste d'initialisation, reprenons l'exemple du chapitre 2 concernant une classe Memoire dont les objets disposent de trois données membres privées : *donnees*, *code*, et *pile*, qui sont toutes les trois des instances de la classe TableEntiers.

Les constructeurs de Memoire peuvent avantageusement utiliser la liste d'initialisation pour initialiser les données membres :

```
Memoire::Memoire (int taille)
 : donnees (32768),
   code (32768),
   pile (32768)
   { ...// etc. }
```

Nous avons vu au chapitre 2 différentes manières de préparer correctement les données membres *donnees*, *code* et *pile* :

- laisser le compilateur initialiser les données membres avec le constructeur par défaut de TableEntiers puis appliquer dans le corps du constructeur Memoire une fonction pour changer les caractéristiques des objets TableEntiers ;
- générer dans le corps du constructeur Memoire des objets TableEntiers temporaires puis les affecter aux données membres ;
- modifier le constructeur par défaut de TableEntiers pour qu'il s'adapte aux données membres de Memoire.

La liste d'initialisation appelle directement le constructeur ad hoc de TableEntiers. Elle évite la surcharge du constructeur de la classe Memoire. Elle évite la génération d'objets temporaires. Elle épargne au concepteur d'une autre classe la nécessité de modifier son constructeur par défaut.

Fonction redéfinie

De même que le constructeur de la classe dérivée intègre la nouvelle donnée membre *delai*, la fonction AfficheToi de la classe de base est insuffisante car elle n'affiche pas cette donnée. Pour éviter la redondance du code d'affichage des parties héritées (code existant dans la fonction AfficheToi de Produit), la nouvelle fonction d'affichage s'appuie sur la fonction héritée de Produit. Contrairement aux constructeurs, l'appel d'une fonction héritée ne se fait pas avant, mais dans le corps même de la fonction :

```
void AfficheToi() { Produit::AfficheToi(); ... }
```

Il est indispensable de préciser avec l'opérateur de portée (Produit::) la version de la fonction AfficheToi qui doit être utilisée. Sans cette précision, la définition serait récursive (AfficheToi s'appelant elle-même) et serait à l'origine d'une boucle infinie.

4.1.3 - Problèmes d'accès aux membres de la classe de base

La structure hiérarchique des classes de C++ favorise les développements séparés. On distingue ainsi le programmeur qui élabore une classe de base (concepteur de la classe de base), le programmeur qui utilise cette classe pour élaborer d'autres classes dérivées (concepteur des classes dérivées), et le programmeur qui ne fait qu'utiliser les classes fournies (utilisateur des classes). Bien sûr, une seule personne peut cumuler les trois rôles. En règle générale, le contrôle des

droits d'accès aux membres privés de la classe de base se fait par le concepteur de cette classe de base, en utilisant les spécificateurs private, protected ou friend. Le contrôle d'accès aux données publiques de la classe de base est ensuite réalisé par le concepteur des classes dérivées qui dispose des spécificateurs d'héritage private et public. Quant à l'utilisateur des classes, il dispose de l'interface publique que les deux premiers ont bien voulu lui laisser... Etudions maintenant les différents modes d'héritage.

Mode d'héritage public

C'est l'héritage que l'on obtient pour la classe DDD quand elle est héritée de la classe BBB avec :

```
class DDD : public BBB { ... /* etc.*/ };
```

Les membres private sont inaccessibles dans la classe dérivée. Cette limitation permet au concepteur de la classe de base de protéger son implémentation quand il fournit seulement le fichier d'en-tête et le module compilé de sa classe. L'utilisateur de la classe de base est alors obligé d'accéder aux membres privés en utilisant l'interface publique préparée par le concepteur, interface normalement validée pour éviter toute mauvaise manipulation des données privées. Et le concepteur de la classe dérivée est dans la même situation : dans aucune des fonctions membres qu'il écrit, il n'a accès aux membres private de la classe de base.

Supposons qu'il veuille passer outre, et décide de modifier la donnée membre prix de la classe ProduitPerissable. Sensible aux moisissures vertes qui se développent sur ses yaourts, il s'empresse de définir une fonction publique Brade, permettant de diminuer le prix de ses denrées de manière persuasive :

```
float ProduitPerissable::Brade(int Pourcent)
// cette fonction applique une ristourne de Pourcent % lorsque le produit reste
// invendu juste avant d'être périmé. Très mal écrite, elle ne vérifie même pas la
// validité du pourcentage donné en argument. Le blocage de la compilation s'en
// trouve justifié
 { prix = prix*(100 - Pourcent) / 100; return prix; }
```

La compilation échoue en signalant que la donnée prix est inaccessible (et c'est tant mieux...)

Bien que déclarée comme fonction membre publique de ProduitPerissable (et censée pouvoir accéder à tous les membres privés de la classe), Brade ne peut manipuler la donnée privée prix, héritée de la classe de base Produit. Pour donner l'accès à cette donnée, le concepteur de la classe de base dispose de deux possibilités :

- il déclasse la donnée de private en protected ;
- il implémente un accesseur en modification dans la classe de base, qui permettra (après validation) la mise à jour de la donnée membre.

Etudions chacune de ces deux approches :

Spécificateur protected

La première solution introduit le spécificateur d'accès protected :

```
class Produit {
 protected:
   float prix;
   ... // etc.
```

Le spécificateur protected a un sens proche de private : un membre défini avec l'attribut protected dans la classe CCC est privé pour CCC. Mais, pour les classes qui héritent avec le mode public de CCC, ce membre reste protected. Ainsi, les classes dérivées accèdent aux données membres protected de leur classe de base comme si elles étaient définies en protected chez elle (équivalence d'accès à private). En cas d'héritage en cascade, le membre protected de la classe de base initiale se transmet en protected dans toutes les classes qui hériteraient des classes héritières, et ainsi de suite.

Le concepteur des classes dérivées doit alors garantir l'intégrité des données membres héritées avec l'accès protected et veiller à ce que ses propres fonctions d'accès n'introduisent aucun effet parasite sur les données membres (comme en provoquerait un pourcentage négatif donné en argument à la fonction Brade).

Accesseur en modification

Avec cette seconde solution, le concepteur de la classe de base doit fournir une fonction membre publique permettant de modifier la donnée membre privée. Cette fonction membre est un accesseur qui se charge alors des vérifications préalables à la modification de la donnée (figure 4.3).

```
void Produit::ChangePrix(float UnPrix)
{ // fonction membre publique de la classe Produit
   if ((UnPrix < 0) || ( UnPrix > 1000)) {
      cerr << "Votre nouveau prix ne parait pas convenir aux règles de "
           << "l'économie libérale - "
           << "Retournez a la case départ sans passer par la banque.";
      exit (1) ;
   }
   else prix = UnPrix;
} // fin fonction ChangePrix

float ProduitPerissable::Brade(int Pourcent)
{  // fonction membre publique de la classe ProduitPerissable
   float x;
   x = Prix()*(100 - Pourcent) / 100;
   ChangePrix(x);
}  // fin fonction Brade
```

Figure 4.3 : utiliser les accesseurs pour manipuler une donnée privée

La fonction Brade utilise les accesseurs en consultation (Prix) et en modification (ChangePrix) pour recalculer et définir le nouveau prix de vente du produit périmé (ou en passe de l'être). Ces accesseurs étant définis comme fonctions membres publiques, ils sont alors utilisables depuis la classe dérivée.

Les deux solutions que nous venons d'étudier sont défendables. S'il est souvent utile de définir des accesseurs en modification pour ses données privées, il n'est pas évident, au moment de la conception de la classe de base, de connaître à l'avance tous les tests requis par les futures classes dérivées pour valider la modification d'une donnée membre. Dans notre exemple, borner le prix à 1000 est probablement moins judicieux que de déclasser prix en protected.

Mode d'héritage private

Nous avons vu, dans l'exemple précédent, une classe dérivée de Produit en utilisant le spécificateur d'accès public. Il est possible de dériver une classe en utilisant le spécificateur private. L'utilisation des membres hérités de la classe de base est plus restreinte qu'avec le spécificateur public. Dans le cas de la classe ProduitPerissable définie figure 4.4, les membres privés de Produit sont toujours inaccessibles, et les membres protected ou public de Produit deviennent private une fois hérités par ProduitPerissable.

```
class ProduitPerissable : private Produit {
 private:
   int delai; // durée de conservation
 public:
   ProduitPerissable (const char * Nom,
                      int Delai,
                      float Prix)
     : Produit (Nom, Prix)
       { delai = Delai; }
}; // fin classe ProduitPerissable
```

Figure 4.4 : classe dérivée en mode private

L'utilisation d'une fonction publique de Produit est donc impossible à l'extérieur de la définition de la classe dérivée, comme en témoigne l'erreur de compilation provoquée par cet exemple :

```
ProduitPerissable P1 ("yaourt", 4, 1.25);
P1.AfficheToi(); // erreur de compilation
```

Comme la fonction AfficheToi n'est pas redéfinie dans ProduitPerissable (figure 4.4), le compilateur ne pourrait utiliser que la définition héritée de la classe de base : celle-ci est inaccessible car devenue private avec ce type d'héritage.

On considère habituellement que les membres privés d'une classe correspondent à son implémentation (manière dont fonctionne la classe en interne), alors que ses membres publics en sont l'interface.

Avec un héritage de type private, le concepteur de la classe dérivée récupère l'interface de la classe de base et s'en sert comme implémentation pour sa classe dérivée, à laquelle il ajoute sa propre interface. De cette manière, le concepteur

d'une hiérarchie de classes arrive à masquer complètement implémentation et interface de la classe de base, en ne laissant accessible au programmeur que l'interface personnalisée de chaque classe dérivée.

Prenons comme exemple une classe d'objets conteneurs d'entiers, TableEntiers. Cette classe va être utilisée comme classe de base par un concepteur de classe dérivée qui désire implémenter un système de gestion de concours. La maquette du programme doit permettre au gestionnaire de consulter une liste de candidats à un concours afin de renseigner (ou de consulter) le rang de sortie de chacun d'entre eux. Le concepteur du programme décide d'associer dans un même objet les rangs de sortie aux codes de chaque étudiant (numéro de sécurité sociale, numéro d'ordre arbitraire ou tout autre code permettant de représenter de manière unique un candidat).

Pour cela, il choisit de s'appuyer sur une classe existante, TableEntiers. Comme les objets de cette classe n'ont qu'une seule donnée membre de type tableau d'entiers (en fait un pointeur sur entiers), il crée une classe dérivée, OrdreConcours, à laquelle il ajoute un deuxième tableau (d'entiers ou tout autre type correspondant au codage des étudiants). La donnée membre héritée de TableEntiers (tab) permet de stocker des entiers. Elle sera donc utilisée pour le rang de chaque candidat, puisque ce rang ne peut être qu'un entier positif (jusqu'à nouvel ordre...). La donnée membre de OrdreConcours (code) servira à stocker le code de chaque candidat. Son type sera défini par le cahier des charges (typeCode pour le moment). Ainsi, pour obtenir la place d'un candidat donné, il suffira de rechercher sa position dans le tableau pointé par rang puis d'extraire son rang à la même position dans le tableau pointé par tab.

Le concepteur de la classe dérivée rajoute enfin des fonctions membres privées (renvoieCode qui permet de retrouver le code d'un étudiant à partir de son nom, adresse ou tout autre renseignement, selon le cahier des charges) ou publique (RenvoieRang qui renvoie le rang de sortie d'un candidat) pour compléter son implémentation.

Le concepteur de la classe dérivée pourra s'appuyer sur une classe de base TableEntiers définie de la façon suivante:

```
typedef typeCode int;
// ... si le cahier des charges donne un code entier à chaque étudiant.

class TableEntiers {
 private:
   int * tab; // tableau d'entiers, usage indéfini
   //etc.
 public:
   int & operator [] (int Rang);
   // etc.
};
```

pour développer sa classe OrdreConcours. Cette classe dérive de Tableau en mode private :

```
class OrdreConcours : private TableEntiers {
 private:
   typeCode * code; // tableau stockant le code des candidats
    // la donnée membre héritée tab servira à stocker le rang de chaque candidat
   typeCode renvoieCode(char * Nom);
    // recherche le code d'un étudiant.
    //etc.
 public:
   RenvoieRang(char * Nom); // utilise [] pour chercher le rang
   // etc.
}
```

Le concepteur de OrdreConcours peut utiliser l'opérateur [] dans ses fonctions membres. L'utilisateur de OrdreConcours ne peut pas utiliser cet opérateur. L'interface de la classe OrdreConcours ne le nécessite pas, puisque ce n'est pas la position réelle des informations dans les tableaux qui intéresse l'utilisateur mais l'association de deux informations (le code – ou nom – de l'élève et son rang de sortie). L'accès au tableau se fera de manière transparente, par l'intermédiaire de la fonction membre RenvoieRang.

Utilisation du spécificateur friend dans la classe de base

```
class Produit {
 private:
   char nom[NbMaxCarac+1];
   float prix;
   void fixeNom(const char * texte);
 public:
   Produit(const char * Nom, float Prix);
   float Prix() const ;
   const char * Nom() const ;
   void AfficheToi();
 friend class ProduitPerissable;
}; // fin définition classe Produit
```

Figure 4.5 : extension explicite des droits d'accès de ProduitPerissable

Le concepteur de la classe de base peut également déclarer explicitement les classes dérivées qui auront le droit d'accéder aux membres privés de cette classe de base. Il utilise pour cela le spécificateur friend (figure 4.5). Dans la classe Produit, la définition :

```
friend class ProduitPerissable;
```

permettra à la classe ProduitPerissable, dérivant de Produit en mode public, d'accéder aux données nom et prix et à la fonction fixeNom comme à ses propres données et fonction privées. Par contre, l'inaccessibilité reste maintenue pour toutes les autres classes dérivées de Produit.

Récapitulatif sur l'héritage des accès

Considérons une classe de base Base, contenant des membres private, protected et public. Cette classe déclare amie sa classe dérivée DeriveFriend. Supposons l'existence des classes dérivées suivantes :

```
class DerivePublic  : public  Base {...};
class DerivePrivate : private Base {...};
class DeriveFriend  : public  Base {...};
```

Pour la classe DerivePublic :

- les membres private hérités de Base sont inaccessibles ;
- les membres protected hérités de Base sont protected (équivalent en accès à private) ;
- les membres public hérités de Base sont public.

Pour la classe DerivePrivate :

- les membres private hérités de Base sont inaccessibles ;
- les membres protected hérités de Base sont private ;
- les membres public hérités de Base sont private.

Pour la classe DeriveFriend :

- les membres private hérités de Base sont accessibles (en private) ;
- les membres protected hérités de Base sont protected ;
- les membres public hérités de Base sont public.

4.1.4 - Surcharge d'une donnée membre

De même qu'il est possible de redéfinir une fonction membre dans la classe dérivée (on utilise le même nom que dans la classe de base), on peut définir une donnée membre dans la classe dérivée ayant le même nom qu'une donnée membre de la classe de base. Deux données membres de même nom sont alors présentes dans chaque objet de la classe dérivée. On y accède en qualifiant la donnée héritée par le nom de la classe de base, comme dans l'exemple de la figure 4.6 (Produit::prix et prix sont les deux données membres des objets ProduitPerissable). Malgré la qualification par l'opérateur de portée, la confusion reste grande entre les deux données (utiliser le même nom pour deux données membres dans une hiérarchie de classe n'est pas toujours une bonne idée).

Prenons l'exemple du marché aux melons de Cavaillon. Selon le degré de maturité des fruits, leur aspect plus ou moins engageant, le prix va subir des fluctuations importantes autour d'un prix de base fixé à l'ouverture du marché. Si les prix s'effondrent à la suite d'une mévente prolongée, le gain financier se limitera au prix réel de vente. Par contre, certains paramètres resteront indexés au prix de base : ce serait le cas du fermage ou de la fiscalité forfaitaire. Il apparaît utile de disposer d'une donnée membre prix dans la classe dérivée ProduitPerissable qui reflétera le prix de marché, alors que la donnée héritée de Produit conservera le prix de base (figure 4.6). Les fonctions, comme Taxes qui

accèdent aux données homonymes doivent préciser avec l'opérateur de portée quelle est la donnée qui doit être utilisée.

```
class Produit {
 protected:
   float prix; // prix de base d'un produit
   // etc. Cf. fig. 4.1...
}; // fin définition classe Produit

class ProduitPerissable : public Produit {
 private:
   float prix; // prix fonction de l'état du produit
   // etc. Cf. fig. 4.2
 public:
   float Taxes();
...
}; // fin définition classe ProduitPerissable

 float ProduitPerissable::Taxes()
 {
   float ttc1 = Produit::prix*1.186; // prix = membre de Produit
   float ttc2 = prix*1.186; // prix = membre de ProduitPerissable
   // etc.
 }
```

Figure 4.6 : redéfinition d'une donnée membre

Pour les fonctions membres redéfinies, l'accès à la fonction de la classe de base se fait également en utilisant l'opérateur de portée. Nous l'avons vu figure 4.2, quand la redéfinition de AfficheToi faisait explicitement appel à la fonction AfficheToi de la classe de base. Nous pouvons dans la suite du programme faire référence à l'une ou l'autre des versions de AfficheToi du moment que l'objet à qui s'adresse le message correspondant à cette fonction appartient bien à la classe ProduitPerissable :

```
ProduitPerissable P1 ("yaourt", 4, 1.25);
P1.AfficheToi(); // affiche le nom, le prix et le délai de consommation
P1.Produit::AfficheToi(); // n'affiche que le nom et le prix
```

4.2 - Héritage des fonctions membres et des opérateurs

4.2.1 - Constructeurs

L'exemple donné figure 4.6 suppose que la donnée membre prix est correctement initialisée. Le constructeur est souvent utilisé pour cette opération. Contrairement aux autres fonctions, les constructeurs ne sont jamais hérités. Pour préparer les données membres héritées, le constructeur de la classe de base doit être invoqué (explicitement, comme dans la figure 4.2) ou implicitement (constructeur

par défaut). Dans tous les cas, le constructeur de la classe de base est invoqué avant celui de la classe dérivée. Si la hiérarchie des classes se poursuit, les constructeurs seront appelés dans l'ordre descendant. A l'inverse, les appels de destructeurs suivront l'ordre ascendant de la hiérarchie des classes (nous verrons plus en détail au paragraphe 4.2.5 le fonctionnement des destructeurs).

Si plusieurs constructeurs sont définis dans la classe de base, ou bien s'il n'existe plus de constructeur par défaut pour cette même classe, le programmeur doit préciser, dans une liste d'initialisation, quel constructeur doit invoquer celui de la classe dérivée :

```
ProduitPerissable (const char * Nom,
                   int Delai,
                   float Prix)
    : Produit (Nom, Prix)
      {...}
```

Dans l'exemple ci-dessus (tiré de la figure 4.4), la liste d'initialisation fait référence au seul constructeur de la classe de base, en lui fournissant comme arguments deux des paramètres passés au constructeur de la classe dérivée.

```
class ProduitSaisonnier : public Produit {
 private:
   const char * const saison;
 public:
   ProduitSaisonnier( const char * Nom,
                      const char * Saison,
                      float Prix)
       : Produit(Nom, Prix), saison(Saison)
       {}
}; // fin classe ProduitSaisonnier
```

Figure 4.7 : constructeur ProduitSaisonnier, *avec liste d'initialisation*

La liste d'initialisation peut être complétée par l'initialisation des données membres, comme le montre la définition d'une deuxième classe dérivée descendant de *Produit* (figure 4.7). Une donnée membre supplémentaire, *saison*, a été définie comme un pointeur constant sur une chaîne de caractères constante (Cf. 2.7.5). Avec une telle définition, le pointeur et la chaîne de caractères doivent être définis une fois pour toutes. On doit donc passer par la liste d'initialisation, qui attribue une valeur à la donnée membre *saison* dès sa définition. Sans cette liste d'initialisation, le compilateur bloquerait toute tentative d'affectation d'une nouvelle (et même première !) valeur au pointeur constant *saison*.

Ce que nous venons d'expliquer sur l'utilisation en cascade des constructeurs dans une hiérarchie de classe doit être pris en compte lors de la définition des constructeurs d'une classe dérivée :

- si le constructeur de la classe de base n'est pas explicitement cité dans la liste d'initialisation, il faut s'assurer qu'il existe un constructeur par défaut (ou un constructeur défini sans aucun argument) dans la classe de base, puisque c'est ce constructeur qui va être automatiquement invoqué par le compilateur ;

– s'il n'y a aucun constructeur explicitement défini dans la classe dérivée, il faut évidemment garder (ou redéfinir) le constructeur par défaut de la classe de base.

Ajoutons un constructeur par défaut à la classe Produit (figure 4.8) de manière à pouvoir exécuter une instruction telle que :

```
Produit P;
```

```
Produit::Produit()
{ // construit un produit aux caractéristiques indéterminées
   prix = 0;
   fixeNom("Dénomination indéterminée ");
}
```

Figure 4.8 : un constructeur par défaut pour la classe Produit

Malgré la présence du constructeur de la figure 4.8, on pourra noter que les instructions :

```
ProduitPerissable PP;
ProduitSaisonnier PS;
```

sont refusées par le compilateur, faute de constructeur par défaut défini dans ces deux classes (ProduitPerissable et ProduitSaisonnier). Les classes dérivées sont bien dans l'incapacité d'hériter des constructeurs de la classe de base.

Nous n'avons pour le moment pas redéfini l'opérateur d'affectation (l'héritage de cet opérateur sera étudié un peu plus loin, en 4.2.3). Supposons que nous voulions maintenant exécuter les instructions suivantes :

```
Produit P ("chocolat", 4.50);
ProduitPerissable PP ("Yaourt", 12, 1.25);
P = PP;
```

Dans une affectation impliquant un objet-source et un objet-cible de même classe, le compilateur copie membre à membre les données de la source sur la cible. Dans l'exemple ci-dessus, l'objet-source PP appartient à une classe dérivée de celle de l'objet cible P. Le compilateur extrait, dans l'objet-source PP, les données membres héritées de Produit pour les copier dans l'objet-cible P (figure 4.9).

Sur la même figure, on voit également l'impossibilité de compiler l'affectation inverse :

```
PP = P;
```

Le compilateur ne peut générer les données manquantes pour remplir la totalité des données membres de PP. L'erreur de compilation signale que le constructeur ProduitPerissable(Produit) n'existe pas. En effet, lorsque l'objet-source est d'un type ou d'une classe différente de celle de l'objet-cible, et que les propriétés de l'héritage ne permettent pas de réaliser quand même l'affectation, le compilateur cherche à convertir l'objet-source dans le type ou la classe de l'objet-cible avant d'effectuer la conversion. Dans le cas d'une affectation vers une instance d'une classe CCC, il cherche un constructeur de CCC acceptant pour argument une valeur du même type que l'opérande droit. S'il le trouve, il génère avec ce constructeur un objet temporaire CCC puis copie membre à membre les données de l'objet

temporaire vers celles de l'objet bénéficiaire de l'affectation (opérande gauche). L'objet temporaire est ensuite détruit[1].

Comme il n'existe pas de constructeur ProduitPerissable (Produit & P), l'affectation PP = P est impossible. Lorsque l'on construit une hiérarchie (Cf. chapitre 5), il peut être utile d'implémenter des constructeurs utilisant comme argument une instance de chaque classe de base.

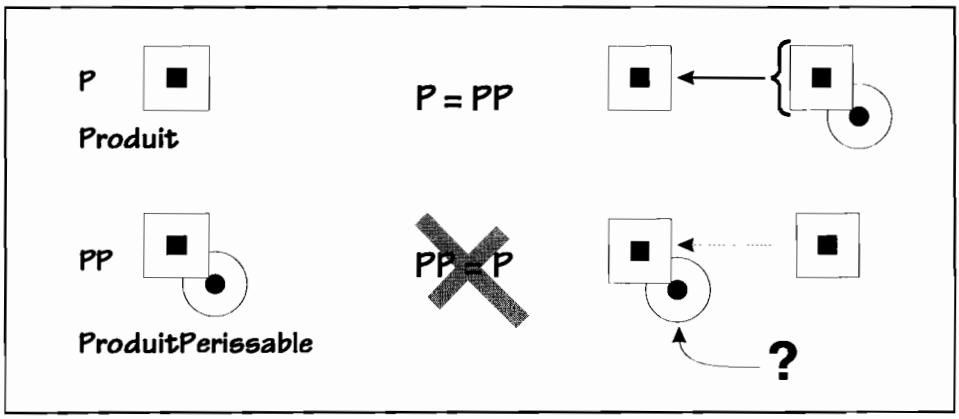


Figure 4.9 : mécanismes de l'affectation entre classe de base et classe dérivée

4.2.2 - Les fonctions virtuelles

Nous disposons maintenant de deux classes dérivées (ProduitPerissable et ProduitSaisonnier) de la classe de base Produit. Complétons la définition de la classe de base par une fonction publique, chargée d'afficher la classe à laquelle appartient l'objet :

```
void Produit::AfficheClasse()
{
   cout << "Ma classe est Produit. " << endl;
}
```

En supposant la déclaration de cette fonction correctement faite dans la définition de Produit, l'exécution de :

```
Produit P1 ("Chocolat", 4.50);
P1.AfficheClasse();
```

provoquera l'affichage de la chaîne :

```
Ma classe est Produit.
```

1 Le chapitre 5 décrit plus en détail les conversions de type entre classes.

Si nous nous en tenons à cette seule définition de la fonction AfficheClasse, les instructions :

```
ProduitPerissable P2 ("Yaourt", 12, 4.50);
P2.AfficheClasse();
ProduitSaisonnier P3 ("Moufles", "Hiver", 120);
P3.AfficheClasse();
```

provoqueront également l'affichage de :

```
Ma classe est Produit.
Ma classe est Produit.
```

Il faut évidemment redéfinir la fonction dans chacune des classes dérivées, de manière à ce qu'elle signale correctement la classe d'appartenance de chaque objet de la hiérarchie (figure 4.10). Avec ces deux redéfinitions de la fonction AfficheClasse, l'affichage de notre exemple précédent devient :

```
Ma classe est ProduitPerissable.
Ma classe est ProduitSaisonnier.
```

```
void ProduitPerissable::AfficheClasse()
{
   cout << "Ma classe est ProduitPerissable. " << endl;
}

void ProduitSaisonnier::AfficheClasse()
{
   cout << "Ma classe est ProduitSaisonnier. " << endl;
}
```

Figure 4.10 : la fonction AfficheClasse *pour les classes dérivées de* Produit

Nous implémentons ainsi le *polymorphisme* du message AfficheClasse() pour les classes Produit, ProduitPerissable et ProduitSaisonnier. Le polymorphisme caractérise la faculté qu'ont des objets de différentes classes d'avoir un comportement semblable en réponse au même message (AfficheClasse() dans notre cas).

Notre problème semble résolu, pour autant qu'à chaque nouvelle classe dérivée, le programmeur pense à redéfinir la fonction AfficheClasse. En réalité, nous pouvons encore trouver un affichage défectueux, en utilisant des pointeurs sur objets des classes Produit, ProduitPerissable et ProduitSaisonnier. Ce cas est illustré figure 4.11. Bien que chaque classe dérivée ait redéfini la fonction AfficheClasse, c'est celle de la classe de base qui est utilisée dans l'instruction :

```
Stock[i]->AfficheClasse();
```

```
Produit P1 ("Chocolat", 4.50);
ProduitPerissable P2 ("Yaourt", 12, 4.50);
ProduitSaisonnier P3 ("Moufles", "Hiver", 120);
Produit * Stock[3] = { &P1, &P2, &P3 };
for (int i = 0; i < 3; i++) Stock[i]->AfficheClasse();

// affichera :
// Ma classe est Produit.
// Ma classe est Produit.
// Ma classe est Produit.
```

Figure 4.11 : polymorphisme incomplet de AfficheClasse

En effet, le compilateur doit préparer un tableau de pointeurs sur Produit. La classe associée à ces pointeurs est donc la classe Produit : toutes les fonctions utilisées sur les objets désignés par ces pointeurs seront tirées de la table des fonctions de Produit. Comme les objets P2 et P3 des classes dérivées ProduitPerissable et ProduitSaisonnier peuvent aussi être considérés comme des instances de la classe Produit (propriété de l'héritage), il est possible d'affecter à des éléments du tableau Stock les adresses de P2 et P3. Par contre, les fonctions associées restent celles de Produit (figure 4.12).

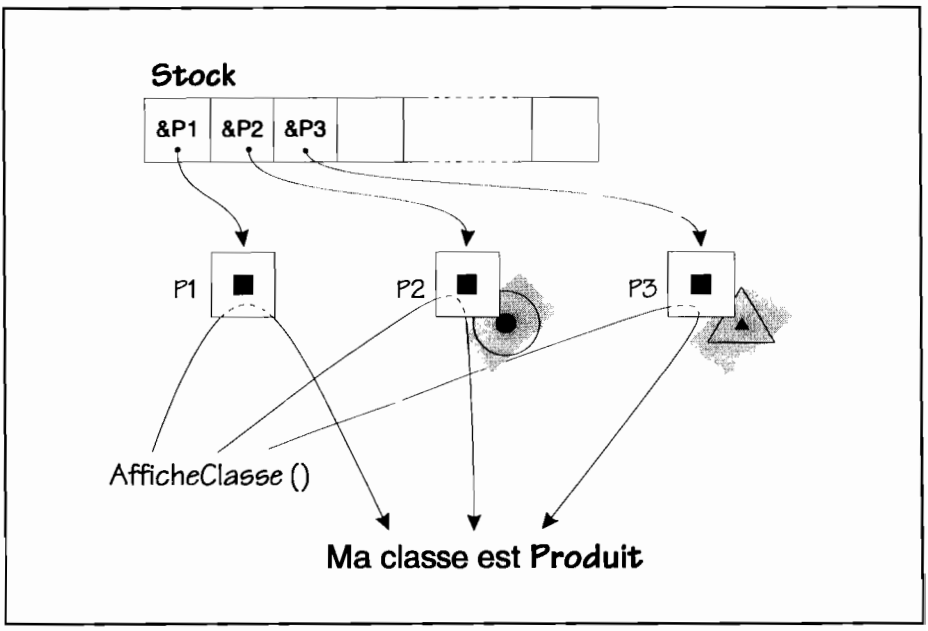


Figure 4.12 : le compilateur choisit toujours AfficheClasse *de* Produit

Pour repousser le choix de la fonction à l'exécution, on doit compléter la déclaration de la fonction AfficheClasse dans la classe de base par le spécificateur virtual :

```
virtual void Produit::AfficheClasse()
{ // AfficheClasse sera aussi virtuelle dans toute classe dérivée de Produit
   cout << "Ma classe est Produit. " << endl;
}
```

Avec cette nouvelle définition de la fonction dans la classe de base, l'exemple donné figure 4.10 affichera :

```
Ma classe est Produit.
Ma classe est ProduitPerissable.
Ma classe est ProduitSaisonnier.
```

Une fonction déclarée virtuelle dans une classe de base, le reste à tous les niveaux de la hiérarchie de classe qui en dérivent (que le spécificateur virtual soit ou non réutilisé dans les surcharges de la fonction).

Avant d'expliquer comment le mécanisme des fonctions virtuelles permet de choisir la bonne fonction au moment de l'exécution, rappelons que, pour un appel de fonction membre non virtuelle, le compilateur génère un appel direct, en utilisant un pointeur sur le code de cette fonction.

En revanche, pour l'appel d'une fonction membre virtuelle, le compilateur va utiliser un appel indirect dont le principe est illustré figure 4.13. Dès qu'une classe *CCC* contient une ou plusieurs fonctions membres virtuelles, la compilation de cette classe génère une table des fonctions virtuelles (*TV* sur notre figure) qui contient, pour chaque fonction virtuelle, un pointeur sur le code de la fonction. De plus, chaque objet de *CCC* contient une donnée membre cachée (notée *PC* sur la figure) qui pointe sur la table des fonctions virtuelles. Si *Obj* est un objet de *CCC* et fv_i une fonction membre virtuelle de cette classe, la compilation de l'appel $Obj.fv_i()$ générera une séquence de code qui utilisera l'indice de la fonction fv_i dans *TV* et la valeur de la donnée membre *PC* pour choisir à l'exécution seulement (puisque la valeur de *PC* ne sera connue qu'à l'exécution) le bon code à exécuter.

On peut déclarer une fonction virtuelle pure f en la définissant sans corps, avec la notation :

```
virtual TypeRenvoyé f(...) = 0;
```

Dans la classe *CCC*, où la fonction est déclarée ainsi, aucune autre déclaration ni définition de f n'est donnée. Le corps de la fonction est donc absent et remplacé par l'écriture *=0*. Dès qu'une classe *CCC* contient au moins une fonction virtuelle pure, elle est *abstraite* : on ne peut instancier aucun objet de cette classe. Une telle classe est utilisée pour rassembler un certain nombre de propriétés communes à différents objets, qui restent malgré tout suffisamment différents pour être répartis entre plusieurs classes qui seront dérivées de *CCC*. Créer une instance dans la classe de base n'a alors pas de sens, puisqu'il s'agit simplement d'un regroupement pratique de quelques propriétés communes à des objets de genre différents. Un grand nombre de bibliothèques de langages orientés objets contiennent des classes abstraites. Les plus connues sont les classes dont héritent des classes de grandeurs comparables (date, entier, flottant, complexe, caractère etc.) ou celles qui dérivent des classes de collections d'objets (*Bag*, *Array*, *Dictionary*...). Nous verrons au chapitre 5 une implémentation d'une classe abstraite *Collection* et de sa descendance.

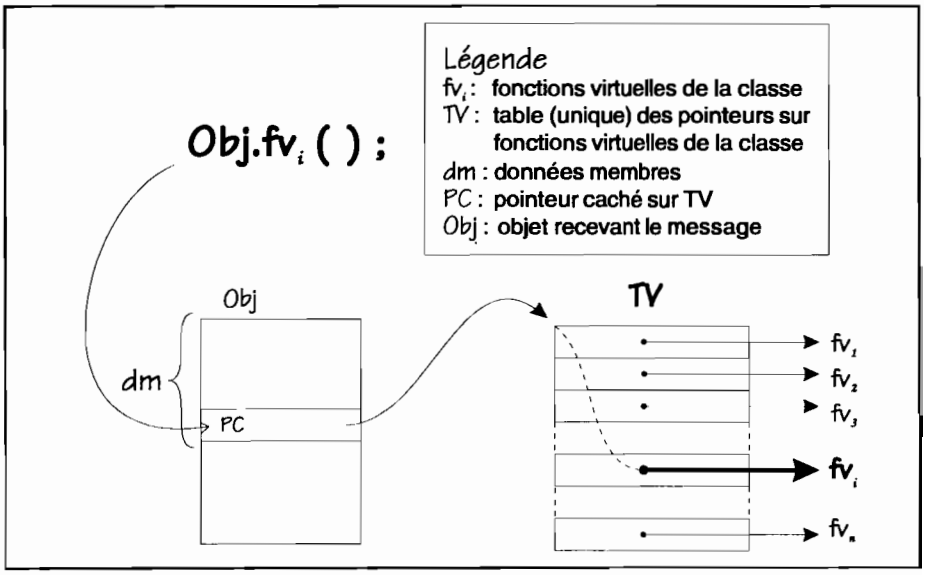


Figure 4.13 : choix d'une fonction virtuelle

Les fonctions virtuelles pures définissent les propriétés qui devront impérativement être redéfinies dans les classes dérivées, pour que des instances puissent être créées dans ces classes. Une classe reste abstraite tant qu'elle contient une fonction virtuelle pure. Examinons par exemple les définitions suivantes :

```
class C {
   virtual void f1 (int X) = 0;
   virtual void f2 (int Y) = 0;
   ... // etc.
};
class SC1 : public C {
   void f1 (int X) {.....}
   ... // etc. f2 n'est pas redéfinie
};
class SC2 : public C {
   void f1 (int X) {.....}
   void f2 (int Y) {.....}
   ... // etc.
};
class SC1A : public SC1 {
   void f2 (int Y) {.....}
   ... // etc.
};
```

La classe SC1 reste abstraite, car elle hérite de la fonction virtuelle pure f2 sans la redéfinir : cette fonction est donc toujours virtuelle pure pour SC1. La classe SC2, qui redéfinit f1 et f2 n'est pas abstraite. La classe SC1A n'est pas abstraite non plus car :

– elle hérite, de SC1, la fonction membre f1 qui n'est pas virtuelle pure,

– elle redéfinit la fonction membre f2 comme une fonction complète.

Remarquons enfin qu'une fonction virtuelle pure peut apparaître à n'importe quel niveau d'une hiérarchie de classes. Si par exemple la classe SC1 définissait la fonction :

```
virtual void f3() = 0;
```

alors toute sous-classe non abstraite de SC1 devrait définir complètement f3.

4.2.3 - Héritage des opérateurs

En ce qui concerne l'héritage, les opérateurs sont comparables aux fonctions : les classes dérivées héritent des opérateurs de la classe de base, avec possibilité de surcharge locale. Les seules exceptions que nous allons détailler concerne les opérateurs qui ne sont pas des fonctions membres de la classe mais sont déclarés comme amis de la classe (comme <<) ou l'affectation.

Opérateurs amis

Nous avons vu au chapitre 3 comment définir l'opérateur << comme ami dans une classe, de manière à ce que le flux de sortie cout (objet ostream) puisse traiter le message inclus dans la séquence suivante :

```
ClasseX obj;
cout << obj;
```

Par défaut, l'opérateur << sait traiter les types prédéfinis. Sans précision supplémentaire, il ignore évidemment comment traiter les objets appartenant aux nouvelles classes créées par le programmeur.

Comme dans le chapitre 3, nous allons ajouter une nouvelle combinaison d'opérandes à celles que << est capable de traiter, en définissant comme fonction amie l'opérateur << (figure 4.14).

Défini comme fonction amie (friend) de Produit, l'opérateur << a accès aux données private et protected de la classe. Nous pouvons donc introduire la donnée membre nom dans son traitement. Avec cette nouvelle définition, l'opérateur << sait réagir aux appels comportant un flux ostream comme opérande de gauche et un objet Produit en opérande de droite. Il sait également réagir aux appels comportant un objet d'une des classes dérivées de Produit en opérande de droite. L'héritage induit une conversion de type implicite de classe dérivée en classe de base. Ainsi, quand un opérateur OP est déclaré ami d'une classe CCC et qu'il est défini par

```
TypeRenvoyé Operator <OP> (CCC X) {.....}
```

les propriétés de l'héritage permettent de l'utiliser avec des objets appartenant aux classes dérivées de CCC.

```
class Produit {
 private:
   void fixeNom(const char * texte);
 protected:
   char nom[NbMaxCarac+1];
   float prix;
 public:
   Produit(const char * Nom, float Prix);
   // etc. Cf. fig. 4.1
   friend ostream & operator << (
      ostream & FluxSortie,
      Produit & P);
}; // fin définition classe Produit

ostream & Produit::operator << (
      ostream & FluxSortie,
      Produit & P)
{
   FluxSortie << "Produit " << P.nom;
   return FluxSortie;
}
```

Figure 4.14 : définition de l'opérateur << comme fonction amie de Produit

Remarquons que, dans la nouvelle définition que nous proposons pour la classe Produit, la donnée membre nom a été déclarée comme membre *protected*. En effet, si nous voulons bénéficier de l'héritage pour utiliser la définition actuelle de l'opérateur << avec toutes les classes dérivées de Produit, il est nécessaire de maintenir l'accès à la donnée nom.

La séquence suivante :

```
Produit P1 ("Chocolat", 12.50);
ProduitSaisonnier P2("Esquimau", "Eté", 15);
ProduitPerissable P3("Huîtres", 4, 42.55);
cout << P1 << endl;
cout << P2 << endl;
cout << P3 << endl;
```

affichera :

```
Produit Chocolat
Produit Esquimau
Produit Huîtres
```

Nous vérifions ainsi que l'opérateur << sait traiter les opérandes appartenant aux classes dérivées de Produit.

De même qu'avec la fonction AfficheClasse étudiée au 4.2.2, le comportement de l'opérateur n'est pas assez précis dans les classes dérivées de Produit. Il est possible de redéclarer, dans chaque classe dérivée, l'opérateur << comme fonction

amie de la classe (figure 4.15) et de le redéfinir avec un comportement adapté à chaque classe.

```
class ProduitPerissable : public Produit {
 // etc. Cf. fig. 4.2
 public:
 // etc. Cf. fig. 4.2
   friend ostream & operator << (
      ostream & FluxSortie,
      ProduitPerissable & P);
}; // fin classe ProduitPerissable

class ProduitSaisonnier : public Produit {
 // etc. Cf. fig. 4.7
 public:
 // etc. Cf. fig. 4.7
   friend ostream & operator << (
      ostream & FluxSortie,
      ProduitSaisonnier & P);
}; // fin classe ProduitSaisonnier

ostream & ProduitPerissable::operator << (
      ostream & FluxSortie,
      ProduitPerissable & P)
{
   FluxSortie << "Produit périssable "
              << P.nom;
   return FluxSortie;
}

ostream & ProduitSaisonnier::operator << (
      ostream & FluxSortie,
      ProduitSaisonnier & P)
{
   FluxSortie << "Produit saisonnier "
              << P.nom;
   return FluxSortie;
}
```

Figure 4.15 : redéfinitions de l'opérateur << pour les classes dérivées de Produit

Maintenant, la séquence examinée précédemment affichera :

```
Produit Chocolat
Produit saisonnier Esquimau
Produit périssable Huîtres
```

Le compilateur dispose maintenant des signatures exactes pour chaque appel de l'opérateur << avec les arguments effectifs P1, P2 et P3. Il n'utilise plus comme auparavant la conversion implicite des arguments effectifs P2 et P3 en Produit.

Par contre, l'exemple de la figure 4.16, inspiré de l'étude de la fonction AfficheClasse (figure 4.11) montre que le polymorphisme de l'opérateur est loin d'être parfait.

```
Produit P1 ("Chocolat", 12.50);
ProduitSaisonnier P2("Esquimau", "Eté", 15);
ProduitPerissable P3("Huîtres",  4,  42.55);
Produit * Stock[3] = { &P1, &P2, &P3 };
for (int i = 0; i < 3; i++) cout << *Stock[i] << endl;

// affichera :
// Produit Chocolat
// Produit Esquimau
// Produit Huîtres
```

Figure 4.16 : polymorphisme incomplet de l'opérateur <<

L'opérateur << est une fonction amie de la classe Produit et de ses classes dérivées. Il ne s'agit pas d'une fonction membre. Sa syntaxe interdit d'ailleurs qu'elle soit jamais fonction membre de la classe Produit :

```
cout << P1;
```

Si l'opérateur << était une fonction membre, alors il serait partie prenante du message << P1 envoyé à l'objet cout. Il s'agirait dans ce cas d'une fonction membre de la classe ostream et non d'une fonction membre de la classe Produit.

Il est donc impossible de définir l'opérateur << comme fonction membre virtuelle de la classe Produit. On pourrait par contre définir une fonction membre opérateur >> déclarée de la manière suivante dans Produit :

```
virtual void operator >> (ostream UnFlux) ;
```

et utilisée dans une instruction comme :

```
P1 >> cout;
```

Le caractère virtuel de la fonction membre opérateur joue alors son rôle et le compilateur repousse le choix de l'opérateur à l'exécution. Ce serait par exemple le cas pour cette instruction inspirée de la figure 4.16 :

```
for (int i = 0; i < 3; i++) *Stock[i] >> cout ;
```

Cependant, la surcharge ainsi proposée pour l'opérateur >> est à déconseiller. En effet, elle rompt la cohérence générale des utilisations de cet opérateur en C++ pour les flux : l'opérateur >> est employée pour les flux d'entrée et non ceux de sortie.

Affectation

Les opérateurs définis comme fonctions membres d'une classe de base sont hérités dans les classes dérivées, comme les autres fonctions membres. La seule

exception à cette règle concerne l'opérateur d'affectation, qui n'est pas à proprement parler hérité, mais complété (un peu comme pour les constructeurs).

Le compilateur fournit par défaut un mécanisme d'affectation. Sans autre précision du programmeur, le compilateur copie membre à membre les données héritées de la classe de base, puis celles de la classe dérivée pour fabriquer une copie conforme de l'objet initial.

Si l'affectation est redéfinie dans la classe de base par le programmeur, le compilateur fournit une affectation par défaut pour chaque classe dérivée qui va :

- appliquer l'affectation surchargée de la classe de base pour les données membres héritées de cette classe ;
- copier membre à membre les données propres à la classe dérivée.

En ce sens, l'affectation a un comportement analogue au constructeur.

Si l'opérateur d'affectation est surchargé dans la classe dérivée, les seuls mécanismes utilisés par le compilateur sont ceux de l'affectation surchargée (figure 4.17).

```
class Produit {
 private:
   void fixeNom(const char * texte);
 protected:
   char nom[NbMaxCarac+1];
   float prix;
 public:
   Produit(const char * Nom, float Prix);
 // etc. Cf. fig. 4.14
   virtual void AfficheToi();
   Produit & operator = (const Produit & P)
   {
      fixeNom(P.nom); prix = P.prix;
      return * this;
   }
}; // fin définition classe Produit
class ProduitPerissable : public Produit {
 private:
   int delai; // durée de conservation
 public:
   ProduitPerissable (const char * N, int Delai, float P);
   void AfficheToi();
   ProduitPerissable & operator =
      (const ProduitPerissable & P)
   { // Danger : cette implémentation oublie les données membres héritées
      delai = P.delai;
      return * this;
   }
}; // fin classe ProduitPerissable
```

Figure 4.17 : redéfinition de l'affectation dans Produit *et sa classe dérivée*

Seule la donnée membre *delai* est traitée dans l'affectation de la classe ProduitPerissable. La séquence suivante :

```
ProduitPerissable P3 ("Huîtres", 4, 42.55);
ProduitPerissable P4 ("Citerne d'huile frelatée",
                      900,
                      12000);
P4 = P3;
P4.AfficheToi();
```

affiche par exemple :

```
Produit périssable : Citerne d'huile frelatée
             prix : 12000
        validité : 4 jours
```

Les membres hérités de la classe de base n'ont pas été copiés car cette opération n'a pas été précisée dans les traitements de l'affectation de ProduitPerissable. Il n'y a pas héritage de l'affectation de la classe de base (qu'elle soit redéfinie par le programmeur ou proposée par défaut par le compilateur). En ce sens, l'affectation a un comportement de fonction membre.

Pour pouvoir utiliser l'affectation définie dans la classe de base, il faut y faire appel explicitement en utilisant l'opérateur de portée (figure 4.18).

```
ProduitPerissable & ProduitPerissable::operator =
   (const ProduitPerissable & P)
{
   Produit::operator = (P);
   delai = P.delai;
   return * this;
}
```

Figure 4.18 : affectation de ProduitPerissable *utilisant l'affectation de* Produit

L'oubli du qualificatif Produit:: provoquerait une récursivité de l'opérateur = et une boucle infinie à l'exécution : sans l'opérateur de portée, l'opérateur = serait pris par le compilateur comme celui de la classe ProduitPerissable.

4.2.4 - Transtypage et héritage

Nous avons montré à plusieurs reprises la capacité d'un objet d'une classe dérivée à réagir comme un objet de la classe de base, vis à vis du compilateur. Avec cette seule déclaration de l'opérateur << (dans la classe Produit) :

```
friend ostream & operator << (ostream & FluxSortie,
                              Produit & P);
```

les opérations suivantes étaient possibles :

```
ProduitSaisonnier P2("Esquimau", "Eté", 15);
ProduitPerissable P3("Huîtres", 4, 42.55);
cout << P2;
cout << P3;
```

Lorsque le compilateur rencontre l'instruction cout << P2, il cherche un opérateur dont la signature correspond à ostream, ProduitSaisonnier. S'il ne le trouve pas, il recherche la signature la plus proche, en utilisant les propriétés de l'héritage (un objet appartient à sa classe, mais peut être aussi considéré comme un objet des classes dont la sienne est une descendante).

L'héritage induit une règle de « *transtypage* » automatique de l'objet, c'est à dire la possibilité de l'utiliser comme étant d'un type différent de son type (ou de sa classe) d'origine.

On peut facilement vérifier que le transtypage (*cast* en anglais) n'a lieu que dans un seul sens. Si l'opérateur << avait été défini uniquement dans la classe ProduitSaisonnier, nous ne pourrions pas l'utiliser dans la séquence suivante :

```
Produit P1 ("Chocolat", 4.50);
cout << P1;
```

L'héritage n'offre pas de mécanisme pour « *transtyper* » un objet d'une classe de base vers ses classes dérivées. Cette impossibilité est illustrée dans le schéma suivant (figure 4.19). Le carré représente graphiquement le moule correspondant aux données membres en mémoire d'un objet Produit. La portion de triangle qui s'y ajoute représente les données supplémentaires de la classe ProduitSaisonnier. Le compilateur, pour « transtyper » un objet ProduitSaisonnier en objet Produit extrait du moule les données correspondant à la figure carrée. Par contre, il est incapable d'inventer les données correspondant à la portion de triangle si on lui demande de « transtyper » un objet Produit en objet ProduitSaisonnier.

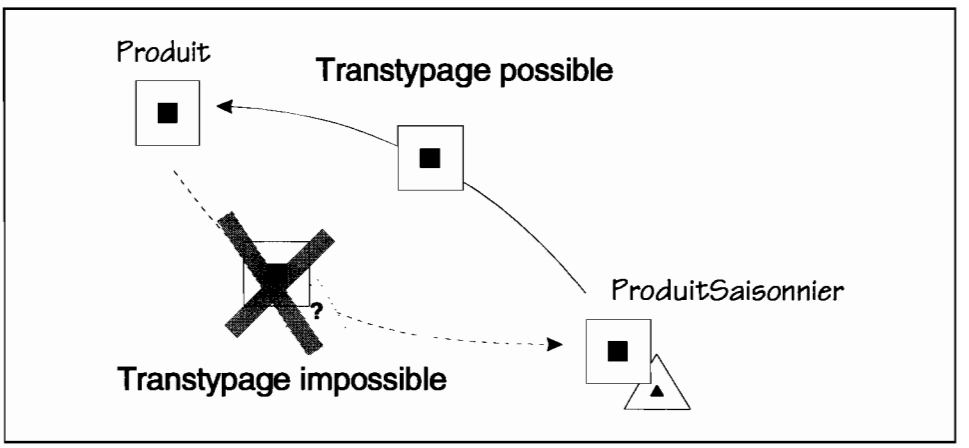


Figure 4.19 : transtypage entre les classes Produit *et* ProduitSaisonnier

Les constructeurs sont également utilisés par le compilateur pour « transtyper » des objets d'un type ou d'une classe vers un autre. La déclaration suivante :

```
Produit::Produit(int Prix);
```

sera utilisée pour le transtypage d'entier (type int) en Produit, comme dans :

```
Produit P;
P = 4;
```

Dans ce cas, le constructeur par défaut est invoqué pour préparer un objet P (première instruction). Pour l'affectation d'un entier vers un Produit, le compilateur cherche un constructeur de la classe Produit acceptant un argument de type entier. S'il en trouve, il prépare un objet temporaire Produit en utilisant ce constructeur avec l'entier 4 comme argument. Il utilise ensuite l'affectation (redéfinie ou non dans la classe Produit) pour recopier les données membres de cet objet temporaire dans P.

Il est possible, si c'est nécessaire, de fournir un constructeur dans chaque classe dérivée acceptant comme argument un objet de la classe de base : dans ce cas, nous disposons d'un transtypage classe de base → classe dérivée (offert par ce constructeur), et du transtypage automatique classe dérivée → classe de base fourni par l'héritage.

4.3 - Héritage multiple : *ProduitFugace*

Nous avons utilisé les classes Produit, ProduitSaisonnier et ProduitPerissable pour décrire des objets aussi divers que des huîtres, des parasols, des moufles, du chocolat ou de l'huile frelatée. Nous connaissons également de nombreux produits qui sont à la fois saisonniers et périssables. Les fruits et légumes (cerises, litchis, artichauts) en sont un exemple.

Pour rassembler de tels objets, nous pourrions définir une troisième classe directement dérivée de Produit, dont les données et fonctions membres seraient largement inspirées des classes ProduitSaisonnier et ProduitPerissable. Cela nous obligerait toutefois à dupliquer une grande partie du code. La mise à jour des classes s'en trouverait alourdie, puisque toute modification d'un traitement propre à la classe ProduitPerissable aurait toute chance d'être nécessaire dans la nouvelle classe ProduitFugace.

Il semble plus pratique d'hériter à la fois de ProduitSaisonnier et de ProduitPerissable de manière à bénéficier de tous les avantages de l'héritage. Nous pourrons ainsi brader en juillet des cerises pourries et solder des lainages mités en mars.

Le langage C++, contrairement à d'autres langages orientés objets, autorise l'héritage multiple. Nous allons voir que cette liberté nécessite un contrôle rigoureux des éléments hérités.

4.3.1 - Créer une classe héritant de deux classes dérivées

Pour garder une certaine concision à nos exemples, malgré la multiplication des classes décrites, nous allons simplifier au maximum la définition des classes Produit, ProduitSaisonnier et ProduitPerissable (figure 4.20).

```
enum Id {Produit,
         ProduitPerissable,
         ProduitSaisonnier,
         ProduitFugace};

class Produit {
 protected:
   float prix;
   char * nom; // le pointeur autorise les descriptions de longueur variable
   void fixeNom(const char * texte);
 public:
   Produit(const char * Nom) {fixeNom(Nom);}
   ~Produit() { delete nom; }
   virtual Id IdClasse() { return Produit; }
}; // fin définition classe Produit

class ProduitPerissable : public Produit {
 private:
   int delai; // durée de conservation
 public:
   ProduitPerissable (const char * Nom, int Delai)
      : Produit (Nom),delai(Delai)
      {}
   Id IdClasse(){ return ProduitPerissable; }
}; // fin classe ProduitPerissable

class ProduitSaisonnier : public Produit {
 private:
   const char * const saison;
 public:
   ProduitSaisonnier(const char * Nom, const char * Saison)
      : Produit(Nom), saison(Saison)
      {}
   Id IdClasse(){ return ProduitSaisonnier; }
}; // fin classe ProduitSaisonnier

void Produit::fixeNom(const char * Texte)
{
   nom = new char[strlen(Texte) + 1];
   strcpy(nom, Texte);
}; // fin définition fixeNom
```

Figure 4.20 : hiérarchie simplifiée de classes dérivées de Produit

Pour définir une classe dérivée à la fois de ProduitSaisonnier et ProduitPerissable, il suffit de mentionner dans sa déclaration les deux classes dont elle hérite (figure 4.21).

```
class ProduitFugace : public ProduitSaisonnier,
                      public ProduitPerissable {
 public:
   Id IdClasse() { return ProduitFugace; }
}; // fin classe ProduitFugace
```

Figure 4.21 : classe ProduitFugace

Telle qu'elle est décrite, notre classe est correcte mais non fonctionnelle : nous n'avons pas encore défini de constructeur. Le compilateur C++ est censé fournir un constructeur par défaut. Mais nous avons vu que, dans le cas de l'héritage, ce constructeur fera alors appel aux constructeurs par défaut des classes de base (Cf. 4.2.1). La déclaration explicite de constructeurs dans chacune des classes ProduitSaisonnier et ProduitPerissable invalide l'utilisation des deux constructeurs par défaut dont a besoin la classe ProduitFugace. Il nous faut donc définir un constructeur explicite pour ProduitFugace.

4.3.2 - Constructeur d'une classe à héritage multiple

En l'absence de constructeurs par défaut dans les classes de base dont dérive ProduitFugace, il faut préciser dans la liste d'initialisation du constructeur ProduitFugace l'appel des deux constructeurs explicites des classes ProduitSaisonnier et ProduitPerissable :

```
ProduitFugace::ProduitFugace(const char * Nom,
                             const char * Saison,
                             int Delai)
   : ProduitPerissable(Nom, Delai),
     ProduitSaisonnier(Nom, Saison)
   {}
```

Malgré l'ordre d'appel dans la liste d'initialisation, les constructeurs seront invoqués dans l'ordre d'héritage (ProduitSaisonnier d'abord, puis ProduitPerissable). La répétition de l'argument Nom dans chacun des constructeurs laisse prévoir une structure en mémoire des données membres un peu particulière pour les objets d'une classe à héritage multiple. En effet, la classe ProduitFugace hérite de toutes les données membres de ses deux classes de base :

- saison, qui lui vient de ProduitSaisonnier ;
- delai, qui lui vient de ProduitPerissable ;
- nom et prix, hérités de ProduitSaisonnier (qui lui-même les hérite de Produit) ;
- nom et prix, cette fois hérités de ProduitPerissable (qui les hérite aussi de Produit).

Nous illustrerons cette particularité en définissant une nouvelle fonction membre AfficheNom qui affichera le nom du produit :

```
void ProduitFugace::AfficheNom()
{
    cout << "Je suis le produit saisonnier périssable "
         << ProduitPerissable::nom << endl;
}
```

Dans cette première définition, la fonction AfficheNom est correcte. Comme l'objet dispose de deux versions différentes de la donnée nom, l'opérateur de portée permet au compilateur de détecter quelle est celle qu'il doit utiliser (même si dans ce cas précis, les deux versions de nom sont identiques). Par contre, la définition suivante :

```
void ProduitFugace::AfficheNom()
{   // cette fonction ne peut être compilée
    cout << "Je suis le produit saisonnier périssable "
         << nom << endl;
}
```

est refusée par le compilateur, qui bloque sur l'appel de nom dont il ne peut résoudre l'ambiguïté. Le compilateur ne sait pas s'il s'agit du membre hérité de Produit par ProduitPerissable ou par ProduitSaisonnier.

Dans l'exemple ci-dessus, la distinction entre les deux versions de nom semble artificielle, puisque le nom d'un produit est unique, qu'il passe par l'héritage de ProduitSaisonnier ou de ProduitPerissable.

Regardons maintenant de nouveau la donnée membre prix définie dans la classe Produit. Dans les classes Produit et ProduitSaisonnier, cette donnée représente le prix public, invariant, de l'objet (nous ne tiendrons pas compte des soldes pour les produits saisonniers). Dans la classe ProduitPerissable, cette donnée sera variable et mise à jour en fonction de l'aspect du produit, du délai de consommation, etc. Nous pouvons imaginer des constructeurs sophistiqués, recalculant la donnée membre prix à partir du prix de base transmis en argument.

Dans ce cas, la distinction entre les deux versions héritées de la donnée prix prend tout son sens : doit-on utiliser le prix de ProduitSaisonnier (invariable) ou celui de ProduitPerissable (variable et pouvant être recalculé à partir de l'argument donné par le programmeur) ?

Comme pour les données membres, le programmeur doit qualifier les fonctions membres héritées (avec l'opérateur de portée) avant de les utiliser dans la définition de la classe ProduitFugace. Si nous supprimons toute définition de IdClasse dans la classe ProduitFugace, l'exécution de :

```
ProduitFugace P ("cerise", "été", 7);
P.IdClasse();
```

est refusée par le compilateur, qui ne peut lever l'ambiguïté régnant entre ProduitPerissable::IdClasse et ProduitSaisonnier::IdClasse. Pour permettre l'exécution d'une fonction doublement héritée, et non surchargée dans la classe dérivée, il faut la qualifier avec l'opérateur de portée :

```
P.ProduitPerissable::IdClasse();
```

Le concepteur de la classe dérivée peut choisir d'hériter préférentiellement l'une ou l'autre des versions des fonctions membres (dans le cas où les classes de base définissent les mêmes fonctions) en redéfinissant la fonction dans la classe dérivée avec un simple appel de la fonction de la classe de base souhaitée :

```
void ProduitFugace::IdClasse()
{
   ProduitPerissable::IdClasse();
}
```

Cela n'empêche en aucun cas l'accès à la fonction définie dans la classe ProduitSaisonnier (il suffit d'utiliser l'opérateur de portée).

4.3.3 - Transtypage et classe de base virtuelle

Gardons toujours la hiérarchie de classes définie dans les paragraphes précédents en y ajoutant les constructeurs par défaut suivants :

```
Produit::Produit() { prix = 0; fixeNom("nom"); }
ProduitPerissable::ProduitPerissable() {}
ProduitSaisonnier::ProduitSaisonnier()
   : saison("saison")
   {}
ProduitFugace::ProduitFugace() {}
```

et tentons d'exécuter la séquence suivante :

```
Produit * Stock[4];
Stock[0] = new Produit;
Stock[1] = new ProduitSaisonnier;
Stock[2] = new ProduitPerissable;
Stock[4] = new ProduitFugace;
```

La compilation bloque à la dernière ligne sur une tentative de conversion de ProduitFugace * en Produit *. Lorsque le compilateur prépare le tableau Stock, il définit chaque case comme un pointeur sur Produit. Nous avons vu au paragraphe 4.2.2 (figure 4.12) que le compilateur savait convertir un pointeur sur un objet d'une classe dérivée en un pointeur sur un objet de la classe de base.

Dans le cas de ProduitFugace, le compilateur dispose en réalité de deux objets Produit pour réaliser le transtypage des pointeurs : l'un est hérité de ProduitPerissable, l'autre de ProduitSaisonnier. Le compilateur ignore vers lequel des deux sous-objets Produit de ProduitFugace il doit pointer (figure 4.22).

Le langage C++ offre la possibilité de résoudre ce conflit en déclarant la classe Produit comme classe de base virtuelle dans chaque classe héritière (ProduitSaisonnier et ProduitPerissable). Un seul sous-objet Produit est alors préparé pour les instances de ProduitFugace.

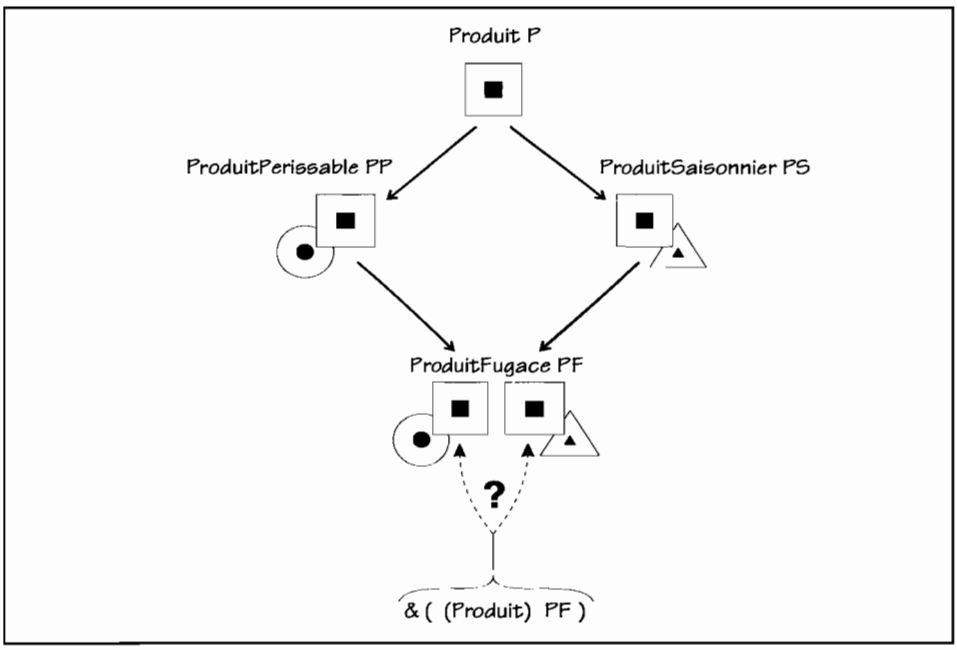


Figure 4.22 : transtypage de pointeurs avec un héritage multiple

Telle qu'elle est maintenant définie figure 4.23, la nouvelle hiérarchie issue de Produit permet de compiler les quelques lignes précédentes (création du tableau Stock). Par contre, si les seuls constructeurs sont ceux présentés dans la figure, le compilateur bloquera sur :

```
ProduitFugace::ProduitFugace() {}
```

Avec la première définition de la hiérarchie issue de Produit, le compilateur aurait hérité des constructeurs par défaut des classes ProduitSaisonnier et ProduitPerissable pour préparer les sous-objets ProduitSaisonnier et ProduitPerissable. Chaque sous-objet comprenant lui-même un sous-objet Produit, l'objet ProduitFugace se retrouvait en final avec :

- un sous-objet ProduitSaisonnier ;
- un sous-objet ProduitPerissable ;
- deux sous-objets Produit.

L'utilisation du spécificateur virtual a précisément pour but de supprimer l'héritage automatique des deux sous-objets Produit. Le compilateur va donc remonter directement (et une seule fois) à la classe Produit pour construire le sous-objet Produit de l'instance de ProduitFugace.

Dans le cas présent, il cherche un constructeur par défaut dans Produit, puisque nous n'avons précisé aucune liste d'initialisation dans notre définition. Ce constructeur n'existant pas (dans notre exemple), le compilateur signale l'erreur.

```
class Produit {
...}; // fin classe Produit

class ProduitPerissable : public virtual Produit {
...}; // fin classe ProduitPerissable

class ProduitSaisonnier : public virtual Produit {
...}; // fin classe ProduitSaisonnier

class ProduitFugace : public ProduitSaisonnier,
                      public ProduitPerissable {
...}; // fin classe ProduitFugace

Produit::Produit(const char * Nom) {fixeNom(Nom);}
ProduitPerissable::ProduitPerissable()
   : Produit ("périssable")
   {}
ProduitSaisonnier::ProduitSaisonnier()
   : Produit("saisonnier")
   {}
ProduitFugace::ProduitFugace() {}
```

Figure 4.23 : utilisation du spécificateur virtual *dans les définitions de classe*

Il faut donc concevoir l'héritage d'une classe unique par l'intermédiaire de classes dérivées virtuelles comme si l'héritage se faisait directement (comme si on avait pratiquement défini Produit comme troisième classe mère de ProduitFugace). La liste d'initialisation du constructeur peut alors comprendre un appel aux constructeurs de toutes les classes héritées :

```
ProduitFugace::ProduitFugace(const char * Nom,
                             const char * Saison,
                             int Delai)
   : Produit(Nom),
     ProduitPerissable(Nom, Delai),
     ProduitSaisonnier(Nom, Saison)
   {}
```

Une autre solution est d'implémenter un constructeur par défaut dans la classe de base (Produit).

4.3.4 - Destructeurs virtuels

La définition de Produit décrite figure 4.20 utilise pour le stockage du membre nom un pointeur sur caractère. L'espace mémoire désigné par ce pointeur est préparé par l'opérateur new, dans la fonction membre fixeNom. Il est donc nécessaire (Cf. 2.4.2) de définir explicitement un destructeur pour la classe Produit. C'est ce que nous avons fait figure 4.20. Modifions maintenant légèrement la classe dérivée ProduitSaisonnier (figure 4.24).

```
class ProduitSaisonnier : public Produit {
 private:
   char * saison;
   void fixeSaison (const char * Saison);
 public:
   ProduitSaisonnier (const char * Nom,
                      const char * Saison)
      : Produit(Nom)
      { fixeSaison(Saison); }
   ~ProduitSaisonnier()
   {
      delete saison;
      cout << "Destructeur ProduitSaisonnier"; // marquage du destructeur
   }
}; // fin classe ProduitSaisonnier

void ProduitSaisonnier::fixeSaison(const char * Saison)
{
   saison = new char(strlen(Saison));
   strcpy(saison, Saison);
}
```

Figure 4.24 : une définition de ProduitSaisonnier *nécessitant un destructeur explicite*

L'introduction d'un pointeur sur char comme donnée membre de ProduitSaisonnier rend nécessaire la définition explicite d'un destructeur. Nous avons vu (paragraphe 4.2.1) que les destructeurs sont appelés en commençant par celui de la classe dérivée pour finir par celui de la classe de base. L'exécution d'un programme contenant dans un bloc l'instruction :

```
{
 ProduitSaisonnier P ("Moufles", "hiver");
 ...// etc.
}
```

génère la création d'un objet ProduitSaisonnier puis sa destruction, en appelant d'abord le destructeur ~ProduitSaisonnier (libération de la place mémoire occupée par saison) puis ~Produit (libération de la place occupée par nom).

Si nous reprenons le tableau Stock :

```
Produit * Stock[4];
Stock[0] = new Produit;
Stock[1] = new ProduitSaisonnier;
```

nous constaterons, grâce au marquage du destructeur ~ProduitSaisonnier, que la destruction du tableau ne libère que l'espace mémoire alloué à nom : le seul destructeur invoqué est celui de la classe Produit. Comme pour les autres fonctions membres, il faudrait préciser avec le mot clé virtual que le choix du destructeur doit être repoussé à l'exécution (figure 4.25).

```
class Produit {
 ...
 public:
 ...
   virtual ~Produit() { delete nom; }
}; // fin classe Produit

class ProduitSaisonnier {
 ...
 public:
 ...
   ~ProduitSaisonnier() { delete saison; }
}; // fin classe ProduitSaisonnier
```

Figure 4.25 : destructeur virtuel

5 Construire et organiser une librairie de classes

Avec ce chapitre, nous abordons les problèmes techniques que doit résoudre le programmeur lorsqu'il entreprend la construction d'une librairie de classes. Notre ouvrage n'est pas un livre de méthodologie : nous n'aborderons donc pas les problèmes de conception : identification des classes à définir et spécification de leurs propriétés. Nous nous situons en aval de la conception et nous nous intéressons au travail du programmeur qui veut implémenter une ou plusieurs hiérarchies de classes déjà spécifiées mais non encore définies en C++.

Nous avons choisi de nous appuyer sur un exemple de hiérarchie de classes-conteneurs : il fait appel à des techniques familières à tous les programmeurs, quelles que soient leurs origines, et il permet de mettre en relief facilement la plupart des caractéristiques de C++ liées à une architecture de classes. On commence par s'intéresser à un exemple simple de conteneur : le tableau. On introduit ensuite une hiérarchie de classes dont la racine est une classe abstraite, que nous appelons Object. On montre alors comment cette hiérarchie permet de répertorier dans un objet-tableau n'importe quel objet descendant de Object. A partir de la nécessité de comparer deux objets différents, on étudie en détail tous les problèmes liés au transtypage, que le programmeur doit impérativement maîtriser pour construire une hiérarchie de classes. On s'intéresse enfin à la définition d'une hiérarchie de plusieurs classes-conteneurs et aux itérateurs associés en étudiant plus particulièrement l'implémentation du polymorphisme.

5.1 - Une classe *Tableau* ?

5.1.1 - Des tableaux d'entiers

La plupart des langages de programmation permettent de définir et manipuler des tableaux. Si `t` est un tableau, on dispose en général d'une primitive d'indexation, souvent notée `t[k]`, qui accède à l'élément de rang `k`. Malheureusement, de nombreux langages ne prévoient pas le contrôle de la validité de `k` et ouvrent ainsi la porte à de nombreuses erreurs de programmation. En C++, on pourra remédier

à cette carence pour gérer, par exemple, des tableaux d'entiers avec une classe étudiée à cet effet. Une telle classe est présentée à la figure 5.1.

```
class TableauEntiers {
 private:
   int * t;        // pointera sur l'espace alloué au tableau
   const int d;    // dimension du tableau
   void alloue (int Dim);
       // fonction privée : initialise les membres t et d
   int valideRang(int Index) const;
       // fonction privée : renvoie Index s'il représente le rang d'un élément
   TableauEntiers & operator =
                      (const TableauEntiers & Source);
       // affectation privée : réservée à l'implémenteur
 public:
   TableauEntiers (int Dimension);
       // construit un tableau de Dimension éléments
   TableauEntiers (const TableauEntiers & Copie);
       // constructeur-copie
   int & operator [] (int R);
       // surcharge de l'indexation
}; // class TableauEntiers
```

Figure 5.1 : la classe TableauEntiers

Avec une telle classe, l'exécution de :

```
TableauEntiers T(50)
```

construira un tableau de 50 éléments et, si K est une variable du type int, l'exécution de :

```
T[K] = 250;
```

vérifiera, avant l'affectation, que la valeur de K est bien dans l'intervalle d'indexation de 0 à 49. Ce contrôle sera effectué par les deux fonctions valideRang et operator[] qui sont définies par :

```
inline int TableauEntiers::valideRang(int Index) const
{  // fonction privée : renvoie Index s'il représente le rang d'un élément
   if (Index < 0  ||  Index >= d) {
      cerr << "classe TableauEntiers : valeur d'index incorrecte\n";
      exit(1);
   }
   return Index;
} // int TableauEntiers::valideRang(int Index)
inline int & TableauEntiers::operator [] (int R)
{  // surcharge de l'indexation
   return t[valideRang(R)];
}
```

Avec une telle classe, on ne peut cependant ranger dans un tableau que des entiers. Pour chaque type de valeur susceptible d'être répertoriée dans un tableau, il faudrait donc définir une classe particulière : classe de tableaux de float, de long int, de String, de Client (si String et Client sont des classes existantes, ...). Une telle approche amènera donc le programmeur à définir autant de classes TableauXXX qu'il manipule de types XXX. Des évolutions récentes de C++ permettent la génération automatique de ces classes par le compilateur : on fait appel au mécanisme des *templates* ou classes-modèles que nous n'étudierons pas dans ce chapitre[1]. Nous nous intéresserons plutôt ici à la technique qui organise les classes utilisées en une arborescence avec, à la racine, une classe dont héritent toutes les autres.

5.1.2 - Un tableau de pointeurs

Si nous voulons organiser nos classes en une hiérarchie d'héritage simple, nous sommes amenés à concevoir une arborescence. Si nous appelons Object[2] la classe-racine de cette arborescence, on peut par exemple définir les classes suivantes :

```
class Object {/* etc. */ ...};
  class Produit : public Object {/* etc. */ ...};
    class ProduitPerissable : public Produit {/* etc. */};
    class ProduitSaisonnier : public Produit {/* etc. */};
```

et exécuter ensuite la séquence suivante :

```
Object * T[100];
T[0] = new Produit;
    // L'objet *T[0] est un Produit
T[1] = new ProduitPerissable;
    // L'objet *T[1] est un ProduitPerissable
T[2] = new ProduitSaisonnier;
    // L'objet *T[2] est un ProduitSaisonnier
```

Parce que l'héritage ClasseDérivée : ClasseDeBase introduit une possibilité de transtypage implicite de pointeur sur ClasseDérivée en pointeur sur ClasseDeBase (Cf. 4.2.4), il est possible d'affecter à un élément du tableau T un pointeur sur une instance d'une sous-classe de Object. Ainsi, l'affectation T[0] = new Produit est-elle licite. Il en va de même pour les deux affectations suivantes de la séquence précédente puisque ProduitPerissable et ProduitSaisonnier héritent toutes deux de Produit qui hérite elle-même de Object. Plus généralement un élément quelconque T[K] de T pourra pointer sur un objet d'une classe CCC quelconque, pourvu que CCC soit une classe de l'arborescence héritée de Object.

---

1 Voir Annexe B.

2 Ce choix de désignation est fait par référence au modèle de hiérarchie du langage Smalltalk. Ce langage, qui ne permet pas l'héritage multiple, fournit une arborescence de classes prédéfinies. A la racine de cette arborescence figure la classe Object qui définit les propriétés communes à toutes les classes. Chaque nouvelle classe construite par le programmeur Smalltalk est une descendante directe ou indirecte de Object.

Si nous écrivons alors une instruction telle que :

```
T[K]->f();
```

le compilateur n'acceptera cette instruction que si f est une fonction membre de la classe *Object*. Si c'est le cas et si f est une fonction virtuelle, la fonction f exécutée sera celle correspondant à la classe *CCC* (Cf. 4.2.2).

Nous pouvons donc envisager de définir une classe unique de tableaux, pourvu que les instances de cette classe enregistrent non pas les objets répertoriés mais des pointeurs sur ces objets. Par rapport à la solution des classes-modèles mentionnée plus haut, cette approche est plus contraignante mais plus puissante.

Elle est plus contraignante car elle implique une approche de bas niveau[3] : le programmeur doit gérer des pointeurs sur les objets qu'il manipule et, dans certains cas, spécifier explicitement le type de l'objet : dans l'exemple précédent, si f est une fonction membre de *CCC* qui n'est pas héritée de *Object*, il faudra écrire explicitement (*CCC* *) T[K] --> f().

Elle est plus puissante parce qu'elle permet de gérer, dans un tableau, des objets de classes différentes, pourvu que ces classes soient dans la hiérarchie héritée de *Object*. Avec une classe-modèle *Tableau*, on enregistre directement les objets dans le tableau et on ne peut utiliser qu'une seule catégorie d'objets dans un tableau donné.

Pour étudier les techniques de construction d'une hiérarchie de classes, nous allons dans un premier temps nous intéresser aux contraintes liées à l'héritage d'une classe *Object* puis nous aborderons la réalisation d'une hiérarchie de classes-conteneurs.

5.2 - Une classe *Object* et sa descendance

Considérons la définition suivante :

```
enum IdClasse { OBJECT };
class Object {
 protected:
   virtual IdClasse classId() const { return OBJECT; }
   virtual const char * className() const
       { return "Object"; }
 public:
   virtual char * AsString() const = 0;
       // renvoie la représentation de l'objet sous forme d'une chaîne
}; // class Object
```

3 La qualification *bas niveau* n'est pas à prendre au sens péjoratif. Elle est inhérente à la philosophie de C++ qui, comme C, est un langage de bas niveau qui permet au programmeur de travailler à un niveau proche des mécanismes de base de la machine. Un langage de plus haut niveau comme Smalltalk n'évite pas les manipulations de pointeurs mais il les cache : le programmeur n'a pas à s'en occuper.

La classe *Object* que nous introduisons définit trois propriétés représentées par des fonctions membres virtuelles. A priori tout objet d'une classe qui héritera de *Object* bénéficiera donc de ces propriétés que nous détaillons ci-après :

- La propriété d'identifier son type est implémentée par la fonction membre *classId* qui renverra un identificateur entier caractérisant de manière unique chaque classe. Pour cela, le type énuméré *IdClasse* doit définir, pour chaque classe de la hiérarchie, une constante entière. Pour l'instant, en l'absence de descendance de *Object*, le type *IdClasse* énumère la seule constante *OBJECT* et la fonction membre *classId* de la classe *Object* renvoie cette constante. On fixe ainsi un comportement par défaut pour toute sous-classe héritée de *Object* qui ne redéfinira pas *classId* : à défaut d'indiquer sa classe, un objet d'une telle sous-classe pourra répondre au message *classId*() qu'il est descendant de *Object*.

- La propriété d'indiquer le nom explicite de la classe est de la même manière définie par la fonction membre *className* qui renverra une chaîne de caractères représentant le nom de la classe.

- Enfin, la dernière propriété est représentée par la fonction virtuelle pure *AsString*. Son rôle est de permettre à chaque objet de fournir une chaîne de caractères donnant la représentation de ses données membres. Cette fonction membre est virtuelle pure dans *Object* : cette classe est donc abstraite et, pour qu'une classe *C* descendante de *Object* soit instanciable, il faudra que *C* redéfinisse la fonction membre *AsString* comme une fonction complète.

On peut déduire de l'étude de cette première version de la classe *Object* que la construction d'une hiérarchie de classes suppose la démarche suivante :

- On fait d'abord l'inventaire des propriétés communes à toutes les classes.

- Parmi les propriétés ainsi inventoriées, on distingue celles pour lesquelles un comportement par défaut peut être défini. Elles sont alors représentées par des fonctions membres virtuelles complètes de la classe-racine de la hiérarchie. Le comportement par défaut est ainsi spécifié une seule fois au plus haut niveau.

- Toutes les autres propriétés communes sont définies par des fonctions virtuelles pures de la classe-racine. On force ainsi chaque classe descendante *C* non abstraite à implémenter ces propriétés conformément aux caractéristiques de *C*.

Pour la classe *Object*, le choix que nous avons fait entre les deux catégories de fonctions virtuelles (complètes ou pures) peut paraître arbitraire. Nous avons ici considéré que la propriété *AsString* ne pouvait s'accommoder d'un comportement par défaut, compte tenu de la variété des objets susceptibles d'hériter de *Object*.

5.2.1 - Deux sous-classes pour la classe *Object*

Examinons la hiérarchie des trois classes définies par la figure 5.2. La classe Ville est héritée de *Object* : chacune de ses instances représentera une ville dont elle fournira le code postal et le nom.

```
enum IdClasse { OBJECT, INTEGER, VILLE };
class Object {
 protected:
   virtual IdClasse classId() const { return OBJECT; }
   virtual const char * className() const
      { return "Object"; }
 public:
   virtual const char * AsString() const = 0;
}; // class Object
class Ville : public Object { // des villes comme objets
 private:
   char codePostal[6];
   char nom[27];
   static char string[61]; // utilisée par AsString
 protected:
   IdClasse classId() const { return VILLE; }
   const char * className() const { return "Ville"; }
 public:
   Ville() { codePostal[0] = '\0'; nom[0] = '\0'; }
   Ville(const char * C, const char * N);
   const char * AsString() const;
}; // class Ville
class Integer : public Object {   // des entiers comme objets
 private:
   int valeur;
   static char string[31]; // utilisée par AsString
 protected:
   IdClasse classId() const { return INTEGER; }
   const char * className() const { return "Integer"; }
 public:
   Integer (int V = 0) : valeur(V) {}
   const char * AsString() const;
}; // class Integer
```

Figure 5.2 : la classe Object et deux de ses sous-classes

Outre un constructeur par défaut qui initialise les données membres correspondantes avec des chaînes vides, elle définit un constructeur qui permet l'instanciation à partir d'un code postal C et d'un nom N :

```
Ville::Ville(const char * C, const char * N)
{
   strncpy (codePostal, C, 5); codePostal[5]='\0';
   strncpy (nom, N, 26); nom[26]='\0';
}
```

La fonction AsString est également définie pour cette classe. La classe Ville est donc instanciable car elle n'a plus de fonction virtuelle pure. La fonction membre AsString est définie par :

```
const char * Ville::AsString() const
{
   sprintf (string, "%s (\"%s\", \"%s\")",
            className(), codePostal, nom);
   return string;
}
```

Elle utilise la fonction sprintf[4] pour construire, dans la variable de classe string une chaîne représentant l'information de l'objet. La variable string est déclarée dans la classe avec le modificateur static. Elle existe donc en un seul exemplaire, partagé entre toutes les instances. Notre implémentation suppose donc que l'environnement d'exécution ne permettra pas l'exécution concurrente de deux appels de AsString. On peut désormais définir et initialiser un tableau T avec :

```
Object * T[3];
T[0] = new Ville("35135", "CHANTEPIE");
T[1] = new Ville;
T[2] = new Ville("06140", "VENCE");
for (int K=0; K<=2; K++)
    cout << T[K]->AsString() <<'\n';
```

L'exécution de cette séquence affichera :

```
Ville("35135", "CHANTEPIE")
Ville("", "")
Ville("06140", "VENCE")
```

Pour gérer de manière sûre des tableaux d'objets héritant de Object, on peut donc envisager de définir sur le modèle de la classe TableauEntiers du paragraphe 5.1, une classe Array dont la donnée membre t qui désignera le tableau sera définie par

```
Object * t; // pointera sur l'espace alloué au tableau
```

On pourrait alors enregistrer des objets d'une classe C quelconque dans une instance de Array. Mais on ne pourrait pas y répertorier des valeurs d'un type prédéfini comme int ou float par exemple. Pour cette raison, nous avons défini dans la figure 5.2 la classe Integer, dérivée de Object. Chaque instance de Integer représente, par sa donnée membre privée valeur, un entier. Si nous définissons un

4 La fonction sprintf s'utilise de la même manière que la fonction printf étudiée au chapitre 1. A la différence de printf, sprintf envoie la chaîne de caractères qu'elle construit non pas à l'écran mais dans une zone-mémoire pointée par son premier argument.

tableau de pointeurs sur Object, on pourra, dans le tableau enregistrer des pointeurs sur des entiers, instances de Integer.

5.2.2 - Un *Integer* peut-il être un *int* ?

Nous savons qu'une instance de la classe Integer représente une valeur du type int. Mais le compilateur, lui, ne le sait pas. On pourrait pourtant le croire car la séquence :

```
Integer E1;
E1 = 5;
```

se compilera sans erreur. En effet, le constructeur sait convertir une valeur du type int en un objet Integer car nous lui avons fourni, avec le constructeur Integer(int) une règle de conversion. Mais si l'on compile :

```
Integer E1(5);
int K = E1;
```

nous obtiendrons une erreur indiquant que le compilateur ne sait pas effectuer l'opération inverse : convertir un Integer en int. Pour la même raison, toutes les opérations arithmétiques comme par exemple K = K + E1 seront refusées par le compilateur.

Nous savons définir une règle de conversion pour passer d'un type TTT à une classe CCC : il suffit de définir un constructeur CCC(TTT). Mais pour passer de Integer à int, on ne peut appliquer le même principe : int n'est pas une classe et on ne peut pas définir de constructeur de ce type.

Cependant le transtypage est un opérateur C++ que l'on peut redéfinir. Pour une classe CCC, le transtypage (TTT) vers le type TTT se définit par une fonction membre opérateur pour laquelle :

- le symbole représentant l'opérateur est l'identificateur du type (TTT),
- aucune valeur de renvoi n'est spécifiée,
- aucun argument n'est défini.

Il nous suffit donc de compléter la classe Integer comme suit :

```
class Integer : public Object {
   ... // etc. Cf. figure 5.2
 public:
   ... // etc. Cf. figure 5.2
   operator int() const { return valeur; }
      // cast de Integer en int
}; // class Integer
```

pour que tous nos problèmes soient réglés.

5.2.3 - Comparer des objets comparables

Si nous voulons implémenter une classe Array, il nous faut prévoir la possibilité de rechercher un objet répertorié dans un tableau. Pour cela, il faudra pouvoir

comparer entre eux deux éléments d'un tableau. Il faut donc que les classes des objets susceptibles d'être répertoriés dans un tableau implémentent les opérateurs == et !=.

Nous pouvons pour cela, compléter la classe *Object* avec deux nouvelles fonctions virtuelles :

```
class Object {
    ... // etc. Cf. figure 5.2
 public:
    ... // etc. Cf. figure 5.2
    virtual int operator == (Object & Op2) const = 0;
    int operator != (Object & Op2) const
        { return ! operator == (Op2); }
}; // class Object
```

L'opérateur == est une fonction virtuelle pure : il devra être obligatoirement défini (ou hérité[5]) comme une fonction membre complète pour toute sous-classe instanciable. L'opérateur != est une fonction virtuelle complète qui n'a pas à être redéfinie dans les sous-classes. Son polymorphisme s'appuie sur celui de l'opérateur ==.

Pour pouvoir comparer entre elles deux instances de *Integer*, nous devons donc surcharger, pour cette classe, l'opérateur ==. Si, comme il paraît assez naturel de le faire, nous définissons cet opérateur par :

```
int Integer::operator == (Integer & Op2) const
{
   return valeur == Op2.valeur;
}
```

le compilateur nous indique que la classe *Integer* reste abstraite. En effet l'opérateur == que nous avons défini pour *Integer* ne surcharge pas la fonction membre virtuelle pure == de la classe *Object* car il n'a pas la même signature. Les types des arguments (*Integer* & et *Object* &) sont différents. Il faut donc spécifier un argument *Object* & pour l'opérateur == de *Integer*. Cependant, nous ne sommes pas tout à fait au bout de nos peines. En effet, la nouvelle définition :

```
int Integer::operator == (Object & Op2) const
{
   return valeur == Op2.valeur;
}
```

provoque une erreur de compilation indiquant que *valeur* n'est pas une donnée membre de *Object*. La solution correcte est donc obtenue avec la définition :

```
int operator == (Object & Op2) const
{
   return valeur == ( (Integer &) Op2).valeur;
}
```

5 En effet, si une classe D dérivée de *Object* implémente cet opérateur comme une fonction membre complète, toute classe descendante de D hérite de cette fonction membre complète.

Avec l'écriture (Integer &) *Op2* nous spécifions un transtypage vers une référence d'Integer : l'argument *Op2*. Nous indiquons ainsi au compilateur qu'il doit traiter l'argument *Op2* comme un objet de la classe Integer. Le compilateur acceptera donc la désignation du membre *valeur* de l'objet *Op2*.

Nous reviendrons au paragraphe 5.2.5 sur le transtypage vers une référence qui mérite de plus longs commentaires. Mais auparavant, terminons le travail en implémentant, sur le même principe, la comparaison à l'égalité pour la classe *Ville*. Les modifications que nous apportons ainsi à notre hiérarchie de classes sont rassemblées à la figure 5.3.

```
enum IdClasse { OBJECT, INTEGER, VILLE };
class Object {
   ... // etc. Cf. figure 5.2
public:
   ... // etc. Cf. figure 5.2
   int operator != (Object & Op2) const
       { return ! operator == (Op2); }
   virtual const char * AsString() const = 0;
}; // class Object
class Integer : public Object {
 private:
   int valeur;
   ... // etc. Cf. figure 5.2
 public:
   ... // etc. Cf. figure 5.2
   int operator == (Object & Op2) const
       { return valeur == ( (Integer &) Op2).valeur; }
}; // class Integer
class Ville : public Object {
 private:
   char codePostal[6];
   char nom[27];
   ... // etc. Cf. figure 5.2
 public:
   ... // etc. Cf. figure 5.2
   int operator == (Object & Op2) const
   {
       return ! strcmp(codePostal,
                       ( (Ville &) Op2).codePostal);
   }
}; // class Ville
```

Figure 5.3 : comparer entre eux des descendants de Object

Maintenant, on peut compiler et exécuter une séquence telle que :

```
Integer E1(5), E2(5.2);
if (E1 == E2)  cout << "grâce au cast en (int), E2 est égal à E1\n";
Ville V1("35135", "CHANTEPIE");
Ville V2("75005", "PARIS 5");
if (V1 != V2) cout << "Chantepie et Paris sont deux villes distinctes\n"
```

5.2.4 - Conversion par constructeur ou par cast ?

Le transtypage, souvent appelé *cast*, est une opération que l'on exprime en C++ par l'écriture :

(UnType) UneValeur

et qui renverra la valeur de UnType que l'on peut obtenir par conversion de UneValeur. Par exemple :

- (int) 3.54 renverra la valeur 3 du type int,
- (long int) 35 renverra la valeur 35 du type long int (en effet, une constante telle que 35 est représentée par défaut avec le type int).

Si nous appelons TTT le type ou la classe correspondant à UnType et VVV le type ou la classe de valeurVVV, la figure 5.4 résume les possibilités offertes par C++ en matière de transtypage.

Transtypage (TTT) valeurVVV		
Si TTT est	**et VVV est**	**alors la conversion ...**
un type prédéfini	un type prédéfini	est définie par le langage, si elle est licite
un type prédéfini	une classe	doit être définie par un opérateur de la classe VVV : VVV::operator TTT();
une classe	un type prédéfini	doit être définie par un constructeur de la classe TTT : TTT::TTT (VVV);
une classe	une classe	doit être définie : – soit par VVV::operator TTT(); – soit par TTT::TTT (VVV);

Figure 5.4 : fonctionnement du cast

Parmi les quatre dernières lignes de ce tableau, les trois premières constituent un rappel de ce que nous avons déjà étudié et elles ne nécessitent pas de nouveau commentaire. En revanche, la situation représentée par la dernière ligne appelle plusieurs remarques. Nous sommes alors dans le cas où TTT et VVV sont des classes et le programmeur a le choix entre :

- disposer d'un opérateur de transtypage de la classe VVV,
- disposer d'un constructeur dans la classe TTT.

Il convient d'abord de remarquer que ce choix est exclusif. Si les deux fonctions membres existent, le compilateur refusera le transtypage car il ne peut décider entre l'une ou l'autre.

Une seule des deux fonctions membres VVV::operator TTT ou TTT::TTT(VVV) doit donc être définie. Si les classes TTT et VVV ont été conçues par le même programmeur, celui-ci aura fait le choix en fonction des caractéristiques de ces classes. Sinon on choisira l'une ou l'autre des deux solutions, selon que l'on a la possibilité d'intervenir dans la définition de la classe TTT ou la classe VVV.

Cependant, une dernière catégorie de transtypage échappe à la classification que nous venons de faire. Etudions-la dans le paragraphe suivant.

5.2.5 - Le transtypage vers une référence

Tous les transtypages effectués au paragraphe précédent impliquent l'exécution d'une séquence de code pour effectuer la conversion :

- séquence définie par le compilateur pour le *cast* d'un type prédéfini vers un autre type prédéfini,
- séquence définie par une fonction membre dans les trois autres cas.

Il existe une autre situation dans laquelle le transtypage n'implique pas une réelle conversion mais vise plutôt à « rassurer » le compilateur. Il s'agit du transtypage vers une référence, de la forme :

```
( UnType & ) UneValeur
```

Quand le compilateur traite une telle conversion, il ne générera aucune séquence de conversion. En particulier, il n'appellera ni constructeur, ni fonction membre. Il considérera que le programmeur a voulu lui indiquer qu'il « sait ce qu'il fait » et qu'il est sûr que UneValeur peut être considérée comme une référence à UnType. Le compilateur traitera alors le résultat du *cast* comme une *lvalue*.

Dans la suite de ce chapitre, nous utiliserons fréquemment le transtypage vers une référence : c'est une opération indispensable pour gérer correctement le polymorphisme dans une hiérarchie de classes.

5.2.6 - Comparer des objets non comparables ?

Revenons à nos comparaisons et, en utilisant les fonctions membres de comparaison décrites en 5.2.3, écrivons la séquence :

```
Integer E(0);
Ville V;
if (E == V) cout << "ça alors !\n"
```

Nous y comparons l'objet E (classe Integer) à l'objet V (classe Ville). Même si cette comparaison n'a guère de sens, le compilateur l'accepte et, si nous l'exécutons,

nous avons la surprise de voir apparaître à l'écran l'affichage *ça alors !*. Le code exécuté a trouvé que les deux objets étaient identiques ! Que s'est-il passé ?

La comparaison est exprimée par E == V. C'est donc E qui reçoit le message == V et c'est la fonction membre *operator* == de Integer qui est exécutée :

```
int Integer::operator == (Object & Op2) const
{
   return valeur == ((Integer &)E2).valeur;
}
```

Dans cette exécution, l'argument Op2 est en fait l'objet V et, comme nous indiquons à la fonction de le considérer comme un objet Integer (avec le transtypage vers une référence d'Integer), le code exécuté accède à ce qu'il « pense » être le membre valeur de cet objet. Or un objet Ville a pour première donnée membre un tableau de six caractères (codePostal) et, pour l'objet V, ce membre est initialisé à une chaîne vide par le constructeur par défaut. Dans son premier élément ce tableau contient donc un code nul ('\0') et, au moment où nous avons exécuté le code, les éléments suivants contenaient aussi des codes nuls. Le code a donc comparé le membre valeur de E (qui est nul) à ce qu'il prend pour un int au début du tableau V.codePostal et qui représente aussi la valeur 0. La conclusion est donc normale, même si elle n'est pas satisfaisante.

Pour remédier à ce problème, nous sommes amenés à modifier notre hiérarchie de classes, comme le montre la figure 5.5.

Par rapport à l'implémentation précédente, l'opérateur == de Object n'est plus virtuel. Il sera donc hérité et aura le même comportement pour toute sous-classe qui ne le redéfinira pas. Nous ne l'avons pas surchargé dans Integer ni dans Ville. Il correspond donc, pour toutes les classes de la figure 5.5, à la définition :

```
int Object::operator == (Object & Op2) const
{ // n'est plus une fonction virtuelle
   sameClassAs(Op2);     // bloque en erreur si Op2 est d'une autre classe
   return isEqual(Op2); // n'est appelé que si Op2 est de la même classe
}
```

Pour tout objet Obj descendant de Object, l'envoi d'un message == AutreObjet, déclenchera le traitement suivant :

- On vérifiera que la classe de AutreObjet est la même que Obj. On appellera pour cela sameClassAs, fonction membre privée (accès *protected*) de Object.
- Si la fonction sameClassAs a bien constaté l'identité des classes, on appellera la fonction isEqual qui réalise la comparaison de deux objets de la même classe. Cette fonction est une fonction membre virtuelle pure privée (accès *protected*) de la classe Object : elle correspond à un traitement spécifique à chaque classe. Son polymorphisme nécessitera sa redéfinition dans chaque sous-classe non abstraite.

```
enum IdClasse { OBJECT, INTEGER, VILLE };
enum Erreurs { CLASS_MISMATCH = 1 };
class Object {
 protected:
   ... // etc. Cf. figure 5.2
   void erreur(Erreurs Num) const;
       // termine l'exécution avec un message d'erreur
   void sameClassAs(Object & Obj) const;
       // bloque l'exécution si le récepteur et Obj ne sont pas de la même classe
   virtual int isEqual(Object & Obj) const = 0;
       // suppose que Obj est de la même classe que le récepteur du message
 public:
   int operator == (Object & Op2) const; // n'est plus virtuelle
   ... // etc. Cf. figure 5.2
}; // class Object
class Integer : public Object {
 private:
   ... // etc. Cf. figure 5.2
 protected:
   ... // etc. Cf. figure 5.2
   int isEqual(Object & Int2) const;
       // quand elle s'exécute, on est sûr que Int2 est un Integer
 public:
   ... // etc. Cf. figure 5.2
}; // class Integer
class Ville : public Object {
 private:
   ... // etc. Cf. figure 5.2
 protected:
   ... // etc. Cf. figure 5.2
   int isEqual(Object & V2) const;
       // quand elle s'exécute, on est sûr que V2 est un objet Ville
 public:
   ... // etc. Cf. figure 5.2
}; // class Ville
```

Figure 5.5 : comparer seulement des objets comparables

Le détail du mécanisme que nous venons de décrire est explicité figure 5.6 avec la définition des fonctions sameClassAs et isEqual.

```
void Object::sameClassAs(Object & Obj) const
{   // bloque l'exécution si le récepteur et Obj ne sont pas de la même classe
    if (classId() != Obj.classId()) erreur(CLASS_MISMATCH);
}
void Object::erreur(Erreurs Num) const
{   // termine l'exécution avec un message d'erreur
    cerr << "classe " << className() << " : erreur " << Num
<<"\n";
    exit(Num);
}
int Integer::isEqual(Object & Int2) const
{   // ne sera exécutée que si on est sûr que Int2 est un Integer
    return valeur == ((Integer &)Int2).valeur;
}
int Ville::isEqual(Object & V2) const
{   // ne sera exécutée que si on est sûr que V2 est un objet Ville
    return ! strcmp(codePostal, ((Ville &) V2).codePostal);
}
```

Figure 5.6 : les fonctions privées sur lesquelles s'appuie l'opérateur ==

La fonction sameClassAs utilise deux appels de classId pour identifier les classes des objets comparés. Si ces deux classes sont distinctes, elle appelle la fonction membre privée (accès *protected*) erreur pour afficher un message d'erreur et terminer l'exécution[6].

Les deux fonctions isEqual des classes Integer et Ville réalisent la comparaison à l'égalité pour chaque classe. Leur utilisation dans l'implémentation de l'opérateur == garantit que les transtypages (Integer &) et (Ville &) seront toujours corrects.

Le lecteur peut juger trop restrictive l'implémentation choisie. S'il souhaite par exemple enregistrer, dans un tableau, des objets de classes différentes descendant de Object, il peut vouloir comparer deux éléments d'un tel tableau entre eux et s'attendre à ce que la comparaison d'un Integer et d'une Ville ne bloque pas l'exécution mais renvoie la valeur *faux*. Il suffirait pour cela que la fonction sameClassAs renvoie *vrai* ou *faux* (sans bloquer l'exécution) et que la fonction isEqual ne soit appelée que si sameClassAs a renvoyé la valeur *vrai*. Nous choisirons cette implémentation quand nous implémenterons une classe Array dans les paragraphes suivants.

---

6 Pour ne pas surcharger notre exemple, nous n'avons pas fait ici ce que l'on programmerait dans une implémentation plus professionnelle. En effet, il serait souhaitable que tous les codes d'erreur identifiés par le type énuméré Erreurs (il n'y en a qu'un, figure 5.5) soient utilisés pour indexer un tableau de chaînes de caractères qui répertorierait les messages d'erreur. Dans ce cas, si T est ce tableau, la fonction erreur pourrait se définir par : void erreur (Erreurs Num) { cerr << T[Num]; exit (Num);}

5.3 - Une classe *Array* pour répertorier des descendants de *Object*

Nous pouvons maintenant définir une classe dont les instances seront des tableaux. Ces tableaux répertorieront des objets dont les classes héritent de *Object*. Cette nouvelle classe, que nous appellerons Array, est présentée figure 5.7.

```
typedef int Boolean;
#define FALSE 0
#define TRUE !FALSE
enum IdClasse { OBJECT, INTEGER, VILLE, ARRAY };
enum Erreurs
     { CLASS_MISMATCH=1, OUT_OF_MEMORY, RANGE_CHECK,
       ASSIGNMENT, EMPTY_ELEMENT };
class Array : public Object {
 private:
   Object * * tableau;       // pointera sur l'espace alloué au tableau
   const int dimension;      // dimension du tableau
   void alloue(int Dim);
       // fonction privée : initialise le membre tableau
   int valideRang(int Index) const;
       // fonction privée : renvoie Index s'il représente le rang d'un élément
 protected:
   IdClasse classId() const { return ARRAY; }
   const char * className() const { return "Array"; }
   Boolean isEqual(Object & T2) const;
   Array & operator = (const Array & Source);
       // affectation privée : réservée à l'implémenteur
 public:
   const char * AsString() const;
   int LastIndex() const { return dimension - 1; }
   Array(int D);   // construit un tableau de D éléments
   Array(const Array & Source);   // constructeur-copie
   ~Array() { delete [] tableau; }
   Object & At(int Rang) const;
     // accesseur en consultation
   void AtPut(int Rang, Object & Obj);
     // accesseur en modification
   Boolean Includes(Object & Obj) const;
     // indique si Obj est répertorié
   Boolean FirstAt(Object & Obj, int & Rang) const;
     // cherche la première occurrence de Obj
     // renvoie FALSE et une valeur négative pour Rang si Obj n'est pas dans le
     // récepteur, sinon renvoie TRUE et affecte à Rang l'index trouvé
}; // class Array : public Object
```

Figure 5.7 : la classe Array *(première version)*

Un objet Array possède deux données membres privées tableau et dimension. La seconde, comme son nom l'indique donnera le nombre d'éléments du tableau et elle est constante. Elle doit donc être initialisée par une liste d'initialisation, comme le montre la définition du constructeur suivant :

```
Array::Array(int D) : dimension(D) { alloue(D); }
    // construit un tableau de D éléments
```

Le membre tableau représente le tableau proprement dit. Il est défini comme un pointeur sur des pointeurs d'Object et il est initialisé par la fonction membre privée alloue qui sera appelée par chaque constructeur :

```
void Array::alloue(int Dim)
{   // fonction privée : initialise le membre tableau
    tableau = new Object * [Dim];
    if (tableau == NULL) erreur(OUT_OF_MEMORY);
    // ici, l'allocation a réussi, on indique ensuite que le tableau est vide
    for (int K = 0; K < Dim; K++) tableau[K] = NULL;
}
```

Après la construction d'un objet Array, le membre tableau pointe donc sur un tableau de dimension pointeurs sur Object. Pour répertorier un objet Obj dans l'élément de rang K, on enregistrera donc l'adresse de Obj dans tableau[K]. Une instance de Array peut donc ainsi répertorier un objet Obj quelconque, pourvu que la classe de Obj hérite de Object.

Nous allons maintenant étudier l'interface proposée par la figure 5.7 pour l'utilisation de la classe Array. Cette interface est représentée par les fonctions membres publiques.

5.3.1 - Répertorier un objet dans un tableau

Nous n'avons pas défini l'opérateur [ ] pour la classe Array. Un tel opérateur ne serait en effet pas commode pour l'utilisateur puisque ce ne sont pas des objets mais des adresses d'objets que l'on enregistre dans les éléments du tableau privé tableau. Avec un tel opérateur, l'utilisateur devrait par exemple écrire

```
Ville V("73410", "ALBENS");
Array T(100);
// on suppose que [] est redéfini pour la classe Array
T[0] = &V;
```

et il devrait constamment avoir à l'esprit que ce ne sont pas les objets mais leurs adresses qu'il manipule. Nous avons décidé de cacher cette manipulation en l'encapsulant. Cela nous permet aussi de mieux la contrôler. En effet, si nous laissions l'utilisateur écrire directement :

```
// on suppose que T est une instance de Array de dimension supérieure à 10
// et que f est une fonction membre de l'objet pointé par T[10]
T[10]->f(...);
```

et si aucun objet n'a été répertorié par T[10], la valeur de T[10] est une adresse indéterminée et le résultat de l'exécution de l'instruction précédente est imprévisible.

Nous avons donc choisi :

- de valider toutes les indexations sur le membre privé tableau avec la fonction membre privée valideRang,
- de gérer l'accès aux éléments du tableau par les fonctions membres AtPut (enregistrement) et At (consultation).

Ces trois fonctions membres sont présentées à la figure 5.8. La fonction valideRang joue le rôle d'un filtre : elle renvoie la valeur qu'elle a reçue en argument si cette valeur est un index valide.

```
int Array::valideRang(int Index) const
{   // fonction privée : renvoie Index s'il représente le rang d'un élément
    if (Index < 0   ||   Index >= dimension)
       erreur(RANGE_CHECK);
    return Index;
} // int Array::valideRang(int Index)
void Array::AtPut(int Rang, Object & Obj)
{   // accesseur en modification
    tableau[valideRang(Rang)] = & Obj;
}
Object & Array::At(int Rang) const
{   // accesseur en consultation
    Object * Ptr = tableau[valideRang(Rang)];
    if (Ptr == NULL) erreur(EMPTY_ELEMENT);
    return *Ptr;
}
```

Figure 5.8 : des accesseurs pour la classe Array

La fonction AtPut reçoit en argument une valeur d'index et un objet à enregistrer. Elle valide l'index puis affecte l'adresse de l'objet à l'élément du tableau désigné par cet index.

La fonction At reçoit en argument une valeur d'index. Elle doit renvoyer l'objet enregistré dans l'élément correspondant. Son implémentation demande une double vérification :

- validation de l'index,
- validation de l'objet : il faut en effet s'assurer qu'un objet a bien été enregistré dans l'élément désigné par l'index.

Cette seconde validation ne peut être effectuée que si, à la construction de l'objet Array, tous les éléments ont reçu une valeur déterminée indiquant l'absence d'objet. C'est pour cette raison que, dans la fonction membre alloue (Cf. paragraphe précédent), nous avons affecté la valeur NULL à chaque élément de tableau.

Avec l'implémentation des fonctions proposée à la figure 5.8, nous disposons d'accesseurs fiables qui renverront un objet ou bloqueront l'exécution si l'élément que l'on consulte n'existe pas ou ne répertorie aucun objet. Cependant, la techni-

que utilisée pour valider la présence d'un objet dans un élément impose d'examiner cet élément avant d'utiliser la valeur de pointage qu'il contient. Ainsi, dans la fonction At, on vérifie que l'élément ne contient pas NULL avant d'utiliser cet élément pour accéder à l'objet qu'il désigne.

Cette opération sera fréquemment exécutée, non seulement pour les accès des fonctions membres que nous venons de définir mais aussi, par exemple, dans la fonction membre Includes :

```
Boolean Array::Includes(Object & Obj) const
{  // indique si Obj est répertorié
   for (int K = 0; K < dimension; K++)
       if (tableau[K] != NULL  &&  *tableau[K] == Obj)
          return TRUE;
   return FALSE;
}
```

Pour chaque accès à un élément de tableau, on doit effectuer la comparaison tableau[K] != NULL avant de comparer éventuellement l'objet pointé par cet élément avec Obj[7]. Si l'on souhaite accélérer ce genre d'opération, il faut pouvoir supprimer cette comparaison. C'est ce que nous allons maintenant étudier.

5.3.2 - Un objet pour représenter l'absence d'objet

Supposons qu'il existe un objet descendant de Object, distinct de tout autre objet et qui représente l'absence d'information dans un élément d'une instance de Array. Si nous appelons cet objet NIL, la fonction alloue doit être réécrite comme suit.

```
void Array::alloue(int Dim)
{  // fonction privée : initialise le membre tableau
   tableau = new Object * [Dim];
   if (tableau == NULL) erreur(OUT_OF_MEMORY);
   // ici, l'allocation a réussi, on indique ensuite que le tableau est vide
   for (int K = 0; K < Dim; K++) tableau[K] = &NIL;
}
```

Par rapport à la version précédente, le changement paraît minime : au lieu d'affecter NULL à chaque élément de tableau on affecte l'adresse de l'objet NIL. Cependant la différence est de taille : maintenant, chaque élément de tableau, qu'il soit « vide » ou non, pointe vers un objet. Et l'on peut donner une version plus performante de la fonction membre Includes :

---

7 La solution qui consisterait à n'effectuer qu'une seule comparaison comme
if (tableau[K] == &Obj)
n'est pas satisfaisante car elle ferait fonctionner l'opérateur == comme un opérateur d'identité et non d'égalité : cet opérateur ne renverrait TRUE que quand un objet est comparé à lui-même.

```
int Array::Includes(Object & Obj)
{  // indique si Obj est répertorié
   for (int K = 0; K < dimension; K++)
       if (*tableau[K] == Obj) return TRUE;
   return FALSE;
}
```

Maintenant, la comparaison à NULL de tableau[K] a disparu de l'itération : celle-ci est donc accélérée. Pour que la fonction Includes fonctionne correctement, il suffit que l'objet NIL réponde au message == en indiquant qu'il n'est égal à aucun autre objet que lui-même.

```
enum IdClasse { OBJECT, INTEGER, VILLE, ARRAY, UNDEFINED
};
enum Erreurs
     { CLASS_MISMATCH=1, OUT_OF_MEMORY, RANGE_CHECK,
       ASSIGNMENT, EMPTY_ELEMENT, NIL_DELETE };

class UndefinedObject : public Object {
 // une seule instance permanente : NIL
 private:
   IdClasse classId() const { return UNDEFINED; }
   const char * className() const
      { return "UndefinedObject"; }
   Boolean isEqual(Object & Obj) const;
   UndefinedObject(UndefinedObject &);
     // constructeur privé : copie interdite
 public:
   UndefinedObject() {}
   virtual const char * AsString() const
     { return "UndefinedObject()"; };
   void operator delete(void *);   // signale une erreur
}; // class UndefinedObject : public Object

UndefinedObject NIL;
   // Objet caractérisant l'absence d'information significative
```

Figure 5.9 : un objet NIL

La figure 5.9 propose une implémentation pour l'objet NIL. Celui-ci doit être un descendant de Object : nous définissons donc une classe dérivée de Object : la classe UndefinedObject dont NIL sera l'unique instance. Cette classe doit être non abstraite : nous devons définir comme des fonctions complètes, les deux fonctions membres virtuelles pures héritées de Object. La fonction membre AsString renvoie la chaîne "UndefinedObject()" et la comparaison est définie par :

```
Boolean UndefinedObject::isEqual(Object & Obj) const
{   // l'égalité n'est reconnue que pour NIL == NIL
    return ( (UndefinedObject *) this == & Obj );
}
```

Cette fonction ne renverra TRUE que si l'objet se compare à lui-même. On remarquera le transtypage effectué avant la comparaison de this avec l'adresse de l'argument Obj. Ce transtypage est indispensable pour éviter une erreur de compilation. En effet, la fonction membre isEqual est une fonction membre constante : pour une telle fonction, le type de this est const UndefinedObject * const qui ne peut être converti en Object *. Le compilateur joue ainsi bien son rôle de « gardien » dans une fonction membre constante. Nous forçons ici le type en toute sécurité car nous allons garantir (Cf. ci-après) que NIL est bien la seule instance de sa classe et que la fonction membre isEqual de cette classe ne peut être appelée que pour cet objet.

L'objet NIL est défini à la fin de la figure 5.9 comme une variable globale. Pour que cette variable soit accessible à chaque utilisateur de la classe, il faudra rassembler dans un fichier-programme Lib.cpp les définitions des classes et des fonctions membres et inclure, dans ce fichier, la définition de NIL. Ce fichier sera compilé, et c'est le résultat de la compilation Lib.o (ou Lib.obj) qui sera fourni à chaque utilisateur.

Pour que NIL soit l'unique instance de sa classe, il faut que l'utilisateur de notre hiérarchie de classes n'instancie aucun autre objet de la classe UndefinedObject. On peut se contenter, comme nous l'avons fait ici, de faire confiance aux utilisateurs. On pourrait aussi gérer, comme nous l'avons vu au chapitre 3, un compteur d'instances qui bloquerait tout constructeur de la classe UndefinedObject dès qu'on essaie de construire une seconde instance.

Cependant, si la confiance en l'utilisateur suppose qu'il ne transgresse pas une règle (*ne pas instancier UndefinedObject*), elle laisse la porte ouverte à des instanciations plus discrètes. Imaginons que l'utilisateur définisse une fonction telle que :

```
void f(UndefinedObject Obj) {....}
```

alors, un appel tel que f(NIL) créerait une copie de l'objet NIL. Pour cette raison, nous déclarons le constructeur-copie comme fonction membre privée dans la classe UndefinedObject. Un appel comme f(NIL) sera refusé à la compilation, le constructeur-copie étant inaccessible.

On remarquera enfin que la classe UndefinedObject déclare l'opérateur *delete*. Nous le définissons de la manière suivante :

```
void UndefinedObject::operator delete(void *)
{   // signale une erreur : on ne doit pas détruire NIL
    cerr << "Tentative de destruction de l'objet NIL\n";
    exit(NIL_DELETE);
}
```

Cet opérateur bloquera l'exécution s'il est appelé. En effet, si l'utilisateur récupère l'adresse de l'objet NIL, il ne doit pas pouvoir détruire cet objet. La présentation de cette dernière fonction membre appelle plusieurs remarques :

- On peut redéfinir, pour une classe donnée, l'opérateur *delete* comme l'opérateur *new*.
- Ces deux opérateurs sont implicitement des fonctions membres statiques car ils n'opèrent pas sur les objets : *new* est appelé avant le constructeur et *delete* après le destructeur[8].
- C'est pour cette raison que nous n'avons pas pu utiliser la fonction membre *erreur* héritée de *Object* dans le code de notre opérateur *delete*. Nous n'avons en effet aucun objet pour qualifier l'appel.

Signalons enfin que, pour que la destruction *delete P* soit correctement effectuée (ou interdite dans le cas de NIL) pour tout objet descendant de *Object* et pointé par un pointeur P de type *Object **, il faut que les destructeurs soient virtuels. C'est pour cette raison que le destructeur de *Object* doit être explicitement défini ainsi :

```
class Object {
   ... // etc.
 public:
   ... // etc.
   virtual ~Object() {}
}; // class Object
```

5.3.3 - Récapitulons les accesseurs de *Array*

Il est sans doute utile de revenir sur les accesseurs de la classe Array pour :

- préciser leur implémentation, en tenant compte de l'objet NIL,
- ajouter quelques fonctions membres supplémentaires.

La figure 5.10 présente la définition de ces fonctions membres. Plusieurs sont nouvelles. Nous allons les commenter mais, auparavant, précisons la définition de la fonction membre At :

```
Object & Array::At(int Rang) const
{  // accesseur en consultation pour les éléments non vides
   Object * Ptr = tableau[valideRang(Rang)];
   if (Ptr == &NIL) erreur(EMPTY_ELEMENT);
   return *Ptr;
}
```

8 En effet, si *CCC* est une classe, dans l'exécution de *new CCC(...)*, l'allocation-mémoire a lieu avant la construction de l'objet. De même, si P pointe sur un objet *CCC*, l'exécution de *delete P* appelle le destructeur avant de libérer la place occupée par l'objet.

```
class Array {
 private:
    ... // etc. Cf. figure 5.7
 public:
    ... // etc. Cf. figure 5.7
    Boolean EmptyAt(int Rang) const;
        // indique si l'élément contient ou non un objet
    Object & At(int Rang) const;
        // accesseur en consultation pour les éléments non vides
    void AtPut(int Rang, Object & Obj);
        // accesseur en modification
    void RemoveAt(int Rang);
        // supprime l'enregistrement de l'objet d'index Rang
    Boolean Includes(Object & Obj) const;
        // indique si Obj est répertorié
    Boolean FirstAt(Object & Obj, int & Rang) const;
        // renvoie FALSE et une valeur <0 pour Rang si Obj n'est pas présent
}; // class Array : public Object
```

Figure 5.10 : les accesseurs de la classe Array

Cette fonction ne renvoie l'objet d'index Rang que si cet objet existe : il faut, pour cela, que l'élément d'index Rang ne désigne pas l'objet NIL. Un utilisateur de la classe Array ne devra donc envoyer un message At(R) à une instance de cette classe que s'il est sûr que l'élément de rang R répertorie un objet. Pour cela, nous avons défini la fonction membre EmptyAt :

```
Boolean Array::EmptyAt(int Rang) const
{   // indique si l'élément contient ou non un objet
    return (tableau[Rang] == &NIL);
}
```

ce qui permettra à l'utilisateur d'écrire une séquence telle que :

```
Array T(100);
... // ici, on enregistre des objets dans T
if ( !T.EmptyAt(10) )  cout << T.At(10).AsString();
```

Le mécanisme de contrôle de la présence ou de l'absence d'un élément est ainsi encapsulé dans les accesseurs : l'utilisateur n'a pas à connaître l'existence de l'objet NIL. Bien sûr, il faut pouvoir aussi enlever un objet. C'est le rôle de la fonction membre RemoveAt :

```
void Array::RemoveAt(int Rang)
{   // supprime l'enregistrement de l'objet d'index Rang
    tableau[valideRang(Rang)] = &NIL;
}
```

Nous supposons ici qu'une instance de Array n'est pas « propriétaire » des objets qu'elle répertorie. Ainsi, l'exécution de T.RemoveAt(K) n'implique pas la destruction de l'objet qui était répertorié par T[K]. Certaines librairies de classes-conteneurs offrent à l'utilisateur le choix entre des conteneurs comme nos ins-

tances de Array et des conteneurs qui sont les seuls à désigner les objets qu'ils répertorient[9] et qui sont alors propriétaires de ces objets.

Enfin les fonctions membres Includes et FirstAt permettent la recherche d'une information. Elles sont définies de la manière suivante :

```
int Array::Includes(Object & Obj) const
{   // indique si Obj est répertorié
    for (int K = 0; K < dimension; K++)
        if (*tableau[K] == Obj) return TRUE;
    return FALSE;
}
int Array::FirstAt(Object & Obj, int & Rang) const
{   // renvoie TRUE et affecte N à Rang si la première occurrence de Obj est dans
    // l'élément d'index Rang. Renvoie FALSE et une valeur négative pour Rang
    // si Obj n'est pas dans le récepteur
    for (Rang = 0; Rang < dimension; Rang++)
        if (*tableau[Rang] == Obj) return TRUE;
    Rang = -1; return FALSE;
}
```

Ces deux fonctions comparent directement chaque élément du tableau à l'objet cherché. Si l'élément est vide, c'est l'objet NIL qui est comparé à cet objet.

Les fonctions Includes et FirstAt effectuent des itérations sur les éléments d'une instance de Array. On peut aussi souhaiter que l'utilisateur de la classe Array puisse lui-même programmer directement des itérations. Nous allons maintenant étudier les techniques à mettre en oeuvre pour de telles opérations.

5.3.4 - Itérer sur le contenu d'une instance de *Array*

Examinons la séquence de la figure 5.11. On y construit un objet Tab, instance de Array. On enregistre ensuite des objets Integer dans les éléments de rangs pairs de Tab. On calcule enfin dans la variable Somme la somme des valeurs des entiers de Tab.

Telle qu'elle est exprimée, l'itération de cumul des entiers appelle deux remarques importantes.

Faut-il transtyper un objet renvoyé par un conteneur ?

Dans l'itération, chaque entier est obtenu avec le message At(Index) adressé à Tab. Le résultat de ce message est normalement du type Object & : nous l'avons transtypé en (Integer &) pour que le compilateur accepte de faire intervenir l'entier dans une addition. En effet, quand un conteneur renvoie une référence sur un objet contenu Obj et que l'on veut adresser un message à Obj, deux cas sont à considérer :

- Le message correspond à une fonction virtuelle de Object. Dans ce cas, aucun transtypage n'est nécessaire et le choix du traitement sera

9 Ces objets sont alors obligatoirement créés avec une allocation dynamique. Ils sont détruits dès qu'ils sont supprimés du conteneur.

fait dynamiquement à l'exécution. On pourrait ainsi écrire, dans l'itération de la figure 5.11 :

```
cout  << Tab.At(Index).AsString();
```

car AsString est une fonction virtuelle disponible pour chaque descendant de Object.

- Le message ne correspond pas à une fonction virtuelle héritée de Object. Il faut alors impérativement indiquer au compilateur le type de l'objet renvoyé pour qu'il puisse valider l'appel de la fonction membre[10].

```
Array Tab(10);
// on enregistre des entiers dans certains éléments de Tab
for (int K = 0; K < 10; K = K + 2)
    Tab.AtPut(K, *(new Integer(K)));
// on calcule la somme des valeurs de Tab
int Somme = 0;
for (int Index = 0; Index <= Tab.LastIndex(); Index++)
    if ( ! Tab.EmptyAt(Index) )
       Somme = Somme + (Integer &) Tab.At(Index);
cout << "Somme des valeurs : " << Somme  <<"\n";
```

Figure 5.11 : une itération sur les éléments d'une instance de Array

Le programmeur doit considérer les éléments vides

Dans l'itération de la figure 5.11, le programmeur traite un à un les éléments du tableau et il doit lui-même exclure du calcul ceux qui sont vides et qui désignent l'objet NIL. Nous l'avons fait en utilisant la fonction membre EmptyAt. Pour libérer l'utilisateur de la classe Array d'une telle contrainte, il faut définir une classe d'itérateurs.

5.3.5 - Une classe d'itérateurs

Pour associer à notre classe Array un mécanisme d'itération, nous choisissons de définir une classe d'itérateurs. Si nous appelons cette classe ArrayIterator, l'exemple décrit figure 5.12 montre comment on pourrait l'utiliser.

10 On constate ici la limitation d'un langage comme C++, dont le code compilé sera certes plus performant mais moins dynamique que la séquence équivalente écrite en Smalltalk et interprétée à l'exécution.

```
Array Tab(100);
... // ici, on enregistre des objets dans Tab
ArrayIterator IterTab(Tab);
    // IterTab est un itérateur sur l'objet Tab
// on veut afficher tous les objets contenus dans Tab
while (IterTab) {
   cout << IterTab.Current().AsString() << '\n';
   IterTab++;
}
```

Figure 5.12 : utiliser un itérateur sur une instance de Array

Dans cet exemple, on instancie l'itérateur IterTab sur le tableau Tab. Puis on exprime une itération pour afficher les objets contenus dans Tab :

- L'itération est contrôlée par l'expression while (IterTab). Pour traiter cette expression, le compilateur cherchera s'il peut convertir IterTab en une valeur entière. En effet, l'énoncé while (Condition) attend un entier pour Condition. Dans notre classe ArrayIterator nous surchargerons donc le transtypage int(), de telle sorte qu'il renvoie une valeur nulle (FALSE) quand tous les objets du conteneur auront été examinés.
- Dans le corps de l'itération, le message Current() est envoyé à l'itérateur pour obtenir chaque objet de Tab. La fonction membre Current de la classe ArrayIterator renverra donc l'objet courant sur lequel l'itérateur est positionné.
- Enfin, pour passer d'un objet au suivant, on exécute IterTab++. Il faut donc surcharger l'opérateur postfixé ++ pour la classe, de telle sorte qu'il fasse passer l'itérateur au prochain élément du conteneur s'il existe.

Conformément à l'interface que nous venons de spécifier, la définition de la classe peut s'écrire comme proposé figure 5.13.

Un objet ArrayIterator identifie le tableau sur lequel il itère par la donnée membre privée leTableau. Il dispose aussi d'une seconde donnée membre privée, courant, qui donnera le rang de l'objet courant dans l'itération. La fonction membre privée prochain est définie par :

```
void ArrayIterator::prochain()
{  // fonction privée, avance courant jusqu'au prochain objet ou la fin du tableau
   courant++;
   while (courant <= leTableau.LastIndex()) {
      if (!leTableau.EmptyAt(courant)) return;
      courant++;
   }
   return;
}
```

Dans cette fonction, on commence par passer à l'élément qui suit l'objet courant (courant++) puis on itère jusqu'à trouver un objet ou arriver à la fin du tableau. Si

celle-ci est atteinte, la donnée membre courant aura dépassé le dernier index du tableau.

```
class ArrayIterator {
 // itérateurs sur un tableau
 private:
   const Array & leTableau; // tableau sur lequel on itère
   int courant; // rang de l'élément courant ou leTableau.dimension
   void prochain(); // avance courant jusqu'au prochain objet s'il existe
 public:
   ArrayIterator (Array & T)
      : courant(-1), leTableau(T)
      { prochain(); }
   Object & Current();
      // renvoie l'objet courant sans avancer l'itérateur
   Object & operator ++ (int);
      // ++ postfixé, renvoie le courant et passe au suivant s'il existe
   operator int();
      // renvoie 0 si on a traité le dernier élément
   void Restart();
      // réinitialise l'itérateur
}; // class ArrayIterator
```

Figure 5.13 : la classe ArrayIterator

On peut maintenant comprendre la définition du constructeur de la figure 5.13. Celui-ci, par sa liste d'initialisation, fixe la donnée membre leTableau et affecte la valeur -1 à courant. Ensuite, le corps du constructeur déclenche une première exécution de la fonction membre prochain. Comme nous venons de le voir, cette exécution passe à l'élément de rang 0 (courant++) puis avance jusqu'au premier objet du conteneur. L'itérateur est ainsi correctement initialisé par son constructeur.

La gestion des membres privés courant et prochain permet aussi de comprendre la définition de l'opérateur de transtypage int() :

```
ArrayIterator::operator int()
{   // renvoie 0 (FALSE) si on a traité le dernier élément
    return !(courant > leTableau.LastIndex());
}
```

La définition de l'opérateur ++ nécessite que l'on précise si la forme surchargée est postfixée ou préfixée. Pour cela on suit la convention C++ qui ajoute à la surcharge postfixée un argument de type int non utilisé par ailleurs et dont le rôle est seulement de permettre la distinction des deux signatures (la surcharge du ++ préfixé ne doit spécifier, elle, aucun argument). La définition de notre opérateur peut donc s'exprimer par :

```
Object & ArrayIterator::operator ++ (int)
{   // ++ postfixé, renvoie le courant et passe au suivant s'il existe
    Object & R = Current();
    prochain();
    return R;
}
```

On notera que la fonction renvoie l'objet courant avant de passer au suivant. Cela permet une concision d'écriture telle que :

```
while (IterTab) {
    cout << IterTab++.AsString() << '\n';
}
```

Enfin, il faut signaler que la figure 5.13 déclare aussi une fonction membre de réinitialisation que l'on peut définir par :

```
void ArrayIterator::Restart()
{   // réinitialise l'itérateur
    courant = -1; prochain();
}
```

Remarquons pour conclure que notre implémentation de la classe ArrayIterator est indépendante de l'implémentation de la classe Array : elle n'utilise que des membres publics de la classe Array. Ce choix nous dispense de déclarer la classe ArrayIterator amie de Array. Il correspond à une décision pédagogique : nous avons jugé que l'étude de la classe ArrayIterator serait ainsi plus simple pour le lecteur. Dans les librairies professionnelles de classes-conteneurs, au contraire, les classes d'itérateurs ont accès aux données privées des conteneurs pour coller à l'implémentation et garantir ainsi les performances.

5.4 - Une hiérarchie de classes-conteneurs

La classe Array, que nous venons de construire est une classe d'objets-conteneurs. On peut concevoir bien d'autres types d'objets-conteneurs : des ensembles, des piles, des suites ordonnées, etc. Généralement, un environnement orienté objets fournit au programmeur un ensemble de classes de ce genre, souvent inspiré de la hiérarchie de classes définies avec le langage Smalltalk. Le langage C++, en tant que tel, n'inclut en l'état actuel de sa définition, aucune autre classe prédéfinie que celles qui gèrent les flux de caractères. Mais la plupart des éditeurs de compilateurs C++ fournissent une librairie de classes conçues selon les principes que nous venons d'exposer. Une telle librairie fournit la plupart du temps plusieurs classes d'objets-conteneurs. Nous allons, pour terminer ce chapitre, nous intéresser aux mécanismes qui gèrent en général l'architecture de ces classes.

5.4.1 - Une classe abstraite : *Collection*

Nous ne construirons pas une hiérarchie complète de classes-conteneurs : cela représenterait un travail considérable qui n'a pas sa place ici. Nous allons seule-

ment, à travers un exemple, illustrer la puissance du polymorphisme dans un environnement de programmation par objets. Si l'on construit une librairie de classes-conteneurs, on est amené à spécifier, au sommet de cette hiérarchie, une classe abstraite qui rassemblera les propriétés communes à toutes nos classes-conteneurs.

Appelons Collection cette classe abstraite. Un exemple simplifié de définition d'une telle classe est présenté figure 5.14.

En effet, notre classe Collection hérite de Object : cela permettra d'obtenir des objets-conteneurs qui contiennent d'autres objets-conteneurs. Un objet-conteneur répertorie des objets descendants de la classe Object : si Collection est une sous-classe de Object, un conteneur descendant de Collection est donc aussi un descendant de Object.

Telle que nous l'avons définie, la classe Collection est abstraite pour trois raisons :

- elle ne redéfinit pas la fonction virtuelle pure isEqual, héritée de Object,
- elle ne redéfinit pas non plus la fonction virtuelle pure AsString, héritée de Object,
- elle introduit une nouvelle fonction virtuelle pure NewIterator.

Toute classe descendant de Collection devra donc, pour être instanciable, définir complètement les trois fonctions membres précédentes.

La classe Collection définit une dernière fonction membre, non virtuelle, la fonction Includes. Cette fonction membre correspond à la propriété générique d'appartenance que l'on retrouvera dans toute sous-classe de Collection. C'est à travers l'implémentation de cette fonction membre que nous mettrons en relief la gestion du polymorphisme. Bien sûr, dans une bibliothèque de classe complète, la classe Collection définirait bien d'autres propriétés génériques.

Toujours sur la figure 5.14, nous trouvons une troisième définition de classe : la classe CollectionIterator, elle aussi abstraite, qui sera la racine de la hiérarchie de nos classes d'itérateurs. A chaque classe d'objets-conteneurs nous ferons correspondre une classe d'itérateurs. Remarquons aussi que la classe CollectionIterator n'est pas dérivée de Object. En effet, les itérateurs sont des objets très particuliers, qui servent à gérer des traitements répétitifs : nous n'avons aucune raison d'envisager de les comparer entre eux ni de les enregistrer dans un conteneur.

Observons pour terminer que la classe CollectionIterator est à la fois déclarée et définie figure 5.14. En effet, la fonction membre NewIterator de la classe Collection fait référence à CollectionIterator par son type renvoyé. Il faut donc indiquer au compilateur que l'identificateur CollectionIterator désigne une classe : c'est le rôle de la déclaration[11].

11 On aurait pu aussi placer la définition de la classe CollectionIterator avant celle de Collection. Nous ne l'avons pas fait pour conserver à la figure une cohérence pédagogique : il est plus naturel de présenter Collection avant CollectionIterator.

```
typedef int Boolean;
enum IdClasse { OBJECT, UNDEFINED, COLLECTION, /* etc. */ };
class Object {
 protected:
   virtual IdClasse classId() const { return OBJECT; }
   virtual const char * className() const
      { return "Object"; }
   Boolean sameClassAs(Object & Obj) const
      { return  classId() == Obj.classId(); };
   void erreur(Erreurs Num) const;
   virtual Boolean isEqual(Object & Obj) const = 0;
 public:
   virtual const char * AsString() const = 0;
   Boolean operator == (Object & Op2) const
   { // comparaison de deux descendants de Object
      return sameClassAs(Op2) && isEqual(Op2);
   }
   Boolean operator != (Object & Op2) const
      { return ! operator == (Op2); }
   virtual ~Object() {}
}; // class Object
class CollectionIterator;
class Collection : public Object {
 // classe abstraite, dont descendront les collections
 protected:
   virtual IdClasse classId() const { return COLLECTION; }
   virtual const char * className() const
      { return "Collection"; }
 public:
   virtual CollectionIterator & NewIterator() const = 0;
      // renvoie un itérateur de la classe appropriée
   Boolean Includes(Object & Obj) const;
      // indique si Obj est répertorié dans la collection
}; // class Collection : public Object
class CollectionIterator {
 // classe abstraite, dont descendront toutes les classes d'itérateurs
 public:
   virtual Object & Current() = 0;
      // renvoie l'objet courant sans avancer l'itérateur
   virtual Object & operator ++ (int) = 0;
      // ++ postfixé, renvoie le courant et passe au suivant s'il existe
   virtual operator int() = 0;
      // renvoie 0 si on a traité le dernier élément
   virtual void Restart() = 0;
      // réinitialise l'itérateur
}; // class CollectionIterator : public Object
```

Figure 5.14 : définition de la classe abstraite Collection

Cette figure rappelle d'abord la classe *Object*, telle que nous l'avions définie. Pour obtenir le polymorphisme complet de la fonction membre Includes, nous allons nous appuyer sur le mécanisme des itérateurs. Intéressons-nous d'abord au rôle de la fonction membre NewIterator.

5.4.2 - La fonction membre *NewIterator*

La figure 5.15 présente la définition de la classe Array intégrée à la hiérarchie de la librairie que nous sommes en train de construire. Par rapport aux définitions du paragraphe 5.3, les changements sont peu nombreux :

- la classe Array n'est plus dérivée directement de *Object*, mais de Collection,
- la fonction membre Includes n'est plus définie : elle sera héritée de Collection,

```
class Array : public Collection {
 private:
   Object * * tableau;      // pointera sur l'espace alloué au tableau
   const int dimension;     // dimension du tableau
   void alloue(int Dim);    // initialise le membre tableau
   int valideRang(int Index) const; // renvoie Index s'il est valide
 protected:
   IdClasse classId() const { return ARRAY; }
   const char * className() const { return "Array"; }
   Boolean isEqual(Object & T2) const ;
   Array & operator = (const Array & Source);
      // réservée à l'implémenteur
 public:
   const char * AsString() const;
   CollectionIterator & NewIterator() const;
   int LastIndex() const { return dimension - 1; }
   Array(int Dimension); // construit un tableau de Dimension éléments
   Array(const Array & Copie); // constructeur-copie
   ~Array() { delete [] tableau; }
   Boolean EmptyAt(int Rang) const;
      // indique si l'élément contient ou non un objet
   Object & At(int Rang) const;
      // accesseur en consultation pour les éléments non vides
   void AtPut(int Rang, Object & Obj);
      // accesseur en modification
   void RemoveAt(int Rang);
      // supprime l'enregistrement de l'objet d'index Rang
   Boolean FirstAt(Object & Obj, int & Rang) const;
      // renvoie FALSE et une valeur négative pour Rang si Obj non trouvé
}; // class Array : public Object
```

Figure 5.15 : la classe Array, *dérivée de* Collection

- la fonction membre NewIterator est déclarée : elle n'est donc plus virtuelle pure et la classe est instanciable.

La fonction NewIterator sert à générer un itérateur pour l'objet-conteneur qui reçoit le message NewIterator. Pour la classe Array, nous la définissons comme suit :

```
CollectionIterator & Array::NewIterator() const
{  // renvoie un itérateur sur l'objet Array récepteur
   ArrayIterator * P = new ArrayIterator(*this);
   return *P;
}
```

Pour comprendre la définition de cette fonction, il faut d'abord supposer que la classe ArrayIterator est définie. Nous ne redécrirons pas cette classe que nous avons étudiée en 5.3.5. Signalons seulement que, dans notre nouvelle librairie de classes-conteneurs, sa définition est identique à celle faite à la figure 5.13, à une exception près : ArrayIterator est dérivée de CollectionIterator.

Plusieurs remarques méritent d'être faites à propos de la fonction NewIterator :

- Bien qu'elle renvoie un objet ArrayIterator son entête indique CollectionIterator comme type renvoyé. En effet, si nous avions déclaré la fonction membre Array::NewIterator avec le type renvoyé ArrayIterator, le compilateur l'aurait considéré comme différente de la fonction virtuelle pure Collection::NewIterator (Cf. figure 5.14) et la classe Array serait restée abstraite.
- Grâce au transtypage classe dérivée → classe de base induit par l'héritage, le compilateur acceptera que l'objet ArrayIterator créé dynamiquement par la fonction membre NewIterator soit renvoyé comme un objet CollectionIterator.
- Remarquons enfin que le type renvoyé n'est pas réellement CollectionIterator mais CollectionIterator &. Ce transtypage vers une référence (Cf. 5.2.5) est indispensable pour que le compilateur se contente de considérer l'objet renvoyé comme une référence à CollectionIterator et n'essaie pas de créer par copie un objet CollectionIterator : il ne le pourrait pas car cette dernière classe est abstraite.

```
Array Tab(10);
... // ici, on enregistre des objets dans certains éléments de Tab
// itération d'affichage des objets de Tab
CollectionIterator & Iter = Tab.NewIterator();
cout << "Contenu du tableau Tab :\n";
while (Iter) {
   cout << Iter.Current().AsString()  << "\n";
   Iter++;
}
delete & Iter; // car Iter a été obtenu par allocation dynamique
```

Figure 5.16 : utilisation de la fonction NewIterator

La figure 5.16 montre une utilisation d'un itérateur obtenu avec la fonction membre NewIterator. On remarquera que, pour le compilateur, l'objet Iter est toujours considéré comme une référence à CollectionIterator. Cependant, l'exécution appellera bien, pour Iter, les fonctions membres de la classe ArrayIterator. En effet, toutes les fonctions membres de nos classes d'itérateurs sont virtuelles. Elles sont donc sélectionnées, à l'exécution, en fonction de la classe de l'itérateur auquel elles s'appliquent.

Il n'est pas inutile, dans cet exemple, de revenir sur la création de l'itérateur Iter. Cette instanciation est obtenue par l'exécution de l'instruction :

```
CollectionIterator & Iter = Tab.NewIterator();
```

Pourtant, sachant que l'itérateur renvoyé est dans notre cas un ArrayIterator le programmeur débutant aurait certainement trouvé plus simple d'écrire :

```
ArrayIterator Iter = Tab.NewIterator();
    // erreur de compilation : Cannot convert 'CollectionIterator' to 'ArrayIterator'
```

Il est normal que le compilateur refuse cette écriture, car la fonction membre virtuelle renvoie par définition un CollectionIterator et il n'y a pas de conversion possible de la classe de base vers la classe dérivée comme nous l'avons vu au chapitre 4. Notre programmeur débutant peut alors améliorer son essai en écrivant :

```
ArrayIterator & Iter = Tab.NewIterator();
    /* erreur de compilation: Cannot initialize 'ArrayIterator &' with
    'CollectionIterator' */
```

Il essuie une nouvelle déconvenue car une référence à un type TTT doit être initialisée avec une variable du type TTT. Bien sûr on peut régler ce problème en écrivant :

```
ArrayIterator & Iter = (ArrayIterator &)
Tab.NewIterator(); // OK, grâce au cast
```

Mais le lecteur conviendra que l'écriture de la figure 5.16 est quand même plus simple. Et, grâce au mécanisme des fonctions virtuelles, Iter, qui est considéré par le compilateur comme une référence à CollectionIterator, sera traité à l'exécution comme l'objet qu'il représente : une instance de ArrayIterator.

5.4.3 - Le polymorphisme de la fonction membre *Includes*

La fonction membre Includes est déclarée figure 5.14 dans la classe Collection par :

```
Boolean Includes(Object & Obj) const;
    // includes polymorphe : indique si Obj est répertorié
```

C'est une fonction membre non virtuelle qui est donc héritée par toute classe descendante de Collection. Elle correspond à un comportement générique de tout conteneur qui pourra, à la réception du message Includes(XXX), indiquer s'il contient ou non l'objet XXX. Cette fonction peut être définie de manière à opérer correctement dans toute classe descendant de Collection. Cette définition est présentée figure 5.17.

```
Boolean Collection::Includes(Object & Obj) const
{  // includes polymorphe : indique si Obj est répertorié
     CollectionIterator & Iter = this->NewIterator();
     while (Iter) {
          if (Iter.Current() == Obj) { // on a trouvé l'objet
               delete & Iter;
               return TRUE;
          }
          Iter++;
     }
     // on a parcouru tout le conteneur sans succès
     delete & Iter;
     return FALSE;
}
```

Figure 5.17 : la fonction membre Includes

Le principe de l'implémentation de cette fonction est simple. Elle utilise un itérateur pour parcourir le conteneur jusqu'à rencontrer l'objet Obj (recherche fructueuse) ou jusqu'à épuisement de l'itérateur (échec de la recherche).

Bien sûr, ce traitement est applicable à toute classe descendante de Collection pourvu que l'on ait défini l'itérateur correspondant : le polymorphisme de Includes repose sur le polymorphisme des itérateurs.

Les mécanismes de l'héritage permettent ainsi de définir, au plus haut niveau d'une hiérarchie, des comportements génériques dont on peut garantir le fonctionnement dans toute classe descendante. C'est un atout précieux pour le programmeur qui peut ainsi rapidement obtenir une classe descendante opérationnelle. Si par la suite, il éprouve le besoin d'améliorer les performances parce qu'il juge que le comportement générique n'est pas optimisé pour une classe particulière, il reste toujours libre de redéfinir la fonction membre correspondante dans cette classe particulière.

5.4.4 - Pour terminer, une implémentation rapide

Pour clore ce chapitre, nous présentons ci-après la définition d'une classe Set et de sa classe d'itérateurs (SetIterator). Un objet Set est un ensemble d'objets. Comme tout ensemble, il ne peut contenir deux objets identiques.

L'implémentation que nous proposons est une implémentation rapide du type de celle qu'on réalise quand on veut très vite mettre la classe à la disposition des utilisateurs. Par la suite on peut changer les membres privés pour améliorer les performances ou supprimer certaines limitations.

Dans notre implémentation rapide, nous avons choisi de nous appuyer sur la classe Array : c'est un objet de cette classe (le membre t) qui sert en fait de conteneur. Nous avons prévu, mais non implémenté, de pouvoir agrandir un ensemble quand l'ajout d'un nouvel objet constate que la capacité maximale est

atteinte. Une amélioration de l'implémentation pourrait, par exemple, remplacer le membre `t` par une table de hachage.

On remarquera aussi que cette première implémentation empêche, par sécurité, l'affectation et la copie. L'opérateur d'affectation et le constructeur copie sont déclarés privés mais *non définis* : toute tentative d'utilisation, même dans une fonction membre, provoquera donc une erreur à l'édition des liens.

```
class SetIterator;

class Set : public Collection {
 // classe d'ensembles : implémentation provisoire, non optimale
 private:
   Array t;          // on enregistre les objets de l'ensemble dans un Array
   int capacite;     // nombre maximal d'éléments
   int cardinal;     // nombre d'éléments de l'ensemble
   void agrandisToi() { erreur(NOT_IMPLEMENTED); }
   static char string[501];
       // pour AsString(), provisoirement limité à 500 caractères
 protected:
   IdClasse classId() const { return SET; }
   const char * className() const { return "Set"; }
   Boolean isEqual(Object & T2) const ;
   Array & operator = (const Array & Source);
       // affectation privée : provisoirement interdite
   Set (const Set &);
       // constructeur copie privé, copie provisoirement interdite
  public:
   const char * AsString() const;
   CollectionIterator & NewIterator() const;
   Set() : cardinal(0), capacite(50), t(50) {}
      // ensemble vide, première capacité 50 avant agrandissement
   void Add (Object & Obj);      // ajoute Obj s'il n'était pas présent
   void Remove (Object & Obj); // enlève Obj s'il était présent
   friend class SetIterator;
       // déclaration friend obligatoire : SetIterator manipule les données privées
}; // class Set

char Set::string[500];

void Set::Add (Object & Obj)
{  // ajoute Obj s'il n'était pas présent
   if (t.Includes(Obj)) return; // déjà là
   if (cardinal == capacite) agrandisToi();
   // maintenant, on est sûr qu'il y a au moins un élément vide : on le cherche
   int Rang = 0;
   while (!t.EmptyAt(Rang)) Rang++;  //
   t.AtPut(Rang, Obj);
   cardinal++;
}
```

```
void Set::Remove (Object & Obj)
{   // enlève Obj s'il était présent
    int Rang;
    if (!t.FirstAt(Obj, Rang)) return; // déjà absent
    t.RemoveAt(Rang);
    cardinal--;
}
Boolean Set::isEqual(Object & T2) const { return FALSE; }
    // implémentation provisoire : pourrait être implémentée au plus
    // haut niveau dans la classe Collection

const char * Set::AsString() const
{   // implémentation limitée par la capacité du membre static string
    char * Ptr = string;
    sprintf(Ptr, "Set( "); Ptr = Ptr+strlen("Set( ");
    CollectionIterator & Iter = NewIterator();
    while (Iter) {
       sprintf(Ptr, "%s ", Iter.Current().AsString());
       Ptr=Ptr+strlen(Iter.Current().AsString())+1;
       Iter++;
    }
    delete & Iter;
    sprintf (Ptr, ")");
    return string;
}

class SetIterator : public CollectionIterator {
 // itérateurs sur un ensemble
 private:
    const Set & ensemble;      // tableau sur lequel on itère
    int courant;      // rang de l'élément courant ou leTableau.dimension
    void prochain();   // avance courant jusqu'au prochain objet s'il existe
 public:
    SetIterator (const Set & S)
        : courant(-1), ensemble(S)
        { prochain(); }
    Object & Current();
        // renvoie l'objet courant sans avancer l'itérateur
    Object & operator ++ (int);
        // ++ postfixé, renvoie le courant et passe au suivant s'il existe
    operator int();
        // renvoie 0 si on a traité le dernier élément
    void Restart();
        // réinitialise l'itérateur
};  // class SetIterator : public CollectionIterator
```

```
CollectionIterator & Set::NewIterator() const
{  // renvoie un itérateur sur une instance de Set
   SetIterator * P = new SetIterator(*this);
   return *P;
}

void SetIterator::prochain()
{  // fonction privée, avance courant jusqu'au prochain objet ou la fin du tableau
   courant++;
   while (courant <= ensemble.t.LastIndex()) {
      if (!ensemble.t.EmptyAt(courant)) return;
      courant++;
   }
   return;
}

Object & SetIterator::Current()
{  // renvoie l'objet courant sans avancer l'itérateur
   return ensemble.t.At(courant);
      // la classe Array validera l'accès grâce à la fonction membre At
}

Object & SetIterator::operator ++ (int)
{  // ++ postfixé, renvoie le courant et passe au suivant s'il existe
   Object & R = Current();
   prochain();
   return R;
}

SetIterator::operator int()
{  // renvoie 0 si on a traité le dernier élément
   if (courant > ensemble.t.LastIndex())
      return FALSE;
   else return TRUE;
}

void SetIterator::Restart()
{  // réinitialise l'itérateur
   courant = -1; prochain()
}
```

6 Développer une application en C++

Dans ce chapitre, nous nous proposons de construire une application complète de simulation d'un distributeur en nous appuyant sur l'ensemble des connaissances acquises dans les chapitres précédents. Afin d'exploiter les facilités de réutilisation du code offertes par C++, nous créerons des classes dérivées de la classe Object définie dans le chapitre précédent, et nous utiliserons les classes implémentées dans ce chapitre, ainsi que la classe String du chapitre 3, modifiée pour être une classe dérivée de Object. Nous nous appuierons sur les connaissances acquises pour regarder plus en détail la conception de l'application et des différentes classes qui la composent.

6.1 - Description du distributeur

Replaçons-nous dans le cadre du chapitre 1, c'est-à-dire celui d'un programmeur devant simuler le comportement d'un distributeur automatique de confiseries. La conception de ce distributeur pourrait aussi bien s'appliquer à celle d'un distributeur de boissons gazeuses, de cigarettes ou encore de journaux. Nous verrons dans la description des produits vendus comment adapter nos classes à d'autres objets que les confiseries.

Le programme devra permettre de simuler le comportement du distributeur en fonction de l'arrivée des clients et prévoir le réapprovisionnement de l'appareil.

Comme dans le chapitre 1, il faut représenter les confiseries, le distributeur, les clients et enfin le réapprovisionnement. De plus, il nous faut décrire les pièces de monnaie introduites par les acheteurs. Ces pièces interviendront dans les processus de paiement et de rendu de monnaie.

Nous allons maintenant analyser le programme à réaliser, et déterminer les classes à concevoir. Nous avons choisi de nous appuyer sur la hiérarchie de classes dérivée de `Object`, de manière à bénéficier des classes générales proposées dans cette librairie. Cette hiérarchie est décrite au chapitre 5 et représentée à la figure 6.1. Nous réutiliserons les classes déjà existantes comme les classes `Array` et `String` (nous avons ajouté la classe mise en oeuvre au chapitre 3 à l'arbores-

cence de la classe *Object*). Ce choix nous obligera à implémenter les fonctions membres virtuelles pures héritées de *Object* dans toutes les classes non abstraites que nous allons construire.

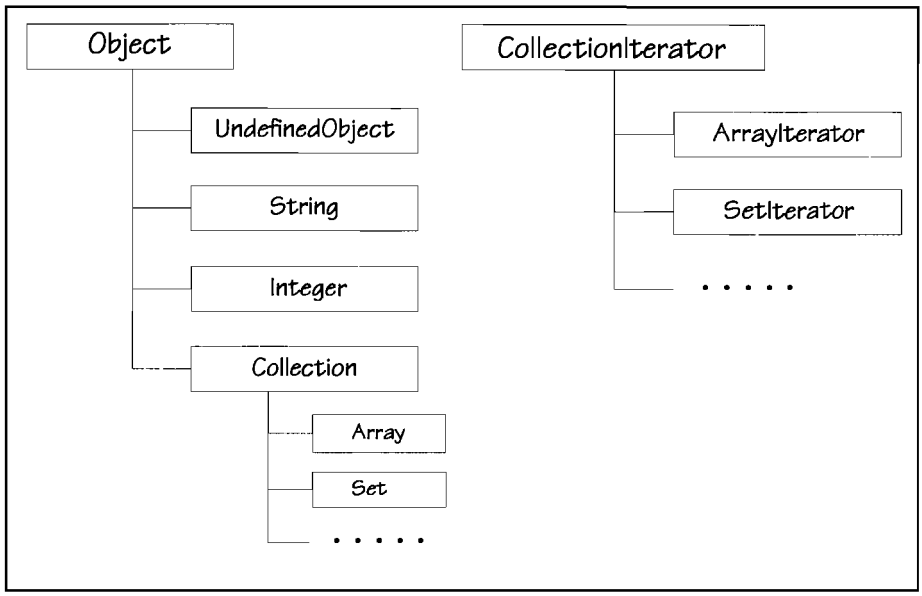


Figure 6.1 : hiérarchie de la classe Object

Pour construire les classes nécessaires à notre application, nous allons décomposer les éléments manipulés dans la simulation en catégorie d'objets simples, chaque catégorie rassemblant des objets ayant un comportement semblable. Une première approche de la conception orientée objet consiste à décrire les objets du monde réel puis à les décomposer en entités élémentaires qui seront représentées par des objets au sens informatique.

De cette manière, si nous désirons décrire une voiture pour une application de dessin ou de C.A.O., nous la décomposerons en un certain nombre de sous-objets la constituant (carrosserie, roues, moteur). Puis, nous analyserons chacun des sous-objets ainsi répertoriés, afin de les décomposer à leur tour (un moteur est un assemblage de carburateur, de pistons, d'arbre, de batterie etc.). Finalement, les constituants élémentaires seront traduits dans l'environnement informatique.

A chaque décomposition, nous réfléchirons au caractère générique de l'objet ainsi décrit et à son utilisation au sein de l'application. Nous nous demanderons si une roue de voiture n'est pas un cas particulier de roue, celle-ci pouvant être de vélo, de moto ou de tracteur, motrice ou non motrice.
Si l'objet roue est décrit avec un ensemble unique de propriétés dans l'application, il est inutile de créer une classe de base destinée à rassembler des propriétés communes aux différents types de roues utilisées. Si la roue ne peut-être qu'une roue de voiture, nous créerons une classe unique *RoueVoiture*.
Par contre, s'il existe de nombreux types de roues ayant des propriétés communes, mais des caractères spécifiques pour quelques unes de leurs propriétés, il est

intéressant de déclarer une classe de base abstraite et d'en dériver autant de classes qu'il y a de types de roues différents dans l'application.
Ainsi, nous implémenterons une classe Roue abstraite dont nous dériverons les classes RoueVoiture, RoueVélo, etc. La classe mère Roue sera une classe abstraire dans laquelle aucun objet ne pourra être instancié, mais qui regroupera tous les propriétés communes aux objets de type Roue, c'est à dire les données et fonctions membres communes à l'ensemble des roues (axe, rayon, matériau, nom etc.).

Dans notre simulation, les éléments importants du distributeur sont les pièces de monnaie, les tubes de stockage de ces pièces, les produits vendus et les « rails » où s'accrochent les produits mis en vente. Par ailleurs, le client sera considéré comme un objet élémentaire, dont les comportements se limiteront à compter son argent de poche et désirer des produits.

6.2 - Les pièces et les produits

Nous allons d'abord étudier les deux objets élémentaires utilisés dans le distributeur : les pièces de monnaie et les produits. Le distributeur contient les produits que les clients achèteront et un certain nombre de pièces de monnaie (introduites par le client pour payer et rendues par le distributeur si le client ne fait pas l'appoint).

Nous avons choisi d'utiliser les classes préexistantes sous Object (String, Array, UndefinedObject etc.). Nous allons implémenter les nouvelles classes de notre application comme des classes dérivées de Object. Ainsi, nous livrons une hiérarchie de classes homogène à nos utilisateurs (directs ou concepteurs de nouvelles classes dérivées). Nous disposerons aussi de toutes les propriétés implémentées dans Object (comparaison d'objets entre eux,...), et de la possibilité de stocker nos nouveaux objets dans les classes conteneurs dérivées de Collection. En contrepartie de ces avantages, il nous faudra implémenter toutes les fonctions virtuelles pures déclarées dans les classes de base.

Si cette solution peut être adoptée dans le cas d'une librairie de classes propriétaire (comme la nôtre) ou cohérente avec nos nouvelles classes, elle est difficile voir impossible à mettre en oeuvre lors de l'utilisation d'une librairie de classes extérieure destinée à un usage trop différent du nôtre : les librairies de classes destinées aux interfaces graphiques ont une arborescence dérivée d'une classe analogue à Object. Toutefois, leur implémentation est ciblée sur l'affichage de fenêtres, de menu ou de boîtes de dialogue. Introduire nos classes dans cette hiérarchie présente alors peu d'intérêt. Dans ce cas, il est conseillé de créer sa propre classe abstraite Object de manière à assurer la cohérence des classes propriétaires.

6.2.1 - Les pièces

L'objet réel pièce est caractérisé par sa valeur dans la devise du pays. Pour que notre simulation puisse fonctionner dans plusieurs pays, et éventuellement gérer simultanément plusieurs devises (distributeur dans la zone de transit d'un aéroport, par exemple), nous ajouterons à la valeur d'une pièce, le nom de la devise ainsi

que l'abréviation courante de cette monnaie. Une première définition de la classe sera la suivante :

```
class Piece : public Object {
 private :
   int valeur;
   String nomMonnaie;
   String nomMonnaieAbr;
 protected :
   IdClasse classId() const { return PIECE; }
   char * className() const { return "Piece"; }
   Boolean isEqual(const Object & E2) const;
 public:
   char * AsString() const { return className(); }
};
```

Ainsi un pièce de 1 franc français, représentée comme une instance de la classe Piece que nous implémentons, aura la structure interne décrite figure 6.2.

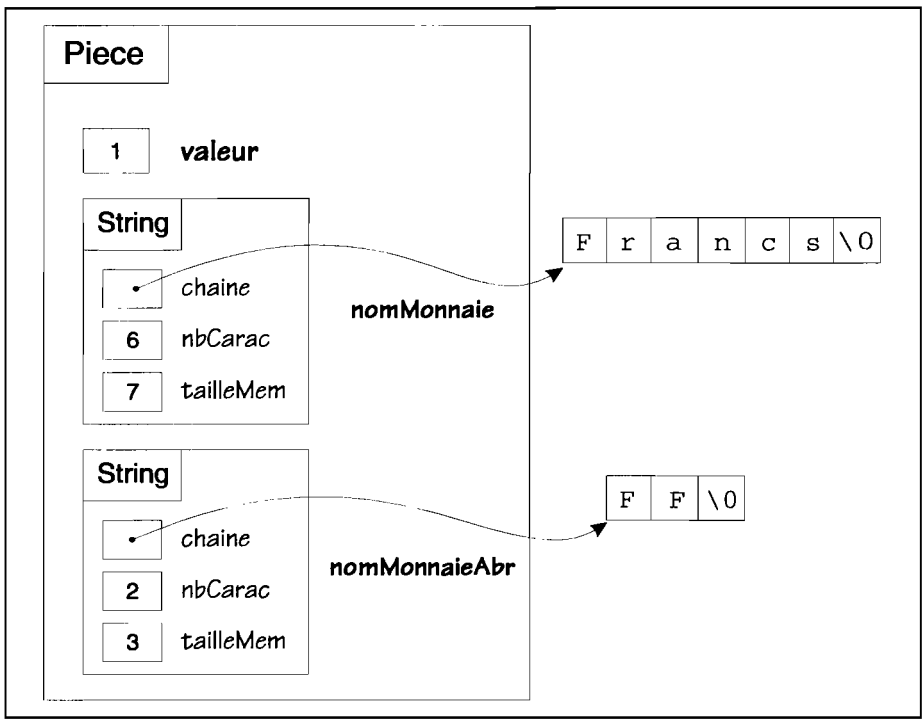


Figure 6.2 : Représentation interne de l'objet Piece

Nous implémentons un seul constructeur avec des valeurs par défaut pour chacun des arguments. Il initialise l'ensemble des données membres tout en jouant le rôle du constructeur par défaut :

```
Piece::Piece(int Valeur = 10,
             String NomMonnaie = "Francs",
             String NomMonnaieAbr = "FF")
   : valeur(Valeur),
     nomMonnaie(NomMonnaie),
     nomMonnaieAbr(NomMonnaieAbr)
   {}
```

Nous garderons le constructeur-copie par défaut. Le compilateur effectue une initialisation membre à membre pour réaliser la copie. Il va donc utiliser, pour les deux données membres de type String, le constructeur-copie de la classe String. Ce constructeur ayant été correctement redéfini au chapitre 3, l'opération de copie d'un objet Piece ne nécessite pas la redéfinition du constructeur copie de Piece.

Toutes les données membres sont déclarées dans la section privée afin que nous puissions contrôler leur utilisation par l'intermédiaire d'accesseurs en consultation et en modification. Contrairement aux billets (dans les époques troubles...), les pièces ont une valeur faciale non modifiable. Nous traduirons cette propriété en éliminant les accesseurs en modification de la classe Piece.

Nous disposons d'une fonction pour afficher les instances de Piece. La fonction AsString, dans la hiérarchie héritée de Object, donne une représentation complète de l'objet sous forme de String. Cette représentation intéresse plus le concepteur de l'application que l'utilisateur final du distributeur. Pour limiter le code au strict nécessaire, nous choisissons de simplifier le corps de cette fonction. La fonction AsString de l'ensemble de nos classes retournera le nom de la classe, ce qui nous permettra d'afficher de façon explicite le type de l'objet.

Nous surchargeons l'opérateur << de manière à renvoyer sur le flux de sortie la représentation traditionnelle d'une pièce de monnaie (le code de cet opérateur est décrit en annexe du présent chapitre). Pour avoir la description complète de l'objet Piece pendant la mise au point de l'application, le programmeur pourra combiner la fonction AsString à l'utilisation directe de l'opérateur << sur un objet Piece. La fonction membre className ne peut être utilisée directement, puisqu'elle est déclarée comme fonction protected. Ainsi, le code complet de la classe Piece, décrit en annexe du présent chapitre, nous permet d'exécuter la séquence :

```
Piece P1(1);
Piece P2(1, "DeutscheMark", "DM");
cout << "P1 :" << P1.AsString() << " de " << P1 << endl;
cout << "P2 :" << P2 << endl;
```

qui affichera :

```
P1 : Piece de 1 FF
P2 : 1 DM
```

6.2.2 - Les produits

Tout produit stocké dans un distributeur est caractérisé par au moins deux informations : son nom et son prix. Nous ajouterons un certain nombre de

propriétés à cette description sommaire. Pour des confiseries, qui sont des produits périssables, nous ajouterons la notion de date de péremption. Pour des journaux, c'est la date d'impression qu'il faudrait connaître, etc. Nous voyons qu'à partir d'un nombre restreint de propriétés communes, nous différencions plusieurs catégories de produits utilisables dans un distributeur. Il nous faut donc rassembler les propriétés communes dans une classe de base abstraite, Produit et dériver une classe, ProduitPerissable, qui nous permettra de représenter les confiseries.

Les classes Produit et ProduitPerissable, que nous définissons, sont différentes de celles définies au chapitre 4. Elles utilisent la classe String (amélioration de la sécurité) et sont héritées de Object (possibilité de comparer les produits entre eux, et de les utiliser dans un objet conteneur comme les instances de Array). .

La classe Produit

Classe abstraite, elle possède deux données membres, prix et nom. Nous choisissons de redéfinir les trois fonctions membres virtuelles classId, AsString et isEqual. Le choix d'implémentation que nous avons fait pour AsString dans le cadre de cette application s'appuie sur la fonction className, qui, elle, reste virtuelle. La fonction membre de AsString héritée de Produit sera utilisée avec profit par toutes les fonctions membres className des classes dérivées. La fonction isEqual est également définie de manière à fournir la base de la comparaison pour les données membres définies dans Produit. Les classes dérivées appelleront cette fonction isEqual avant de compléter la comparaison par des tests sur les données membres supplémentaires.

Comme nous souhaitons garder le caractère abstrait de la classe Produit, nous ne définissons pas la fonction membre className. La définition de la classe (dont le code est fourni en annexe du présent chapitre) est :

```
class Produit : public Object {
 private:
   String nom;
   int prix;
 protected:
   IdClasse classId() const { return PRODUIT; }
   virtual Boolean isEqual(const Object & E2) const
   {
      return (nom == ((Produit&)E2).nom &&
              prix == ((Produit&)E2).prix);
   }
 public:
   Produit(String Nom = "", int Prix = 0)
         : nom(Nom), prix(Prix)
   {
      nom.ToUpper();
   }
```

```
    virtual ~Produit(){};
    char * AsString() const { return className(); }
    const String & Nom() const { return nom; }
    int Prix() const { return prix; }
    Boolean NouveauPrix(int NouveauPrix);
};
```

L'unique constructeur de la classe sert également de constructeur par défaut. La classe étant abstraite, nous mettons en oeuvre un destructeur virtuel. Comme pour la classe Piece, nous fournissons des accesseurs en consultation, Nom et Prix qui retournent respectivement le nom et le prix du produit et un accesseur en modification NouveauPrix sur le membre prix.

La classe ProduitPerissable

En complément du nom et du prix hérités de Produit, nous ajoutons une information indiquant la date limite de consommation du produit. Nous implémentons celle-ci comme un objet de type Date. Cette classe n'étant pas encore définie, nous la déclarons avec la syntaxe suivante :

```
class Date;
```

dans le code précédant la définition de la classe ProduitPerissable.
Il nous faudra ensuite la définir complètement (Cf. 6.3.2) afin de pouvoir implémenter les traitements liés aux objets de la classe ProduitPerissable défini ainsi :

```
class ProduitPerissable : public Produit {
  private:
   Date dateFin;
  protected:
   IdClasse classId() const
   {
      return PRODUIT_PERISSABLE;
   }
   char * className() const
   {
      return "ProduitPerissable";
   }
   virtual Boolean isEqual(const Object & E2) const
   {
      return
         (Produit::isEqual(E2) &&
          dateFin == ((ProduitPerissable&)E2).dateFin);
   }
  public:
   ProduitPerissable(String Nom = "", int Prix = 0,
                     String DateFin = "00/00/0000")
      : Produit(Nom, Prix),dateFin(DateFin) {}
   ProduitPerissable(Produit & MonProduit, Date DateFin)
      : Produit(MonProduit),dateFin(DateFin){}
```

```
    ~ProduitPerissable(){};
    Date DateFin() const { return dateFin; }
    friend ostream & operator << (ostream & stream,
                    const ProduitPerissable & Source);
};
```

La représentation interne d'un produit périssable est donnée par la figure 6.3. Comme dans la classe Produit, nous définissons un constructeur unique pour ProduitPerissable, qui appelle le constructeur de la classe Produit, puis appelle un constructeur de la classe Date prenant comme argument une chaîne de caractères de type String.

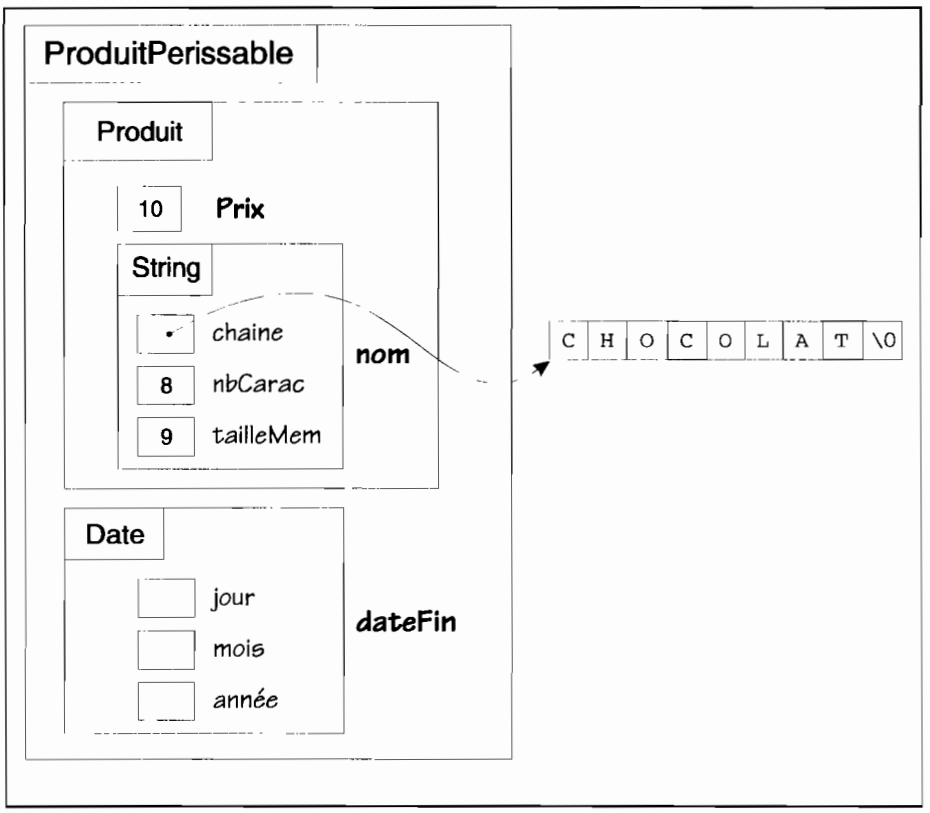


Figure 6.3 : Représentation interne de l'objet ProduitPerissable

Nous déclarons ensuite un accesseur en consultation pour le membre dateFin, donnant la limite de consommation du produit. Bien que la classe Date ne soit pas encore définie, nous élaborons petit à petit son cahier des charges en imaginant son interface. Jusqu'à présent, nous supposons qu'elle définit un constructeur acceptant un objet String comme argument, un accesseur en consultation sans compter les quatre fonctions virtuelles héritées de Object.

Nous redéfinissons enfin les fonctions virtuelles classId, className, isEqual et AsString de ProduitPerissable, ainsi que l'opérateur << de sortie sur un flux

C++. Comme nous l'avons décidé, la fonction AsString se contente de renvoyer le nom de la classe en appelant la fonction membre className, sans décrire plus précisément l'objet, alors que la version de l'opérateur << adaptée aux arguments de type ProduitPerissable renvoie les caractéristiques du produit dans un flux.

Le code de cette classe, récapitulé en annexe du présent chapitre, permet d'exécuter la séquence :

```
ProduitPerissable Prod1("chocolat", 10, "04/01/1995");
cout << Prod1.AsString() << " : " << Prod1 << endl;
```

qui affichera :

```
ProduitPerissable : CHOCOLAT (10 FF, 04/01/1995)
```

6.2.3 - La classe *Date* : une classe annexe

Même si cette classe n'est qu'une classe annexe pour notre application, un objet de type Date peut se rencontrer dans la plupart des applications, C'est pourquoi nous la plaçons dans la librairie de classes initiale (Cf. figure 6.1). Cette classe étant une classe élémentaire proche des types C++ natifs, elle hérite directement de la classe Object. Comme pour toute classe de base, son implémentation et son interface avec le programmeur doivent être étudiées avec soin, même si la première implémentation des fonctions membres peut être élémentaire (figure 6.4).

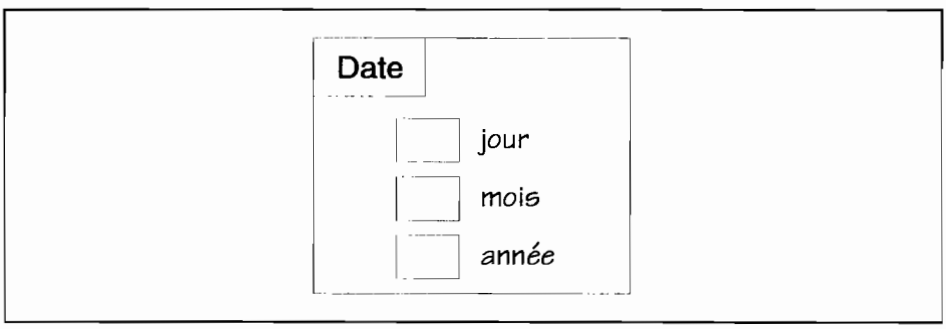


Figure 6.4 : Représentation interne de l'objet Date

Ici, nous nous contentons d'une classe élémentaire, nous permettant d'effectuer les traitements nécessaires sur les dates pour les différentes classes de notre application.

Pour répondre aux besoins exprimés au paragraphe 6.2.2, nous devons pouvoir construire une date à partir d'une chaîne de caractères que nous écrirons sous la forme "*20/01/1901*" (format courant d'une date européenne) en utilisant le constructeur :

```
Date(String D = "00/00/0000");
```

Ce constructeur sert également de constructeur par défaut. La valeur choisie par défaut correspond à une date indéfinie (tous les champs à zéro). Le corps du

constructeur effectue les tests de vérification de format de l'argument avant de définir les données membres.

Un objet Date doit également pouvoir être construit à partir des éléments le constituant, c'est à dire le jour, le mois et l'année, avec le constructeur suivant :

```
Date::Date(int Jour, TypMois Mois, int Annee);
```

le type TypMois étant défini par l'énumération suivante :

```
enum TypMois { Incorrect, Janvier, Fevrier, Mars, Avril,
               Mai, Juin, Juillet, Aout, Septembre,
               Octobre, Novembre, Decembre };
```

Nous surchargeons également pour notre classe l'opérateur de sortie en le déclarant ami de la classe Date.

6.2.4 - Une amélioration de l'implémentation de la classe *Object*

Pour chacune des classes que nous venons de créer, il a fallu redéfinir les fonctions membres virtuelles et virtuelles pures de la classe Object. Mais il nous a également fallu créer les constantes de type IdClasse. Cette implémentation oblige le concepteur de classes dérivées à compléter l'énumération des types IdClasse dans un fichier en-tête de la hiérarchie. Il est préférable, par sécurité, de ne pas toucher à ce fichier en-tête. Pour ajouter ses propres classes, ou pour permettre à une équipe concepteurs de classes dérivées de travailler simultanément sur la hiérarchie, nous modifions l'implémentation de la classe Object en ajoutant dans l'énumération IdClasse une valeur nommée à partir de laquelle les concepteurs de l'équipe se distribueront des tranches de numéros pour les classes qu'ils construisent. L'énumération IdClasse devient :

```
enum IdClasse { OBJECT, ARRAY, STRING, DATE,
                UNDEFINED, IDC_USER };
```

Si un programmeur dispose de la tranche k à k + 10 pour définir ses classes, il écrira pour sa première classe implémentée :

```
#define MACLASSE IDC_USER + k
```

Ainsi, il ne modifie pas le fichier en-tête de la bibliothèque de classes livrée avec Object. Le concepteur de la hiérarchie de classes issues de Object garde la possibilité de faire évoluer son implémentation en ajoutant d'autres classes sans perturber les applications développées avec la version précédente de sa hiérarchie. Il le fait en insérant ou en supprimant l'identifiant de ses propres classes avant la valeur nommée IDC_USER. Par exemple, s'il veut ajouter une classe NewClass à sa hiérarchie d'origine, il ajoutera l'identificateur NEWCLASS dans l'énumération :

```
enum IdClasse { OBJECT, ARRAY, STRING, DATE, NEWCLASS,
                UNDEFINED, IDC_USER };
```

Avec la directive de pré compilation #define MACLASSE IDC_USER + k, le type de MACLASSE est automatiquement int. La fonction virtuelle classId déclarée dans Object doit impérativement renvoyer un type int si l'on veut qu'elle puisse être correctement définie dans les classes futures des utilisateurs. Nous la définissons donc de la façon suivante :

```
class Object {
    // etc.
    // ancienne version
    // virtual IdClasse classId () { return OBJECT; }
    // nouvelle version
    virtual int classId () { return OBJECT; }
}
```

La modification que nous apportons à la classe *Object* et à sa descendance oblige à la fois l'équipe implémentant la hiérarchie de classes initiale et celle concevant des applications qui utiliseront cette hiérarchie à avoir une discipline stricte. En effet, pour que le système reste fonctionnel, les identificateurs doivent tous être différents. Il faut donc que chaque équipe, voire chaque développeur, dispose d'une tranche de numéros déterminés pour ses identificateurs. Les nouveaux identificateurs peuvent être regroupés dans un fichier spécifique pour être connus de l'ensemble des concepteurs du projet. Cela permet en fin de projet de lisser les valeurs de tous les identificateurs pour qu'elles forment une suite continue de valeurs entières.

6.3 - Le distributeur

A ce stade, nous sommes capables de représenter les deux entités élémentaires qui interviennent dans le fonctionnement du distributeur, *Piece* et *Produit*. Nous allons maintenant étudier plus précisément le fonctionnement du distributeur. Comme nous l'avons indiqué en introduction, notre distributeur gère des produits et des pièces.

Les produits sont placés sur des rails ou dans des cases à l'intérieur du distributeur. Ils sont identifiés par des numéros, qui permettent au client de choisir son produit. Par la suite, nous utiliserons le terme de rail pour décrire un élément contenant plusieurs occurrences du même produit.

Si un rail ne contient qu'un type de produit, on peut trouver plusieurs rails ayant le même type de produit.

Les pièces contenues dans le distributeur dépendent de la devise et du choix du constructeur du distributeur. Ainsi, en France, la majorité des distributeurs excluent les pièces de 5, 10, 20 et 50 centimes pour limiter la taille des espaces de stockage.

A l'intérieur du distributeur, afin de faciliter la collecte des pièces, leur traitement ultérieur et rendre la monnaie, les pièces sont mises dans des tubes qui contiennent un type unique de pièces. Pour chaque tube, il nous faut connaître le type et le nombre de pièces présentes ainsi que la capacité maximale pouvant y être stocké.

6.3.1 - Une classe *Sac*

Nous allons ajouter une classe conteneur qui nous permette d'identifier l'objet contenu dans le conteneur, de connaître le nombre d'exemplaires présents et le

maximum que l'on peut y mettre. Nous appelons cette classe Sac[1]. Une fois, cette classe créée, nous pourrons commencer l'implémentation du distributeur.

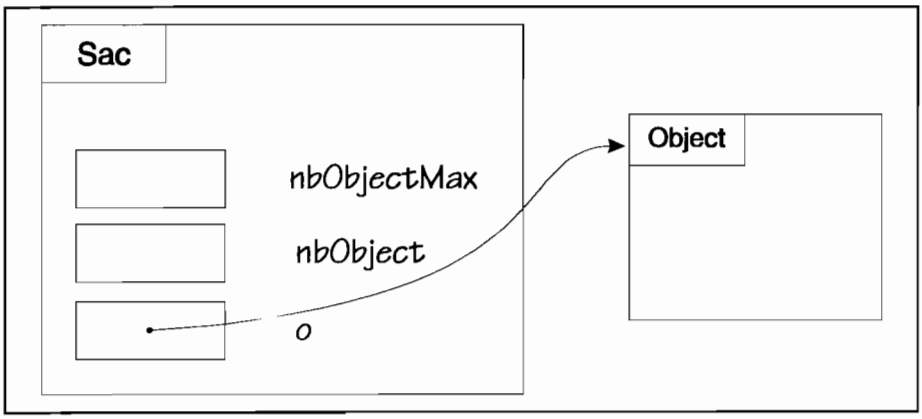


Figure 6.5 : Représentation interne de l'objet Sac

Cette classe définit trois données membres (figure 6.5) :

- *o*, pointeur sur l'objet contenu (pointeur sur un objet constant pour que cet objet ne soit pas modifié) ;
- nbObjectMax, nombre d'objets pouvant être contenus ,
- nbObject, nombre d'objets présents .

L'objet pointé par le membre *o* doit appartenir à une classe dérivée de *Object*. Dans le cas d'un tube de pièces, la donnée membre *o* pointera sur un élément de type *Piece*.

La classe *Sac* n'est pas du type *Collection*, puisqu'elle ne contient qu'un objet. Elle va donc hériter directement de *Object*. Après avoir redéfini les quatre fonctions virtuelles héritées de *Object*, nous implémentons un constructeur qui permet d'initialiser un sac :

```
Sac::Sac(int NbObjMax, int NbObj, Object & AnObject)
```

avec sa capacité (NbObjMax), le nombre d'objets présents au départ (NbObj) et une référence à l'objet contenu (AnObject).

Nous complétons cette classe avec quelques accesseurs et une série de fonctions membres et d'opérateurs chargés d'ajouter ou de retrancher des objets[2] :

```
class Sac : public Object {
 private:
   int nbObjectMax;
   int nbObject;
   const Object * o;
```

1 Cette classe *Sac* n'est pas une classe *Bag* au sens Smalltalk, c'est à dire un fourre-tout où des objets différents peuvent être stockés de façon non ordonnée et en plusieurs exemplaires.

2 On notera que les opérateurs ++ et -- surchargés sont les opérateurs postfixés (C++ distingue les deux formes en imposant un argument int, non utilisé, dans la forme postfixée).

```
  protected:
    IdClasse classId() const { return SAC; }
    char * className() const { return "Sac"; }
    int isEqual(const Object & T2) const { return FALSE;}
  public:
    char * AsString() const { return className();}
    // Accesseurs en consultation
    int NbObjectMax() const { return nbObjectMax; }
    int NbObject() const { return nbObject; }
    const Object & ObjectSac() const { return *o; }
    // Constructeur
    Sac(int NbObjMax, int NbObj, Object & AnObject);
    // Opérateurs de manipulation du nombre d'objets
    Sac operator ++ (int); // ajoute un objet au sac
    Sac operator -- (int); // enlève un objet du sac
    Sac operator + (int NbElem); // ajoute NbElem objets au sac
    Sac operator += (int NbElem); // ajoute NbElem objets au sac
    Sac operator - (int NbElem); // enlève NbElem du sac
    Sac operator -= (int NbElem); // enlève NbElem du sac
    void NouveauNbObject(int NbObject);
       // remplace le nombre actuel d'objets par NbObject
    void Remplir(); // remplit le sac au maximum de sa capacité
    // Opérateur de flux de sortie
    friend ostream & operator << (ostream & stream,
                                  const Sac & Source);
};
```

Nous pouvons maintenant utiliser la classe Sac pour représenter les rails contenant un produit ou les tubes de pièces.

6.3.2 - Structure interne du distributeur

Un distributeur contient plusieurs tubes de pièces de monnaie et plusieurs rails de produits. Ce n'est qu'à l'instanciation d'un objet de la classe Distributeur que l'on saura exactement le nombre de rails et de tubes du distributeur. Nous allons donc utiliser deux pointeurs sur des objets de type Array. Le premier tableau Array gérera la monnaie : chacun de ses éléments pointera sur un objet Sac (de pièces). Le deuxième tableau Array gérera les produits : chacun de ses éléments pointera sur un objet Sac (de produits). Nous avons schématisé la représentation interne du distributeur en figure 6.6.

6.3.3 - Traitements disponibles

Constructeur et destructeur

L'application que nous mettons en oeuvre simule un distributeur de confiserie, de manière à valider certains choix techniques du fabricant. Nous proposons, dans la classe Distributeur, un seul constructeur qui va préparer un distributeur selon les

spécifications précisées dans ses arguments, mais remplira aléatoirement ses rails par des produits tirés d'une liste fournie en argument. Il va de soi qu'il faudrait développer d'autres constructeurs pour permettre une utilisation plus générale de la classe Distributeur.

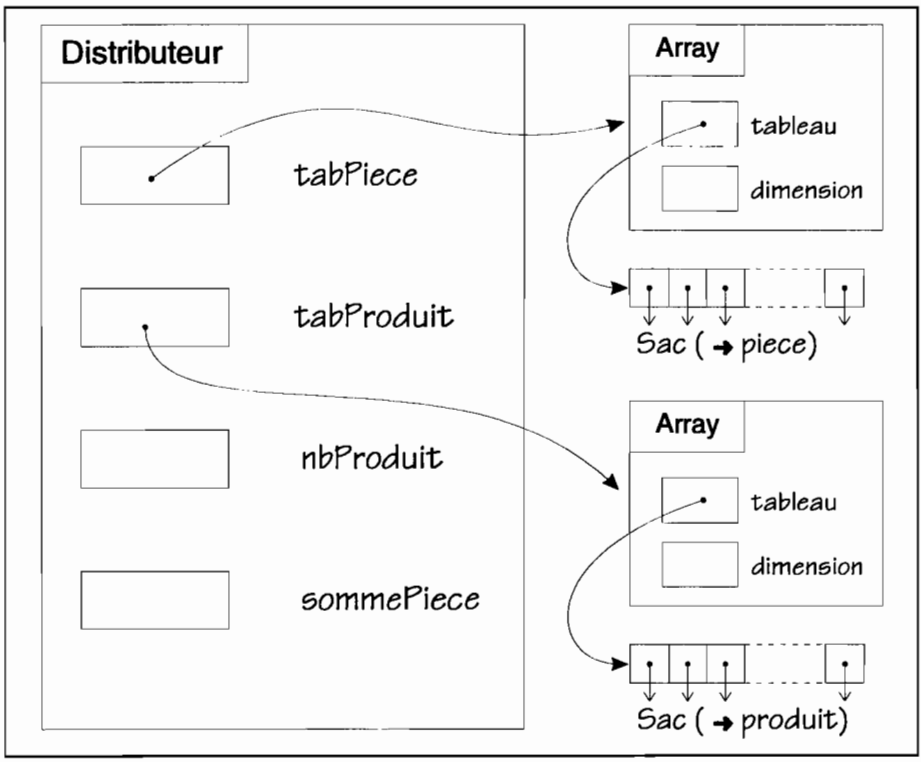


Figure 6.6 : Représentation interne de l'objet Distributeur

Le constructeur proposé est défini en annexe du présent chapitre. Sa déclaration est :

```
Distributeur(int NbSacProduit,
             int CapaciteSacProd,
             const Array & ListeProduitDisponible,
             int CapaciteSacPiece,
             const Array & ListePieceDisponible);
```

Il attend en argument des informations concernant la gestion des produits :

- NbSacProduit, nombre de rails soutenant les produits vendus ;
- CapaciteSacProd, nombre de produits maximum que l'on peut ranger sur un rail ;
- ListeProduitDisponible, tableau des produits que l'on pourra mettre dans ce distributeur ;

Concernant la gestion des pièces, les arguments sont :

- CapaciteSacPiece, nombre de pièces accumulables dans un tube et
- ListePieceDisponible, tableau rassemblant les pièces acceptées par le distributeur.

Ce constructeur ne permet pas de faire la distinction entre chaque rail de produit ou entre chaque tube de pièce en ce qui concerne la capacité de stockage. Lors de la simulation, il peut s'avérer utile d'augmenter la taille de tel ou tel tube de pièces fréquemment utilisées, ou de tenir compte de l'épaisseur des produits qui permet d'en loger plus ou moins par rail, dans la réalité. Il suffira au programmeur chargé de mettre au point l'application d'ajouter les constructeurs simulant les chaînes de fabrication des distributeurs automatiques.

Le constructeur que nous proposons effectue les initialisations requises et tire aléatoirement dans ListeProduitDisponible les produits présentés au client.

Il n'est pas utile de définir un constructeur par copie : le constructeur fournit par défaut utilisera le constructeur copie redéfini dans Array pour gérer correctement l'allocation dynamique des tableaux pointés par les données membres tabPiece et tabProduit.

Le destructeur se contente de libérer l'espace alloué pour les deux pointeurs sur le type Array, tabProduit et tabPiece.

Des fonctions d'accès aux différents membres

Afin de simplifier les notations, nous avons défini un certain nombre de fonctions membres qui facilitent l'accès aux membres de la classe, et plus particulièrement aux informations des objets Piece et Produit contenus dans le distributeur.

```
class Distributeur {
    // etc.
protected:
    Sac & sacProduitAt(int Rang);
        // Renvoie le sac de produits à la position Rang dans le distributeur
    const Sac & sacProduitAt(int Rang) const;
        // Renvoie le sac de produits... (pour un distributeur constant)
    Sac & sacPieceAt(int Rang);
        // Renvoie le sac de pièces à la position Rang dans le distributeur
    const Sac & sacPieceAt(int Rang) const;
        // Renvoie le sac de pièces...(pour un distributeur constant)
public:
    // etc.
    const Produit & ProduitAt(int Rang) const;
        // Renvoie le produit du sac de produits à la position Rang
    const Piece & PieceAt(int Rang) const;
        // Renvoie la pièce du sac de pièces à la position Rang
    Boolean EmptyProduitAt(int Rang) const;
        // Renvoie TRUE si le sac de produits à la position Rang est vide
    Boolean EmptyPieceAt(int Rang) const;
        // Renvoie TRUE si le sac de pièces à la position Rang est vide
```

```
    int PrixAt(int Rang) const;
        // Renvoie le prix du produit contenu dans le sac à la position Rang
    int ValeurAt(int Rang) const;
        // Renvoie la valeur de la pièce contenu dans le sac à la position Rang
    int NbProduitAt(int Rang) const;
        // Renvoie le nombre de produits contenus dans le sac à la position Rang
    int NbProduitMaxAt(int Rang) const;
        // Renvoie la capacité du sac de produits à la position Rang
    int NbPieceAt(int Rang) const;
        // Renvoie le nombre de pièces contenues dans le sac à la position Rang
    int NbPieceMaxAt(int Rang) const;
        // Renvoie la capacité du sac de pièces à la position Rang
    Boolean EstPayable(const Array & ListePiece) const;
        // Renvoie TRUE si les pièces contenues dans ListePiece peuvent être
        // stockées dans les tubes du distributeur
};
```

Les fonctions d'affichage

Nous redéfinissons l'opérateur de sortie afin de pouvoir afficher le distributeur complet, ainsi que deux fonctions annexes, AfficheSacProduit et AfficheSacPiece, qui affichent les sacs de produits et de pièces.

Les services fournis par le distributeur

Le seul service fourni par ce distributeur simple est effectué par la fonction membre :

```
void PaiementClient(const Array & ArraySacPiece,
                    int RangProduit)
```

qui assure le stockage des pièces et la distribution du produit. Elle utilise la fonction membre EstPayable pour vérifier que les pièces du client (contenues dans ArraySacPiece) pourront être stockées dans les tubes du distributeur.

Dans un modèle plus complexe, nous pourrions également gérer le rendu de la monnaie, le choix de la devise...

6.4 - Les clients

Enfin, il nous reste à implémenter une classe Client. Un client devra pouvoir choisir un produit dans le distributeur, savoir s'il peut payer le produit (connaissant le contenu de son porte-monnaie) et le payer.

6.4.1 - Représentation du client

Un client va donc être représenté par son porte-monnaie. Dans la réalité, un porte-monnaie est un fourre-tout, mais afin de faciliter les traitements, nous le représenterons comme une série de tubes de pièces de monnaie. Comme pour le distributeur, ce n'est qu'à l'instanciation d'un objet de la classe Client que l'on

connaîtra le nombre de types de pièces. Nous allons donc utiliser un pointeur sur un objet de type Array, qui gérera la monnaie : chacun de ses éléments pointera sur un objet Sac (de pièces). La variable d'instance sommePiece permet de connaître, à tout instant, la somme totale contenue dans le porte-monnaie du client. Nous avons schématisé la représentation interne du distributeur en figure 6.7.

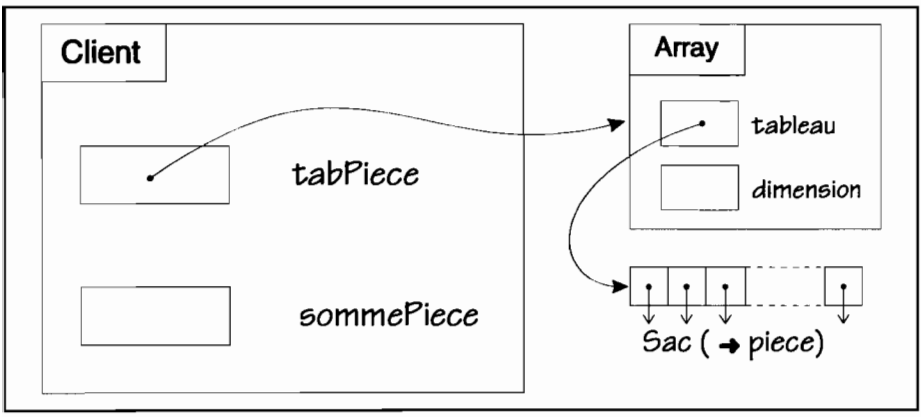


Figure 6.7 : Représentation interne de l'objet Client

6.4.2 - Les actions du client

Constructeur et destructeur

La classe Client est une classe propre à notre application. Elle n'a pas (pour l'instant) vocation à être diffusée pour une utilisation générale. Nous nous contentons donc d'un unique constructeur, simulant le contenu de différents porte-monnaie à chaque instanciation, par simple tirage aléatoire d'un nombre de pièces NbPiece parmi une liste ListePiece de monnaies autorisées :

```
Client(int NbPiece, Array & ListePiece);
```

Le constructeur par copie n'est pas redéfini : le constructeur par défaut utilise le constructeur copie de la classe Array.

Le destructeur se contente de libérer l'espace alloué pour le pointeur sur le membre tabPiece, de type Array.

Fonction d'affichage

Nous redéfinissons l'opérateur de sortie dans un flux, afin de pouvoir afficher le porte-monnaie du client.

Les actions du client

Le client choisit de façon aléatoire le produit qu'il va consommer par un appel de la fonction ChoixProduit, sur le même principe que dans le chapitre 1. Ensuite,

la fonction PeutPayer permet de savoir s'il possède assez d'argent et s'il peut faire l'appoint (un distributeur plus évolué pourrait rendre la monnaie).

Enfin, il est capable de payer le distributeur en initialisant une structure de communication ListePiece, qui est un tableau de sac de pièces représentant les pièces que le client donne pour payer.

6.5 - Exemple d'exécution : la fonction *main*

Le fournisseur de confiseries désire optimiser le passage de ses livreurs sur chaque appareil. Il espère ainsi réapprovisionner en une seule fois chaque distributeur. Partant de l'hypothèse de départ (le client choisit aléatoirement le produit qu'il veut consommer), le distributeur de confiseries veut vérifier :

- que l'absence prématurée d'un produit ne conduira pas une trop grande quantité de clients à abandonner leur achat (on définit un indice de satisfaction en mesurant la fuite des clients après trois choix successifs refusés par le distributeur) ;
- que ses tubes de pièce supporteront le paiement de tous les produits du distributeur.

Nous choisissons ainsi de simuler l'arrivée de clients jusqu'à ce qu'il n'y ait plus que ValeurArret produits dans le distributeur. Nous étudierons uniquement l'indice de satisfaction dans la fonction main proposée en exemple. L'outil de simulation est prêt : il ne reste qu'à l'utiliser.

```
#include "c_Client.h"
#include "c_Distri.h"

/* Les valeurs constantes de la simulation */
// Nombre de pièces différentes possibles
const int NbTypePiece = 5;
// Nombre de produits différents disponibles
const int NbTypeProduit = 7;
// Nombre de produits différents dans le distributeur
const int NbProdDistri = 3;
// Capacité d'un sac de produits du distributeur
const int CapaciteSacProduit = 5;
// Capacité d'un sac de pièces du distributeur
const int CapaciteSacPiece = 50;
// Capacité du porte-monnaie (sac de pièces) du client
const int CapacitePorteMonnaie = 20;
// Nombre de produits restant à vendre dans le distributeur lors du remplissage
const int ValeurArret = 5;

void main()
{
    // Instanciation d'un tableau de dimension NbTypePiece de pointeurs sur
    // Piece qui donne l'ensemble des pièces utilisables
    Array ListePiece(NbTypePiece);
```

```
// Initialisation du tableau ListePiece
ListePiece.AtPut(0,*(new Piece(1)));
ListePiece.AtPut(1,*(new Piece(2)));
ListePiece.AtPut(2,*(new Piece(5)));
ListePiece.AtPut(3,*(new Piece(10)));
ListePiece.AtPut(4,*(new Piece(20)));
/* Instanciation d'un tableau de dimension NbTypeProduit de pointeurs sur
   Produit qui donne l'ensemble des produits pouvant être introduit
   dans le distributeur */
Array ListeProduit(NbTypeProduit);
// Initialisation du tableau ListeProduit
ListeProduit.AtPut(0,*(new ProduitPerissable
   ("Barre chocolaté",5,"13/12/1993")));
ListeProduit.AtPut(1,*(new ProduitPerissable
   ("Bonbons aux fruits",10)));
ListeProduit.AtPut(2,*(new ProduitPerissable
   ("Chips",5)));
ListeProduit.AtPut(3,*(new ProduitPerissable
   ("Cacahuetes",7)));
ListeProduit.AtPut(4,*(new ProduitPerissable
   ("Bonbons à la menthe",10)));
ListeProduit.AtPut(5,*(new ProduitPerissable
   ("Réglisse",6)));
ListeProduit.AtPut(6,*(new ProduitPerissable
   ("Chewing Gum",2)));
// Construction du distributeur
Distributeur Distri(NbProdDistri,
                    CapaciteSacProduit,ListeProduit,
                    CapaciteSacPiece,ListePiece);
// Affichage du distributeur
cout << Distri;
// tableau de stockage de la satisfaction client
// Position 0 = nombre de clients très satisfaits => Premier choix
// Position 1 = nombre de clients satisfaits => Deuxième choix
// Position 2 = nombre de clients mécontents => Troisième choix
// Position 3 = nombre de clients insatisfaits => Plus de trois choix
int NbClient[4]={0,0,0,0};
int Rang;       // Rang du produit choisi par le client
int NbChoix;    // Nombre de choix avant satisfaction
int NbPiece;    // Nombre de pièces différentes
while (Distri.NbProduit() > ValeurArret) {
   // Construction d'un client
   Client Client1(CapacitePorteMonnaie, ListePiece);
   // Allocation d'une structure d'échange entre le client et le distributeur de
   // type Array de dimension NbTypePiece de pointeur sur Sac de Pieces
   Array ListePieceEchange(NbTypePiece);
```

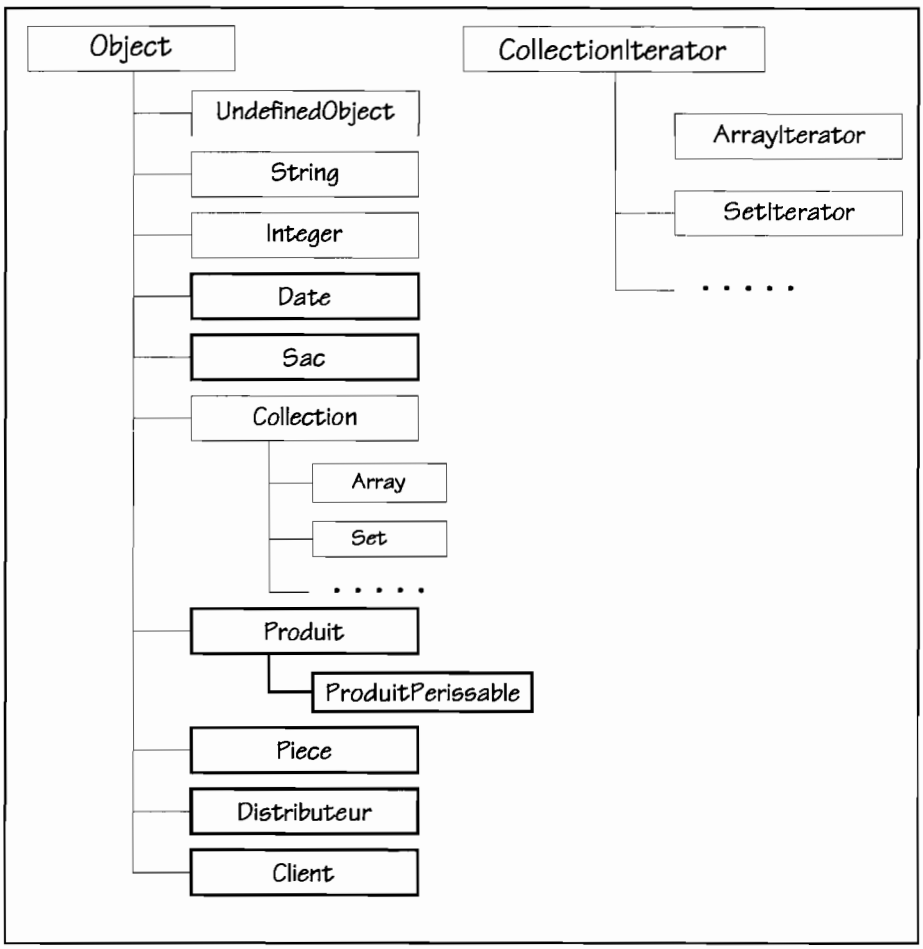
```
        // Initialisation de ListePieceEchange en utilisant ListePiecePossible
        for (i = 0; i < NbTypePiece ; i++)
           ListePieceEchange.AtPut(i,*(new Sac(50,0,
              *(new Piece ((Piece&)ListePiece.At(i))))));
        // Initialisation à zéro du nombre de choix
        NbChoix = 0;
        // Initialisation du nombre de pièces
        NbPiece = NbTypePiece;
        // Choix correct si le produit est encore dans le distributeur
        // et si le client peut le payer
        do {
           Rang = Client1.ChoixProduit(NbProdDistri);
           NbChoix++;
           // Sortie de la boucle de choix : client instatisfait
           if (NbChoix > 3) break;
        }
        while (Distri.EmptySacProduitAt(Rang) ||
               ! Client1.PeutPayer(Distri.PrixAt(Rang),
                                   ListePieceEchange,
                                   &NbPiece));
        // Retour au début de la boucle principale
        if (NbChoix > 3) continue;
        // Affichage du choix
        cout << (ProduitPerissable&) Distri.ProduitAt(Rang)
             << endl;
        // Mise à jour de l'indice de satisfaction
        if (NbChoix > 3) NbClient[3]++;
        else NbClient[NbChoix-1]++;
        // Mise à jour de la structure d'échange par le client
        Client1.PaieDistributeur(ListePieceEchange);
        // Insertion des pièces dans le distributeur et libération
        // du produit par le distributeur
        Distri.PaiementClient(ListePieceEchange,Rang);
    }
    int NbClientTotal = 0;
    for (i = 0; i < 4; i++)
       NbClientTotal += NbClient[i];
    cout << "Distributeur contenant " << ValeurArret << " produits"
         << endl << "Satisfaction des clients " << endl;
    for (i = 0; i < 4; i++)
       cout << "   Niveau " << i << " de satisfaction "
            << NbClient[i]*100/NbClientTotal << "%"
            << endl;
}
```

Une des exécutions de ce programme de simulation nous donne :

```
Distributeur
    Description des 15 produits
    0 : Sac de contenance 5 et contenant 5 objets de type
        ProduitPerissable
        BONBONS    LA MENTHE (10,0/0/0)
    1 : Sac de contenance 5 et contenant 5 objets de type
        ProduitPerissable
        BARRE CHOCOLAT (5,13/12/1993)
    2 : Sac de contenance 5 et contenant 5 objets de type
        ProduitPerissable
        CACAHUETES (7,0/0/0)

    Aucune pièce

Les achats
    CACAHUETES (7,0/0/0)
    BONBONS    LA MENTHE (10,0/0/0)
    CACAHUETES (7,0/0/0)
    CACAHUETES (7,0/0/0)
    BARRE CHOCOLAT (5,13/12/1993)
    BARRE CHOCOLAT (5,13/12/1993)
    CACAHUETES (7,0/0/0)
    CACAHUETES (7,0/0/0)
    BARRE CHOCOLAT (5,13/12/1993)
    BONBONS    LA MENTHE (10,0/0/0)

Distributeur contenant 5 produits
Satisfaction des clients
Niveau 0 de satisfaction : 10%
Niveau 1 de satisfaction : 90%
Niveau 2 de satisfaction : 0%
Niveau 3 de satisfaction : 0%
```

6.6 - Annexe : les classes de l'application

6.6.1 - La hiérarchie de classes



6.6.2 - La classe *Piece*

Le fichier d'en-tête *c_Piece.h*

```
#ifndef _PIECE
#define _PIECE

#include "c_String.h"
#include "c_Date.h"
```

```
class Piece : public Object {
 private:
   int valeur;              // valeur faciale de la pièce
   String nomMonnaie;       // nom courant de la monnaie, ex. : Francs
   String nomMonnaieAbr;    // abréviation de la monnaie, ex. : FF
 protected:
   IdClasse classId() const { return PIECE; }
   char * className() const { return "Piece"; }
   Boolean isEqual(const Object & E2) const;
 public:
   char * AsString() const { return className(); }
   Piece(int Valeur=10,
         const String NomMonnaie = "Francs",
         const String NomMonnaieAbr = "FF");
         // Constructeur
   // Accesseurs
   const String & NomMonnaie() const
      { return nomMonnaie; }
   const String & NomMonnaieAbr() const
      { return nomMonnaieAbr; }
   int Valeur() const { return valeur; }
   // Surcharge de l'opérateur de sortie
   friend ostream & operator << (ostream & stream,
                                 const Piece & Source);
};
#endif
```

***Le fichier** c_Piece.cpp*

```
#include <string.h>
#include <stdio.h>
#include <strstream.h>
#include <iostream.h>
#include "c_Object.h"
#include "c_Piece.h"

// Comparaison
Boolean Piece::isEqual(const Object & E2) const
{  // deux pièces sont identiques si elles ont les mêmes données membres
   return (valeur==((Piece&)E2).valeur &&
           nomMonnaie==((Piece&)E2).nomMonnaie &&
           nomMonnaieAbr==((Piece&)E2).nomMonnaieAbr);
}
```

```
// Surcharge de l'opérateur de sortie
ostream & operator << (ostream & stream,
                       const Piece & Source)
{
   return stream << Source.valeur
                 << " " << Source.nomMonnaieAbr;
}
// Constructeur
Piece::Piece(int Valeur,
             const String NomMonnaie,
             const String NomMonnaieAbr)
   : valeur(Valeur), nomMonnaie(NomMonnaie),
     nomMonnaieAbr(NomMonnaieAbr)
   {}
```

6.6.3 - La classe *Produit*

Le fichier d'en-tête c_Prod.h

```
#ifndef _PROD
#define _PROD
#include "c_String.h"
#include "c_Date.h"
class Produit : public Object {
 private:
   String nom; // nom du produit
   int prix;   // prix du produit
 protected:
   IdClasse classId() const { return PRODUIT; }
   char * className() const { return "Produit"; }
   Boolean isEqual(const Object & E2) const;
 public:
   Produit(String Nom="", int Prix=0);
   virtual ~Produit(){}; // Destructeur virtuel
   const String & Nom() const { return nom; }
   int Prix() const { return prix; }
   Boolean NouveauPrix(int NouveauPrix);
};

class ProduitPerissable : public Produit {
 private:
   Date dateFin; // date de péremption du produit
 protected:
   IdClasse classId() const
      { return PRODUIT_PERISSABLE; }
   char * className() const
      { return "ProduitPerissable"; }
```

```
    Boolean isEqual(const Object & E2) const;
  public:
    char * AsString() const { return className(); }
    ProduitPerissable(String Nom="",
                      int Prix=0,
                      String DateFin="")
       : Produit(Nom, Prix),dateFin(DateFin) {}
    ProduitPerissable(Produit & MonProduit, Date DateFin)
       : Produit(MonProduit),dateFin(DateFin){}
    Date DateFin() const { return dateFin; }
    friend ostream & operator <<
       (ostream & stream,
        const ProduitPerissable & Source);
};
#endif
```

Le fichier c_Prod.cpp

```
#include <iostream.h>
#include "c_Object.h"
#include "c_Prod.h"
#include "c_Date.h"


Boolean Produit::isEqual(const Object & E2) const
{ // Comparaison
   return (nom==((Produit&)E2).nom &&
           prix==((Produit&)E2).prix);
}

Produit::Produit(String Nom, int Prix)
   : nom(Nom), prix(Prix)
{ // Constructeur (le nom est immédiatement mis en majuscules)
   nom.ToUpper();
}

Boolean Produit::NouveauPrix(int NouveauPrix)
{  // Accesseur en modification de prix - Hypothèse sur un prix compris entre 0
   // et 30 pour un produit distribué en machine...
   if (NouveauPrix <= 0 || NouveauPrix >= 30)
      return FALSE;
   prix = NouveauPrix;
   return TRUE;
}
```

```
Boolean ProduitPerissable::isEqual(const Object & E2)
   const
{ // Comparaison
   return (Produit::isEqual(E2) &&
           dateFin == ((ProduitPerissable&)E2).dateFin );
}

ostream & operator << (ostream & stream,
                       const ProduitPerissable & Source)
{ // Surcharge de l'opérateur de sortie sur un flux
   return stream << Source.Nom().Chaine()
                 << " (" << Source.Prix()
                 << "," << Source.DateFin() << ")";
}
```

6.6.4 - La classe *Date*

***Le fichier d'en-tête** c_Date.h*

```
#ifndef _DATE
#define _DATE
#include <iostream.h>
#include "c_String.h"
enum TypMois { Incorrect, Janvier, Fevrier, Mars, Avril,
               Mai, Juin, Juillet, Aout, Septembre,
               Octobre, Novembre, Decembre };
   // l'identifiant Incorrect permet d'initialiser une date indéterminée
class Date : public Object {
 private:
   int jour;       // de 0 à 31
   TypMois  mois;  // utilise le type énuméré TypMois
   int annee;      // sur 4 chiffres (1994)
 protected:
   IdClasse classId() const { return DATE; }
   char * className() const { return "Date"; }
   Boolean isEqual(const Object & E2) const;
 public:
   char * AsString() const { return "Date"; }
   Date(String D="00/00/0000");
   Date(int Jour, TypMois Mois, int Annee);
   friend ostream & operator << (ostream & stream,
                                 const Date & Source);
};
#endif
```

***Le fichier** c_Date.cpp*

```
#include <stdlib.h>
#include <string.h>
#include "c_Object.h"
#include "c_Date.h"

Boolean Date::isEqual(const Object & E2) const
{ // Comparaison
   return (annee==((Date&)E2).annee &&
           jour==((Date&)E2).jour &&
           mois==((Date&)E2).mois);
}

Date::Date(String D)
{ // Constructeur à partir d'une chaîne String au format JJ/MM/AAAA
   if (D.SubString(2,1) != "/" || D.SubString(5,1) != "/"
       || strlen(D.SubString(6,4).Chaine()) < 4) {
      // Test du format de la chaîne
      jour = 0; mois = Incorrect; annee = 0;
   }
   else {
      jour = atoi((D.SubString(0,2)).Chaine());
      if (jour < 0 || jour > 31 ) { // valeur de jour incorrecte
          jour = 0; mois = Incorrect; annee = 0; }
      else {
         int tempMois =
            atoi((D.SubString(3,2)).Chaine());
         if (tempMois < Janvier
             || tempMois > Decembre) {
            // valeur de mois incorrecte
            tempMois = Incorrect; annee = 0; }
         else {
            mois = (TypMois) tempMois;
            annee = atoi((D.SubString(6,4)).Chaine());
         }
      }
   }
}

Date::Date(int Jour, TypMois Mois, int Annee)
{ // Constructeur à partir des éléments d'une date
   if (Jour < 0 || Jour > 31) {
        // Validation rapide : une classe Date plus élaborée devrait effectuer
        // un test en fonction du mois et de l'année
        jour = 0; mois = Incorrect; annee = 0;
   }
   else { jour = Jour; mois = Mois; annee = Annee; }
}
```

```
ostream & operator << (ostream & stream,
                       const Date & Source)
{ // Surcharge de l'opérateur de sortie sur un flux
   return stream << Source.jour
                 << "/" << Source.mois
                 << "/" << Source.annee;
}
```

6.6.5 - La classe *Sac*

Le fichier d'en-tête *c_Sac.h*

```
#ifndef _SAC
#define _SAC
#include <iostream.h>
#include "c_Collec.h"
class Sac : public Object {
 private:
   int nbObjectMax;   // capacité du sac
   int nbObject;      // nombre d'objets présents
   const Object * o; // pointeur sur l'objet contenu dans le sac
 protected:
   IdClasse classId() const { return SAC; }
   char * className() const { return "Sac"; }
   int isEqual(const Object & T2) const { return FALSE;}
 public:
   char * AsString() const { return className();}
   // Accesseurs en consultation des données membres
   int NbObjectMax() const { return nbObjectMax; }
   int NbObject() const { return nbObject; }
   const Object & ObjectSac() const;
   Boolean Empty() const { return ( nbObject == 0 ); }
   Sac(int NbObjectMax, int NbObject, Object & AObject);
      // Constructeur
   // Opérateurs de remplissage ou de vidage du sac
   Sac operator ++ (int);
   Sac operator -- (int);
   Sac operator + (int NbElem);
   Sac operator += (int NbElem);
   Sac operator - (int NbElem);
   Sac operator -= (int NbElem);
   void NouveauNbObject(int NbObject);
   void Remplir();
   friend ostream & operator << (ostream & stream,
                                 const Sac & Source);
};
#endif
```

***Le fichier** c_Sac.cpp*

```
#include "c_sac.h"

Sac::Sac(int NbObjectMax, int NbObject, Object & AObject)
{  // Constructeur
   nbObjectMax = NbObjectMax;
   nbObject = NbObject;
   o = &AObject;
}

const Object & Sac::ObjectSac() const
{  // Accesseur sur l'objet
   const Object * Ptr = o;
   if (Ptr == &NIL) erreur(EMPTY_ELEMENT);
   return *Ptr;
}

Sac Sac::operator ++ (int)
{  // Incrémente le nombre d'objets du sac de 1
   if (nbObject >= nbObjectMax) erreur(RANGE_CHECK);
   nbObject++;
   return * this;
}

Sac Sac::operator -- (int)
{  // Décrémente le nombre d'objets du sac de 1
   if (nbObject <= 0) erreur(RANGE_CHECK);
   nbObject--;
   return * this;
}

Sac Sac::operator + (int NbElem)
{  // Ajoute NbElem objets au sac et renvoie le sac récepteur du message
   if ( nbObject + NbElem <= nbObjectMax)
      nbObject += NbElem;
   return * this;
};

Sac Sac::operator += (int NbElem)
{  // Idem précédent - Ecriture plus compacte
   return  operator + (NbElem);
}

Sac Sac::operator - (int NbElem)
{  // Enlève NbElem objets au sac et renvoie le sac récepteur du message
   if ( nbObject - NbElem >= 0)
      nbObject -= NbElem;
   return * this;
};
```

```
Sac Sac::operator -= (int NbElem)
{  // Idem précédent - Ecriture plus compacte
   return operator - (NbElem);
}

void Sac::NouveauNbObject(int NbObject)
{  // Modifie le nombre d'objets du sac en NbObject
   if (NbObject > nbObjectMax) NbObject = nbObjectMax;
   if (NbObject < 0) NbObject = 0;
   nbObject = NbObject;
}

void Sac::Remplir()
{  // Remplit le sac au maximum de sa capacité
   nbObject = nbObjectMax;
}

ostream & operator << (ostream & stream,
                       const Sac & Source)
{  // Surcharge de l'opérateur de sortie sur un flux
   return stream << "Sac de contenance " << Source.nbObjectMax
                 << " et contenant " << Source.nbObject
                 << " objet"
                 << (Source.nbObject>1 ? "s":" " )
                 << "de type " << Source.o->AsString();
}
```

6.6.6 - La classe *Distributeur*

Le fichier d'en-tête c_Distri.h

```
#ifndef _DISTRIBUTEUR
#define _DISTRIBUTEUR
#include "c_Array.h"
#include "c_Prod.h"
#include "c_Sac.h"
#include "c_Piece.h"

class Distributeur : public Object {
 private:
   int nbProduit;
      // Nombre total de produits dans le distributeur
   int sommePiece;
      // Somme totale d'argent contenue dans le distributeur
   Array * tabProduit;
      // Pointeur sur un tableau de sacs de produits
```

```
    Array * tabPiece;
        // Pointeur sur un tableau de sacs de pièces
  protected:
    IdClasse classId() const { return DISTRIBUTEUR; }
    char * className() const { return "Distributeur"; }
    Boolean isEqual(const Object & T2) const
        { return FALSE;}
    Sac & sacProduitAt(int Rang);
        // Renvoie le sac de produits à la position Rang dans le distributeur
    const Sac & sacProduitAt(int Rang) const;
        // Renvoie le sac de produits... (pour un distributeur constant)
    Sac & sacPieceAt(int Rang);
        // Renvoie le sac de pièces à la position Rang dans le distributeur
    const Sac & sacPieceAt(int Rang) const;
        // Renvoie le sac de pièces... (pour un distributeur constant)
  public:
    char * AsString() const { return className();}
    // Constructeur et destructeur
    Distributeur(int NbProduit,
                 int CapaciteSacProd,
                 const Array & ListeProduit,
                 int CapaciteSacPiece,
                 const Array & ListePiece);
    ~Distributeur();
    //Accesseurs en consultation
    int NbProduit() const { return nbProduit; }
    int SommePiece() const { return sommePiece; }
    const Produit & ProduitAt(int Rang) const;
        // Renvoie le produit du sac de produits à la position Rang
    const Piece & PieceAt(int Rang) const;
        // Renvoie le pièce du sac de pièces à la position Rang
    int PrixAt(int Rang) const;
        // Renvoie le prix du produit contenu dans le sac à la position Rang
    int ValeurAt(int Rang) const;
        // Renvoie la valeur de la pièce contenu dans le sac à la position Rang
    Boolean EmptySacProduitAt(int Rang) const;
        // Renvoie TRUE si le sac de produits à la position Rang est vide
    Boolean EmptySacPieceAt(int Rang) const;
        // Renvoie TRUE si le sac de pièces à la position Rang est vide
    int NbProduitAt(int Rang) const;
        // Renvoie le nombre de produits contenus dans le sac à la position Rang
    int NbProduitMaxAt(int Rang) const;
        // Renvoie la capacité du sac de produits à la position Rang
    int NbPieceAt(int Rang) const;
        // Renvoie le nombre de pièces contenues dans le sac à la position Rang
    int NbPieceMaxAt(int Rang) const;
        // Renvoie la capacité du sac de pièces à la position Rang
```

```
    Boolean EstPayable(const Array & ArraySacPiece) const;
        // Renvoie TRUE si les pièces contenues dans ListePiece peuvent être
        // stockées dans les tubes du distributeur
    // Service
    void PaiementClient(const Array & ArraySacPiece,
                        int RangProduit);
    // Fonctions d'affichage
    void AfficheSacProduit(ostream & stream) const;
    void AfficheSacPiece(ostream & stream) const;
    friend ostream & operator <<
        (ostream & stream, const Distributeur & Source);
};
#endif
```

Le fichier c_Distri.cpp

```
#include <stdlib.h>
#include "c_Sac.h"
#include "c_Piece.h"
#include "c_Distri.h"

Sac & Distributeur::sacProduitAt(int Rang)
{   // Renvoie le sac de produit correspondant au rail numéro Rang
    return ((Sac&)tabProduit->At(Rang));
}

const Sac & Distributeur::sacProduitAt(int Rang) const
{   // Idem pour un distributeur constant
    return ((Sac&)tabProduit->At(Rang));
}

Sac & Distributeur::sacPieceAt(int Rang)
{   // Renvoie le sac de pièces correspondant au tube de pièces numéro Rang
    return ((Sac&)tabPiece->At(Rang));
}

const Sac & Distributeur::sacPieceAt(int Rang) const
{   // Idem pour un distributeur constant
    return ((Sac&)tabPiece->At(Rang));
}
```

```
Distributeur::Distributeur(int NbSacProduit,
                           int CapaciteSacProd,
                           const Array & ListeProduit,
                           int CapaciteSacPiece,
                           const Array & ListePiece)
{  // Constructeur - Suppose que tous les rails ont la même taille de stockage, et
   // les tubes de pièces la même capacité.
   nbProduit = NbSacProduit * CapaciteSacProd;
   sommePiece = 0;
   randomize(); // Réinitialisation du tirage aléatoire
   tabProduit = new Array(NbSacProduit);
   for ( int i = 0; i < NbSacProduit; i++) {
      // Tirage aléatoire d'un produit pour chaque rail
      int Rang =
         EntierAleatoireInf(ListeProduit.LastIndex());
      if (ListeProduit.EmptyAt(Rang))
         erreur(EMPTY_ELEMENT);
      tabProduit->AtPut(
         i,
         *(new Sac(CapaciteSacProd,
                   CapaciteSacProd,
                   *(new ProduitPerissable(
                     (ProduitPerissable&)ListeProduit.At(
                       Rang))))));
   }
   tabPiece = new Array(ListePiece.LastIndex()+1);
   for (i = 0; i <= ListePiece.LastIndex(); i++) {
      // Initialisation des tubes de pièces à partir de ListePiece
      if (ListePiece.EmptyAt(i)) erreur(EMPTY_ELEMENT);
      tabPiece->AtPut(
         i,
         *(new Sac(CapaciteSacPiece,
                   0,
                   *(new Piece ((Piece&)ListePiece.At(
                      i))))));
   }
}

Distributeur::~Distributeur()
{  // Destructeur - Libère l'espace alloué par tabProduit et tabPiece
   delete tabProduit;
   delete tabPiece;
}

const Produit & Distributeur::ProduitAt(int Rang) const
{  // Renvoie le produit stocké dans le rail numéro Rang
   return (Produit&) (sacProduitAt(Rang).ObjectSac());
}
```

```
const Piece & Distributeur::PieceAt(int Rang) const
{  // Renvoie la pièce de monnaie stockée dans le tube numéro Rang
   return (Piece&) (sacPieceAt(Rang).ObjectSac());
}

int Distributeur::PrixAt(int Rang) const
{  // Renvoie le prix du produit stocké sur le rail numéro Rang
   return ProduitAt(Rang).Prix();
}

int Distributeur::ValeurAt(int Rang) const
{  // Renvoie la valeur faciale de la pièce du tube numéro Rang
   return PieceAt(Rang).Valeur();
}

Boolean Distributeur::EmptySacProduitAt(int Rang) const
{  // Renvoie TRUE si le rail numéro Rang est vide
   return sacProduitAt(Rang).Empty();
}

Boolean Distributeur::EmptySacPieceAt(int Rang) const
{  // Renvoie TRUE si le tube numéro Rang est vide
   return sacPieceAt(Rang).Empty();
}

int Distributeur::NbProduitAt(int Rang) const
{  // Renvoie le nombre de produits restant dans le rail numéro Rang
   return sacProduitAt(Rang).NbObject();
}

int Distributeur::NbProduitMaxAt(int Rang) const
{  // Renvoie la capacité maximale du rail numéro Rang
   return sacProduitAt(Rang).NbObjectMax();
}

int Distributeur::NbPieceAt(int Rang) const
{  // Renvoie le nombre de pièces restant dans le tube numéro Rang
   return sacPieceAt(Rang).NbObject();
}

int Distributeur::NbPieceMaxAt(int Rang) const
{  // Renvoie la capacité maximale du tube numéro Rang
   return sacPieceAt(Rang).NbObjectMax();
}
```

```
Boolean Distributeur::EstPayable
    ( const Array & ArraySacPiece ) const
{   // Renvoie TRUE si la liste de pièces ArraySacPiece peut rentrer dans les
    // tubes du distributeur
    for ( int i = 0;i < ArraySacPiece.LastIndex(); i++)
       if (((Sac&)ArraySacPiece.At(i)).NbObject() +
           NbPieceAt(i) > NbPieceMaxAt(i) )
       return FALSE;
    return TRUE;
}

void Distributeur::PaiementClient(
    const Array & ArraySacPiece, int RangProduit)
{   // Valide l'achat d'un client, qui paye avec ArraySacPiece le produit numéro
    // RangProduit. Suppose que la liste de pièces a été validée par EstPayable,
    // et qu'un produit est encore présent dans le rail numéro Rang
    for ( int i = 0; i < ArraySacPiece.LastIndex(); i++) {
       sacPieceAt(i) +=
          ((Sac&)ArraySacPiece.At(i)).NbObject();
       sommePiece +=
          ((Sac&)ArraySacPiece.At(i)).NbObject()*
          ((Piece&)
            ((Sac&)ArraySacPiece.At(i)).ObjectSac())
            .Valeur();
    }
    sacProduitAt(RangProduit)--;
    nbProduit--;
}

void Distributeur::AfficheSacProduit(ostream & stream)
    const
{   // Affiche le contenu des rails de produits du distributeur
    for ( int i = 0; i <= tabProduit->LastIndex(); i++)
       if (! EmptySacProduitAt(i))
          stream << "  " << i << " :" << sacProduitAt(i)
                 << endl << "   "
                 << (ProduitPerissable&) ProduitAt(i)
                 << endl;
}
```

```
void Distributeur::AfficheSacPiece(ostream & stream)
   const
{  // Affiche le contenu des tubes de pièces du distributeur
   for (int i = 0; i <= tabPiece->LastIndex(); i++)
      if (! EmptySacPieceAt(i))
         stream << "  " << i << " :" << sacPieceAt(i)
                << " " << PieceAt(i) << endl;
   stream << "  soit au total : " << sommePiece << endl;
}

ostream & operator << (ostream & stream,
                       const Distributeur & Source)
{  // Description complète du distributeur : utilise les fonctions précédentes
   // AfficheSacProduit et AfficheSacPiece
   stream << Source.className() << endl;
   if (Source.nbProduit <= 0 )
      stream << "  Aucun produit" << endl;
   else {
      stream << "  Description de"
             << (Source.nbProduit > 1 ? "s " : " ")
             << Source.nbProduit << " produits" << endl;
      Source.AfficheSacProduit(stream);
   }
   stream << endl;
   if (Source.sommePiece == 0 )
      stream << "  Aucune pièce" << endl;
   else {
      stream << "  Description des pièces" << endl;
      Source.AfficheSacPiece(stream);
   }
   return stream;
}
```

6.6.7 - La classe *Client*

Le fichier d'en-tête c_Client.h

```
#ifndef _CLIENT
#define _CLIENT
#include <iostream.h>
#include "c_Array.h"
class Client : public Object {
 private:
   Array * tabPiece;
      // Pointeur sur un tableau de sacs de pièces
   int sommePiece;
      // Somme totale d'argent contenue dans le porte-monnaie
```

```
  protected:
    static Array & ListeProduitInit();
    IdClasse classId() const { return CLIENT; }
    char * className() const { return "Client"; }
    Boolean isEqual(const Object & T2) const
       { return FALSE;}
  public:
    char * AsString() const { return className();}
    // Constructeur et destructeur
    Client(int NbPiece, Array & ListePiece);
    ~Client();
    int SommePiece() { return sommePiece;}
       // Accesseur
    void PaieDistributeur(Array & TabSacPiece);
       // Met l'argent pour le distributeur
    Boolean PeutPayer(int PrixTotal,
                      Array & ListePiece,
                      int * Index);
       // Renvoie TRUE si le client a assez d'argent pour payer
       // le produit en faisant l'appoint
    int ChoixProduit(int Max);
       // Choisit le produit
    friend ostream & operator << (ostream & stream,
                                  const Client & Source);
       // Fonction d'affichage
};
#endif
```

Le fichier c_Client.cpp

```
#include <stdlib.h>
#include "c_Client.h"
#include "c_Sac.h"
#include "c_Piece.h"

Client::Client(int NbPiece,  Array & ListePiece)
{  // Constructeur de Client. Pour les besoins de la simulation, ce constructeur
   // tire aléatoirement NbPiece appartenant à la liste ListePiece.
   sommePiece = 0;
   randomize();  // Initialisation du tirage aléatoire
   tabPiece = new Array(ListePiece.LastIndex()+1);
   for (int i = 0; i <= ListePiece.LastIndex(); i++) {
      // Remplit le tabPiece avec les types de pièces contenus dans ListePiece
      if (ListePiece.EmptyAt(i)) erreur(EMPTY_ELEMENT);
      tabPiece->AtPut
        (i,
         *(new Sac(
               NbPiece, 1,
               *(new Piece((Piece&)ListePiece.At(i)))))
        );
```

```
    for (i = 0; i <= NbPiece; i++) {
        // Tirage aléatoire des pièces à placer dans tabPiece
        int Rang =
         EntierAleatoireInf(ListePiece.LastIndex()+1);
        ((Sac&)tabPiece->At(Rang))++;
        sommePiece +=
         ((Piece&)ListePiece.At(Rang)).Valeur();
    }

Client::~Client()
{   // Destructeur
    delete tabPiece;
}

int Client::ChoixProduit(int Max)
{   // Fonction de choix aléatoire d'un produit
    return EntierAleatoireInf(Max);
}

void Client::PaieDistributeur(Array & TabSacPiece)
{   // Valide le paiement du distributeur en actualisant les données membres
    // (sommePiece décrémentée du prix du produit payé et tabPiece décrémenté
    // des pièces utilisées pour payer le distributeur et décrite par TabSacPiece)
    for ( int i = 0; i < TabSacPiece.LastIndex(); i++) {
        ((Sac&)tabPiece->At(i)) -=
          ((Sac&)TabSacPiece.At(i)).NbObject();
        sommePiece -= ((Sac&)TabSacPiece.At(i)).NbObject()
           *((Piece&)((Sac&)TabSacPiece.At(i)).
             ObjectSac()).Valeur();
      }
}

Boolean Client::PeutPayer(int PrixTot, Array &
ListePiece, int * NbElem)
{   // Renvoie TRUE si le client peut payer PrixTot avec les pièces de son porte-
    // monnaie. Si c'est possible, stocke les pièces nécessaires dans ListePiece.
    // Fonction récursive - NbElem représente l'index le plus haut à partir duquel
    // on examine les pièces utilisables pour payer.
    int Index = * NbElem - 1;
    if (sommePiece < PrixTot) return FALSE;
        // Le porte-monnaie du client ne contient même pas la somme nécessaire
```

```
    while (((Sac&)tabPiece->At(Index)).NbObject() <= 0
            || ((Piece&)((Sac&)tabPiece->
              At(Index)).ObjectSac()).Valeur() >PrixTot ) {
        // On cherche la pièce de valeur la plus élevée
        // inférieure à la somme à payer
        Index --;
        if ( Index < 0) return FALSE;
            // Toutes les pièces restant dans le porte-monnaie
            // dépassent la somme à payer
    }
    // A ce niveau, Index représente la première pièce
    // du porte-monnaie inférieure à PrixTot.
    int ValeurPiece = ((Piece&)
     ((Sac&)tabPiece->At(Index)).ObjectSac()).Valeur();
        // ValeurPiece est une variable temporaire qui représente la valeur de la
        // pièce à la position Index
    int NbPiece = ((Sac&)tabPiece->At(Index)).NbObject();
        // NbPiece est une variable temporaire qui représente le nombre de pièces
        // à la position Index
    while ( ValeurPiece <= PrixTot && NbPiece-- > 0) {
        // On met la pièce de rang Index dans ListePiece,
        // et on décrémente PrixTot de sa valeur.
        ((Sac&) ListePiece.At(Index))++;
        PrixTot -= ValeurPiece;
    // Tant que la pièce de rang Index peut être utilisée pour payer,
    // (et tant qu'il y 'en a), on la met dans ListePiece
    }
    if ( PrixTot == 0) return TRUE; // Sortie de la récursivité si la
                                    // somme à payer est atteinte.
    * NbElem = Index + 1; // ... sinon
    return PeutPayer(PrixTot,ListePiece,NbElem);
        // On renvoie le résultat de PeutPayer avec comme arguments, la somme
        // restant dûe PrixTot, ListePiece en cours de confection et NbElem qui
        // représente le rang maximum des pièces à examiner.
}

ostream & operator << (ostream & stream,
                       const Client & Source)
{   // Surcharge de l'opérateur de sortie sur un flux.
    stream << Source.className() << endl
           << "  Description du porte monnaie Client" << endl;
```

```
    for (int i=0; i <= Source.tabPiece->LastIndex(); i++)
       if ( ! Source.tabPiece->EmptyAt(i))
          stream << "  " << i << " : "
                 << (Sac&) Source.tabPiece->At(i)
                 << " de "
                 << (Piece&) ((Sac&) Source.tabPiece->
                      At(i)).ObjectSac()
                 << endl;
    return stream  << "  Somme totale : "
                   << Source.sommePiece << endl;
}
```

Annexe A : compléments C/C++

A1 - Séquences d'échappement

En C++, les constantes caractères se représentent entre apostrophes, les constantes chaînes entre guillemets. Ces conventions imposent évidemment de définir un mode de représentation des caractères apostrophe ou guillemet, quand on veut les représenter en tant que tels. C++ offre aussi la possibilité de coder des caractères spéciaux comme le saut de page, le retour en arrière, le retour chariot, etc. La table suivante fournit la liste de ces *séquences d'échappement*.

`\0`	fin de chaîne de caractères
`\n`	passage à la ligne
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\b`	retour en arrière (un caractère)
`\r`	retour en début de ligne
`\f`	saut de page
`\a`	alerte (signal sonore)
`\\`	barre inverse, plus familièrement appelée antislash
`\?`	point d'interrogation ?
`\'`	apostrophe '
`\"`	guillemet "

Il est également possible de représenter un caractère par son code octal ou hexadécimal dans la table utilisée. L'échappement utilisé se compose du caractère antislash \ suivi des trois chiffres octaux (au plus) représentant son code, ou du caractère antislash \ suivi de x, suivi des caractères hexadécimaux représentant le caractère. Ces deux séquences d'échappement sont résumées dans la liste suivante :

`\ooo`	nombre octal ooo
`\xhhh`	nombre hexadécimal hhh

A2 - Mot-clé *static*

Le mot clé static correspond à plusieurs sens selon sa position syntaxique.

Dans une fonction

Les variables définies dans le corps d'une fonction sont dites *automatiques*. Elles sont créées automatiquement en début d'exécution d'un appel de la fonction et détruites en fin d'exécution de cet appel. L'attribut automatic est représenté par le mot-clé auto que l'on n'utilise pratiquement jamais car il est implicite :

```
void f1(...)
{
   auto int Ctr; // équivalent à int Ctr
   ... // etc.
}
```

On peut, dans une fonction, définir une variable locale avec l'attribut static. Dans ce cas, la variable est créée dans l'environnement global du programme mais elle n'est pas accessible en dehors de la fonction. Bien entendu, une variable static conserve sa valeur entre deux exécutions successives de la fonction où elle est définie :

```
int f2()
{
   static int Ctr = 10; // l'initialisation n'est faite qu'au premier appel
   Ctr = Ctr + 1;
   return Ctr;
}
void main()
{
   cout << f2(); // affichera 11
   cout << f2(); // affichera 12
}
```

Dans un en-tête de fichier

Exemple :

```
static int nbInstances;
static int compteInstances();
```

Une variable ou une fonction déclarée avec le mot clé static à une portée limitée au module dans lequel elle est déclarée. Elle n'est donc pas accessible à l'édition de liens par les autres modules.

Dans une définition de classe

Exemple :

```
class String {
 private:
   static int memInstances;
   static int memTotale();
... // etc.
}

String::memInstances = 0;
```

La variable qualifiée par static est une donnée membre de classe (variable de classe). Elle existe donc en un seul exemplaire, quel que soit le nombre d'instances créées dans la classe. Elle doit être déclarée dans le corps de la classe, et impérativement définie et initialisée à l'extérieur de la classe. Par défaut, l'initialisation peut être omise. Dans ce cas, la variable de classe prend la valeur 0.

La fonction qualifiée par static est une fonction membre de classe (méthode de classe) qui peut être appelée même si on ne l'associe à aucun autre objet. Dans ce cas, l'appel doit être qualifié par le nom de la classe :

```
cout << String::memTotale();   // appel en l'absence de tout objet
String Chaine;
... // etc.
cout << Chaine::memTotale();   // appel associé à un objet
```

Les membres static peuvent être private, public ou protected.

A3 - Arguments de la fonction *main*

La fonction main peut récupérer les arguments donnés à la ligne de commande, au lancement du programme. La syntaxe de sa définition[1] devient alors :

```
int main(int argc char* argv[])
{
    ... // etc.
    return valeurRenvoyee; // valeurRenvoyee est de type int
}
```

Le paramètre *argc* récupère le nombre d'arguments de la ligne de commande. Le paramètre *argv*[] est un tableau de chaînes de caractères, chacune de ces chaînes représentant un des arguments de la ligne de commande, et se terminant par convention par le caractère *\0*.

Le nombre de paramètres *argc* inclut le nom utilisé pour l'appel du programme dans la ligne de commande. Les paramètres sont rangés dans le tableau *argv* dans l'ordre de saisie sur la ligne de commande. Le paramètre de rang *argc* dans *argv* (paramètre qui suit le dernier donné sur la ligne de commande) est toujours initialisé avec la valeur NULL.

Supposons un programme *prog.exe* dont la fonction main est définie comme suit :

```
int main(int argc, char* argv[])
{
  cout << argc << endl;
  for (int i = 0; i < argc; cout << argv[i++]) << '\t');
  return;
}
```

L'exécution de la ligne de commande :

```
prog.exe 1 2 3
```

provoquera l'affichage sur le flux standard de sortie de :

```
4
prog.exe1     2     3
```

1 Le langage C++ n'impose pas de type renvoyé particulier pour la fonction main. La plupart des implémentations supportent les types renvoyés *void* et *int* et les deux formes :

```
int main() { ... }
int main(int argc, char * argv[]) { ... }
```

A4 - Priorités des opérateurs

Le tableau suivant récapitule les opérateurs du langage C++, en les classant par priorités décroissantes. Pour chaque niveau de priorité l'évaluation se fait de gauche à droite ou de droite à gauche selon l'indication d'associativité (troisième colonne)

Priorité	**Opérateurs**	**Associativité**
1 (la plus haute)	`()` *appel de fonction* `[] -> :: .`	gauche à droite

2	*opérateurs unaires*
	`! ~ + - ++ -- & *`
	`sizeof new delete`
		droite à gauche

3	`.* ->*`	gauche à droite
4	`* / %`	gauche à droite
5	`+ -`	gauche à droite
6	`<< >>`	gauche à droite
7	`< <= > >=`	gauche à droite
8	`== !=`	gauche à droite
9	`&`	gauche à droite
10	`^`	gauche à droite
11	`|`	gauche à droite
12	`&&`	gauche à droite
13	`||`	gauche à droite
14	`?:`	droite à gauche
15	`= *= /* %= += -= &=` `^= |= <<= >>=`	droite à gauche
16	`,`	gauche à droite

Annexe B : les *templates*

Le langage C++ est en voie de standardisation et les dernières propositions de normalisation incluent un mécanisme de génération automatique de classes paramétrées, celui des *templates*. De nombreux compilateurs supportent déjà cette fonctionnalité, que nous décrivons brièvement dans cette annexe.

B1 - Pourquoi les templates ?

Avant les *templates*, pour obtenir en C++, le polymorphisme d'une classe `Tableau` et ranger, dans un objet de cette classe, aussi bien des entiers que des réels ou des chaînes de caractères, il fallait définir autant de classes `Tableau` que de types de valeurs à enregistrer dans un tableau.

Une autre solution (Cf. chapitre 5) est d'intégrer toutes les classes de valeurs dans une hiérarchie dont la racine est une classe, souvent appelée `Object`, dont héritent toutes les autres classes. On définit alors la classe `Tableau` comme une classe dont les instances peuvent répertorier des pointeurs sur des instances de `Object` et donc pointer sur n'importe quel objet. Cette solution impose de nombreuses manipulations de pointeurs et ne permet pas de gérer simplement des types de valeurs comme `int`, `float`, etc., qui ne correspondent pas à des classes.

Avec les *templates*, tout devient plus simple : nous allons maintenant étudier leur utilisation, à travers un exemple.

B2 - Une classe *Bag* en C++

Supposons que nous voulons construire une classe dont les objets permettent d'enregistrer des valeurs non forcément distinctes. Une telle classe existe dans de nombreux environnements objets et elle est inspirée de la classe `Bag` de Smalltalk. Une instance d'une telle classe est représentée de manière imagée figure 1.

L'objet peut être représenté comme un « sac » dans lequel chaque valeur peut figurer plusieurs fois. Restreignons provisoirement les valeurs du sac à des entiers et définissons une première classe qui représentera ces entiers :

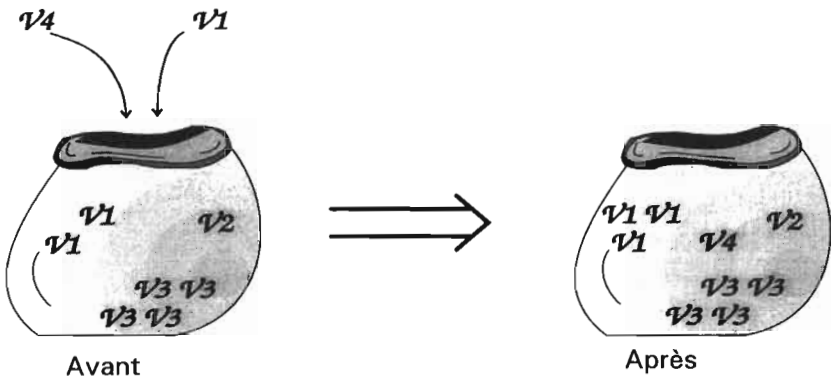


Figure 1 : on ajoute au sac les valeurs v1 et v4

```
class ValeurComptee {
// permet de représenter des valeurs avec répétition
 private:
  int valeur;    // valeur répertoriée
  int ctr;       // compteur de répétitions
 public:
  // accès aux membres privés
  int Valeur() { return valeur; }
  int Ctr() { return ctr; }
  // constructeur pour la première occurrence de V
  ValeurComptee(int V)
   { valeur = V; ctr = 1; }
  // constructeur par défaut
  ValeurComptee() {};
  // ajouter une répétition
  void UneDePlus() { ctr++; }
}; // class ValeurComptee
```

Une instance de *ValeurComptee* est l'association d'un entier (*valeur*) et d'un compteur d'occurrences de cet entier (*ctr*). La classe *ValeurComptee* fournit deux constructeurs dont l'un construit l'objet correspondant à l'enregistrement d'une première occurrence de la valeur V. Nous pouvons maintenant définir la classe souhaitée, en représentant un sac avec un tableau d'instances de la classe *ValeurComptee*, comme le suggère la figure 2.

Comme le montre la figure 2, une instance de la classe *SacDeValeurs* est représentée par un tableau t d'objets de la classe *ValeurComptee*, dont la dimension est indiquée par la donnée-membre dim. La donnée-membre *nbValeurs* indiquera

le nombre (< dim) de valeurs répertoriées dans le sac. Le constructeur *SacDeValeurs(int C)* crée un sac vide au départ, représenté par un tableau de C éléments.

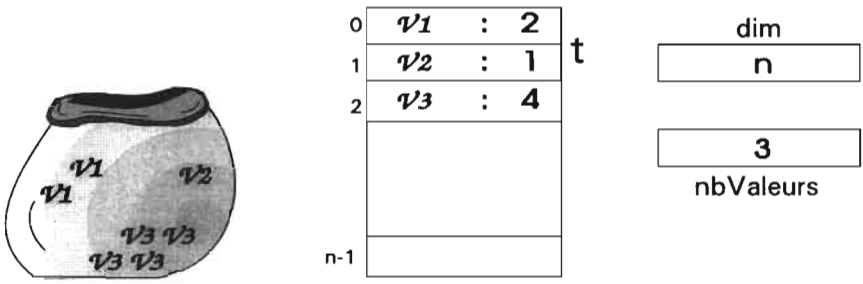


Figure 2 : implémentation d'un sac

Nous définissons la classe *SacDeValeurs*, comme suit.

```
class SacDeValeurs { // Sac d'objets
 private:
   ValeurComptee * t; // tableau des valeurs
   int dim; // dimension du tableau
   int nbValeurs; // nombre de valeurs enregistrées;
   int rang(int V); // accès à V dans t
       // renvoie le rang de V dans t ou une indication d'absence
 public:
   // constructeur d'un sac vide, de la capacité C
   SacDeValeurs(int C)
   {
      t = new ValeurComptee[C];
      dim = C; nbValeurs = 0;
   }
   // destructeur
   ~SacDeValeurs() { delete [] t; }
   void AjouteValeur(int V); // ajoute V dans le sac
   void AfficheToi(char * Titre);
       // affiche le contenu du sac à l'écran
}; // class SacDeValeurs
```

Nous ne détaillerons pas ici l'implémentation des fonctions-membres qui ne sont pas précisées dans la définition de la classe *SacDeValeurs*. Si nous supposons cependant que la définition de la classe est complète, on peut compiler et exécuter une séquence telle que :

```
SacDeValeurs MonSac(5);
MonSac.AjouteValeur(4);
MonSac.AjouteValeur(2);
MonSac.AjouteValeur(3);
MonSac.AjouteValeur(4);
MonSac.AfficheToi("MonSac");
```

qui produira, par exemple, l'affichage :

```
Contenu du SacDeValeurs MonSac
4 présent 2 fois
2 présent 1 fois
3 présent 1 fois
```

B3 - Définir une classe paramétrée

Si la classe *SacDeValeurs* nous convient, comment pouvons-nous l'utiliser pour représenter des sacs de valeurs réelles, ou de valeurs d'une classe que nous aurons nous-mêmes définie ? Avant les *templates*, il n'y avait pas d'autre solution que celle de définir autant de classes *SacDeValeursXXX* que nous avons de types de valeurs à gérer ainsi.

Avec les *templates*, nous pouvons maintenant définir une seule classe paramétrée et laisser le compilateur faire le travail à notre place :

```
template <class Objet> class SacDeValeurs {
// Sac d'objets
 private:
   class ValeurComptee { // Objet Répété
    private:
     Objet valeur;   // valeur répertoriée
     int ctr;        // nombre d'occurrences de cette valeur
    public:
     Objet Valeur() { return valeur; }
     int Ctr() { return ctr; }
     // constructeurs
     ValeurComptee() {};
     ValeurComptee(Objet V);
     void UneDePlus () { ctr++; }
    }; // class ValeurComptee
   ValeurComptee * t; // tableau des valeurs
   int dim; // dimension du tableau
   int nbValeurs; // nombre de valeurs enregistrées;
   int rang (Objet V);
 public:
   SacDeValeurs(int Capacite);
   ~SacDeValeurs() { delete [] t; }
   void AjouteValeur(Objet V);
   void AfficheToi(const char * Titre);
}; // class SacDeValeurs
```

Par rapport à la version précédente, la spécification :

```
template <class Objet>
```

transforme la classe *SacDeValeurs* en un modèle de classe (*class template*), qui décrit comment le compilateur doit construire les classes paramétrées avec l'argument *Objet*. On notera l'emboîtement de la classe *ValeurComptee* dans la classe *SacDeValeurs*, qui fait que *ValeurComptee* est aussi un modèle de classe, avec le même argument *Objet*. La portée d'une classe emboîtée est limitée à la classe emboîtante. Ainsi, en dehors de la définition de la classe *SacDeValeurs*, la classe *ValeurComptee* devra être désignée par *SacDeValeurs::ValeurComptee*.

Comment utiliser un modèle de classe ? Il suffit de préciser la valeur du *template*, dans la déclaration des objets souhaités :

```
class CodeVille  { ... };
// un objet CodeVille associe un code postal et la ville correspondante
SacDeValeurs <int>::ValeurComptee A(11);
SacDeValeurs <int> Sac1(5);
SacDeValeurs <float> Sac2(10);
SacDeValeurs <CodeVille> Sac3(10);
```

B4 - Modèles de fonctions

On notera enfin que le mécanisme des templates peut être utilisé pour définir des fonctions génériques dont un ou plusieurs arguments doivent être de type variable. Supposons que la fonction *AfficheToi* de la classe *SacDeValeurs* est définie par :

```
template <class Objet>
void SacDeValeurs<Objet>::AfficheToi(const char * Titre)
{
   cout << "Contenu du SacDeValeurs "
        << Titre << "\n";
   for (int K = 0; K < nbValeurs; K++) {
        Affiche(t[K].Valeur()); // t[K].Valeur() est du type Objet
        cout <<"présent "  <<t[K].Ctr()  <<" fois\n";
   }
} // SacDeValeurs<Objet>::AfficheToi()
```

Dans cette fonction, l'affichage d'une valeur de type *Objet* est réalisé par appel à la fonction *Affiche* qui, si *Objet* correspond à int ou à float sera définie par :

```
void Affiche(int E) { cout << E; }
void Affiche(float R) { cout << R; }
```

Mais si *Objet* est une classe dont les valeurs ne sont pas acceptées par l'opérateur << des flux C++

, il nous faudra soit redéfinir cet opérateur, ce qui peut impliquer un travail complexe, soit, tout simplement définir le modèle de fonction (*function template*) :

```
template <class Valeur>
void Affiche(Valeur V) { V.AfficheToi(); }
```

en s'assurant, bien entendu, que chaque classe susceptible de correspondre à *Objet* définit bien la fonction-membre *AfficheToi* adéquate.

Index

, 221
! 6
!= 6, 221
#define 16-17, 47, 94
#include 8, 17, 30-31
% 221
& 28, 58, 221
&& 6, 221
* 28, 221
& 221
*= 221
+ 221
++ 188, 221
+= 221
- 221
-- 188, 221
-= 221
-> 29, 142, 221
. 221
/ 221
/= 221
:: 48, 221
; 2
< 6
<< 19-20, 26, 124, 221
<<= 221
<= 6
== 6, 150-151
> 6
>= 6
>> 19-20, 221
>>= 221
[] 61
^ 221
^= 221
{} 4
|| 221
| 6
~ 51

A

accesseur 156, 160
  en modification 110
accolade *voir* {}
accès
  privé 23-24
  public 23-24
adresse 28
affectation 3, 59, 62, 64, 82-84, 117, 126-127
affichage *voir* sdout, stderr
allocation dynamique *voir* mémoire
argument
  main 220
  nombre variable 44-46
  par défaut 43
  référence 64
Array 153-154, 160, 169, 172
ASCII 3
automatic 218

B

bloc 4, 27, 53, 55

C

C (langage) 1
caractère 73
  char 2
  spéciaux 4,217
cast *voir* transtypage

cerr 38-39
char 2
chaîne 3, 71
  affichage à l'écran 9
cin 19-21, 88, 91
class 24-25, 31
classe 20, 71
  abstraite 122-123, 147, 167
  Array 153-154, 160, 169, 172
  Collection 121, 166-167
  constructeur 26, 104
  conteneur
  conteneurs 112, 142, 166
  Date 183, 185
  d'itérateur 162
  de base 103, 113, 121, 134
  déclaration 167
  définition 25
  dérivée 105, 113, 131, 136
  hiérarchie 141, 166
  instance 27
  Integer 146-147
  librairie 139
  Memoire 108
  Object 141-142, 168, 186
  paramétrée 226
  Piece 180
  Produit 104, 124, 136
  Sac 187
  SetIterator 173
  static 219
  String 72, 92, 95-100, 102, 181
  TableEntiers 34, 37, 49, 112
  UndefinedObject 158
  variable de 91
  virtuelle 134
clrscr 12
Collection 166-167
comparaison 148, 152
compilateur 13, 108, 120, 134
  passage des paramètres 56
compilation
  bibliothèque 17
  séparée 13
concaténation 84
const 11, 18, 21, 66-68, 77
constante 16, 37, 67
  déclaration 67
  partagée 67
  transmettre 67
constructeur 26, 34, 46, 61, 64, 104, 107, 115
  appel 37
  classe dérivée 106
  conversion 149
  conversion de type 61
  copie 63-64, 75
  défaut, 65
  définir 35
  héritage multiple 132
  liste d'initialisation 107
  par défaut 33-34, 39, 107, 116, 132, 136
  plusieurs 40
conversion 118
  règle de 63
copie
  constructeur 63-64, 75
  donnée membre 127
  membre à membre 63-64
cout 19-21, 23, 26, 88, 91, 93, 126, 129

D

Date 183, 185
débogage 94
déclaration (de constante) 66-68
définition (de constante) 68
delete 29-30, 51-53, 55, 160, 221
  redéfinir 160
delete[] *voir delete*
destructeur 137
  explicite 50-51
  par défaut 50
  virtuel 136
différence 87
do 4, 30
donnée membre 21-23, 26, 33, 73
  accès 108, 112, 114
  copie 127
  homonyme 115
  héritage 105, 108, 114-115
  héritage multiple 133
  initialisation 37, 65, 107
  privée 24, 110
  redéfinition 115
  surcharge 114

double 2

E

éditeur de liens 13
égalité 87, 152
élément
  vide 163
else 6
encapsuler 22
endl 89
exit 25
extern 14, 18, 67

F

fichier
  objet 14
  partage de constante 66
  source 13-14, 66
float 2
flux 89
  d'entrée *voir* stdin
  de sortie *voir* sdout, stderr
fonction 7
  amie 126
  argument 43
  constante 77
  déclaration 8, 48
  définition 8
  en ligne 47, 77
  get 90
  inline 47
  main 7, 9-10, 30, 32, 220
  modèle 227
  nom 8
  non constante 77
  paramètres 8
  paramètre (transmettre) 55
  paramètre constant 67
  printf 9, 11
  putback 90
  signature 43, 147
  strcpy 72
  valeur renvoyée 11, 58-59
fonction membre 22-23, 26-27, 36, 115, 142
  argument 43
  constante 77
  déclaration 8, 48
  définition 8
  en ligne 47, 77
  héritage 114
  héritage multiple 133
  inline 47
  modèle 227
  nom 8
  non constante 77
  paramètres 8
  paramètre (transmettre) 55
  paramètre constant 67
  privée 24, 36
  redéfinition 108, 115, 123
  signature 43, 147
  valeur renvoyée 11, 58-59
  virtuelle 118, 121, 123, 143, 147, 162
  virtuelle pure 122, 151
for 4, 41-42
free 51
friend 89, 113, 123, 125, 128

G

garbage collector 55
get 90

H

héritage 103
  affectation 127
  fonction membre 115
  fonction virtuelle 118
  multiple 130, 135
  multiple (transtypage) 135
  opérateur 115, 123
  pointeurs 135
  private 111, 114
  public 109, 113-114
  transtypage 128-129

I

if 6
indexation 112
initialisation 107-108
  liste 66, 107
inline 47-48
instance 21, 27, 34, 37, 91
instanciation 21, 36
instruction

switch 48
do 30
for 41-42
while 3-4
int 2, 146
Integer 146
intervalle 140
iostream.h 88
istream 88, 90
itérateur 162, 164, 171
itération 3
classe 162
do 30
for 41-42
sur instance de Array 162
while 3-4

L

liste d'initialisation 108, 116, 132
long 2
lvalue 59, 81

M

macro 45, 47-48
va_arg 46
va_end 46
va_list 46
va_start
main 7, 9-10, 12, 30, 32, 220
message 18-19
récepteur 80
module 13, 15
mémoire
allocation dynamique 27-30, 36, 38, 40-41, 52, 54-55, 74
destructeur 137
libération 50
new 28, 38, 55
place 28
méthode
de classe 93

N

new 38, 51-52, 55, 134, 160, 221
redéfinir 160
NIL 157-159
NULL 38, 158

O

Object 141-142, 151, 158, 167-168, 186
objet 18-19, 34
absence 157
comparaison 146, 148
créer 33
désigner 80
héritage multiple 135
initialiser 35
temporaire 62, 68, 117
opérateur 86, 115, 123
| 6
!= 6, 87, 221
% 221
%= 79
& 28, 42, 60-61, 170, 221
&& 6, 221
&< 79
&= 221
() 221
* 28, 42, 221
*= 79, 221
+ 84-85, 221
++ 42, 188, 221
+= 79, 221
- 221
-- 188, 221
-= 79, 221
-> 29, 221
. 78, 221
/ 221
/= 79, 221
:: 48, 78, 221
< 6, 87
<< 23, 26, 88-90, 123-126, 221
<<= 79, 221
<= 6, 87
= 82-84, 87
== 6, 150-151
> 6, 87
>= 6, 87
>> 88, 221
>>= 79, 221
[] 80-82, 112-113, 221
^ 221
^= 79, 221
|| 221

|= 79
|| 6
affectation 59, 82-84
ami 123
argument 126
binaire 79
comparaison 86
d'indexation 61, 80-81
delete 52, 160, 221
de transtypage 165
entrée 90
entreé/sortie 88
new 52, 160, 221
opérande 124
priorité 221
scope 48
sizeof 45, 78, 221
surcharge 78, 80, 82, 86, 88, 90
unaire 42, 79
operator 123, 126
ostream 88, 123, 125, 128

P

paramètre 67
copie 55
effectif 56, 58
empêcher modification 68
formel 39, 56, 58, 63
synonyme 58
transmission 63
valeur par défaut 39
Piece 180
pointeur 28, 64
constant 66, 68-69
delete 29
destruction 29
et tableau 41
initialisation 28, 41
invariable 41
NULL 39
sur caractère 73
sur *char* 137
sur paramètre 56
sur valeur constante 66, 69
tableau 36, 40
variable 42
pointeurs
héritage multiple 135
polymorphisme 119-120, 139, 151, 171-172
portée 48, 53
printf 9, 11
private 25, 31, 106, 109, 111, 114
Produit 104
programme 10
mise au point 94
module 12, 16
protected 109-111, 114, 123-124, 151
protocole 19
préprocesseur 17
public 31, 106, 109, 111, 113-114
constructeur 35
héritage 109
putback 90

R

rand 8
return 7, 12, 60
référence 170
externe 66

S

sac 224
schéma itératif *voir* itération
SetIterator 173
short 2
signature 43, 147
sizeof 45, 221
sprintf 145
static 15, 92-93, 218
stdarg.h 46
stderr 26
stdin 88
stdio.h 9
stdout 26, 88
strcmp 86
strcpy 72
String 72, 92, 95-100, 102, 181
constructeur 75
définition 74, 78
instanciation 76
struct 20, 23, 31
structure
programme 7
surcharge 78, 80, 83-84, 88, 90

+ 85
affectation 82-84
comparaison 86
donnée membre 114
opérateur 84-85
switch 48
séparateur
d'instructions 2
séquence d'échappement 217

T

tableau 41, 139, 147
d'entiers 33
d'objets 154
de caractères 72
de pointeur 120, 141
template 141, 223-224, 226-227
this 80
transmission
par copie 57-58
par référence 57
transtypage 128-129, 134, 139, 146, 148-149, 152, 162
vers référence 150, 170
type
conversion 61
prédéfini 149
typedef 112

U

UndefinedObject 158

V

va_arg 45-46
va_end 45-46
va_list 45-46
va_start 45
valeur 2
renvoyée 7, 11-12
variable
affecter 2
automatique 13
de bloc 50
de classe 92, 94
durée de vie 53
dynamique 54-55
définition 53
globale 15, 54, 91, 159
initialisation 54
locale 54-55, 60
NIL 159
portée 53
référence 60
virtual 120-121, 137-138
void 12, 26

W

while 3-4

MASSON Éditeur
120, boulevard Saint-Germain
75280 Paris Cedex 06
Dépôt légal : mai 1994

SNEL S.A.
Rue Saint-Vincent 12 – 4020 Liège
avril 1994

SYSTÈMES D'EXPLOITATION

UNIX. Programmation avancée. M.J. Rochkind.
UNIX SYSTEME V. Système et environnement. A.-B. Fontaine et Ph. Hammes.
- LES SYSTÈMES D'EXPLOITATION. Structure et concepts fondamentaux. C. Lhermitte.

LE SYSTÈME D'EXPLOITATION PICK. M. Bull.

MICROPROCESSEURS ET ARCHITECTURE DE L'ORDINATEUR

- TECHNOLOGIE DES ORDINATEURS pour les IUT et BTS. Avec exercices. P.-A. Goupille.
- ARCHITECTURE DES ORDINATEURS. Des techniques de base aux techniques avancées. G. Blanchet et B. Dupouy.

PRINCIPES DE FONCTIONNEMENT DES ORDINATEURS. R. Dowsing et F. Woodhams.
- IMPLANTATION DES FONCTIONS USUELLES EN 68000. F. Bret.

LES MICROPROCESSEURS 80286/80386. Nouvelles architectures PC. A.-B. Fontaine et F. Barrand.

RÉSEAUX ET TÉLÉCOMS

L'UNIVERS DES RÉSEAUX ETHERNET. Concepts, produits, mise en pratique. N. Turin.
- TÉLÉMATIQUE. Téléinformatique et réseaux. M. Maiman.

TÉLÉCOMS ET RÉSEAUX. M. Maiman.
RNIS. Description technique. Ph. Chailley et D. Seret.
UNE INTRODUCTION À TCP/IP. J. Davidson.
X 25. Protocoles pour les réseaux à commutation de paquets. R.J. Deasington.
OSI. LES NORMES DE COMMUNICATION ENTRE SYSTÈMES OUVERTS. J. Henshall et S. Shaw.
LES RÉSEAUX LOCAUX. Comparaison et choix. J.S. Fritz, C.F. Kaldenbach, L.M. Progar.
ORDINATEURS INTERFACES ET RÉSEAUX DE COMMUNICATION. S. Collin.
RNIS. Concept et développement J. Ronayne.
LE RÉSEAU SNA. K.C.E. Gee.
RÉSEAUX ET MICRO-ORDINATEURS. Ph. Jesty.

INTELLIGENCE ARTIFICIELLE

INTELLIGENCE ARTIFICIELLE. E. Rich.
PROGRAMMATION DES SYSTÈMES EXPERTS EN PASCAL. B. Sawyer et D. Foster.
SYSTÈMES EXPERTS. Concepts et exemples. J.L. Alty et M.J. Coombs.

INFORMATIQUE POUR L'ENTREPRISE

COMPRENDRE, CONCEVOIR ET UTILISER LES SIAD. A. Checroun.
INFORMATIQUE DOCUMENTAIRE. A. Deweze.
- INGÉNIERIE DES DONNÉES. Bases de données, systèmes d'information, modèles et langages. E. Pichat et R. Bodin.
- MODÉLISATION DANS LA CONCEPTION DES SYSTÈMES D'INFORMATION. Avec exercices commentés. Acsiome.

L'AUDIT INFORMATIQUE. Méthodes, règles, normes. M. Thorin.
APPLICATION SYSTEM. Un système IBM de 4e génération. J. Rambaud.
COMPRENDRE LES BASES DE DONNÉES. Théorie et pratique. A. Mesguich et B. Normier.

INFORMATIQUE INDUSTRIELLE ET APPLICATIONS SCIENTIFIQUES

GRAPHISME DANS LE PLAN ET DANS L'ESPACE AVEC TURBO PASCAL 4.0. R. Dony.
- SPÉCIFICATION ET CONCEPTION DES SYSTÈMES. Une méthodologie. J.-P. Calvez.
- SPÉCIFICATION ET CONCEPTION DES SYSTÈMES. Études de cas. J.-P. Calvez.

PROGRAMMATION EN INFOGRAPHIE. Principes, exercices et programmes en C. L. Ammeraal.
- INFOGRAPHIE ET APPLICATIONS. T. Liebling et H. Röthlisberger.

MÉTHODES DE DÉVELOPPEMENT D'UN SYSTÈME À MICROPROCESSEURS. A. Amghar.
INFORMATIQUE GRAPHIQUE DANS LE BATIMENT ET L'ARCHITECTURE. G. Lauret.
- EXERCICES DE RECONNAISSANCE DE FORMES PAR MICRO-ORDINATEUR. Ph. Fabre.

SYSTÈMES TEMPS RÉEL EN ADA. Une approche virtuelle et asynchrone. L. Briand.
BASES DE DONNÉES POUR LE GÉNIE LOGICIEL. C. Godart et F. Charoy.
NORMES DE MÉTAFICHIERS ET D'INTERFACES POUR L'INFOGRAPHIE. D.B. Arnold et P.R. Bono.

MANUELS INFORMATIQUES MASSON

comprendre et utiliser C++ pour programmer objets

G. CLAVEL I. TRILLAUD L. VEILLON

C++ est un langage exigeant, mais sa souplesse et son efficacité sont telles qu'il devient l'un des plus utilisés dans la programmation objet.

Afin de ne pas égarer le lecteur dans le labyrinthe de ses spécifications, cet ouvrage propose un apprentissage progressif, basé non sur une présentation classique de ses fonctionnalités mais sur la découverte des concepts objets de base. Cette progression introduit les mécanismes de C++ au moment opportun, lorsque le besoin auquel ils répondent apparaît clairement. Aucune connaissance préalable n'est donc requise, si ce n'est une pratique élémentaire de la programmation.

La première partie rappelle les éléments de base de C et introduit les mécanismes correspondants de C++ (classes, objets, messages, instanciation). La seconde partie initie à la conception et la définition d'une classe, puis à l'utilisation de l'héritage. Enfin, la dernière partie est consacrée à la conception et l'utilisation d'un ensemble de classes. On y étudie les techniques de construction d'une librairie de classes puis un exemple complet d'application.

L'ouvrage fournira les connaissances de base à la fois aux programmeurs concernés par les particularités de l'application qu'ils construisent et à ceux qui conçoivent et implémentent des classes dans diverses applications.

*Polytechnicien, **Gilles CLAVEL** est directeur-consultant de la société IMA-informatique, spécialisée dans la technologie objets. Il est également professeur à l'Institut National Agronomique.*

***Isabelle TRILLAUD** est ingénieur en informatique, diplômée de l'Institut National Agronomique. Elle est responsable des études et projets au sein de la société IMA-informatique.*

***Luc VEILLON** est ingénieur de recherche, directeur du centre de calcul de l'Institut National Agronomique, où il collabore aux travaux orientés objets de la chaire d'informatique.*