Cours de C/C++

Christian Casteyde

Cours de C/C++

par Christian Casteyde

Historique des versions

Version 1.39.0	31/12/2000	Revised by: CC

Mise en conformité des en-têtes C++ des exemples avec la norme. Correction des exemples utilisant des noms réservés par la librairie standard. Complément sur les exceptions. Corrections sur l'instanciation des template et précisions sur leur édition de liens. Première ébauche de description de la librairie standard C++.

Version 1.38.1	14/10/2000	Revised by: CC

Précisions sur les classes de base virtuelles. Corrections orthographiques.

Version 1.38.0	01/10/2000	Revised by: CC

Corrections typographiques. Précisions sur les opérateurs & et *.

Version 1.37	23/08/2000	Revised by: CC

Passage au format de fichier SGML. Ajout des liens hypertextes. Corrections mineures.

Version 1.36	27/07/2000	Revised by: CC

Complément sur les parenthèses dans les définitions de macros. Corrections sur la numérotation des paragraphes.

Version 1.35	10/07/2000	Revised by: CC

Corrections sur les déclarations using.

Version 1.34	09/07/2000	Revised by: CC

Passage en licence FDL. Ajout de la table des matières.

Version 1.33	22/60/2000	Revised by: CC

Correction d'une erreur dans le paragraphe sur les paramètres template template. Corrections orthographiques diverses.

Version 1.32	17/06/2000/	Revised by: CC

Correction d'une erreur dans le programme d'exemple du premier chapitre. Correction d'une erreur dans un exemple sur la dérivation. Précisions sur le comportement du mot clef const. Corrections orthographiques diverses.

Version 1.31	12/02/2000	Revised by: CC

Corrections mineurs. Ajout du paragraphe sur la spécialisation d'une fonction membre d'une classe template.

Version 1.30	05/12/1999	Revised by: CC

Ajout de la licence. Modifications mineures du formatage.

Version <1.30	<1998	Revised by: CC

Version initiale.

Table des matières

Liste des tableaux

Liste des exemples

Avant-propos

Le présent document est un cours de C et de C++. Il s'adresse aux personnes qui ont déjà quelques notions de programmation dans un langage quelconque. Les connaissances requises ne sont pas très élevées cependant : il n'est pas nécessaire d'avoir fait de grands programmes pour lire ce document. Il suffit d'avoir vu ce qu'est un programme et compris les grands principes de la programmation.

La description de toutes les fonctions de la librairie C n'est pas donnée dans ce cours. De même, les fonctions les plus courantes de la norme POSIX (telles que les fonctions de manipulation des fichiers par exemple) ne sont pas décrites, car bien que présentes sur quasiment tous les systèmes d'exploitation, elles sont spécifiques à cette norme et n'appartiennent pas au langage en soi. Seules les fonctions incontournables des librairies seront donc présentées ici. Si vous désirez plus de renseignements, reportez-vous à la documentation des environnements de développement, des kits de développement des systèmes d'exploitation (SDK) et à la bibliographie. En revanche, la librairie standard C++ est décrite en détail, car elle fait désormais partie du langage d'une part, et fournit une illustration exemplaire des fonctionnalités du C++ d'autre part.

Ce document a pour but de présenter le langage C++ tel qu'il est décrit par la norme ISO 14882 du langage C++. Cependant, bien que cette norme ait été publiée en 1999, le texte officiel n'est pas librement disponible. Comme je ne veux pas cautionner le fait qu'un texte de norme internationnal ne soit pas accessible à tous, je me suis rabattu sur le document du projet de normalisation du langage, datant du 2 décembre 1996 et intitulé Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++ (http://www.cygnus.com/misc/wp/dec96pub/). Je serai reconnaissant à quiconque pourrait me procurer le texte officiel de cette norme, afin que je puisse mettre en conformité ce cours. Ceci ne règlerait toutefois pas le problème de la rétention d'information pratiquée par les groupes de l'ISO.

Notez que les compilateurs qui respectent cette norme se comptent encore sur les doigts d'une main, et que les informations et exemples donnés ici peuvent ne pas s'avérer exactes avec certains produits. En particulier, certains exemples ne compileront pas avec les compilateurs les plus mauvais. Notez également que certaines constructions du langage n'ont pas la même signification avec tous les compilateurs, parce qu'elles ont été implémentées avant que la norme ne les spécifie complètement. Ces différences peuvent conduire à du code non portable, et ont été signalées à chaque fois dans ce document dans une note. Le fait que les exemples de ce cours ne fonctionnent pas avec de tels compilateurs ne peut donc pas être considéré comme une erreur de ce document, mais plutôt comme une non-conformité des outils utilisés, qui sera sans doute levée dans les versions ultérieures de ces produits.

Après avoir tenté de faire une présentation rigoureuse du sujet, j'ai décidé d'arranger ce document dans un ordre plus pédagogique. Il est à mon avis impossible de parler d'un sujet un tant soit peu vaste dans un ordre purement mathématique, c'est à dire un ordre où les notions sont introduites une à une, à partir des notions déjà connues (chaque fonction, opérateur, etc… n'apparaît pas avant sa définition dans le document). Un tel plan nécessiterait de couper le texte en morceaux qui ne sont plus thématiques. J'ai donc pris la décision de présenter les choses par ordre logique, et non par ordre de nécessité syntaxique.

Les conséquences de ce choix sont les suivantes :

- il faut admettre certaines choses, quitte à les comprendre plus tard ;
- il faut lire deux fois ce document. Lors de la première lecture, on voit l'essentiel, et lors de la deuxième lecture, on comprend les détails (de toutes manières, je félicite celui qui comprend toutes les subtilités du langage à la première lecture).

Enfin, ce document n'est pas une référence et contient certainement des erreurs. Toute remarque est donc la bienvenue. Je tâcherai de corriger les erreurs que l'on me signalera dans la mesure du possible, et d'apporter les modifications nécessaires si un point est obscur. En revanche, il est possible que les réclamations concernant la

forme de ce document ne soient pas prises en compte, parce que j'ai des contraintes matérielles et logicielles que je ne peux pas éviter. En particulier, je maintiens ce document sous un unique format, et je m'efforce d'assurer la portabilité du document sur différents traitements de texte.

Afin de reconnaître les différentes éditions de ce document, un historique des révisions a été inclus en première page. La dernière version de ce document peut être trouvée sur mon site Web (http://casteyde.christian.free.fr).

Chapitre 1. Introduction

Le C++ est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Il souffre cependant de la réputation d'être compliqué et illisible. Cette réputation est en partie justifiée. La complexité du langage est inévitable lorsque l'on cherche à avoir beaucoup de fonctionnalités. Quant à la lisibilité des programmes, elle dépend essentiellement de la bonne volonté du programmeur.

Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, ce langage est, avec le C, idéal pour ceux qui doivent assurer la portabilité de leurs programmes au niveau des fichiers sources (pas des exécutables).

Les principaux avantages du C++ sont les suivants :

- grand nombre de fonctionnalités ;
- performances du C ;
- facilité d'utilisation des langages objets ;
- portabilité des fichiers sources ;
- facilité de conversion des programmes C en C++ ;
- contrôle d'erreurs accru.

On dispose donc de quasiment tout : puissance, fonctionnalité, portabilité et sûreté. La richesse du contrôle d'erreurs du langage, basé sur un typage très fort, permet de signaler un grand nombre d'erreurs à la compilation. Toutes ces erreurs sont autant d'erreurs que le programme ne fait pas à l'exécution. Le C++ peut donc être considéré comme un " super C ". Le revers de la médaille est que les programmes C ne se compilent pas directement en C++ : quelques adaptations sont nécessaires. Cependant, celles-ci sont minimes, puisque les syntaxes du C++ est basée sur celle du C. On remarquera que tous les programmes C peuvent être modifiés pour compiler à la fois en C et en C++.

Ce document est organisé en trois grandes parties. La première partie (chapitres 2 à 7) traite des fonctionnalités communes au C et au C++, en différenciant bien celles qui ne sont disponibles qu'en C++. Cette partie présente essentiellement la syntaxe de base du C et du C++. La deuxième partie (chapitres 8 à 12) ne traite que du C++. Les sujets suivants y sont traités : programmation orientée objet, exceptions, identification dynamique des types, espaces de nommage et template. Enfin, la dernière partie (chapitres 13 à FIXME) présente la librairie standard C++.

Le C présenté dans la première partie est compilable en C++. Ceci signifie qu'il n'utilise pas certaines fonctionnalités douteuses du C. Ceux qui désirent utiliser la première partie comme un cours de C doivent donc savoir qu'il s'agit d'une version épurée de ce langage. Les appels de fonctions non déclarées ou les appels de fonctions avec trop de paramètres ne sont donc pas considérées comme des pratiques de programmation valables dans ce cours.

En ce qui concerne la syntaxe, elle sera donnée sauf exception avec la convention suivante : ce qui est entre crochets ('[' et ']') est facultatif. De plus, quand plusieurs éléments de syntaxe sont séparés par une barre verticale ('|'), l'un de ces éléments seulement doit être présent (c'est un " ou " exclusif). Enfin, les points de suspension désigneront une itération éventuelle du motif précédent.

Par exemple, si la syntaxe d'une commande est la suivante :

```
[fac|rty|sss] zer[(kfl[,kfl[...]])];
```

les combinaisons suivantes seront syntaxiquement correctes :

```
zer;
fac zer;
rty zer;
zer(kfl);
sss zer(kfl,kfl,kfl,kfl);
```

mais la combinaison suivante sera incorrecte :

```
fac sss zer()
```

pour les raisons suivantes :

- *fac* et *sss* sont mutuellement exclusifs, bien que facultatifs tous les deux ;
- au moins un *kfl* est nécessaire si les parenthèses sont mises ;
- il manque le point virgule finale.

Rassurez-vous, il n'y aura pratiquement jamais de syntaxe aussi compliquée. Je suis sincèrement désolé de la complexité de cet exemple.

Chapitre 2. Première approche du C++

Le C/C++ est un langage *procédural*, du même type que le Pascal par exemple. Cela signifie que les instructions sont exécutées linéairement et regroupées en blocs : les *fonctions* et les *procédures* (les procédures n'existent pas en C/C++, ce sont des fonctions qui ne retournent pas de valeur).

Tout programme a pour but d'effectuer des opérations sur des données. La structure fondamentale est donc la suivante :

ENTRÉE DES DONNÉES
(clavier, souris, fichier, autres périphériques)
|
TRAITEMENT DES DONNÉES
|
SORTIE DES DONNÉES
(écran, imprimante, fichier, autres périphériques)

Ces diverses étapes peuvent être dispersées dans le programme. Par exemple, les entrées peuvent se trouver dans le programme même (l'utilisateur n'a dans ce cas pas besoin de les saisir). Notez que ce processus peut être répété autant de fois que nécessaire pendant l'exécution d'un programme. Par exemple, les programmes graphiques traitent les événements système et graphiques au fur et à mesure qu'ils apparaissent. Les données qu'ils reçoivent sont fournies par le système sont couramment appelés des *messages*, et la boucle de traitement de ces données la *boucle des messages*.

Les données sont stockées dans des *variables*, c'est à dire des zones de la mémoire. Comme leur nom l'indique, les variables peuvent être modifiées (par le traitement des données). Des opérations peuvent être effectuées sur les variables, mais pas n'importe lesquelles. Par exemple, on ne peut pas ajouter des pommes à et des bananes, sauf à définir cette opération bien précisément. Les opérations dépendent donc de la nature des variables. Afin de réduire les risques d'erreurs de programmation, les langages comme le C/C++ donnent un *type* à chaque variable (par exemple : *pomme* et *banane*). Lors de la *compilation* (phase de traduction du texte *source* du programme en *exécutable*), ces types sont utilisés pour vérifier si les opérations effectuées sont autorisées. Le programmeur peut évidemment définir ses propres types.

Le langage fournit des types de base et des opérations prédéfinies sur ces types. Les opérations qui peuvent être faites sont soit l'application d'un *opérateur*, soit l'application d'une fonction sur les variables. Logiquement parlant, il n'y a pas de différence. Seule la syntaxe change :

```
a=2+3
```

est donc strictement équivalent à :

```
a=ajoute(2,3)
```

Évidemment, des fonctions utilisateur peuvent être définies. Les opérateurs ne peuvent être que redéfinis : il est impossible d'en définir de nouveaux (de plus, la redéfinition des opérateurs n'est faisable qu'en C++).

Cette première partie est donc consacrée à la définition des types, la déclaration des variables, la construction et l'appel de fonctions, et aux entrées-sorties de base (clavier et écran).

2.1. Les commentaires en C++

Les commentaires sont nécessaires et très simples à faire. Tout programme doit être commenté. Attention cependant, trop de commentaires tue le commentaire, parce que les choses importantes sont noyées dans les banalités.

Il existe deux types de commentaires en C++ : les commentaires de type C et les commentaires de fin de ligne (qui ne sont disponibles qu'en C++).

Les commentaires C commencent avec la séquence barre oblique - étoile. Les commentaires se terminent avec la séquence inverse : une étoile suivie d'une barre oblique.

Exemple 2-1. Commentaire C

```
/*   Ceci est un commentaire C   */
```

Ces commentaires peuvent s'étendre sur plusieurs lignes.

En revanche, les commentaires de fin de lignes s'arrêtent à la fin de la ligne courante, et pas avant. Ils permettent de commenter plus facilement les actions effectuées sur la ligne courante, avant le commentaire. Les commentaires de fin de ligne commencent par la séquence constituée de deux barres obliques (ils n'ont pas de séquence de terminaison, puisqu'ils ne se terminent qu'à la fin de la ligne courante). Par exemple :

Exemple 2-2. Commentaire C++

```
action quelconque     // Ceci est un commentaire C++
action suivante
```

2.2. Les types prédéfinis du C/C++

Il y a plusieurs types prédéfinis. Ce sont :

- le type vide : void. Ce type est utilisé pour spécifier le fait qu'il n'y a pas de type. Ceci a une utilité pour faire des procédures (fonctions ne renvoyant rien) et les pointeurs sur des données non typées (voir plus loin) ;
- les booléens : bool, qui peuvent prendre les valeurs true et false (en C++ uniquement, ils n'existent pas en C) ;
- les caractères : char ;
- les caractères longs : wchar_t (en C++ seulement, ils n'existent pas en C) ;
- les entiers : int ;
- les réels : float ;
- les réels en double précision : double ;
- les tableaux à une dimension, dont les indices sont spécifiés par des crochets ('[' et ']'). Pour les tableaux de dimension supérieure ou égale à 2, on utilisera des tableaux de tableaux ;
- les structures, unions et énumérations (voir plus loin).

Les types entiers (int) peuvent être caractérisés d'un des mots-clés long ou short. Ces mots-clés permettent de modifier la taille du type, c'est à dire la plage de valeurs qu'ils peuvent couvrir. De même, les réels en double précision peuvent être qualifiés du mot-clé long, ce qui augmente leur plage de valeurs. On ne peut pas utiliser le mot-clé short avec les double. On dispose donc de types additionnels :

- les entiers longs : long int, ou long (int est facultatif) ;
- les entiers courts : short int, ou short ;
- les réels en quadruple précision : long double.

La taille des types n'est spécifiée dans aucune norme, sauf pour le type char. En revanche, les inégalités suivantes sont toujours vérifiées :

char ☐ short int ☐ int ☐ long int float ☐ double ☐ long double

où l'opérateur " <= " signifie ici " a une plage de valeur plus petite ou égale que ". La taille des caractères de type char est toujours de un octet.

Les types char et int peuvent être signés ou non. Un nombre signé peut être négatif, pas un nombre non signé. Lorsqu'un nombre est signé, la valeur absolue du plus grand nombre représentable est plus petite. Par défaut, un nombre est signé (sauf les type char et wchar_t, qui peuvent être soit signés, soit non signés, selon le compilateur utilisé). Pour préciser qu'un nombre n'est pas signé, il faut utiliser le mot-clé unsigned. Pour préciser qu'un nombre est signé, on peut utiliser le mot-clé signed. Ces mots-clés peuvent être intervertis librement avec les mots-clés long et short.

Exemple 2-3. Types signés et non signés

```
unsigned char
signed char
unsigned int
signed int
unsigned long int
long unsigned int
```

Les valeurs accessibles avec les nombres signés ne sont pas les mêmes que celles accessibles avec les nombres non signés. En effet, un bit est utilisé pour le signe dans les nombres signés. Par exemple, puisque le type char est codé sur 8 bits, on peut coder les nombres allant de 0 à 256 avec ce type en non signé (il y a 8 chiffres binaires, chacun peut valoir 0 ou 1, on a donc 2 puissance 8 combinaisons possibles, ce qui fait 256). En signé, les valeurs s'étendent de -128 à 127 (un des chiffres binaires est utilisé pour le signe, il en reste 7 pour coder le nombre, donc il reste 128 possibilités dans les positifs comme dans les négatifs. 0 est considéré comme positif. En tout, il y a autant de possibilités.).

Le type int doit être capable de représenter les entiers utilisés par la machine sur laquelle le programme tournera. Par exemple, sur les machines 16 bits ils sont codés sur 16 bits (les valeurs accessibles vont donc de -32768 à 32768, ou de 0 à 65535 si l'entier n'est pas signé). C'est le cas sur les PC en mode réel (c'est à dire sous DOS) et sous Windows 3.x. Sur les machines fonctionnant en 32 bits, le type int est stocké sur 32 bits : l'espace des valeurs disponibles est donc 65536 fois plus large. C'est le cas sur les PC en mode protégé 32 bits (Windows 95 ou NT, DOS Extender, Linux), sur la plupart des machines UNIX et sur les Macintosh. Sur les machines 64 bits, le type int est 64 bits (DEC Alpha par exemple). On constate donc que la portabilité des types de base est très aléatoire. En pratique cependant, ils ont souvent la même taille pour toutes les machines 32 bits (la majorité). Sur ces machines, les entiers longs sont codés la plupart du temps sur 32 bits et les entiers courts sur 16 bits. Les caractères sont souvent codés sur 8 bits. Le type wchar_t est équivalent à l'un des types entiers, il est souvent codé sur 16 bits. Enfin, le type float est généralement codé sur 4 octets et les types double et long double sont identiques et codés sur 8 octets.

Le C++ (et le C++ uniquement) considère le type char comme le type de base des caractères. Les caractères n'ont pas de notion de signe associée. Cependant, les caractères peuvent être considérés comme des entiers à tout instant, mais il n'est pas précisé si ce type est signé ou non. Ceci dépend du compilateur. L'interprétation du type char en tant que type intégral n'est pas le comportement de base du C++, par conséquent, le langage distingue les versions signées et non signées de ce type de la version dont le signe n'est pas spécifié. Ceci signifie que le compilateur traite les types char, unsigned char et signed char comme des types différents. Cette distinction n'a pas lieu d'être au niveau des plages de valeurs si l'on connaît le signe du type char, mais elle est très importante dans la détermination de la *signature* des fonctions (la signature des fonctions sera définie plus loin dans ce cours).

Si l'on veut faire du code portable (c'est à dire qui compilera et fonctionnera sans modification du programme sur tous les ordinateurs), il faut utiliser des types de données qui donneront les mêmes intervalles de valeurs sur tous les ordinateurs. Il est donc recommandé de définir ses propres types (par exemple int8, int16, int32) dont la taille et le signe seront fixe. Lorsque le programme devra être porté, seule la définition de ces types sera à changer, pas le programme. En pratique, si l'on veut faire du code portable entre les machines 16 bits et les machines 32 bits, on ne devra pas utiliser le type int seul : il faudra toujours indiquer la taille de l'entier utilisé : short (16 bits)

ou long (32 bits).

2.3. Notation des nombres

Les entiers se notent de la manière suivante :

- base 10 (décimale) : avec les chiffres de '0' à '9', et les signes '+' (facultatif) et '-'.

  Exemple 2-4. Notation des entiers en base 10

  12354, -2564

- base 16 (hexadécimale) : avec les chiffres '0' à '9' et 'A' à 'F' ou a à f (A=a=10, B=b=11, … F=f=15). Les entiers notés en hexadécimal devront toujours être précédés de " 0x " (qui indique la base). On ne peut pas utiliser le signe '-' avec les nombres hexadécimaux.

  Exemple 2-5. Notation des entiers en base 16

  0x1AE

- base 8 (octale) : avec les chiffres de '0' à '7'. Les nombres octaux doivent être précédés d'un 0 (qui indique la base). Le signe '-' ne peut pas être utilisé.

  Exemple 2-6. Notation des entiers en base 8

  01, 0154

Les flottants (pseudo réels) se notent de la manière suivante :

[signe] chiffres [.[chiffres]][e|E [signe] exposant]

où signe indique le signe. On emploie les signes '+' (facultatif) et '-' aussi bien pour la mantisse que pour l'exposant. 'e' ou 'E' permet de donner l'exposant du nombre flottant. L'exposant est facultatif. Si on ne donne pas d'exposant, on doit donner des chiffres derrière la virgule avec un point et ces chiffres.

Les chiffres après la virgule sont facultatifs, mais pas le point. Si on ne met ni le point, ni la mantisse, le nombre est un entier décimal.

Exemple 2-7. Notation des réels

-123.56, 12e-12, 2

" 2 " est entier, " 2. " est réel.

Les caractères se notent entre guillemets simples :

'A', 'c', '('

On peut donner un caractère non accessible au clavier en donnant son code en octal, précédé du caractère '\'. Par exemple, le caractère 'A' peut aussi être noté '\0101'. Attention à ne pas oublier le 0 du nombre octal. Il est aussi possible d'utiliser certains caractères spéciaux, dont les principaux sont :

'\a'	Bip sonore
'\b'	Backspace
'\f'	Début de page suivante
'\r'	Retour à la ligne (sans saut de ligne)
'\n'	Passage à la ligne
'\t'	Tabulation
'\v'	Tabulation verticale

D'autres séquences d'échappement sont disponibles, afin de pouvoir représenter les caractères ayant une signification particulière en C :

'\\' Le caractère \
'\"' Le caractère '

Note: Attention ! Il n'y a pas de chaînes de caractères. Les chaînes de caractères sont en fait des tableaux de caractères. Cependant, on pourra créer des tableaux de caractères constants en donnant la chaîne entre doubles guillemets :

"Exemple de chaîne de caractère..."

Les caractères spéciaux peuvent être utilisés directement dans les chaînes de caractères :

"Ceci est un saut de ligne :\nCeci est à la ligne suivante."

Si une chaîne de caractère est trop longue pour tenir sur une seule ligne, on peut concaténer plusieurs chaînes en les juxtaposant :

"Ceci est la première chaîne " "ceci est la deuxième."

produit la chaîne de caractères complète suivante :

"Ceci est la première chaîne ceci est la deuxième."

Note: Attention : il ne faut pas mettre de caractère nul dans une chaîne de caractères. Ce caractère est en effet le caractère de terminaison de toute chaîne de caractères.

Vous trouverez plus loin pour de plus amples informations sur les chaînes de caractères et les tableaux.

Enfin, les versions longues des différents types cités précédemment (wchar_t, long int et long double) peuvent être notées en faisant précéder ou suivre la valeur de la lettre 'L'. Cette lettre doit précéder la valeur dans le cas des caractères et des chaînes de caractères, et la suivre quand il s'agit des entiers et des flottants. Exemple :

```
L"Ceci est une chaîne de wchar_t."
2.3e5L
```

2.4. La déclaration des variables

Les variables simples se déclarent avec la syntaxe suivante :

type identificateur;

où type est le type de la variable et identificateur est son nom. Il est possible de créer et d'initialiser une série de variables dès leur création avec la syntaxe suivante :

type identificateur[=valeur][, identificateur[=valeur][...]];

Exemple 2-8. Déclaration de variables

```
int i=0, j=0;    /* Déclare et initialise deux entiers à 0 */
```

```
double somme;    /* Déclare une variable réelle */
```

Les variables peuvent être déclarées quasiment n'importe où dans le programme. Ceci permet de ne déclarer une variable temporaire que là où l'on en a besoin.

> **Note:** Ceci n'est vrai qu'en C++. En C pur, on est obligé de déclarer les variables au début des fonctions ou des instructions composées (voir plus loin). Il faut donc connaître les variables temporaires nécessaires à l'écriture du morceau de code qui suit leur déclaration.

La déclaration d'un tableau se fait en faisant suivre le nom de l'identificateur d'une paire de crochet, contenant le nombre d'élément du tableau :

```
type identificateur[taille]([taille](…));
```

> **Note:** Attention ! Les caractères '[' et ']' étant utilisés par la syntaxe des tableaux, ils ne signifient plus les éléments facultatifs ici. Ici, et ici seulement, les éléments facultatifs sont donnés entre parenthèses.

Dans la syntaxe précédente, type représente le type des éléments du tableau.

Exemple 2-9. Déclaration d'un tableau

```
int MonTableau[100];
```

MonTableau est un entier de 100 entiers. On référence les éléments des tableaux en donnant l'indice de l'élément entre crochet :

```
MonTableau[3]=0;
```

Les tableaux à plus d'une dimension sont en fait des tableaux de tableaux. On prendra garde au fait que dans la déclaration d'un tableau à plusieurs dimensions, la dernière dimension indiquée est la dimension du tableau dont on fait un tableau. Ainsi, dans l'exemple suivant :

```
int Matrice[5][4];
```

Matrice est un tableau de dimension 5 dont les éléments sont des tableaux de dimension 4. L'ordre de déclaration des dimensions est donc inversé : 5 est la taille de la dernière dimension et 4 est la taille de la première dimension. L'élément suivant :

```
int Matrice[2];
```

est donc le deuxième élément de ce tableau à 5 dimensions, et est lui-même un tableau à 4 dimensions.

On prendra garde au fait qu'en C/C++, les indices des tableaux varient de 0 à taille-1. Il y a donc bien taille éléments dans le tableau. Dans l'exemple donné ci-dessus, l'élément MonTableau[100] n'existe pas : y accéder plantera le programme. C'est au programmeur de vérifier que ses programmes n'utilisent jamais les tableaux avec des indices plus grand que leur taille.

Un autre point auquel il faudra faire attention est la taille des tableaux à utiliser pour les chaînes de caractères. Une chaîne de caractères se termine obligatoirement par le caractère nul ('\0'), il faut donc réserver de la place pour lui. Par exemple, pour créer une chaîne de caractères de 100 caractères au plus, il faut un tableau pour 101 caractères (déclaré avec " char chaine[101]; ").

2.5. Les instructions

Les instructions sont identifiées par le point virgule. C'est ce caractère qui marque la fin d'une instruction.

Exemple 2-10. Instruction vide

```
;    /* Instruction vide : ne fait rien ! */
```

Les opérations possibles à l'intérieur d'une instruction sont les suivantes :

- affectation :

variable = valeur

Note: L'affectation n'est pas une instruction. C'est une *opération* qui *renvoie la valeur affectée*. On peut donc effectuer des affectations multiples.

Exemple 2-11. Affectation multiple

```
i=j=k=m=0;   /* Annule les variables i, j, k et m. */
```

- autres opérations :

  valeur op valeur

  où op est l'une des opérations suivantes : +, -, *, /, %, &, |, ^, ~, <<, >>.

  Note: '/' représente la division euclidienne pour les entiers, et la division classique pour les flottants.

  '%' représente la congruence (modulo). '|' et '&' représentent respectivement le *ou* et le *et* binaire (c'est à dire bit à bit : 1 et 1 = 1, 0 et x = 0, 1 ou x = 1 et 0 ou 0 = 0). '^' représente le *ou exclusif* (1 xor 1 = 0 xor 0 = 0 et 1 xor 0 = 1). '~' représente la négation binaire (1 <-> 0). '<<' et '>>' effectuent un décalage binaire vers la gauche et la droite respectivement, d'un nombre de bits égal à la valeur du second opérande.

- affections composées. Ces opérations permettent de réaliser une opération normale et une affectation en une seule étape :

  variable op_aff valeur

  avec op_aff l'un des opérateurs suivants : '+=', '-=', '*=', etc...

  Cette syntaxe est strictement équivalente à :

  variable = variable op valeur

Exemple 2-12. Affectation composée

```
i*=2;        /* Multiplie i par 2 : i = i * 2. */
```

Il est possible de créer un bloc d'instructions, en entourant les instructions de ce bloc avec des accolades. Un bloc d'instructions est considéré comme une instruction unique. Il est inutile de mettre un point virgule pour marquer l'instruction, puisque le bloc lui-même est une instruction.

Exemple 2-13. Instruction composée

```
{
  i=1;
  j=i+3*g;
}
```

2.6. Les fonctions

Le C++ ne permet de faire que des fonctions, pas de procédures. Une procédure peut être faite en utilisant une fonction ne renvoyant pas de valeur, ou en ignorant la valeur retournée.

2.6.1. Définition des fonctions

La définition des fonctions se fait comme suit :

```
type identificateur(paramètres)
{
...   /* Instructions de la fonction. */
}
```

type est le type de la valeur renvoyée, identificateur est le nom de la fonction, et paramètres est une liste de paramètres. La syntaxe de la liste de paramètres est la suivante :

```
type variable [= valeur] [, type variable [= valeur] [...]]
```

où type est le type du paramètre *variable* qui le suit et valeur sa valeur par défaut. La valeur par défaut d'un paramètre est la valeur que ce paramètre prend lors de l'appel de la fonction si aucune autre valeur n'est fournie.

> **Note:** L'initialisation des paramètres de fonctions n'est possible qu'en C++, le C n'accepte pas cette syntaxe.

La valeur de la fonction à renvoyer est spécifiée en utilisant la commande return, dont la syntaxe est :

```
return valeur;
```

Exemple 2-14. Définition de fonction

```
int somme(int i, int j)
{
  return i+j;
}
```

Si une fonction ne renvoie pas de valeur, on lui donnera le type void. Si elle n'attend pas de paramètres, sa liste de paramètres sera void ou n'existera pas. Il n'est pas nécessaire de mettre une instruction return à la fin d'une fonction qui ne renvoie pas de valeur.

Exemple 2-15. Définition de procédure

```
void rien()     /* Fonction n'attendant pas de paramètres */
{              /* et ne renvoyant pas de valeur. */
  return;      /* Cette ligne est facultative. */
}
```

2.6.2. Appel des fonctions

L'appel d'une fonction se fait en donnant son nom, puis les valeurs de ses paramètres entre parenthèses. Attention ! S'il n'y a pas de paramètres, il faut quand même mettre les parenthèses, sinon la fonction n'est pas appelée.

Exemple 2-16. Appel de fonction

```
int i=somme(2,3);
rien();
```

Si la déclaration comprend des valeurs par défaut pour des paramètres (C++ seulement), ces valeurs sont utilisées lorsque ces paramètres ne sont pas fournis lors de l'appel. Si un paramètre est manquant, alors tous les paramètres qui le suivent doivent être eux aussi manquants. Il en résulte que seuls les derniers paramètres d'une fonction peuvent avoir des valeurs par défaut. Par exemple :

```
int test(int i = 0, int j = 2)
{
  return i/j;
}
```

L'appel de la fonction test(8) est valide. Comme on ne précise pas le dernier paramètre, *j* est initialisé à 2. Le résultat obtenu est donc 4. De même, l'appel test() est valide : dans ce cas *i* vaut 0 et *j* vaut 2. En revanche, il est impossible d'appeler la fonction test en ne précisant que la valeur de *j*. Enfin, l'expression " int test(int i=0, int j) {...} " serait invalide, car si on ne passait pas deux paramètres, *j* ne serait pas initialisé.

2.6.3. Déclaration des fonctions

Toute fonction doit être *déclarée* avant d'être appelée pour la première fois. La *définition* d'une fonction peut faire office de *déclaration*.

Il peut se trouver des situations où une fonction doit être appelée dans une autre fonction définie avant elle. Comme cette fonction n'est pas définie au moment de l'appel, elle doit être déclarée. De même, il est courant d'avoir à appeler une fonction définie dans un autre fichier que le fichier d'où se fait l'appel. Encore une fois, il est nécessaire de déclarer ces fonctions.

Le rôle des déclarations est donc de signaler l'existence des fonctions aux compilateurs afin de les utiliser, tout en reportant leur définition de ces fonctions plus loin ou dans un autre fichier.

La syntaxe de la déclaration d'une fonction est la suivante :

type identificateur(paramètres);

où type est le type de la valeur renvoyée par la fonction, identificateur est son nom et paramètres la liste des types des paramètres que la fonction admet, séparés par des virgules.

Exemple 2-17. Déclaration de fonction

```
int Min(int, int);        /* Déclaration de la fonction minimum */
                          /* définie plus loin. */
/* Fonction principale. */
int main(void)
{
  int i = Min(2,3);       /* Appel à la fonction Min, déjà
                             déclarée. */
  return 0;
}

/* Définition de la fonction min. */
int Min(int i, int j)
{
  if (i<j) return i;
  else return j;
}
```

En C++, il est possible de donner des valeurs par défaut aux paramètres dans une déclaration, et ces valeurs peuvent être différentes de celles que l'on peut trouver dans une autre déclaration. Dans ce cas, les valeurs par défaut utilisées sont celles de la déclaration visible lors de l'appel de la fonction.

2.6.4. Surcharge des fonctions

Il est interdit en C de définir plusieurs fonctions qui portent le même nom. En C++, cette interdiction est levée, moyennant quelques précautions. Le compilateur peut différencier deux fonctions en regardant le type des paramètres qu'elle reçoit. La liste de ces types s'appelle la *signature* de la fonction. En revanche, le type du résultat de la fonction ne permet pas de l'identifier, car le résultat peut être converti en une valeur d'un autre type avant d'être utilisé après l'appel de cette fonction.

Il est donc possible de faire des fonctions de même nom (on dit que ce sont des fonctions *surchargées*) si et seulement si toutes les fonctions portant ce nom peuvent être distinguées par leurs signatures. La fonction qui sera appelée sera choisie parmi les fonctions de même nom, et ce sera celle dont la signature est la plus proche des valeurs passées en paramètre lors de l'appel.

Exemple 2-18. Surcharge de fonctions

```
float test(int i, int j)
{
  return (float) i+j;
}

float test(float i, float j)
{
  return i*j;
}
```

Ces deux fonctions portent le même nom, et le compilateur les acceptera toutes les deux. Lors de l'appel de test(2,3), ce sera la première qui sera appelée, car 2 et 3 sont des entiers. Lors de l'appel de test(2.5,3.2), ce sera la deuxième, parce que 2.5 et 3.2 sont réels. Attention ! Dans un appel tel que test(2.5,3), le flottant 2.5 sera converti en entier et la première fonction sera appelée. Il convient donc de faire très attention aux mécanismes de surcharges du langage, et de vérifier les règles de priorité utilisées par le compilateur.

On veillera à ne pas utiliser des fonctions surchargées dont les paramètres ont des valeurs par défaut, car le compilateur ne pourrait pas faire la distinction entre ces fonctions. D'une manière générale, le compilateur dispose d'un ensemble de règles (dont la présentation dépasse le cadre de ce cours) qui lui permettent de déterminer la meilleure fonction étant donné un jeu de paramètres. Si, lors de la recherche de la fonction à utiliser, le compilateur trouve des ambiguïtés, il générera une erreur.

2.6.5. Fonctions inline

Le C++ dispose du mot-clé inline, qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration de la fonction, il donne l'autorisation au compilateur de ne pas instancier cette fonction. Cela signifie qu'il a le droit de remplacer l'appel d'une fonction par le code correspondant. Si la fonction est grosse ou si elle est appelée souvent, le programme devient plus gros, puisque la fonction est réécrite à chaque fois qu'elle est appelée. En revanche, il devient nettement plus rapide, puisque les mécanismes d'appel de fonctions, de passage des paramètres et de la valeur de retour sont ainsi évités. De plus, le compilateur peut effectuer des optimisations additionnelles qu'il n'aurait pas pu faire si la fonction n'était pas inlinée. En pratique, on réservera cette technique pour les petites fonctions appelées dans du code devant être rapide (à l'intérieur des boucles par exemple), ou pour les fonctions permettant de lire des valeurs dans des variables.

Cependant, il faut se méfier. Le mot-clé inline donne l'autorisation au compilateur de faire des fonctions inline. Il n'y est pas obligé. La fonction peut donc très bien être implémentée classiquement. Pire, elle peut être implémentée des deux manières, selon les mécanismes d'optimisation du compilateur.

De plus, il faut connaître les restrictions des fonctions inline :

- elles ne peuvent pas être récursives ;
- elles ne sont pas instanciées, donc on ne peut pas faire de pointeur sur une fonction inline.

Si l'une de ces deux conditions n'est pas vérifiée pour une fonction, le compilateur l'implémentera classiquement (elle ne sera donc pas inline).

Enfin, du fait que les fonctions inline sont insérées telles quelles aux endroits où elles sont appelées, il est nécessaires qu'elles soient complètement définies avant leur appel. Ceci signifie que, contrairement aux fonctions classiques, il n'est pas possible de se contenter de les déclarer pour les appeler, et de fournir leur définition dans un fichier séparé. Dans ce cas en effet, le compilateur générerait des références externes sur ces fonctions, et n'insérerait pas leur code. Ces références ne seraient pas résolues à l'édition de lien, car il ne génére également pas les fonctions inline, puisqu'elles sont supposées être insérées sur place lorsqu'on les utilise. Les notions de compilation dans des fichiers séparés et d'édition de liens seront présentées en détail dans le Chapitre 7.

Exemple 2-19. Fonction inline

```
inline int Max(int i, int j)
{
  if (i>j)
    return i;
  else
    return j;
```

```
}
```

Pour ce type de fonction, il est tout à fait justifié d'utiliser le mot-clé inline.

2.6.6. Fonctions statiques

Par défaut, lorsqu'une fonction est définie dans un fichier C/C++, elle peut être utilisée dans tout autre fichier pourvu qu'elle soit déclarée avant son utilisation. Dans ce cas, la fonction est dite *externe*. Il peut cependant être intéressant de définir des fonctions locales à un fichier, soit afin de résoudre des conflits de noms (entre deux fonctions de même nom et de même signature mais dans deux fichiers différents), soit parce que la fonction est uniquement d'intérêt local. Le C et le C++ fournissent donc le mot-clé static, qui, une fois placé devant la définition et les éventuelles déclarations d'une fonction, la rend unique et utilisable uniquement dans ce fichier. À part ce détail, les fonctions statiques s'utilisent exactement comme des fonctions classiques.

Exemple 2-20. Fonction statique

```
// Déclaration de fonction statique :
static int locale1(void);

/* Définition de fonction statique : */
static int locale2(int i, float j)
{
  return i*i+j;
}
```

2.6.7. Fonctions prenant un nombre de paramètres variable

En général, les fonctions ont un nombre constant de paramètres. Pour les fonctions qui ont des paramètres par défaut en C++, le nombre de paramètres peut apparaître variable à l'appel de la fonction, mais en réalité, la fonction utilise toujours le même nombre de paramètres.

Cependant, le C et le C++ disposent d'un mécanisme qui permet au programmeur de réaliser des fonctions dont le nombre et le type des paramètres est variable. Nous verrons plus loin que les fonctions d'entrée - sortie du C sont des fonctions dont la liste des arguments n'est pas fixée, ceci afin de pouvoir réaliser un nombre d'entrées - sorties arbitraire, et ce sur n'importe quel type prédéfini.

En général, les fonctions dont la liste des paramètre est arbitrairement longue disposent d'un critère pour savoir quel est le dernier paramètre. Ce critère peut être le nombre de paramètres, qui peut être fourni en premier paramètre à la fonction, ou une valeur de paramètre particulière qui détermine la fin de la liste par exemple. On peut aussi définir les paramètres qui suivent le premier paramètre à l'aide d'une chaîne de caractère.

Pour indiquer au compilateur qu'une fonction peut accepter une liste de paramètres variable, il faut simplement utiliser des points de suspensions dans la liste des paramètres :

```
type identificateur(paramètres, ...)
```

dans les déclarations et la définition de la fonction. Dans tous les cas, il est nécessaire que la fonction ait au moins un paramètre classique. Ces paramètres doivent impérativement être avant les points de suspensions.

La difficulté apparaît en fait dans la manière de récupérer les paramètres de la liste de paramètres dans la définition de la fonction. Les mécanismes de passage des paramètres étant très dépendants de la machine (et du compilateur), un jeu de macros a été défini dans le fichier d'en-tête stdarg.h pour faciliter l'accès aux paramètres de la liste. Pour en savoir plus sur les macros et les fichiers d'en-tête, consulter le Chapitre 6. Pour l'instant, sachez seulement qu'il faut ajouter la ligne suivante :

```
#include <stdarg.h>
```

au début de votre programme. Ceci permet d'utiliser le type va_list et les expressions va_start, va_arg et va_end pour récupérer les arguments de la liste de paramètres variable, un à un.

Le principe est simple. Dans la fonction, vous devez déclarer une variable de type va_list. Puis, vous devez initialiser cette variable avec la syntaxe suivante :

va_start(variable, paramètre);

où variable est le nom de la variable de type va_list que vous venez de créer, et *paramètre* est le dernier paramètre classique de la fonction. Dès que variable est initialisée, vous pouvez récupérer un à un les paramètres à l'aide de l'expressions suivantes :

va_arg(variable, type)

qui renvoie le paramètre en cours avec le type type et met à jour variable pour passer au paramètre suivant. Vous pouvez utiliser cette expression autant de fois que vous le désirez, elle retourne à chaque fois un nouveau paramètre. Lorsque le nombre de paramètres correct a été récupéré, vous devez détruire la variable variable à l'aide de la syntaxe suivante :

va_end(variable);

Il est possible de recommencer les étapes suivantes autant de fois que l'on veut, la seule chose qui compte est de bien faire l'initialisation avec va_start et de bien terminer la procédure avec va_end à chaque fois.

> **Note:** Il existe une restriction sur les types des paramètres des listes variables d'arguments. Lors de l'appel des fonctions, un certain nombre de traitements sur les paramètres a lieu. En particulier, des *promotions* implicites ont lieu, ce qui se traduit par le fait que les paramètres réellement passés aux fonctions ne sont pas du type déclaré. Le compilateur continue de faire les vérifications de type, mais en interne, un type plus grand peut être utilisé pour passer les valeurs des paramètres. En particulier, les types char et short ne sont pas utilisés : les paramètres sont toujours promus aux type int ou long int. Ceci implique que les seuls types que vous pouvez utiliser sont les types cibles des promotions et les types qui ne sont pas sujets aux promotions (pointeurs, structures et unions). Les types cibles dans les promotions sont déterminés comme suit :
>
> - les types char, signed char, unsigned char, short int ou unsigned short int sont promus en int si ce type est capable d'accepter toutes leurs valeurs. Si int est insuffisant, unsigned int est utilisé ;
> - les types des énumérations (voir plus loin pour la définition des énumérations) et wchar_t sont promus en int, unsigned int, long ou unsigned long selon leurs capacités. Le premier type capable de conserver la plage de valeur du type à promouvoir est utilisé ;
> - les valeurs des champs de bits sont converties en int ou unsigned int selon la taille du champ de bit (voir plus loin pour la définition des champs de bits) ;
> - les valeurs de type float sont converties en double.

Exemple 2-21. Fonction à nombre de paramètres variable

```
#include <stdarg.h>

/* Fonction effectuant la somme de "compte" paramètres : */
double somme(int compte, ...)
{
  double resultat=0;      /* Variable stockant la somme. */
  va_list varg;           /* Variable identifiant le prochain
                          paramètre. */
  va_start(varg, compte); /* Initialisation de la liste. */
  do                      /* Parcours de la liste. */
  {
    resultat=resultat+va_arg(varg, double);
    compte=compte-1;
  } while (compte!=0);
  va_end(varg);           /* Terminaison. */
  return resultat;
}
```

La fonction somme effectue la somme de compte flottants (float ou double) et la renvoie dans un double. Pour plus de détails sur la structure de contrôle do ... while, voir Section 3.1.4.

2.7. La fonction main

Lorsqu'un programme est chargé, son exécution commence par l'appel d'une fonction spéciale du programme. Cette fonction doit impérativement s'appeler " main " (*principal* en anglais) pour que le compilateur puisse savoir que c'est cette fonction qui marque le début du programme. La fonction main est appelée par le système d'exploitation, elle ne peut pas être appelée par le programme, c'est à dire qu'elle ne peut pas être récursive.

Exemple 2-22. Programme minimal

```
int main()   /* Plus petit programme C/C++. */
{
  return 0;
}
```

La fonction main doit renvoyer un code d'erreur d'exécution du programme, le type de ce code est int. Elle peut aussi recevoir des paramètres du système d'exploitation. Ceci sera expliqué plus loin. Pour l'instant, on se contentera d'une fonction main ne prenant pas de paramètres.

> **Note:** Il est spécifié dans la norme du C++ que la fonction main ne doit pas renvoyer le type void. En pratique cependant, tous les compilateurs l'acceptent aussi.
>
> La valeur 0 retournée par la fonction main indique que tout s'est déroulé correctement. En réalité, la valeur du code de retour peut être interprétée différemment selon le système d'exploitation utilisé. La librairie C définit donc les constantes EXIT_SUCCESS et EXIT_FAILURE, qui permettent de supprimer l'hypothèse sur la valeur à utiliser respectivement en cas de succès et en cas d'erreur.

2.8. Les fonctions d'entrée-sortie de base

Les deux fonctions de base d'entrée-sortie du C sont printf et scanf. printf (" print formatted " en anglais) permet d'afficher des données à l'écran. scanf (" scan formatted ") permet de les lire à partir du clavier. Elles attendent toutes les deux une chaîne de caractères en premier paramètre. Cette chaîne est appelée *chaîne de format*, et elle permet de spécifier le type, la position et le format (précision, etc…) des données à traiter.

2.8.1. La fonction printf

Elle s'emploie comme suit : d

printf(chaîne de format [, valeur [, valeur […]]])

Elle renvoie le nombre de caractères affichés.

On peut passer autant de valeurs que l'on veut, pour peu qu'elles soient toutes référencées dans la chaîne de format.

La chaîne de format peut contenir du texte, mais surtout elle doit contenir autant de *formateurs* que de variables à afficher. Si ce n'est pas le cas, le programme plantera. Les formateurs sont placés dans le texte là où les valeurs des variables doivent être affichées.

La syntaxe des formateurs est la suivante :

%[[indicateur]…][largeur][.précision][taille] type

Un formateur commence donc toujours par le caractère %. Pour afficher ce caractère sans faire un formateur, il faut le dédoubler (%%).

Le *type* de la variable à afficher est obligatoire lui aussi. Les types utilisables sont les suivants :

Tableau 2-1. Types pour les chaînes de format de printf

	Type de données à afficher	**Caractère de formatage**
Numériques	Entier décimal signé	d
	Entier décimal non signé	u ou i
	Entier octal non signé	o
	Entier héxadécimal non signé	x (avec les caractères 'a' à 'f') ou X (avec les caractères 'A' à 'F')
	Flottants	f, e, g, E ou G
Caractères	Caractère isolé	c
	Chaîne de caractères	s
Pointeurs	Pointeur	p

> **Note:** Voir le Chapitre 4 pour plus de détails sur les pointeurs. Le format des pointeurs dépend de la machine.
>
> Les valeurs flottantes infinies sont remplacées par les mentions +INF et -INF. Un non-nombre IEEE (Not-A-Number) donne +NAN ou -NAN.

Les autres paramètres sont facultatifs.

Les valeurs disponibles pour le paramètre de *taille* sont les caractères suivants :

Tableau 2-2. Options pour les types des chaînes de format

Option	**Type utilisable**	**Taille du type**
F	Pointeur	Pointeur FAR (DOS uniquement)
N	Pointeur	Pointeur NEAR (DOS uniquement)
h	Entier	short int
l	Entier ou flottant	long int ou double
L	Flottant	long double

Exemple 2-23. Utilisation de printf

```
#include <stdio.h>  /* Ne pas chercher à comprendre cette ligne
                       Elle est nécessaire pour utiliser les
                       fonctions printf et scanf.              */
int main(void)
{
  int i = 2;
  printf("Voici la valeur de i : %d.", i);
  return 0;
}
```

Les paramètres *indicateurs*, *largeur* et *précisions* sont moins utilisés. Il peut y avoir plusieurs paramètres indicateurs, ils permettent de modifier l'apparence de la sortie. Les principales options sont :

- '-' : justification à gauche de la sortie, avec remplissage à droite par des 0 ou des espaces ;
- '+' : affichage du signe pour les nombres positifs ;
- espace : les nombres positifs commencent tous par un espace.

Le paramètre *largeur* permet de spécifier la largeur minimum du champ de sortie, si la sortie est trop petite, on complète avec des 0 ou des espaces. Enfin, le paramètre *précision* spécifie la précision maximum de la sortie (nombre de chiffres à afficher).

2.8.2. La fonction scanf

La fonction scanf permet de faire une ou plusieurs entrées. Comme la fonction printf, elle attend une chaîne de format en premier paramètre. Il faut ensuite passer les variables devant contenir les entrées dans les paramètres qui suivent. Sa syntaxe est la suivante :

```
scanf(chaîne de format, &variable [, &variable [...]]);
```

Elle renvoie le nombre de variables lues.

Ne cherchez pas à comprendre pour l'instant la signification du symbole & se trouvant devant chacune des variables. Sachez seulement que s'il est oublié, le programme plantera.

La chaîne de format peut contenir des chaînes de caractères. Toutefois, si elle contient autre chose que des formateurs, le texte saisi par l'utilisateur devra correspondre impérativement avec les chaînes de caractères indiquées dans la chaîne de format. scanf cherchera à reconnaître ces chaînes, et arrêtera l'analyse à la première erreur.

La syntaxe des formateurs pour scanf diffère un peu de celle de ceux de printf :

```
%[*][largeur][taille]type
```

Seul le paramètre *largeur* change par rapport à printf. Il permet de spécifier le nombre maximum de caractères à prendre en compte lors de l'analyse du paramètre. Le paramètre '*' est facultatif, il indique seulement de passer la donnée entrée et de ne pas la stocker dans la variable destination. Cette variable doit quand même être présente dans la liste des paramètres de scanf.

2.9. Exemple de programme complet

Pour utiliser les fonctions printf et scanf, il faut mettre au début du programme la ligne suivante :

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

Sans entrer dans les détails, disons simplement que cette ligne permet d'inclure un fichier contenant les déclarations des fonctions printf et scanf. Voir le chapitre sur le préprocesseur pour de plus amples détails.

Le programme suivant est donné à titre d'exemple. Il calcule la moyenne de deux nombres entrés au clavier et l'affiche :

Exemple 2-24. Programme complet simple

```
#include <stdio.h>  /* Autorise l'emploi de printf et de scanf. */
```

```
long double x, y;

int main(void)
{
   printf("Calcul de moyenne\n");   /* Affiche le titre. */
   printf("Entrez le premier nombre : ");
   scanf("%Lf", &x);            /* Entre le premier nombre. */
   printf("\nEntrez le deuxième nombre : ");
   scanf("%Lf", &y);            /* Entre le deuxième nombre. */
   printf("\nLa valeur moyenne de %Lf et de %Lf est %Lf.\n",
         x, y, (x+y)/2);
   return 0;
}
```

Dans cet exemple, les chaînes de format spécifient des flottants (f) en quadruple précision (L).

Chapitre 3. Le C/C++ un peu plus loin

Dans cette partie nous allons aborder d'autres aspects du langage indispensable à la programmation. Il s'agit des structures de contrôle (if, while, goto, etc…), et des types de donnés complexes (array, struct, union). Des précisions seront également données sur les différentes classes de variables utilisables en C/C++.

3.1. Les structures de contrôle

Le C/C++ dispose de toutes les structures de contrôle nécessaire à la programmation. Leur syntaxe est donné ci-dessous.

3.1.1. La structure conditionnelle if

Syntaxe :

```
if (test) opération;
```

test est une expression dont la valeur est booléenne ou entière. Toute valeur non nulle est considérée comme vraie. Si le test est vrai, opération est exécuté. Ce peut être une instruction ou un bloc d'instructions. Une variante permet de spécifier l'action à exécuter en cas de test faux :

```
if (test) opération1;
else opération2;
```

Note: Attention ! Les parenthèses autour de test sont nécessaires !

Les opérateurs de comparaison sont les suivants :

Tableau 3-1. Opérateurs de comparaison

==	égalité
!=	inégalité
<	infériorité
>	supériorité
<=	infériorité ou égalité
>=	supériorité ou égalité

Les opérateurs logiques applicables aux expressions booléennes sont les suivants :

Tableau 3-2. Opérateurs logiques

&&	et logique
||	ou logique

!	négation logique

Il n'y a pas d'opérateur ou exclusif logique.

Exemple 3-1. Test conditionnel if

```
if (a<b && a!=0)
{
  min=a;
  nouveau_min=1;
}
```

3.1.2. La boucle for

Syntaxe :

for (initialisation ; test ; itération) opération;

initialisation est une instruction (ou un bloc d'instructions) exécutée avant le premier parcours de la boucle du for. test est une expression dont la valeur déterminera la fin de la boucle. itération est l'opération à effectuer en fin de boucle, et opération constitue le traitement de la boucle. Chacune de ces parties est facultative.

La séquence d'exécution est la suivante :

```
initialisation
test : saut en fin du for ou suite
       opération
       itération
       retour au test
fin du for.
```

Exemple 3-2. Boucle for

```
somme = 0;
for (i=0; i<=10; i=i+1) somme = somme + i;
```

> **Note:** En C++, il est possible que la partie initialisation déclare une variable. Dans ce cas, la variable déclarée n'est définie qu'à l'intérieur de l'instruction for. Par exemple,
>
> ```
> for (int i=0; i<10; i++);
> ```
>
> est strictement équivalent à :
>
> ```
> {
>   int i;
>   for (i=0; i<10; i++);
> }
> ```
>
> Ceci signifie que l'on ne peut pas utiliser la variable i après l'instruction for, puisqu'elle n'est définie que dans cette instruction. Ceci permet de réaliser des variables muettes, qui ne servent qu'à l'instruction for dans laquelle elles sont définies.
>
> **Note:** Cette règle n'est pas celle utilisée par la plupart des compilateurs C++. La règle qu'ils utilisent spécifie que la variable déclarée dans la partie initialisation de l'instruction for reste déclarée après cette instruction. La différence est subtile, mais importante. Ceci pose assurément des problèmes de compatibilité avec les programmes C++ écrits pour ces compilateurs, puisque dans un cas la variable doit être redéclarée et dans l'autre cas elle ne le doit pas. Il est donc recommandé de ne pas déclarer de variables dans la partie initialisation des instructions for pour assurer une portabilité maximale.

3.1.3. Le while

Syntaxe :

while (test) opération;

opération est effectuée tant que test est vérifié. Comme pour le if, les parenthèses autour du test sont nécessaires. L'ordre d'exécution est :

test
opération

Exemple 3-3. Boucle while

```
somme = i = 0;
while (somme<1000)
{
  somme = somme + 2*i/(5+i);
  i = i+1;
}
```

3.1.4. Le do

Syntaxe :

```
do opération;
while (test);
```

opération est effectuée jusqu'à ce que test ne soit plus vérifié. L'ordre d'exécution est :

opération
test

Exemple 3-4. Boucle do

```
p = i = 1;
do
{
  p = p * i;
  i = i +1;
} while (i!=10);
```

3.1.5. Le branchement conditionnel

Syntaxe :

```
switch (valeur)
{
case cas1:
  [opération;
   [break;]
  ]
case cas2:
  [opération;
   [break;]
  ]
  ⋮
case casN:
  [opération;
```

```
   [break;]
  ]
[default:
  [opération;
   [break;]
  ]
]
}
```

valeur est évalué en premier. Son type doit être entier. Selon le résultat de l'évaluation, l'exécution du programme se poursuit au cas de même valeur. Si aucun des cas ne correspond et si default est présent, l'exécution se poursuit après default. Si en revanche default n'est pas présent, on sort du switch.

L'opération qui suit le case approprié ou default est exécutée. Puis, l'opération *du cas suivant* est exécutée (on ne sort donc pas du switch). Pour forcer la sortie du switch, on doit utiliser le mot-clé break.

Exemple 3-5. Branchement conditionnel switch

```
i= 2;
switch (i)
{
case 1 :
case 2 :   /* Si i=1 ou 2, la ligne suivante sera exécutée. */
  i=2-i;
  break;
case 3 :
  i=0;    /* Cette ligne ne sera jamais exécutée. */
default :
  break;
}
```

Note: Il est interdit d'effectuer une déclaration de variable dans un des case d'un switch.

3.1.6. Le saut

Syntaxe :

goto étiquette;

et

étiquette:

étiquette est le point d'arrivée du saut goto. Elle peut avoir n'importe quel nom d'identificateur, et est toujours suivi de deux points (:).

Il n'est pas possible d'effectuer des sauts en dehors d'une fonction. En revanche, il est possible d'effectuer des sauts en dehors et à l'intérieur des blocs d'instructions sous certaines conditions. Si la destination du saut se trouve après une déclaration, cette déclaration ne doit pas comporter d'initialisations. De plus, ce doit être la déclaration d'un type simple (c'est à dire une déclaration qui ne demande pas l'exécution de code) comme les variables, les structures ou les tableaux. Enfin, si, au cours d'un saut, le contrôle d'exécution sort de la portée d'une variable, celle-ci est détruite.

Note: Ces dernières règles sont particulièrement importantes en C++ si la variable est un objet dont la classe a un constructeur ou un destructeur non trivial. Voir le Chapitre 8 pour plus de détails à ce sujet.

Autre règle spécifique au C++ : il est impossible d'effectuer un saut à l'intérieur d'un bloc de code en exécution protégée try {}. Voir aussi le Chapitre 9 concernant les exceptions.

3.1.7. Les commandes de rupture de séquence

Syntaxe :

continue;

ou

```
break;
```

ou

```
return [valeur];
```

ou

```
goto étiquette;
```

À part le goto, qui a déjà été vu, il y a d'autres commandes de *rupture de séquence* (c'est à dire de changement de la suite des instructions à exécuter).

return permet de quitter immédiatement la fonction en cours. break permet de passer à l'instruction suivant l'instruction while, do, for ou switch la plus imbriquée (celle dans laquelle on se trouve).

continue saute directement à la dernière ligne de l'instruction while, do ou for la plus imbriquée. Cette ligne est l'accolade fermante. C'est à ce niveau que les tests de continuation sont faits pour for et do, ou que le saut au début du while est effectué (suivi immédiatement du test). On reste donc dans la structure dans laquelle on se trouvait au moment de l'exécution de continue, contrairement à ce qui se passe avec le break.

Exemple 3-6. Rupture de séquence par continue

```
/* Calcule la somme des 1000 premiers entiers pairs : */
somme_pairs=0;
for (i=0; i<1000; i=i+1)
{
  if (i%2==1) continue;
  somme_pairs=somme_pairs + i;
}
```

3.2. Retour sur les types

En dehors des types de variables simples, le C/C++ permet de créer des types plus complexes.

3.2.1. Les structures

Les types complexes peuvent se construire à l'aide de *structures*. Pour cela, on utilise le mot-clé struct. Sa syntaxe est la suivante :

```
struct [nom_structure]
{
  type champ;
  [type champ;
  […]]
};
```

Il n'est pas nécessaire de donner un nom à la structure. La structure contient plusieurs autres variables, appelées *champs*. Leur type est donné dans la déclaration de la structure. Ce type peut être n'importe quel autre type, même une structure.

La structure ainsi définie peut alors être utilisée pour définir une variable dont le type est cette structure.

Pour cela, deux possibilités :

- faire suivre la définition de la structure par l'identificateur de la variable ;

Exemple 3-7. Déclaration de variable de type structure

```
struct Client
{
  unsigned char Age;
  unsigned char Taille;
} Jean;
```

ou, plus simplement :

```
struct
{
  unsigned char Age;
  unsigned char Taille;
} Jean;
```

Dans le deuxième exemple, le nom de la structure n'est pas mis.

- déclarer la structure en lui donnant un nom, puis déclarer les variables avec la syntaxe suivante :

  [struct] nom_structure identificateur;

Exemple 3-8. Déclaration de structure

```
struct Client
{
  unsigned char Age;
  unsigned char Taille;
};

struct Client Jean, Philippe;
Client Christophe;   // Valide en C++ mais invalide en C
```

Dans cet exemple, le nom de la structure doit être mis, car on utilise cette structure à la ligne suivante. Pour la déclaration des variables Jean et Philippe de type structure client, le mot-clé struct a été mis. Ceci n'est pas nécessaire en C++, mais l'est en C. Le C++ permet donc de déclarer des variables de type structure exactement comme si le type structure était un type prédéfini du langage. La déclaration de la variable Christophe ci-dessus est invalide en C.

Les éléments d'une structure sont accédés par un point, suivi du nom du champ de la structure à accéder. Par exemple, l'âge de Jean est désigné par Jean.Age.

Note: Le typage du C++ est plus fort que celui du C, parce qu'il considère que deux types ne sont identiques que s'ils ont le même nom. Alors que le C considère que deux types qui ont la même structure sont des types identiques, le C++ les distingue. Ceci peut être un inconvénient, car des programmes qui pouvaient être compilés en C ne le seront pas forcément par un compilateur C++. Considérons l'exemple suivant :

```
int main(void)
{
  struct st1
  {
    int a;
  } variable1 = {2};
  struct
  {
    int a;
  } variable2;     /* variable2 a exactement la même structure
                      que variable1, */
  variable2 = variable1;   /* mais ceci est ILLÉGAL en C++ ! */
  return 0;
}
```

Bien que les deux variables aient exactement la même structure, elles sont de type différents ! En effet, variable1 est de type " st1 ", et variable2 de type " " (la structure qui a permis de la

construire n'a pas de nom). On ne peut donc pas faire l'affectation. Pourtant, ce programme était compilable en C pur…

Note: Il est possible de ne pas donner de nom à une structure lors de sa définition sans pour autant déclarer une variable. De telles structures anonymes ne sont utilisables que dans le cadre d'une structure incluse dans une autre structure :

```
struct struct_principale
{
  struct
  {
    int champ1;
  };
  int champ2;
};
```

Dans ce cas, les champs des structures imbriquées seront accédés comme s'il s'agissait de champs de la structure principale. La seule limitation est que, bien entendu, il n'y ait pas de conflit entre les noms des champs des structures imbriquées et ceux des champs de la structure principale. S'il y a conflit, il faut donner un nom à la structure imbriquée qui pose problème, en en faisant un vrai champ de la structure principale.

3.2.2. Les unions

Les *unions* constituent un autre type de structure. Elles sont déclarées avec le mot-clé union, qui a la même syntaxe que struct. La différence entre les structures et les unions est que les différents champs d'une union occupent le même espace mémoire. On ne peut donc, à tout instant, n'utiliser qu'un des champs de l'union.

Exemple 3-9. Déclaration d'une union

```
union entier_ou_reel
{
  int entier;
  float reel;
};

union entier_ou_reel x;
```

x peut prendre l'aspect soit d'un entier, soit d'un réel. Par exemple :

```
x.entier=2;
```

affecte la valeur 2 à x.entier, ce qui détruit x.reel.

Si, à présent, on fait :

```
x.reel=6.546;
```

la valeur de x.entier est perdue, car le réel 6.546 a été stocké au même emplacement mémoire que l'entier x.entier.

Les unions, contrairement aux structures, sont assez peu utilisées, sauf en programmation système où l'on doit pouvoir interpréter des données de différentes manières selon le contexte. Dans ce cas, on aura avantage à utiliser des unions de structures anonymes et à accéder aux champs des structures, chaque structure permettant de manipuler les données selon une de leur interprétation possible.

Exemple 3-10. Union avec discriminant

```
struct SystemEvent
{
```

```
  int iEventType;   /* Discriminant de l'événement.
                       Permet de choisir comment l'interpréter. */
  union
  {
    struct
    {                  /* Structure permettant d'interpréter */
      int iMouseX;       /* les événements souris. */
      int iMouseY;
    };
    struct
    {                  /* Structure permettant d'interpréter */
      char cCharacter;   /* les événements clavier. */
      int iShiftState;
    };
    // etc…
  };
};
```

3.2.3. Les énumérations

Les *énumérations* sont des types *intégraux* (c'est à dire qu'ils sont basés sur les entiers), pour lesquels chaque valeur dispose d'un nom unique. Leur utilisation permet de définir les constantes entières dans un programme et de les nommer. La syntaxe des énumérations est la suivante :

```
enum enumeration
{
  nom1 [=valeur1]
  [, nom2 [=valeur2]
  […]]
};
```

Dans cette syntaxe, enumeration représente le nom de l'énumération et nom1, nom2, etc… représentent les noms des énumérés. Par défaut, les énumérés reçoivent les valeurs entières 0, 1, etc… sauf si une valeur explicite leur est donnée dans la déclaration de l'énumération. Dès qu'une valeur est donnée, le compteur de valeurs se synchronise avec cette valeur, si bien que l'énuméré suivant prendra la valeur augmentée de 1.

Exemple 3-11. Déclaration d'une énumération

```
enum Nombre
{
  un=1, deux, trois, cinq=5, six, sept
};
```

Dans cet exemple, les énumérés prennent respectivement leur valeurs. Comme quatre n'est pas défini, une resynchronisation a lieu lors de la définition de cinq.

Les énumérations suivent les mêmes règles que les structures et les unions en ce qui concerne la déclaration des variables : on doit répéter le mot-clé enum en C, ce n'est pas nécessaire en C++.

3.2.4. Les champs de bits

Il est possible de définir des *champs de bits* et de donner des noms aux bits de ces champs. Pour cela, on utilisera le mot-clé struct et on donnera le type des groupes de bits, leur nom, et enfin leur étendue :

Exemple 3-12. Déclaration d'un champs de bits

```
struct champ_de_bits
{
  int var1;              /* Définit une variable classique. */
  int bits1a4   : 4;  /* Premier champ : 4 bits. */
  int bits5a10  : 6;  /* Deuxième champ : 6 bits. */
  unsigned int bits11a16 : 6;  /* Dernier champ : 6 bits. */
};
```

La taille d'un champ de bits ne doit pas excéder celle d'un entier. Pour aller au-delà, on créera un deuxième champ de bits. La manière dont les différents groupes de bits sont placés en mémoire dépend du compilateur et n'est pas normalisée.

Les différents bits ou groupes de bits seront tous accessibles comme des variables classiques d'une structure ou d'une union :

```
struct champ_de_bits essai;

int main(void)
{
  essai.bits1a4 = 3;
  /* suite du programme */
  return 0;
}
```

3.2.5. Initialisation des structures et des tableaux

Les tableaux et les structures peuvent être initialisées, tout comme les types classiques peuvent l'être. La valeur servant à l'initialisation est décrite en mettant les valeurs des membres de la structure ou du tableau entre accolades, en les séparant par des virgules :

Exemple 3-13. Initialisation d'une structure

```
/* Définit le type Client : */
struct Client
{
  unsigned char Age;
  unsigned char Taille;
  unsigned int Comptes[10];
};

/* Déclare et initialise la variable John : */
struct Client John={35, 190, {13594, 45796, 0, 0, 0, 0, 0, 0, 0, 0}};
```

La variable John est ici déclarée comme étant de type Client et initialisée comme suit : son âge est de 35, sa taille de 190 et ses deux premiers comptes de 13594 et 45796. Les autres comptes sont nuls.

Il n'est pas nécessaire de respecter l'imbrication du type complexe au niveau des accolades, ni de fournir des valeurs d'initialisations pour les derniers membres d'un type complexe. Les valeurs par défaut qui sont utilisées dans ce cas sont les valeurs nulles du type du champ non initialisé. Ainsi, la déclaration de John aurait pu se faire ainsi :

```
struct Client John={35, 190, 13594, 45796};
```

3.2.6. Les alias de types

Le C/C++ dispose d'un mécanisme de création d'*alias*, ou de synonymes des types complexes. Le mot-clé à utiliser est typedef. Sa syntaxe est la suivante :

```
typedef définition alias;
```

où alias est le nom que doit avoir le synonyme du type et définition est sa définition. Pour les tableaux, la

syntaxe est particulière :

typedef type_tableau type[(taille)]([taille](…));

type_tableau est alors le type des éléments du tableau.

Exemple 3-14. Définition de type simple

```
typedef unsigned int mot;
```

mot est strictement équivalent à unsigned int.

Exemple 3-15. Définition de type tableau

```
typedef int tab[10];
```

tab est le synonyme de " tableau de 10 entiers ".

Exemple 3-16. Définition de type structure

```
typedef struct client
{
  unsigned int Age;
  unsigned int Taille;
} Client;
```

Client représente la structure client. Attention à ne pas confondre le nom de la structure (" struct client ") avec le nom de l'alias (" Client ").

> **Note:** Pour comprendre la syntaxe de typedef, il suffit de raisonner de la manière suivante. Si on dispose d'une expression qui permet de déclarer une variable d'un type donné, alors il suffit de placer le mot-clé typedef devant cette expression pour faire en sorte que l'identificateur de la variable devienne un identificateur de type. Par exemple, si on supprime le mot-clé typedef dans la déclaration du type Client ci-dessus, alors Client devient une variable dont le type est struct client.

Une fois ces définitions d'alias effectuées, on peut les utiliser comme n'importe quel type, puisqu'ils représentent des types :

```
unsigned int i = 2, j;   /* Déclare deux unsigned int */
tab Tableau;   /* Déclare un tableau de 10 entiers */
Client John;   /* Déclare une structure client */

John.Age = 35; /* Initialise la variable John */
John.Taille = 175;
for (j=0; j<10; j = j+1) Tableau[j]=j;  /* Initialise Tableau */
```

3.2.7. Transtypages

Il est parfois utile de changer le type d'une valeur. Considérons l'exemple suivant : la division de 5 par 2 renvoie 2. En effet, 5/2 fait appel à la division euclidienne. Comment faire pour obtenir le résultat avec un nombre réel ? Il faut faire 5./2, car alors 5. est un nombre flottant. Mais que faire quand on se trouve avec des variables entières (i et j par exemple) ? Le compilateur signale une erreur après i dans l'expression i./j ! Il faut changer le type de l'une des deux variables. Cette opération s'appelle le *transtypage*. On la réalise simplement en faisant précéder l'expression à transtyper du type désiré entouré de parenthèses :

(type) expression

Exemple 3-17. Transtypage en C

```
int i=5, j=2;
((float) i)/j
```

Dans cet exemple, i est transtypé en flottant avant la division. On obtient donc 2.5.

3.3. Les classes de stockage

Les variables en C/C++ peuvent être créées de différentes manières. Elles sont classées en différents types de variables, appelés *classes de stockage*.

La classification la plus simple que l'on puisse faire des variables est la classification locale - globale. Les variables *globales* sont déclarées en dehors de tout bloc d'instructions, dans la zone de déclaration globale du programme. Les variables *locales* en revanche sont créées à l'intérieur d'un bloc d'instructions. Les variables locales et globales ont des durées de vie, des portées et des emplacements en mémoire différents.

La *portée* d'une variable est la zone du programme dans laquelle elle est accessible. La portée des variables globales est tout le programme, alors que la portée des variables locales est le bloc d'instructions dans lequel elles ont été créées.

La *durée de vie* d'une variable est le temps pendant lequel elle existe. Les variables globales sont créées au début du programme et détruites à la fin, leur durée de vie est donc celle du programme. En général, les variables locales ont une durée de vie qui va du moment où elles sont déclarées jusqu'à la sortie du bloc d'instructions dans lequel elles ont été déclarées. Cependant, il est possible de faire en sorte que les variables locales survivent à la sortie de ce bloc d'instructions. D'autre part, la portée d'une variable peut commencer avant sa durée de vie si cette variable est déclarée après le début du bloc d'instructions dans lequel elle est déclarée. La durée de vie n'est donc pas égale à la portée d'une variable.

La *classe de stockage* d'une variable permet de spécifier sa *durée de vie* et sa *place en mémoire* (sa portée est toujours le bloc dans lequel la variable est déclarée). Le C/C++ dispose d'un éventail de classes de stockage assez large et permet de spécifier le type de variables que l'on désire utiliser :

- auto : la classe de stockage par défaut. Les variables ont pour portée le bloc d'instructions dans lequel elles ont été crées. Elles ne sont accessibles que dans ce bloc. Leur durée de vie est restreinte à ce bloc. Ce mot-clé est facultatif, la classe de stockage auto étant la classe par défaut ;
- static : cette classe de stockage permet de créer des variables dont la portée est le bloc d'instructions en cours, mais qui, contrairement aux variables auto, ne sont pas détruites lors de la sortie de ce bloc. À chaque fois que l'on rentre dans ce bloc d'instructions, les variables statiques existeront et auront pour valeurs celles qu'elles avaient avant que l'on quitte ce bloc. Leur durée de vie est donc celle du programme, et elles conservent leur valeurs. Un fichier peut être considéré comme un bloc. Ainsi, une variable statique d'un fichier ne peut pas être accédée à partir d'un autre fichier. Ceci est utile en compilation séparée (voir plus loin) ;
- register : cette classe de stockage permet de créer une variable dont l'emplacement se trouve dans un registre du microprocesseur. Il faut bien connaître le langage machine pour correctement utiliser cette classe de variable. En pratique, cette classe est très peu utilisée ;
- volatile : cette classe de variable sert lors de la programmation système. Elle indique qu'une variable peut être modifiée en arrière plan par un autre programme (par exemple par une interruption, par un thread, par un autre processus, par le système d'exploitation ou par un autre processeur dans une machine parallèle). Cela nécessite donc de recharger cette variable à chaque fois qu'on y fait référence dans un registre du processeur, et ce *même si elle se trouve déjà dans un de ces registres* (ce qui peut arriver si on a demandé au compilateur d'optimiser le programme) ;
- const : cette classe est utilisée pour rendre le contenu d'une variable non modifiable. En quelque sorte, la variable devient ainsi une variable en lecture seule. Attention, une telle variable n'est pas forcément une constante : elle peut être modifiée soit par l'intermédiaire d'un autre identificateur, soit par une entité extérieure au programme (comme pour les variables volatile). Quand ce mot-clé est appliqué à une structure, aucun des champs de la structure n'est accessible en écriture. Bien qu'il puisse paraître étrange de vouloir rendre " constante " une " variable ", ce mot-clé a une utilité. En particulier, il permet de faire du code plus sûr ;
- mutable : cette classe de stockage, disponible uniquement en C++, ne sert que pour les membres des

structures. Elle permet de passer outre la constance éventuelle d'une structure pour ce membre. Ainsi, un champ de structure déclaré mutable peut être modifié même si la structure est déclarée const ;

- extern : cette classe est utilisée pour signaler que la variable peut être définie dans un autre fichier. Elle est utilisée dans le cadre de la compilation séparée (voir le Chapitre 7 pour plus de détails).

Pour déclarer une classe de stockage particulière, il suffit de faire précéder la déclaration de la variable par l'un des mots-clés auto, static, register, etc… On n'a le droit de n'utiliser que les classes de stockage non contradictoires. Par exemple, register et extern sont incompatibles, de même que register et volatile, et const et mutable. Par contre, static et const, de même que const et volatile, peuvent être utilisées simultanément.

Exemple 3-18. Déclaration d'une variable locale statique

```
int appels(void)
{
  static int n = 0;
  return n = n+1;
}
```

Cette fonction mémorise le nombre d'appels qui lui ont été faits dans la variable n et renvoie ce nombre.

```
int appels(void)
{
  int n = 0;
  return n =n + 1;
}
```

Cette fonction renverra toujours 1. La variable n est créée, initialisée, incrémentée et détruite à chaque appel. Elle ne survit pas à la fin de l'instruction return.

Exemple 3-19. Déclaration d'une variable constante

```
const i=3;
```

i prend la valeur 3 et ne peut plus être modifiée.

Les variables globales qui sont définies sans le mot-clé const sont traitées par le compilateur comme des variables de classe de stockage extern par défaut. Ces variables sont donc accessibles à partir de tous les fichiers du programme. En revanche, cette règle n'est pas valide pour les variables définies avec le mot-clé const. Ces variables sont automatiquement déclarées static par le compilateur, ce qui signifie qu'elles ne sont accessibles que dans le fichier dans lequel elles ont été déclarées. Pour les rendre accessibles aux autres fichiers, il faut impérativement les déclarer avec le mot-clé extern avant de les définir.

Exemple 3-20. Déclaration de constante externes

```
int i = 12;          /* i est accessible de tous les fichiers. */
const int j = 11;    /* Synonyme de "static const int j = 11;". */

extern const int k;  /* Déclare d'abord la variable k… */
const int k = 12;    /* puis donne la définition. */
```

Notez que toutes les variables définies avec le mot-clé const doivent être initialisées lors de leur définition. En effet, on ne peut pas modifier la valeur des variables const, elles doivent donc avoir une valeur initiale. Enfin, les variables statiques non initialisées prennent la valeur nulle.

Les mots-clés const et volatile demandent au compilateur de réaliser des vérifications additionnelles lors de l'emploi des variables qui ont ces classes de stockage. En effet, le C/C++ assure qu'il est interdit de modifier (du moins sans magouiller) une variable de classe de stockage const, et il assure également que toute les références à une variable de classe de stockage volatile se feront sans optimisations dangereuses. Ces vérifications sont basées sur le type des variables manipulées. Dans le cas des types de base, ces vérifications sont simples et de compréhension immédiate. Ainsi, les lignes de code suivantes :

```
const int i=3;
int j=2;
```

```
i=j;   /* Illégal : i est de type const int. */
```

génèrent une erreur parce qu'on ne peut pas affecter une valeur de type int à une variable de type const int.

En revanche, pour les types complexes (pointeurs et références en particulier), les mécanismes de vérifications sont plus fins. Nous verrons quels sont les problèmes soulevés par l'emploi des mots-clés const et volatile avec les pointeurs et les références dans le chapitre suivant.

Enfin, en C++ uniquement, le mot-clé mutable permet de rendre un champ de structure const accessible en écriture :

Exemple 3-21. Utilisation du mot-clé mutable

```
struct A
{
  int i;          // Non modifiable si A est const.
  mutable int j;  // Toujours modifiable.
};

const A a={1, 1};   // i et j valent 1.

int main(void)
{
  a.i=2;          // ERREUR ! a est de type const A !
  a.j=2;          // Correct : j est mutable.
  return 0;
}
```

Chapitre 4. Les pointeurs et références

Les pointeurs sont des variables très utilisées en C et en C++. Ils doivent être considérés comme des variables, il n'y a rien de sorcier derrière les pointeurs. Cependant, les pointeurs ont un domaine d'application très vaste.

Les références sont des identificateurs synonymes d'autres identificateurs, qui permettent de manipuler certaines notions introduites avec les pointeurs plus souplement. Elles n'existent qu'en C++.

4.1. Notion d'adresse

Tout objet manipulé par l'ordinateur est stocké dans sa mémoire. On peut considérer que cette mémoire est constituée d'une série de " cases ", cases dans lesquelles sont stockées les valeurs des variables ou les instructions du programme. Pour pouvoir accéder à un objet (la valeur d'une variable ou les instructions à exécuter par exemple), c'est à dire au contenu de la case mémoire dans laquelle cet objet est enregistré, il faut connaître le numéro de cette case. Autrement dit, il faut connaître l'emplacement en mémoire de l'objet à manipuler. Cet emplacement est appelé l'*adresse* de la case mémoire, et par extension, l'*adresse de la variable* ou l'*adresse de la fonction* stockée dans cette case et celles qui la suivent.

Toute case mémoire a une adresse unique. Lorsque l'on utilise une variable ou une fonction, le compilateur manipule l'adresse de cette dernière pour y accéder. C'est lui qui connaît cette adresse, le programmeur n'a pas à s'en soucier.

4.2. Notion de pointeur

Une adresse est une valeur. Cette valeur est constante, car en général un objet ne se déplace pas en mémoire.

Un *pointeur* est une variable qui contient l'adresse d'un objet, par exemple l'adresse d'une autre variable. On dit que le pointeur *pointe* sur la variable *pointée*. Ici, *pointer* signifie " faire référence à ". La valeur d'un pointeur peut changer : cela ne signifie pas que la variable pointée est déplacée en mémoire, mais plutôt que le pointeur pointe sur autre chose.

Afin de savoir ce qui est pointé par un pointeur, les pointeurs disposent d'un type. Ce type est construit à partir du type de l'objet pointé. Ceci permet au compilateur de vérifier que les manipulations réalisées en mémoire par l'intermédiaire du pointeur son valides. Le type des pointeur se lit " pointeur de … ", où les points de suspension représentent le nom du type de l'objet pointé.

Les pointeurs se déclarent en donnant le type de l'objet qu'ils devront pointer, suivi de leur identificateur précédé d'une étoile :

```
int *pi;   // pi est un pointeur d'entier.
```

Note: Si plusieurs pointeurs doivent être déclarés, l'étoile doit être répétée :

```
int *pi1, *pi2, j, *pi3;
```

Ici, `pi1`, `pi2` et `pi3` sont des pointeurs d'entiers et `j` est un entier.

Il est possible de faire un pointeur sur une structure dans une structure en indiquant le nom de la structure comme type du pointeur :

```
typedef struct nom
{
```

```
  nom *pointeur;   /* Pointeur sur une structure "nom". */
  ...
} MaStructure;
```

Ce type de construction permet de créer des listes de structures, dans lesquelles chaque structure contient l'adresse de la structure suivante dans la liste.

Il est également possible de créer des pointeurs sur des fonctions, et d'utiliser ces pointeurs pour paramétrer un algorithme avec l'action de la fonction pointée. Nous détaillerons plus loin ce type d'utilisation des pointeurs.

4.3. Déréférencement, indirection

Un pointeur ne servirait strictement à rien s'il n'y avait pas de possibilité d'accéder à l'adresse d'une variable ou d'une fonction (on ne pourrait alors pas l'initialiser), ou s'il n'y avait pas moyen d'accéder à l'objet référencé par le pointeur (la variable pointée ne pourrait pas être manipulée, ou la fonction pointée ne pourrait pas être appelée).

Ces deux opérations sont respectivement appelées *indirection* et *déréférencement*. Il existe deux opérateurs permettant de récupérer l'adresse d'un objet et d'accéder à l'objet pointé. Ces opérateurs sont respectivement & et *.

Il est très important de s'assurer que les pointeurs que l'on manipule sont tous initialisés (c'est à dire contiennent l'adresse d'un objet valide, et pas n'importe quoi). En effet, accéder à un pointeur non initialisé revient à lire, ou plus grave encore, à écrire dans la mémoire à un endroit complètement aléatoire (selon la valeur initiale du pointeur lors de sa création). En général, on initialise les pointeurs dès leur création, ou, s'ils doivent être utilisés ultérieurement, on les initialise avec le pointeur nul. Ceci permettra de faire ultérieurement des tests sur la validité du pointeur, ou au moins de détecter les erreurs. En effet, l'utilisation d'un pointeur initialisé avec le pointeur nul génère souvent une faute de protection du programme, que tout bon débogueur est capable de détecter. Le pointeur nul se note NULL.

> **Note:** NULL est une macro définie dans le fichier d'en-tête stdlib.h. En C, elle représente la valeur des pointeurs non initialisés. Malheureusement, cette valeur peut ne pas être égale à l'adresse 0 (certains compilateurs utilisent la valeur -1 pour NULL par exemple). C'est pour cela que cette macro a été définie, afin de représenter, selon le compilateur, la bonne valeur. Voir le Chapitre 6 pour plus de détails sur les macros et sur les fichiers d'en-tête.
>
> La norme du C++ fixe la valeur nulle des pointeurs à 0. Par conséquent, les compilateurs C/C++ qui définissent NULL comme étant égal à -1 posent un problème de portabilité certain, puisque un programme C qui utilise NULL n'est plus valide en C++. Par ailleurs, un morceau de programme C++ compilable en C qui utiliserait la valeur 0 ne serait pas correct en C.
>
> Il faut donc faire un choix : soit utiliser NULL en C et 0 en C++, soit utiliser NULL partout, quitte à redéfinir la macro NULL pour les programmes C++ (solution qui me semble plus pratique).

Exemple 4-1. Déclaration de pointeurs

```
int i=0;      /* Déclare une variable entière. */
int *pi;      /* Déclare un pointeur sur un entier. */
pi=&i;    /* Initialise le pointeur avec l'adresse de cette
             variable. */
*pi = *pi+1;  /* Effectue un calcul sur la variable pointée par pi,
              c'est à dire sur i lui-même, puisque pi contient
              l'adresse de i. */

        /* À ce stade, i ne vaut plus 0, mais 1. */
```

Il est à présent facile de comprendre pourquoi il faut répéter l'étoile dans la déclaration de plusieurs pointeurs :

```
int *p1, *p2, *p3;
```

signifie syntaxiquement : p1, p2 et p3 sont des pointeurs d'entiers, mais aussi *p1, *p2 et *p3 sont des entiers.

Si l'on avait écrit :

```
int *p1, p2, p3;
```

seul p1 serait un pointeur d'entier. p2 et p3 seraient des entiers.

L'accès aux champs d'une structure par le pointeur sur cette structure se fera avec l'opérateur '->', qui remplace '(*).'.

Exemple 4-2. Utilisation de pointeurs de structures

```
struct Client
{
  int Age;
};

Client structure1;
Client *pstr = &structure1;
pstr->Age = 35;   /* On aurait pu écrire (*pstr).Age=35; */
```

4.4. Notion de référence

En plus des pointeurs, le C++ permet de créer des références. Les *références* sont des synonymes d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

Note: Les références n'existent qu'en C++. Le C ne permet pas de créer des références.

Par exemple, si " id " est le nom d'une variable, il est possible de créer une référence " ref " de cette variable. Les deux identificateurs id et ref représentent alors la même variable, et celle-ci peut être accédée et modifiée à l'aide de ces deux identificateurs indistinctement.

Toute référence doit se référer à un identificateur : il est donc impossible de déclarer une référence sans l'initialiser. De plus, la déclaration d'une référence ne crée pas un nouvel objet comme c'est le cas pour la déclaration d'une variable par exemple. En effet, les références se rapportent à des identificateurs déjà existants. La syntaxe de la déclaration d'une référence est la suivante :

```
type &référence = identificateur;
```

Après cette déclaration, référence peut être utilisé partout où identificateur peut l'être. Ce sont des synonymes.

Exemple 4-3. Déclaration de références

```
int i=0;
int &ri=i;   // Référence sur la variable i.
ri=ri+i;         // Double la valeur de i (et de ri).
```

est possible de faire des références sur des valeurs numériques. Dans ce cas, les références doivent être déclarées comme étant constantes, puisqu'une valeur est une constante :

```
const int &ri=3;  // Référence sur 3.
int &error=4;     // Erreur ! La référence n'est pas constante.
```

4.5. Lien entre les pointeurs et les références

Les références et les pointeurs sont étroitement liés. En effet, si l'on utilise une référence pour manipuler un objet, cela revient exactement à manipuler un pointeur constant contenant l'adresse de l'objet manipulé. Les

références permettent simplement d'obtenir le même résultat que les pointeurs avec un plus grande facilité d'écriture.

Par exemple, considérons le morceau de code suivant :

```
int i=0;
int *pi=&i;
*pi=*pi+1;   // Manipulation de i via pi.
```

et faisons passer l'opérateur & de la deuxième ligne à gauche de l'opérateur d'affectation :

```
int i=0;
int &*pi=i;   // Ceci génère une erreur de syntaxe mais nous
              // l'ignorons pour les besoins de l'explication.
*pi=*pi+1;
```

Maintenant, comparons avec le morceau de code équivalent suivant :

```
int i=0;
int &ri=i;
ri=ri+1;          // Manipulation de i via ri.
```

Nous constatons que la référence ri peut être identifiée avec l'expression *pi, qui représente bel et bien la variable i. Ainsi, ri représente exactement i. Ceci permet de comprendre l'origine de la syntaxe de déclaration des références.

4.6. Passage de paramètres par variable ou par valeur

Il y a deux méthodes pour passer des variables en paramètres dans une fonction : le *passage par valeur* ou le *passage par variable*. Ces méthodes sont décrites ci-dessous.

4.6.1. Passage par valeur

La valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

Si l'expression passée en paramètre est une variable, son contenu est copié dans la variable locale. Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.

Le C ne permet de faire que des passages par valeur.

Exemple 4-4. Passage de paramètre par valeur

```
int i=2;

void test(int j)   /* j est la copie de la valeur passée en
                      paramètre */
{
  j=3;            /* Modifie j, mais pas i. */
  return;
}

int main(void)
{
  test(i);        /* Le contenu de i est copié dans j.
                     i n'est pas modifié. Il vaut toujours 2. */
  test(2);        /* La valeur 2 est copiée dans j. */
  return 0;
}
```

4.6.2. Passage par variable

La deuxième technique consiste à passer non plus la valeur des variables comme paramètre, mais à passer les variables elles-mêmes. Il n'y a donc plus de copie, plus de variables locales. Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.

Le C ne permet pas de faire ce type de passage de paramètres (le C++ le permet en revanche).

Exemple 4-5. Passage de paramètre par variable en Pascal

```
Var i : integer;

Procedure test(Var j : integer)
Begin
          {La variable j est strictement égale
           à la variable passée en paramètre.}
  j:=2;    {Ici, cette variable est modifiée.}
End;

Begin
  i:=3;    {Initialise i à 3}
  test(i); {Appelle la fonction. La variable i est passée en
           paramètres, pas sa valeur. Elle est modifiée par
           la fonction test.}

    {Ici, i vaut 2.}
End.
```

Puisque la fonction attend une variable en paramètre, on ne peut plus appeler test avec une valeur (test(3) est maintenant interdit, car 3 n'est pas une variable : on ne peut pas le modifier).

4.6.3. Avantages et inconvénients des deux méthodes

Les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètre.

Les passages par variables sont plus rapides et plus économes en mémoire que les passages par valeur, puisque les étapes de la création de la variable locale et la copie de la valeur ne sont pas faites. Il faut donc éviter les passages par valeur dans les cas d'appels récursifs de fonction ou de fonctions travaillant avec des grandes structures de données (matrices par exemple).

4.6.4. Comment passer les paramètres par variable en C ?

Il n'y a qu'une solution : passer l'adresse de la variable. Ceci constitue donc une application des pointeurs.

Voici comment l'exemple Exemple 4-5 serait programmé en C :

Exemple 4-6. Passage de paramètre par variable en C

```
void test(int *pj) /* test attend l'adresse d'un entier… */
{
  *pj=2;          /* … pour le modifier. */
  return;
}

void main(void)
{
  int i=3;
  test(&i);   /* On passe l'adresse de i en paramètre. */
```

```
  /* Ici, i vaut 2. */
  return;
}
```

À présent, il est facile de comprendre la signification de & dans l'appel de scanf : les variables à entrer sont passées par variable.

4.6.5. Passage de paramètres par référence

La solution du C est exactement la même que celle du Pascal du point de vue sémantique. En fait, le Pascal procède exactement de la même manière en interne, mais la manipulation des pointeurs est masquée par le langage. Cependant, plusieurs problèmes se posent au niveau syntaxique :

- la syntaxe est lourde dans la fonction, à cause de l'emploi de l'opérateur * devant les paramètres ;
- la syntaxe est dangereuse lors de l'appel de la fonction, puisqu'il faut systématiquement penser à utiliser l'opérateur & devant les paramètres. Un oubli devant une variable de type entier et la valeur de l'entier est utilisée à la place de son adresse dans la fonction appelée (plantage assuré, essayez avec scanf).

Le C++ permet de résoudre tous ces problème à l'aide des références. Au lieu de passer les adresses des variables, il suffit de passer les variables elles-mêmes en utilisant des paramètres sous la forme de références. La syntaxe des paramètres devient alors :

type &identificateur [, type &identificateur […]]

Exemple 4-7. Passage de paramètre par référence en C++

```
void test(int &i)   // i est une référence du paramètre constant.
{
  i = 2;    // Modifie le paramètre passé en référence.
  return;
}

int main(void)
{
  int i=3;
  test(i);
  // Après l'appel de test, i vaut 2.
  // L'opérateur & n'est pas nécessaire pour appeler
  // test.
  return 0;
}
```

4.7. Arithmétique des pointeurs

Il est possible d'effectuer des opérations arithmétiques sur les pointeurs.

Les seules opérations valides sont les opérations externes (addition et soustraction des entiers) et la soustraction de pointeurs. Elles sont définies comme suit (la soustraction d'un entier est considérée comme l'addition d'un entier négatif) :

p + i = adresse contenue dans p + i*taille(élément pointé par p)

et :

p1 - p2 = adresse contenue dans p1 - adresse contenue dans p2

Si p est un pointeur d'entier, p+1 est donc le pointeur sur l'entier qui suit immédiatement celui pointé par p. On retiendra surtout que l'entier qu'on additionne au pointeur est multiplié par la taille de l'élément pointé pour obtenir la nouvelle adresse.

Le type du résultat de la soustraction de deux pointeurs est très dépendant de la machine cible et du modèle mémoire du programme. En général, on ne pourra jamais supposer que la soustraction de deux pointeurs est un entier (que les chevronnés du C me pardonnent, mais c'est une erreur *très grave*). En effet, ce type peut être insuffisant pour stocker des adresses (une machine peut avoir des adresses sur 64 bits et des données sur 32 bits). Pour résoudre ce problème, le fichier d'en-tête stdlib.h contient la définition du type à utiliser pour la différence de deux pointeurs. Ce type est nommé ptrdiff_t.

Exemple 4-8. Arithmétique des pointeurs

```
int i, j;
ptrdiff_t delta = &i - &j;  /* Correct */
int error = &i - &j;        /* Peut marcher, mais par chance. */
```

Il est possible de connaître la taille d'un élément en octets en utilisant l'opérateur sizeof. Il a la syntaxe d'une fonction :

```
sizeof(type|expression)
```

Il attend soit un type, soit une expression. La valeur retournée est soit la taille en octets du type, soit celle du type de l'expression. Dans le cas des tableaux, il renvoie la taille totale du tableau. Si son argument est une expression, celle-ci n'est pas évaluée (donc si il contient un appel à une fonction, celle-ci n'est pas appelée). Par exemple :

```
sizeof(int)
```

renvoie la taille d'un entier en octet, et :

```
sizeof(2+3)
```

renvoie la même taille, car 2+3 est de type entier. 2+3 n'est pas calculé.

> **Note:** L'opérateur sizeof renvoie la taille des types en tenant compte de leur alignement. Ceci signifie par exemple que même si un compilateur espace les éléments d'un tableau afin de les aligner sur des mots mémoire de la machine, la taille des éléments du tableau sera celle des objets de même type qui ne se trouvent pas dans ce tableau (ils devront donc être alignés eux aussi). On a donc toujours l'égalité suivante :
>
> sizeof(tableau) = sizeof(élément) * nombre d'éléments

4.8. Utilisation des pointeurs avec les tableaux

Les tableaux sont étroitement liés aux pointeurs parce que, de manière interne, l'accès aux éléments des tableaux se fait par manipulation de leur adresse de base, de la taille des éléments et de leurs indices. En fait, l'adresse du n-ième élément d'un tableau est calculée avec la formule :

```
Adresse_n = Adresse_Base + n*taille(élément)
```

où taille(élément) représente la taille de chaque élément du tableau et Adresse_Base l'adresse de base du tableau. Cette adresse de base est l'adresse du début du tableau, c'est donc à la fois l'adresse du tableau et l'adresse de son premier élément.

Ce lien apparaît au niveau du langage dans les conversions implicites de tableaux en pointeurs, et dans le passage des tableaux en paramètre des fonctions.

4.8.1. Conversions des tableaux en pointeurs

Afin de pouvoir utiliser l'arithmétique des pointeurs pour manipuler les éléments des tableaux, le C++ effectue les conversions implicites suivantes lorsque nécessaire :

- tableau vers pointeur d'élément ;
- pointeur d'élément vers tableau.

Ceci permet de considérer les expressions suivantes comme équivalentes :

identificateur[n]

et :

*(identificateur + n)

si identificateur est soit un identificateur de tableau, soit celui d'un pointeur.

Exemple 4-9. Accès aux éléments d'un tableau par pointeurs

```
int tableau[100];
int *pi=tableau;

tableau[3]=5;   /* Le 4ème élément est initialisé à 5 */
*(tableau+2)=4; /* Le 3ème élément est initialisé à 4 */
pi[5]=1;        /* Le 5ème élément est initialisé à 1 */
```

> **Note:** Le langage C++ impose que l'adresse suivant le dernier élément d'un tableau doit toujours être valide. Ceci ne signifie absolument pas que la zone mémoire référencée par cette adresse est valide, bien au contraire, mais plutôt que cette adresse est valide. Il est donc garantit que cette adresse ne sera pas le pointeur NULL par exemple, ni tout autre valeur spéciale qu'un pointeur ne peut pas stocker. Il sera donc possible de faire des calculs d'arithmétique des pointeurs avec cette adresse, même si elle ne devra jamais être déréférencée, sous peine de voir le programme planter.
>
> On prendra garde à certaines subtilités. Les conversions implicites sont une facilité introduite par le compilateur, mais en réalité, les tableaux ne sont pas des pointeurs, ce sont des variables comme les autres, à ceci près : leur type est convertible en pointeur sur le type de leurs éléments. Il en résulte parfois quelques ambiguïtés lorsque l'on manipule les adresses des tableaux. En particulier, on a l'égalité suivante :
>
> &tableau == tableau
>
> en raison du fait que l'adresse du tableau est la même que celle de son premier élément. Il faut bien comprendre que dans cette expression, une conversion a lieu. Cette égalité n'est donc pas exacte en théorie. En effet, si c'était le cas, on pourrait écrire :
>
> *&tableau == tableau
>
> puisque les opérateurs * et & sont conjugués. D'où :
>
> tableau == *&tableau = *(&tableau) == *(tableau) == t[0]
>
> ce qui est faux (le type du premier élément n'est en général pas convertible en type pointeur.).

4.8.2. Paramètres de fonction de type tableau

La conséquence la plus importante de la conversion tableau vers pointeur se trouve dans le passage par variable des tableaux dans une fonction. Lors du passage d'un tableau en paramètre d'une fonction, la conversion implicite a lieu, les tableaux sont donc toujours passés par variable, jamais par valeur. Il est donc faux d'utiliser des pointeurs pour les passer en paramètre, car le paramètre aurait le type pointeur de tableau. On ne modifierait pas le tableau, mais bel et bien le pointeur du tableau. Le programme aurait donc de fortes chances de planter.

Par ailleurs, certaines caractéristiques des tableaux peuvent être utilisées pour les passer en paramètre dans les fonctions.

Il est autorisé de ne pas spécifier la taille de la dernière dimension des paramètres de type tableau dans les déclarations et les définitions de fonctions. En effet, la borne supérieure des tableaux n'a pas besoin d'être précisée pour manipuler leurs éléments (on peut malgré tout la donner si cela semble nécessaire).

Cependant, pour les dimensions deux et suivantes, les tailles des premières dimensions restent nécessaires. Si elles n'étaient pas données explicitement, le compilateur ne pourrait pas connaître le rapport des dimensions. Par exemple, la syntaxe :

```
int tableau[][];
```

utilisée pour référencer un tableau de 12 entiers ne permettrait pas de faire la différence entre les tableaux de deux lignes et de six colonnes et les tableaux de trois lignes et de quatre colonnes (et leurs transposés respectifs). Une référence telle que :

```
tableau[1][3]
```

ne représenterait rien. Selon le type de tableau, l'élément référencé serait le quatrième élément de la deuxième ligne (de six éléments), soit le dixième élément, ou bien le quatrième élément de la deuxième ligne (de quatre éléments), soit le huitième élément du tableau. En précisant tous les indices sauf un, il est possible de connaître la taille du tableau pour cet indice à partir de la taille globale du tableau, en la divisant par les tailles sur les autres dimensions (2 = 12/6 ou 3 = 12/4 par exemple).

Le programme d'exemple suivant illustre le passage des tableaux en paramètre :

Exemple 4-10. Passage de tableau en paramètre

```
int tab[10][20];

void test(int t[][20])
{
  /* Utilisation de t[i][j] … */
  return;
}

int main(void)
{
  test(tab);   /* Passage du tableau en paramètre. */
  return 0;
}
```

4.9. Références et pointeurs constants et volatiles

L'utilisation des mots-clés const et volatile avec les pointeurs et les références est un peu plus compliquée qu'avec les types simples. En effet, il est possible de déclarer des pointeurs sur des variables, des pointeurs constants sur des variables, des pointeurs sur des variables constantes et des pointeurs constants sur des variables constantes (bien entendu, il en est de même avec les références). La position des mots-clés const et volatile dans les déclarations des types complexes est donc extrêmement importante. En général, le mot-clé const ou volatile caractérise ce qui le suit dans la déclaration. Ainsi :

```
const int * pi;
```

permet de déclarer un pointeur d'entier constant. Mais :

```
int j;
int * const pi=&j;
```

déclare pi comme étant constant, et de type pointeur d'entier.

Note: Les déclarations C++ peuvent devenir très compliquées et difficiles à lire. Il existe une astuce qui permet de les interpréter facilement. Lors de l'analyse de la déclaration d'un identificateur X, il faut toujours commencer par une phrase du type " X est un … ". Pour trouver la suite de la phrase, il suffit de lire la déclaration en partant de l'identificateur et de suivre l'ordre imposé par les priorités des opérateurs. L'ordre des priorités peut être modifié par la présence de parenthèses. L'annexe B donne les priorités de tous les opérateurs du C++.

Ainsi, dans l'exemple suivant :

```
const int *pi[12];
void (*pf)(int * const pi);
```

la première déclaration se lit de la manière suivante : " pi (pi) est un tableau ([]) de 12 (12) entiers (int) constants (const) ". La deuxième déclaration se lit : " pf (pf) est un pointeur (*) de fonction (()) de pi (pi), qui est lui-même une constante (const) de type pointeur (*) d'entier (int). Cette fonction ne renvoie rien (void) ".

Le C et le C++ n'autorisent que les écritures qui conservent ou augmentent les propriétés de constance et de volatilité. Par exemple, le code suivant est correct :

```
char *pc;
const char *cpc;

cpc=pc;   /* Le passage de pc à cpc augmente la constance. */
```

parce qu'elle signifie que si l'on peut écrire dans une variable par l'intermédiaire du pointeur pc, on peut s'interdire de le faire en utilisant cpc à la place de pc. En revanche, si on n'a pas le droit d'écrire dans une variable, on ne peut en aucun cas se le donner.

Cependant, les règles du langage relatives à la modification des variables peuvent parfois paraître étranges. Par exemple, le langage interdit une écriture telle que celle-ci :

```
char *pc;
const char **ppc;

ppc = &pc;   /* Interdit ! */
```

Pourtant, cet exemple ressemble beaucoup à l'exemple précédent. On pourrait penser que le fait d'affecter un pointeur de pointeur de variable à un pointeur de pointeur de variable constante revient à s'interdire d'écrire dans une variable qu'on a le droit de modifier. Mais en réalité, cette écriture va contre les règles de constances, parce qu'elle permettrait de modifier une variable constante. Pour s'en convaincre, il faut regarder l'exemple suivant :

```
const char c='a';      /* La variable constante. */
char *pc;              /* Pointeur par l'intermédiaire duquel
                       nous allons modifier c. */
const char **ppc=&pc;  /* Interdit, mais supposons que ce ne le
                       soit pas. */
*ppc=&c;               /* Parfaitement légal. */
*pc='b';               /* Modifie la variable c. */
```

Que s'est-il passé ? Nous avons, par l'intermédiaire de ppc, affecté l'adresse de la constante c au pointeur pc. Malheureusement, pc n'est pas un pointeur de constante, et ceci nous a permis de modifier la constante c.

Afin de gérer correctement cette situation (et les situations plus complexes qui utilisent des triples pointeurs ou encore plus d'indirection), le C et le C++ interdisent l'affectation de tout pointeur dont les propriétés de constance et de volatilité sont moindres de celles du pointeur cible. La règle exacte est la suivante :

1. On note cv les différentes qualifications de constance et de volatilité possibles (à savoir : const volatile, const, volatile ou aucune classe de stockage).
2. Si le pointeur source est un pointeur cvs,0 de pointeur cvs,1 de pointeur … de pointeur cvs,n-1 de type Ts cvs,n, et que le pointeur destination est un pointeur cvd,0 de pointeur cvd,1 de pointeur … de pointeur cvd,n-1 de type Td cvs,n, alors l'affectation de la source à la destination n'est légale que si :
   - les types source Ts et destination Td sont compatibles ;

- il existe un nombre entier strictement positif N tel que, quel que soit j supérieur ou égal à N, on ait :
  - si const apparaît dans cvs,j, alors const apparaît dans cvd,j ;
  - si volatile apparaît dans cvs,j, alors volatile apparaît dans cvd,j ;
  - et tel que, quel que soit 0<k<N, const apparaisse dans cvd,k.

Ces règles sont suffisamment compliquées pour ne pas être apprises. Les compilateurs se chargeront de signaler les erreurs s'il y en a en pratique. Par exemple :

```
const char c='a';
const char *pc;
const char **ppc=&pc;  /* Légal à présent. */
*ppc=&c;
*pc='b';               /* Illégal (pc a changé de type). */
```

L'affectation de double pointeur est à présent légale, parce que le pointeur source a changé de type (on ne peut cependant toujours pas modifier le caractère c).

Il existe une exception notable à ces règles : l'initialisation des chaînes de caractères. Les chaînes de caractères telles que :

```
"Bonjour tout le monde !"
```

sont des chaînes de caractères constantes. Par conséquent, on ne peut théoriquement affecter leur adresse qu'à des pointeurs de caractères constants :

```
const char *pc="Coucou !"; /* Code correct. */
```

Cependant, il a toujours été d'usage de réaliser l'initialisation des chaînes de caractères de la même manière :

```
char *pc="Coucou !";       /* Théoriquement illégal, mais toléré. */
```

Par compatibilité, le langage fournit donc une conversion implicite entre " const char * " et " char * ". Cette facilité ne doit pas pour autant vous inciter à transgresser les règles de constance : utilisez les pointeurs sur les chaînes de caractères constants autant que vous le pourrez (quitte à réaliser quelques copies de chaînes lorsqu'un pointeur de caractère simple doit être utilisé). Sur certains systèmes, l'écriture dans une chaîne de caractère constante peut provoquer un plangage immédiat du programme.

En C++, les références constantes sont très utiles pour réaliser des passages de variables par paramètres sans pour autant donner à la fonction le droit de modifier les paramètres. Ce type de situation apparaît souvent lorsque l'on veut passer en paramètre une grosse structure ou toute autre variable dont la copie n'est pas nécessaire (et donc déconseillée si l'on tient à avoir de bonnes performances).

Exemple 4-11. Passage de paramètres constant par référence

```
typedef struct
{
  ...
} structure;
```

```
void ma_fonction(const structure & s)
{
  ...
  return ;
}
```

Dans cet exemple, s est une référence sur une structure constante. Le code se trouvant à l'intérieur de la fonction ne peut donc pas utiliser la référence s pour modifier la structure (on notera cependant que c'est la fonction elle-même qui s'interdit l'écriture dans la variable s. const est donc un mot-clé " coopératif ". Il n'est pas possible à un programmeur d'empêcher ses collègues d'écrire dans ses variables avec le mot-clé const. Nous verrons dans le Chapitre 8 que le C++ permet de pallier à ce problème grâce à une technique appelée l'encapsulation.).

Un autre avantage des références constantes pour les passages par variables est que si le paramètre n'est pas une variable, ou s'il n'est pas du bon type, une variable locale du type du paramètre est créée et initialisée avec la valeur du paramètre transtypé.

Exemple 4-12. Création d'un objet temporaire lors d'un passage par référence

```
void test(const int &i)
{
     ...    // Utilisation de la variable i
            // dans la fonction test. La variable
            // i est créée si nécessaire.
  return ;
}

int main(void)
{
  test(3);   // Appel de test avec une constante.
  return 0;
}
```

Au cours de cet appel, une variable locale est créée (la variable i de la fonction test), et 3 lui est affecté.

4.10. Les chaînes de caractères : pointeurs et tableaux à la fois !

On a vu dans le premier chapitre que les chaînes de caractères n'existaient pas en C/C++. Ce sont en réalité des tableaux de caractères dont le dernier caractère est le caractère nul.

Ceci a plusieurs conséquences. La première, c'est que les chaînes de caractères sont aussi des pointeurs sur des caractères, ce qui se traduit dans la syntaxe de la déclaration d'une chaîne de caractères constante :

```
const char *identificateur = "chaîne";
```

identificateur est déclaré ici comme étant un pointeur de caractère, puis il est initialisé avec l'adresse de la chaîne de caractères constante "chaîne".

La deuxième est le fait qu'on ne peut pas faire, comme en Pascal, des affectations de chaînes de caractères, ni des comparaisons. Par exemple, si " nom1 " et " nom2 " sont des chaînes de caractères, l'opération :

```
nom1=nom2;
```

n'est pas l'affectation du contenu de nom2 à nom1. C'est une affectation de pointeur : le pointeur nom1 est égal au pointeur nom2 et pointent sur *la même chaîne* ! Une modification de la chaîne pointée par nom1 entraîne donc la modification de la chaîne pointée par nom2...

De même, le test nom1==nom2 est un test entre pointeurs, pas entre chaînes de caractères. Même si deux chaînes sont égales, le test sera faux si elles ne sont pas au même emplacement mémoire.

Il existe dans la librairie C de nombreuses fonctions permettant de manipuler les chaînes de caractères. Par exemple, la copie d'une chaîne de caractères dans une autre se fera avec la fontion strcpy, la comparaison de

deux chaînes de caractères pourra être réalisée à l'aide de la fonction strcmp. Je vous invite à consulter la documentation de votre environnement de développement pour découvrir toutes les fonctions de manipulation des chaînes de caractères.

4.11. Allocation dynamique de mémoire

Les pointeurs sont surtout utilisés pour créer un nombre quelconque de variables, ou des variables de taille quelconque, en cours d'exécution du programme. Normalement, une variable est créée automatiquement lors de sa déclaration. Ceci est faisable parce que les variables à créer ainsi que leurs tailles sont connues au moment de la compilation (c'est le but des déclarations). Par exemple, une ligne comme :

```
int tableau[10000];
```

signale au compilateur qu'une variable tableau de 10000 entiers doit être créée. Le programme s'en chargera donc automatiquement lors de l'exécution.

Mais supposons que le programme gère une liste de clients. On ne peut pas savoir à l'avance combien de clients seront entrés, le compilateur ne peut donc pas faire la réservation de l'espace mémoire automatiquement. C'est au programmeur de le faire. Cette réservation de mémoire (appelée encore *allocation*) doit être faite pendant l'exécution du programme. La différence d'avec la déclaration de tableau précédente, c'est que le nombre de clients et donc la quantité de mémoire à allouer, est variable. Il faut donc faire ce qu'on appelle une *allocation dynamique de mémoire*.

4.11.1. Allocation dynamique de mémoire en C

Il existe deux principales fonctions C permettant de demander de la mémoire au système d'exploitation et de la lui restituer. Elles utilisent toutes les deux les pointeurs, parce qu'une variable allouée dynamiquement n'a pas d'identificateur, étant donné qu'elle n'est pas déclarée. Les pointeurs utilisés par ces fonctions C n'ont pas de type. On les référence donc avec des pointeurs non typés. Leur syntaxe est la suivante :

```
malloc(taille)
free(pointeur)
```

malloc (abréviation de " Memory ALLOCation ") alloue de la mémoire. Elle attend comme paramètre la taille de la zone de mémoire à allouer et renvoie un pointeur non typé (void *).

free (pour " FREE memory ") libère la mémoire allouée. Elle attend comme paramètre le pointeur sur la zone à libérer et ne renvoie rien.

Lorsqu'on alloue une variable typée, on doit faire un transtypage du pointeur renvoyé par malloc en pointeur de ce type de variable.

Pour utiliser les fonctions malloc et free, vous devez mettre au début de votre programme la ligne :

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

Son rôle est similaire à celui de la ligne #include <stdio.h>. Vous verrez sa signification dans le chapitre concernant le préprocesseur.

Exemple 4-13. Allocation dynamique de mémoire en C

```
#include <stdio.h>   /* Autorise l'utilisation de printf
                        et de scanf. */
#include <stdlib.h>  /* Autorise l'utilisation de malloc
```

```
                    et de free. */

int (*pi)[10][10];   /* Déclare un pointeur d'entier, qui sera
                    utilisé comme un tableau de n matrices
                    10*10. */
int main(void)
{
  unsigned int taille;   /* Taille du tableau (non connue
                        à la compilation). */
  printf("Entrez la taille du tableau : ");
  scanf("%u",&taille);
  pi = (int (*)[10][10]) malloc(taille * sizeof(int)*10*10);

  /* Ici se place la section utilisant le tableau. */

  free(pi);        /* Attention à ne jamais oublier de restituer
                    la mémoire allouée par vos programmes ! */
  return 0;
}
```

4.11.2. Allocation dynamique en C++

En plus des fonctions malloc et free du C, le C++ fournit d'autres moyens pour allouer et restituer la mémoire. Pour cela, il dispose d'opérateurs spécifiques : new, delete, new[] et delete[]. La syntaxe de ces opérateurs est respectivement la suivante :

```
new type
delete pointeur
new type[taille]
delete[] pointeur
```

Les deux opérateurs new et new[] permettent d'allouer de la mémoire, et les deux opérateurs delete et delete[] de la restituer.

La syntaxe de new est très simple, il suffit de faire suivre le mot-clé new du type de la variable à allouer, et l'opérateur renvoie directement un pointeur sur cette variable avec le bon type. Il n'est donc plus nécessaire d'effectuer un transtypage après l'allocation, comme c'était le cas pour la fonction malloc. Par exemple, l'allocation d'un entier se fait comme suit :

```
int *pi = new int;  // Équivalent à (int *) malloc(sizeof(int)).
```

La syntaxe de delete est encore plus simple, puisqu'il suffit de faire suivre le mot-clé delete du pointeur sur la zone mémoire à libérer :

```
delete pi;          // Équivalent à free(pi);
```

Les opérateurs new[] et delete[] sont utilisés pour allouer et restituer la mémoire pour les types tableaux. Ce ne sont pas les mêmes opérateurs que new et delete, et la mémoire allouée par les uns ne peut pas être libéré par les autres. Si la syntaxe de delete[] est la même que celle de delete, l'emploi de l'opérateur new[] nécessite de donner la taille du tableau à allouer. Ainsi, on pourra créer un tableau de 10000 entiers de la manière suivante :

```
int *Tableau=new int[10000];
```

et détruire ce tableau de la manière suivante :

```
delete[] Tableau;
```

Note: Il est important d'utiliser l'opérateur delete[] avec les pointeurs renvoyés par l'opérateur

new[] et l'opérateur delete avec les pointeurs renvoyés par new. De plus, on ne devra pas non plus mélanger les mécanismes d'allocation mémoire du C et du C++ (utiliser delete sur un pointeur renvoyé par malloc par exemple). En effet, le compilateur peut allouer une quantité de mémoire supérieure à celle demandée par le programme afin de stocker des données qui lui permettent de gérer la mémoire. Ces données peuvent être interprétées différemment pour chacune des méthodes d'allocation, si bien qu'une utilisation erronée peut entraîner soit la perte des blocs de mémoire, soit une erreur, soit un plantage.

L'opérateur new[] alloue la mémoire et crée les objets dans l'ordre croissant des adresses. Inversement, l'opérateur delete[] détruit les objets du tableau dans l'ordre décroissant des adresses avant de libérer la mémoire. Les opérateur delete et delete[] ne doivent pas générer d'erreur lorsqu'on leur passe en paramètre un pointeur nul.

Lorsqu'il n'y a pas assez de mémoire disponible, les opérateurs new et new[] peuvent se comporter de deux manières selon l'implémentation. Le comportement le plus répandu est de renvoyer un pointeur nul. Cependant, la norme C++ indique un comportement différent : si l'opérateur new manque de mémoire, il doit appeler un gestionnaire d'erreur. Ce gestionnaire ne prend aucun paramètre et ne renvoie rien. Selon le comportement de ce gestionnaire d'erreur, plusieurs actions peuvent être faites :

- soit ce gestionnaire peut corriger l'erreur d'allocation et rendre la main à l'opérateur new (le programme n'est donc pas terminé), qui effectue une nouvelle tentative pour allouer la mémoire demandée ;
- soit il ne peut rien faire. Dans ce cas, il peut mettre fait à l'exécution du programme ou lancer une exception std::bad_alloc, qui remonte alors jusqu'à la fonction appelant l'opérateur new.

L'opérateur new est donc susceptible de lancer une exception std::bad_alloc. Voir le Chapitre 9 pour plus de détails à ce sujet.

Il est possible de remplacer le gestionnaire d'erreur appelé par l'opérateur new à l'aide de la fonction std::set_new_handler, déclarée dans le fichier d'en-tête new. Cette fonction attend en paramètre un pointeur sur une fonction qui ne prend aucun paramètre et ne renvoie rien. Elle renvoie l'adresse du gestionnaire d'erreur précédent.

> **Note:** La fonction std::set_new_handler et la classe std::bad_alloc font partie de la librairie standard C++. Comme leurs noms l'indiquent, ils sont déclarés dans l'espace de nommage std::, qui est réservé pour les fonctions et les classes de la librairie standard. Voyez aussi le Chapitre 11 pour plus de détails sur les espaces de nommages. Si vous ne désirez pas utiliser les mécanismes des espaces de nommage, vous devrez inclure le fichier d'en-tête new.h au lieu de new.
>
> Attendez vous à ce qu'un jour, tous les compilateurs C++ lancent une exception en cas de manque de mémoire lors de l'appel à l'opérateur new, car c'est ce qu'impose la norme. Si vous ne désirez pas avoir à gérer les exceptions dans votre programme et continuer à recevoir un pointeur nul en cas de manque de mémoire, vous pouvez fournir un deuxième paramètre de type std::nothrow_t à l'opérateur new. La librairie standard définit l'objet constant std::nothrow à cet usage.

Les opérateurs C++ d'allocation et de désallocation de la mémoire devront être utilisés de préférence aux fonctions malloc et free car ils permettent un meilleur contrôle des types. De plus, nous verrons qu'ils permettent d'initialiser correctement les types définis par l'utilisateur dans le chapitre traitant de la couche objet du C++.

4.12. Pointeurs et références de fonctions

4.12.1. Pointeurs de fonctions

Il est possible de faire des pointeurs de fonctions. Un pointeur de fonction contient l'adresse du début du programme constituant la fonction. Il est possible d'appeler une fonction dont l'adresse est contenue dans un pointeur de fonction avec l'opérateur d'indirection *.

Pour déclarer un pointeur de fonction, il suffit de considérer les fonctions comme des variables. Leur déclaration est identique à celle des tableaux, en remplaçant les crochets par des parenthèses :

type (*identificateur)(paramètres);

où type est le type de la valeur renvoyée par la fonction, identificateur est le nom du pointeur de la fonction et paramètres est la liste des types des variables que la fonction attend comme paramètres, séparés par des virgules.

Exemple 4-14. Déclaration de pointeur de fonction

```
int (*pf)(int, int);   /* Déclare un pointeur de fonction. */
```

pf est un pointeur de fonction attendant comme paramètres deux entiers et renvoyant un entier.

Il est possible d'utiliser typedef pour créer un alias du type pointeur de fonction :

```
typedef int (*PtrFonct)(int, int);
PtrFonct pf;
```

PtrFonct est le type des pointeurs de fonctions.

Si f est une fonction répondant à ces critères, on peut alors initialiser pf avec l'adresse de f. De même, on peut appeler la fonction pointée par pf avec l'opérateur d'indirection.

Exemple 4-15. Déréférencement de pointeur de fonction

```
#include <stdio.h>   /* Autorise l'emploi de scanf et de printf. */

int f(int i, int j)  /* Définit une fonction. */
{
      return i+j;
}

int (*pf)(int, int); /* Déclare un pointeur de fonction. */

int main(void)
{
  int l, m;         /* Déclare deux entiers. */
  pf = &f;          /* Initialise pf avec l'adresse de la fonction */
                    /* f. */
  printf("Entrez le premier entier : ");
  scanf("%u",&l);   /* Initialise les deux entiers. */
  printf("\nEntrez le deuxième entier : ");
  scanf("%u",&m);

/* Utilise le pointeur pf pour appeler la fonction f
  et affiche le résultat : */

  printf("\nLeur somme est de : %u\n", (*pf)(l,m));
  return 0;
}
```

L'intérêt des pointeurs de fonction est de permettre l'appel d'une fonction parmi un éventail de fonctions au choix.

Par exemple, il est possible de faire un tableau de pointeurs de fonctions et d'appeler la fonction dont on connaît

l'indice de son pointeur dans le tableau.

Exemple 4-16. Application des pointeurs de fonctions

```
#include <stdio.h>  /* Autorise l'emploi de scanf et de printf. */

/* Définit plusieurs fonctions travaillant sur des entiers : */

int somme(int i, int j)
{
  return i+j;
}

int multiplication(int i, int j)
{
  return i*j;
}

int quotient(int i, int j)
{
  return i/j;
}

int modulo(int i, int j)
{
  return i%j;
}

typedef int (*fptr)(int, int);
fptr ftab[4];

void main(void)
{
  int i,j,n;
  ftab[0]=&somme;          /* Initialise le tableau de pointeur */
  ftab[1]=&multiplication; /* de fonctions. */
  ftab[2]=&quotient;
  ftab[3]=&modulo;
  printf("Entrez le premier entier : ");
  scanf("%u",&i);          /* Demande les deux entiers i et j. */
  printf("\nEntrez le deuxième entier : ");
  scanf("%u",&j);
  printf("\nEntrez la fonction : ");
  scanf("%u",&n);          /* Demande la fonction à appeler. */
  printf("\nRésultat : %u.\n", (*(ftab[n]))(i,j) );
  return;
}
```

4.12.2. Références de fonctions

Les références de fonctions sont acceptées en C++. Cependant, leur usage est assez limité. Elles permettent parfois de simplifier les écritures dans les manipulations de pointeurs de fonctions. Mais il n'est pas possible de définir des tableaux de références, le programme d'exemple donné ci-dessus ne peut donc pas être réécrit avec des références.

Les références de fonctions peuvent malgré tout être utilisées à profit dans le passage des fonctions en paramètre dans une autre fonction. Par exemple :

```
#include <stdio.h>  // Autorise l'emploi de scanf et de printf.

// Fonction de comparaison de deux entiers :

int compare(int i, int j)
{
  if (i<j) return -1;
  else if (i>j) return 1;
  else return 0;
}

// Fonction utilisant une fonction en tant que paramètre :

void trie(int tableau[], int taille, int (&fcomp)(int, int))
{
  // Effectue le tri de tableau avec la fonction fcomp.
  // Cette fonction peut être appelée comme toute les autres
  // fonctions :
  printf("%d", fcomp(2,3));
    ⋮
  return ;
}

int main(void)
{
  int t[3]={1,5,2};
  trie(t, 3, compare);   // Passage de compare() en paramètre.
  return 0;
}
```

4.13. Paramètres de la fonction main - ligne de commande

L'appel d'un programme se fait normalement avec la syntaxe suivante :

nom param1 param2 [...]

où nom est le nom du programme à appeler et param1, etc... sont les paramètres de la ligne de commande. De plus, le programme appelé peut renvoyer un code d'erreur au programme appelant (soit le système d'exploitation, soit un autre programme). Ce code d'erreur est en général 0 quand le programme s'est déroulé correctement. Toute autre valeur indique qu'une erreur s'est produite en cours d'exécution.

La valeur du code d'erreur est renvoyée par la fonction main. Le code d'erreur doit toujours être un entier. La fonction main peut donc (et même normalement doit) être de type entier :

int main(void) ...

Les paramètres de la ligne de commandes peuvent être récupérés par la fonction main. Si vous désirez les récupérer, la fonction main doit attendre deux paramètres :

- le premier est un entier, qui représente le nombre de paramètres ;
- le deuxième est un tableau de chaînes de caractères (donc en fait un tableau de pointeurs, ou encore un pointeur de pointeurs de caractères).

Les paramètres se récupèrent avec ce tableau. Le premier élément pointe toujours sur la chaîne donnant le nom du programme. Les autres éléments pointent sur les paramètres de la ligne de commande.

Exemple 4-17. Récupération de la ligne de commande

```
#include <stdio.h>     /* Autorise l'utilisation des fonctions */
                       /* printf et scanf. */
```

```
int main(int n, char *params[])  /* Fonction principale. */
{
  int i;

  /* Affiche le nom du programme : */
  printf("Nom du programme : %s.\n",params[0]);

  /* Affiche la ligne de commande : */
  for (i=1; i<n; i++)
    printf("Argument %d : %s.\n",i, params[i]);
  return 0;          /* Tout s'est bien passé : on renvoie 0 ! */
}
```

4.14. DANGER

Je vais répéter ici les principaux dangers que l'on encourt lorsqu'on utilise les pointeurs. Bien que tous ces dangers existent, il faut vivre avec car il est impossible de programmer en C/C++ sans pointeurs. Il suffit de faire attention pour les éviter.

Les pointeurs sont, comme on l'a vu, très utilisés en C/C++. Il faut donc bien savoir les manipuler.

Mais ils sont très dangereux, car ils permettent d'accéder à n'importe quelle zone mémoire, s'ils ne sont pas correctement initialisés. Dans ce cas, ils pointent n'importe où. Accéder à la mémoire avec un pointeur non initialisé peut altérer soit les données du programme, soit le code du programme lui-même, soit le code d'un autre programme ou celui du système d'exploitation. Ceci conduit dans la majorité des cas au plantage du programme, et parfois au plantage de l'ordinateur si le système ne dispose pas de mécanisme de protection efficace.

VEILLEZ À TOUJOURS INITIALISER LES POINTEURS QUE VOUS UTILISEZ.

Pour initialiser un pointeur qui ne pointe sur rien (c'est le cas lorsque la variable pointée n'est pas encore créée ou lorsqu'elle est inconnue lors de la déclaration du pointeur), on utilisera le pointeur prédéfini NULL.

VÉRIFIEZ QUE TOUTE DEMANDE D'ALLOCATION MÉMOIRE A ÉTÉ SATISFAITE.

La fonction malloc renvoie le pointeur NULL lorsqu'il n'y a plus ou pas assez de mémoire. Le comportement des opérateurs new et new[] est différent. Théoriquement, ils doivent lancer une exception si la demande d'allocation mémoire n'a pas pu être satisfaite. Cependant, la plupart des compilateurs font en sorte qu'ils renvoient le pointeur nul du type de l'objet à créer.

S'ils renvoient une exception, le programme sera arrêté si aucun traitement particulier n'est fait. Bien entendu, le programme peut traiter cette exception s'il le désire, mais en général, il n'y a pas grand chose à faire en cas de manque de mémoire. Vous pouvez consulter le chapitre traitant des exceptions pour plus de détails à ce sujet.

Dans tous les cas,

LORSQU'ON UTILISE UN POINTEUR, IL FAUT VÉRIFIER S'IL EST VALIDE

(par un test avec NULL ou le pointeur nul, ou en analysant l'algorithme).

Chapitre 5. Comment faire du code illisible ?

Il est facile, très facile, de faire des programmes illisibles en C ou en C++. À ce propos, deux choses peuvent être dites :

1. Ça n'accroît pas la vitesse du programme. Si l'on veut aller plus vite, il faut revoir l'algorithme ou changer de compilateur (inutile de faire de l'assembleur : les bons compilateurs se débrouillent mieux que les être humains sur ce terrain. L'avantage de l'assembleur est que là, au moins, on est sûr d'avoir un programme illisible.).
2. Ça augmente les chances d'avoir des bogues.

Cependant, si vous voulez vous amuser, voici quelques conseils :

5.1. De nouveaux opérateurs

Il existe des opérateurs merveilleux.

Le premier est l'opérateur ternaire (?:). C'est le seul opérateur qui attende 3 paramètres (à part l'opérateur fonctionnel () des fonctions, qui admet n paramètres).

Il permet de remplacer le if. Sa syntaxe est la suivante :

```
test ? expression1 : expression2
```

Dans la syntaxe donnée ci-dessus, test est évalué en premier. Son résultat doit être booléen ou entier. Si test est vrai (ou si sa valeur est non nulle), expression1 est calculée et sa valeur est renvoyée. Sinon, c'est la valeur de expression2 qui est renvoyée. Par exemple :

```
min=(i<j)?i:j;   /* Calcule le minimum de i et de j. */
```

Ça reste encore lisible. C'est pour cela que l'opérateur virgule (,) a été inventé. Il remplace les blocs d'instructions. Sa syntaxe est :

```
expression1,expression2[,expression3[…]]
```

Les expressions sont évaluées de gauche à droite (expression1, expression2, etc…), puis le type et la valeur de la dernière expression sont utilisés pour renvoyer le résultat. Par exemple :

```
double r = 5;
int i = r*3,1;   /* À la fin de l'instruction, r vaut 5
                    et i vaut 1. r*3 est calculé pour rien. */
```

Enfin, il y a les opérateurs d'incrémentation et de décrémentation (++ et —). Ils s'appliquent comme des préfixes ou des suffixes sur les variables. Lorsqu'ils sont en préfixe, la variable est incrémentée ou décrémentée, puis sa valeur est renvoyée. S'ils sont en suffixe, la valeur de la variable est renvoyée, puis la variable est incrémentée ou décrémentée. Par exemple :

```
int i=2,j,k;

j=++i;   /* À la fin de cette instruction, i et j valent 3. */
k=j++;   /* À la fin de cette ligne, k vaut 3 et j vaut 4. */
```

5.2. Quelques conseils

Voici les conseils que je donne pour faire du code illisible :

- abusez des opérateurs vus précédemment ;
- placez-les dans les structures de contrôles. Notamment, utilisez l'opérateur virgule pour faire des instructions composées dans les tests du `while` et dans tous les membres du `for`. Il est souvent possible de mettre le corps du `for` dans les parenthèses ;
- si nécessaire, utiliser les expressions composées ({ et }) dans les structures de contrôles ;
- placez toutes les déclarations de variables globales ensemble ;
- ne commentez rien, ou mieux, donnez des commentaires sans rapport avec le code ;
- choisissez des noms de variables aléatoires (pensez à une phrase, et prenez les premières ou les deuxièmes lettres des mots au hasard) ;
- faites des fonctions à rallonge ;
- ne soignez pas l'apparence de votre programme (pas d'indentations, plusieurs lignes ensembles) ;
- regroupez-les toutes dans un même fichier, par ordre de non-appariement ;
- rajoutez des parenthèses là où elles ne sont pas nécessaires ;
- rajoutez des transtypages là où ils ne sont pas nécessaires.

Exemple 5-1. Programme parfaitement illisible

```
/* Que fait ce programme ? */
#include <stdio.h>
int main(void)
  {
int zkmlpf,geikgh,wdxaj;
   scanf("%u",&zkmlpf);for (wdxaj=0,
  geikgh=0;
     ((wdxaj+=++geikgh),geikgh)<zkmlpf;);
  printf("%u",wdxaj); return 0;
}
```

Vous l'aurez compris : il est plus simple de dire ici ce qu'il ne faut pas faire que de dire comment il faut faire. Je ne prétend pas imposer à quiconque une méthodologie quelconque, car chacun est libre de programmer comme il l'entend. En effet, certaines conventions de codages sont aussi absurdes qu'inutiles et elles ont l'inconvénient de ne plaire qu'à celui qui les a écrites (et encore…). C'est pour cette raison que je me suis contenté de lister les sources potentielles d'illisibilité des programmes. Sachez donc simplement que si vous utilisez une des techniques données dans ce paragraphe, vous devriez vous assurer que c'est réellement justifié et revoir votre code. Pour obtenir des programmes lisibles, il faut simplement que chacun y mettre un peu du sien, c'est aussi une marque de politesse envers les autres programmeurs.

Chapitre 6. Le préprocesseur C

6.1. Définition

Le *préprocesseur* est un programme qui analyse un fichier texte et qui lui fait subir certaines transformations. Ces transformations peuvent être l'*inclusion d'un fichier*, la *suppression* d'une zone de texte ou le *remplacement* d'une zone de texte.

Le préprocesseur effectue ces opérations en suivant des ordres qu'il lit dans le fichier en cours d'analyse.

Il est appelé automatiquement par le compilateur, avant la compilation, pour traiter les fichiers à compiler.

6.2. Les commandes du préprocesseur

Toute commande du préprocesseur commence :

- en début de ligne ;
- par un signe dièse (#).

Les commandes sont les suivantes :

6.2.1. Inclusion de fichier

L'inclusion de fichier permet de factoriser du texte commun à plusieurs autres fichiers (par exemple des déclarations de types, de constantes, de fonctions, etc…). Le texte commun est mis en général dans un fichier portant l'extension .h (pour " header ", fichier d'en-tête de programme).

Syntaxe :

```
#include "fichier"
```

ou :

```
#include <fichier>
```

fichier est le nom du fichier à inclure. Lorsque son nom est entre guillemets, le fichier spécifié est recherché dans le répertoire courant (normalement le répertoire du programme). S'il est encadré de crochets, il est recherché d'abord dans les répertoires spécifiés en ligne de commande avec l'option -I, puis dans les répertoires du chemin de recherche des en-têtes du système (ces règles ne sont pas fixes, elles ne sont pas normalisées).

Le fichier inclus est traité lui aussi par le préprocesseur.

La signification de la ligne #include <stdio.h> au début de tous les programmes utilisant les fonctions scanf et printf devient alors claire. Si vous ouvrez le fichier stdio.h, vous y verrez la déclaration de toutes les fonctions et de tous les types de la librairie d'entrée - sortie standard. De même, les fonctions malloc et free sont déclarées dans le fichier d'en-tête stdlib.h et définies dans la librairie standard. L'inclusion de ces fichiers permet donc de déclarer ces fonctions afin de les utiliser.

6.2.2. Remplacement de texte

Syntaxe :

```
#define texte remplacement
```

texte est le texte à remplacer. Le texte de remplacement est remplacement. Il est facultatif (dans ce cas, c'est le texte vide). À chaque fois que texte sera rencontré, il sera remplacé par remplacement.

Cette commande servira à définir les constantes.

Exemple 6-1. Définition de constantes

```
#define VRAI 1
#define FAUX 0
```

VRAI sera remplacé systématiquement par 1 et FAUX par 0 dans la suite du texte.

Le préprocesseur définit un certain nombre de chaînes de remplacement automatiquement. Ce sont les suivantes :

- __LINE__ : donne le numéro de la ligne courante ;
- __FILE__ : donne le nom du fichier courant ;
- __DATE__ : renvoie la date du traitement du fichier par le préprocesseur ;
- __TIME__ : renvoie l'heure du trautement du fichier par le préprocesseur ;
- __cplusplus : définie uniquement dans le cas d'une compilation C++. Sa valeur doit être 199711L pour les compilateurs compatibles avec le projet de norme du 2 décembre 1996. En pratique, sa valeur est dépendante de l'implémentation utilisée, mais on pourra utiliser cette chaîne de remplacement pour distinguer les parties de code écrites en C++ de celles écrites en C.

> **Note:** Si __FILE__, __DATE__, __TIME__ et __cplusplus sont bien des constantes pour un fichier donné, ce n'est pas le cas de __LINE__. En effet, cette dernière " constante " change bien évidemment de valeur à chaque ligne. On peut considérer qu'elle est redéfinie automatiquement par le préprocesseur à chaque début de ligne.
>
> On fera par ailleurs une distinction bien nette entre les constantes littérales définies avec la directive #define du préprocesseur et les constantes définies avec le mot-clé const. En effet, les constantes littérales ne réservent pas de mémoire. Ce sont des valeurs immédiates, définies par le compilateur. En revanche, les variables de classe de stockage const ont malgré tout une place mémoire réservée. Il est possible, par un transtypage de pointeur, de modifier leur valeur (opération très déconseillée et de surcroît certainement inutile), et elles peuvent être modifiées par l'environnement si elles sont de type volatile. Ce sont donc des variables accessibles en lecture seule plutôt que des constantes. On ne pourra jamais supposer qu'une variable ne change pas de valeur sous prétexte qu'elle a la classe de stockage const, alors qu'évidemment, une vraie constante déclarée avec la directive #define du préprocesseur conservera toujours sa valeur (pourvu qu'on ne la redéfinisse pas).

6.2.3. Définition d'un identificateur

Il est possible de déclarer un identificateur. Pour cela, on utilise aussi #define :

```
#define identificateur
```

Pour détruire l'identificateur, il faut utiliser #undef :

```
#undef identificateur
```

Lorsqu'un identificateur est défini, il est possible de faire des tests dessus (voir section suivante).

6.2.4. Suppression de texte

Syntaxe :

```
#ifdef identificateur
  ⋮
#endif
```

Le texte compris entre le #ifdef (c'est à dire " if defined ") et le #endif est laissé tel quel si l'identificateur identificateur est connu du préprocesseur. Sinon, il est supprimé. L'identificateur peut être déclaré en utilisant simplement la commande #define vue ci-dessus.

Il existe d'autres commandes conditionnelles :

```
#ifndef     (if not defined …)
#elif       (sinon, si … )
#if         (si … )
```

qui permettent de faire des fichiers dont certaines parties ne seront pas compilées. C'est ce qu'on appelle la *compilation conditionnelle*. La directive #if attend en paramètre une expression constante. Le texte qui la suit est inclus dans le fichier si et seulement si cette expression est non nulle. Par exemple :

```
#if (__cplusplus==199711L)
  ⋮
#endif
```

permet d'inclure un morceau de code C++ strictement à la norme décrite dans le projet de norme du 2 décembre 1996.

Une autre application courante des directives de compilation est la suivante :

```
#ifndef DejaLa
#define DejaLa

Texte à n'inclure qu'une seule fois au plus.

#endif
```

qui permet d'éviter que le texte ne soit inclus plusieurs fois, à la suite de plusieurs appels de #include. En effet, au premier appel, DejaLa n'est pas connu du préprocesseur. Il est donc déclaré et le texte est inclus. Lors de tout autre appel ultérieur, DejaLa existe, et le texte est n'est pas inclus. Ce genre d'écriture se rencontre dans les fichiers d'en-tête, pour lesquels en général on ne veut pas qu'une inclusion multiple ait lieu.

6.2.5. Autres commandes

Le préprocesseur est capable d'effectuer d'autres actions que l'inclusion et la suppression de texte. Les directives qui permettent d'effectuer ces actions sont indiquées ci-dessous :

- # : ne fait rien (directive nulle) ;
- #error message : permet de stopper la compilation en affichant le message d'erreur donné en paramètre ;
- #line numéro [fichier] : permet de changer le numéro de ligne courant et le nom du fichier courant lors de la compilation ;
- #pragma texte : permet de donner des ordres dépendant de l'implémentation au compilateur. Toute implémentation qui ne reconnaît pas un ordre donné dans une directive #pragma doit l'ignorer pour éviter des messages d'erreurs.

6.3. Les macros

Le préprocesseur peut, lors du mécanisme de remplacement de texte, prendre des paramètres. Ces paramètres sont placés sans modifications dans le texte de remplacement. Le texte de remplacement est alors appelé *macro*.

Syntaxe :

```
#define macro(identificateur[,identificateur[…]]) définition
```

Exemple 6-2. Macros MIN et MAX

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
```

Pour poursuivre une définition sur la ligne suivante, terminez la ligne courante par le signe '\'.

Le mécanisme des macros permet de faire l'équivalent des fonctions générales, comme, qui fonctionnent pour tous les types ordonnés. Ainsi, la macro MAX renvoie le maximum de ses deux paramètres, qu'ils soient entiers, longs ou réels. Cependant, on prendra garde au fait que les parmamètres passés à une macro sont évalués par celle-ci à chaque fois qu'ils sont utilisés dans la définition de la macro. Ceci peut poser des problèmes de performances, ou pire, provoquer des effets de bords indésirables. Par exemple, l'utilisation suivante de la macro MIN :

```
MIN(f(3), 5)
```

provoque le remplacement suivant :

```
((f(3))<(5))?(f(3)):(5))
```

soit deux appels de la fonction f si f(3) est inférieur à 5, et un seul appel sinon. Si la fonction f ainsi appelée modifie des variables globales, le résultat de la macro ne sera certainement pas celui attendu, puisque le nombre d'appels est variable pour une même expression. On évitera donc, autant que faire se peut, d'utiliser des expressions ayant des effets de bords en paramètres d'une macro. Les écritures du type :

```
MIN(++i, j)
```

sont donc à prohiber.

On mettra toujours des parenthèses autour des paramètres de la macro. En effet, ces paramètres peuvent être des expressions composées, qui doivent être calculées complètement avant d'être utilisée dans la macro. Les parenthèses forcent ce calcul. Si on ne les met pas, les règles de priorités peuvent générer une erreur de logique dans la macro elle-même. De même, on entourera de parenthèses les macros renvoyant une valeur, afin de forcer leur évaluation complète avant toute utilisation dans une autre expression. Par exemple :

```
#define mul(x,y) x*y
```

est une macro fausse. La ligne :

```
mul(2+3,5+9)
```

sera remplacée par :

```
2+3*5+9
```

ce qui vaut 26, et non pas 70 comme on l'aurait attendu. La bonne macro est :

```
#define mul(x,y) ((x)*(y))
```

car elle donne le texte suivant :

```
((2+3)*(5+9))
```

et le résultat est correct. De même, la macro :

```
#define add(x,y) (x)+(y)
```

est fausse, car l'expression suivante :

```
add(2,3)*5
```

est remplacée textuellement par :

```
(2)+(3)*5
```

dont le résultat est 17 et non 25 comme on l'aurait espéré. Cette macro doit donc se déclarer comme suit :

```
#define add(x,y) ((x)+(y))
```

Ainsi, les parenthèses assurent un comportement cohérent de la macro. Comme on le voit, les parenthèses peuvent alourdir les définitions des macros, mais elles sont absolument nécessaires.

Le résultat du remplacement d'une macro par sa définition est, lui aussi, soumis au préprocesseur. Par conséquent, une macro peut utiliser une autre macro ou une constante définie avec #define. Cependant, ce mécanisme est limité aux macros qui n'ont pas encore été remplacées afin d'éviter une récursion infinie du préprocesseur. Par exemple :

```
#define toto(x) toto((x)+1)
```

définit la macro toto. Si plus loin on utilise " toto(3) ", le texte de remplacement final sera " toto((3)+1) " et non pas l'expression infinie " (...(((3)+1)+1...)+1 ".

Le préprocesseur définit automatiquement la macro defined, qui permet de tester si un identificateur est connu du préprocesseur. Sa syntaxe est la suivante :

```
defined(identificateur)
```

La valeur de cette macro est 1 si l'identificateur existe, 0 sinon. Elle est utilisée principalement avec la directive #if. Il est donc équivalent d'écrire :

```
#if defined(identificateur)
  ⋮
#endif
```

à la place de :

```
#ifdef identificateur
  ⋮
#endif
```

Cependant, defined permet l'écriture d'expressions plus complexes que la directive #if.

6.4. Manipulation de chaînes de caractères dans les macros

Le préprocesseur permet d'effectuer des opérations sur les chaînes de caractères. Tout argument de macro peut être transformé en chaîne de caractères dans la définition de la macro s'il est précédé du signe #. Par exemple, la macro suivante :

```
#define string(s) #s
```

transforme son argument en chaîne de caractères. Par exemple :

```
string(2+3)
```

devient :

```
"2+3"
```

Lors de la transformation de l'argument, toute occurrence des caractères " et \ est transformée respectivement en \" et \\ pour conserver ces caractères dans la chaîne de caractères de remplacement.

Le préprocesseur permet également la concaténation de texte grâce à l'opérateur ##. Les arguments de la macro qui sont séparés par cet opérateur sont concaténés (sans être transformés en chaînes de caractères cependant). Par exemple, la macro suivante :

```
#define nombre(chiffre1,chiffre2) chiffre1##chiffre2
```

permet de construire un nombre à deux chiffres :

```
nombre(2,3)
```

est remplacé par le nombre décimal 23. Le résultat de la concaténation est ensuite analysé pour d'éventuels remplacements additionnels par le préprocesseur.

6.5. Les trigraphes

Le jeu de caractère utilisé par le langage C++ comprend toutes les lettres en majuscules et en minuscules, tous les chiffres et les caractères suivants :

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

Malheureusement, certains environnements sont incapables de gérer quelques-uns de ces caractères. C'est pour résoudre ce problème que les *trigraphes* ont été créés.

Les trigraphes sont des séquences de trois caractères commençant par deux points d'interrogations. Ils permettent de remplacer les caractères qui ne sont pas accessibles sur tous les environnements. Vous n'utiliserez donc sans doute jamais les trigraphes, à moins d'y être forcé. Les trigraphes disponibles sont définis ci-dessous :

Tableau 6-1. Trigraphes

Trigraphe	**Caractère de remplacement**
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Chapitre 7. Modularité

La *modularité* est le fait, pour un programme, d'être écrit en plusieurs morceaux relativement indépendants les uns des autres. La modularité a d'énormes avantages lors du développement d'un programme. Cependant, elle implique un processus de génération de l'exécutable assez complexe. Dans ce chapitre, nous allons voir l'intérêt de la modularité, les différentes étapes qui permettent la génération de l'exécutable, et enfin l'influence de ces étapes sur la syntaxe du langage.

7.1. Pourquoi faire une programmation modulaire ?

Ce qui coûte le plus cher en informatique, c'est le développement de logiciel, pas le matériel. En effet, développer un logiciel demande du temps, de la main d'½uvre, et n'est pas facile (il y a toujours des erreurs). De plus, les logiciels développés sont souvent spécifiques à un type de problème donné. Pour chaque problème, il faut tout refaire.

Ce n'est pas un très bon bilan. Pour éviter tous ces inconvénients, une branche de l'informatique a été développée : le *génie logiciel*. Le génie logiciel donne les grands principes à appliquer lors de la réalisation d'un programme, de la conception à la distribution, et sur toute la durée de vie du projet. Ce sujet dépasse largement le cadre de ce cours, aussi je ne parlerais que de l'aspect codage seul, c'est à dire ce qui concerne le C/C++.

Au niveau du codage, le plus important est la programmation modulaire. Les idées qui en sont à la base sont les suivantes :

- diviser le travail en plusieurs équipes ;
- créer des morceaux de programme indépendants de la problématique globale, donc réutilisables pour d'autres logiciels ;
- supprimer les risques d'erreurs qu'on avait en reprogrammant ces morceaux à chaque fois.

Je tiens à préciser que les principes de la programmation modulaire ne s'appliquent pas qu'aux programmes développés par des équipes de programmeurs. Ils s'appliquent aussi aux programmeurs individuels. En effet il est plus facile de décomposer un problème en ses éléments, forcément plus simples, que de le traiter dans sa totalité (dixit Descartes).

Pour parvenir à ce but, il est indispensable de pouvoir découper un programme en sous-programmes indépendants, ou presque indépendants. Pour que chacun puisse travailler sur sa partie de programme, il faut que ces morceaux de programme soient dans des fichiers séparés.

Pour pouvoir vérifier ces morceaux de programme, il faut que les compilateurs puissent les compiler indépendamment, sans avoir les autres fichiers du programme. Ainsi, le développement de chaque fichier peut se faire relativement indépendamment de celui des autres. Cependant, cette division du travail implique des opérations assez complexes pour générer l'exécutable.

7.2. Étapes impliquées dans la génération d'un exécutable

Les phases du processus qui conduit à l'exécutable à partir des fichiers source d'un programme sont décrites ci-dessous.

Au début de la génération de l'exécutable, on ne dispose que des fichiers source du programme, écrit en C, C++ ou tout autre langage (ce qui suit n'est pas spécifique au C/C++). En général, la première étape est le traitement des fichiers source avant compilation. Dans le cas du C et du C++, il s'agit des opérations effectuées par le *préprocesseur* (remplacement de macros, suppression de texte, inclusion de fichiers…).

Vient ensuite la *compilation séparée*, qui est le fait de compiler séparément les fichiers sources. Le résultat de la compilation d'un *fichier source* est un *fichier objet*. Les fichiers objet contiennent la traduction du code des fichiers source en *langage machine*. Ils contiennent aussi d'autres informations, par exemple les données initialisées et les informations qui seront utilisées lors de la création du fichier exécutable à partir de tous les fichiers objet générés.

Enfin, la dernière étape est le regroupement de toutes les données et de tout le code des fichiers objet, ainsi que la résolution des références inter-fichiers. Cette étape est appelée *édition de liens* (" linking " en anglais). Le résultat de l'édition de liens est le *fichier image*, qui pourra être chargé en mémoire par le système d'exploitation. Les fichiers exécutables et les librairies dynamiques sont des exemples de fichiers image.

Toutes ces opérations peuvent être réunies soit au niveau du compilateur, soit grâce à un programme appelé **make**. Le principe de **make** est toujours le même, même si aucune norme n'a été définie en ce qui le concerne. **make** lit un fichier (le fichier (" makefile "), dans lequel se trouvent toutes les opérations nécessaires pour compiler un programme. Puis, il les exécute si c'est nécessaire. Par exemple, un fichier qui a déjà été compilé et qui n'a pas été modifié depuis ne sera pas recompilé. C'est plus rapide. **make** se base sur les dates de dernière modification des fichiers pour savoir s'ils ont été modifiés (il compare les dates des fichiers source et des fichiers objets). La date des fichiers est gérée par le système d'exploitation : il est donc important que l'ordinateur soit à l'heure.

7.3. Compilation séparée en C/C++

La compilation séparée se fait au niveau du fichier. Il existe trois grand types de fichiers en C/C++ :

- les fichiers C, qui ne contiennent que du code C ;
- les fichiers C++, qui contiennent du code C++ et éventuellement du code C si ce dernier est suffisamment propre ;
- les fichiers d'en-têtes, qui contiennent toutes les déclarations et définitions communes à plusieurs fichiers sources.

On utilise une extension différente pour les fichiers C et les fichiers C++ afin de les différencier. Les conventions utilisées dépendent du compilateur. Cependant, on peut en général établir les règles suivantes :

- les fichiers C ont l'extension .c ;
- les fichiers C++ prennent l'extension .cc, ou .C (majuscule) sur UNIX, ou .cpp sur les PC sous DOS ou Windows (ces deux systèmes ne faisant pas la différence entre les majuscules et les minuscules dans leur systèmes de fichiers) ;
- les fichiers d'en-tête ont l'extension .h, parfois .hpp (en-tête C++).

> **Note:** Les fonctions inline doivent impérativement être définies dans les fichiers où elles sont utilisées, puisqu'en théorie, elles sont recopiées dans les fonctions qui les utilisent. Ceci implique de placer leur définition dans les fichiers d'en-tête .h ou .hpp. Comme le code des fonctions inline est normalement inclus dans le code des fonctions qui les utilisent, les fichiers d'en-têtes contenant du code inline peuvent être compilés séparément sans que ces fonctions ne soient définies plusieurs fois. Par conséquent, l'éditeur de lien ne générera pas d'erreur (alors qu'il l'aurait fait si on avait placé le code d'une fonction non inline dans un fichier d'en-tête inclus plusieurs fois). Certains programmeurs considèrent qu'il n'est pas bon de placer des définitions de fonctions dans des fichiers d'en-têtes, il placent donc toutes leurs fonctions inline dans des fichiers portant l'extension .inl. Ces fichiers sont ensuite inclus soit dans les fichiers d'en-tête .h, soit dans les fichiers .c ou .cpp qui utilisent les fonctions inline.

7.4. Syntaxe des outils de compilation

Il existe évidemment un grand nombre de compilateurs C/C++ pour chaque plate-forme. Ils ne sont malheureusement pas compatibles au niveau de la ligne de commande. Le même problème apparaît pour les éditeurs de lien (linker en anglais) et pour **make**. Cependant, quelques principes généraux peuvent être établis. Dans la suite, je supposerais que le nom du compilateur est " **cc** ", que celui du préprocesseur est " **cpp** ", celui de l'éditeur de lien est " **ld** " et que celui de make est " **make** ".

En général, les différentes étapes de la compilation et de l'édition de liens sont regroupées au niveau du compilateur, ce qui permet de faire les phases de traitement du préprocesseur, de compilation et d'édition de liens en une seule commande. Les lignes de commandes des compilateurs sont donc souvent compliquées et très peu portable. En revanche, la syntaxe de make est un peu plus portable.

7.4.1. Syntaxe des compilateurs

Le compilateur demande en général les noms des fichiers source à compiler et les noms des fichiers objet à utiliser lors de la phase d'édition de liens. Lorsque l'on spécifie un fichier source, le compilateur utilisera le fichier objet qu'il aura créé pour ce fichier source en plus des fichiers objet donnés dans la ligne de commande. Le compilateur peut aussi accepter en ligne de commande le chemin de recherche des librairies du langage et des fichiers d'en-tête. Enfin, différentes options d'optimisations sont disponibles (mais très peu portable). La syntaxe (simplifiée) des compilateurs est souvent la suivante :

```
cc [fichier.o […]] [[-c] fichier.c […]] [-o exécutable]
   [-Lchemin_librairies] [-llibrairie […]] [-Ichemin_include]
```

fichier.c est le nom du fichier à compiler. Si l'option -c le précède, le fichier sera compilé, mais l'éditeur de lien ne sera pas appelé. Si cette option n'est pas présente, l'éditeur de lien est appelé, et le programme exécutable formé est enregistré dans le fichier a.out. Pour donner un autre nom à ce programme, il faut utiliser l'option -o, suivie du nom de l'exécutable. Il est possible de donner le nom des fichiers objet déjà compilé (" fichier.o ") pour que l'éditeur de liens les lie avec le programme compilé.

L'option -L permet d'indiquer le chemin du répertoire des librairies de fonctions prédéfinies. Ce répertoire sera ajouté à la liste des répertoires indiqués dans la variable d'environnement LIBRARY_PATH. L'option -l demande au compilateur d'utiliser la librairie spécifiée, si elle ne fait pas partie des librairies utilisées par défaut. De même, l'option -I permet de donner le chemin d'accès au répertoire des fichiers à inclure (lors de l'utilisation du préprocesseur). Les chemins ajoutés avec cette option viennent s'ajouter aux chemins indiqués dans les variables d'environnement C_INCLUDE_PATH et CPLUS_INCLUDE_PATH pour les programmes compilés respectivement en C et en C++.

L'ordre des paramètres sur la ligne de commande est significatif. La ligne de commande est exécutée de gauche à droite.

Exemple 7-1. Compilation d'un fichier et édition de liens

```
cc -c fichier1.c
cc fichier1.o programme.cc -o lancez_moi
```

Dans cet exemple, le fichier C fichier1.c est compilé en fichier1.o, puis le fichier C++ programme.cc est compilé et lié au fichier1.o pour former l'exécutable lancez_moi.

7.4.2. Syntaxe de make

La syntaxe de make est très simple :

```
make
```

En revanche, la syntaxe du fichier makefile est un peu plus compliquée et peu portable. Cependant, les fonctionnalités de base sont gérées de la même manière par la plupart des programme **make**.

Le fichier makefile est constitué d'une série de lignes d'informations et de lignes de commandes (de l'interpréteur de commande UNIX ou DOS). Les commandes doivent toujours être précédées d'un caractère de tabulation horizontale.

Les lignes d'informations donnent des renseignements sur les dépendances des fichiers (en particulier : quels sont les fichiers objet qui doivent être utilisés pour créer l'exécutable ?). Les lignes d'informations demandent à **make** de compiler les fichiers dont dépend l'exécutable avant de créer celui-ci. Les lignes de commande indiquent comment effectuer la compilation (et éventuellement d'autres tâches).

La syntaxe des lignes d'informations est la suivante :

nom:dépendance

où nom est le nom du fichier destination, et dépendance est la liste des noms des fichiers dont dépend le fichier nom, séparés par des espaces.

Les commentaires dans un fichier makefile se font avec le signe dièse (#).

Exemple 7-2. Fichier makefile sans dépendances

```
# Compilation du fichier fichier1.c :
	cc - c fichier1.c

# Compilation du programme principal :
	cc -o Lancez_moi fichier1.o programme.c
```

Exemple 7-3. Fichier makefile avec dépendances

```
# Indique les dépendances :
Lancez_moi: fichier1.o programme.o

# Indique comment compiler le programme :
# (le symbole $@ représente le nom de la destination, ici, Lancez_moi)
	cc -o $@ fichier1.o programme.o

#compile les dépendances :
fichier1.o: fichier1.c
	cc -c fichier1.c

programme.o: programme1.c
	cc -c programme.c
```

7.5. Problèmes syntaxiques relatifs à la compilation séparée

Pour que le compilateur puisse compiler les fichiers séparément, il faut que vous respectiez les conditions suivantes :

- chaque type ou variable utilisé doit être déclaré ;
- toute fonction non déclarée doit renvoyer un entier (en C seulement, en C++, l'utilisation d'une fonction non déclarée génère une erreur).

Ces conditions ont des répercussions sur la syntaxe des programmes. Elles seront vues dans les paragraphes suivants.

7.5.1. Déclaration des types

Les types doivent toujours être déclarés, comme d'habitude. Par exemple, il est interdit d'utiliser une structure client sans l'avoir définie avant sa première utilisation.

7.5.2. Déclaration des variables

Les variables qui sont définies dans un autre fichier doivent être déclarées avant leur première utilisation. Pour cela, on les spécifie comme étant des variables externes, avec le mot-clé extern :

```
extern int i;   /* i est un entier qui est déclaré et
                   créé dans un autre fichier.
                   Ici, il est simplement déclaré.
                */
```

Inversement, si une variable ne doit pas être accédée par un autre module, il faut déclarer cette variable statique. Ainsi, même si un autre fichier utilise le mot-clé extern, il ne pourra pas y accéder.

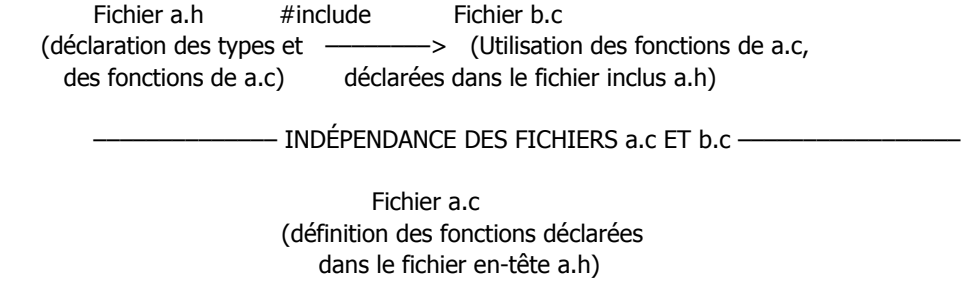
7.5.3. Déclaration des fonctions

Lorsqu'une fonction se trouve définie dans un autre fichier, il est nécessaire de la déclarer. Pour cela, il suffit de donner sa déclaration (le mot-clé extern est également utilisable, mais facultatif) :

```
int factorielle(int);
/*
  factorielle est une fonction attendant comme paramètre
  un entier et renvoyant une valeur entière.
  Elle est définie dans un autre fichier.
 */
```

Il faudra bien faire la distinction entre les fichiers compilés séparément et les fichiers inclus par le préprocesseur. Ces derniers ne sont en effet pas séparés : ils sont compilés avec les fichiers dans lesquels ils sont inclus. Il est donc possible d'inclure du code dans les fichiers d'en-tête.

Les programmes modulaires auront donc typiquement la structure suivante :



Compilation : a.c donne a.o, b.c donne b.o ;
Édition de liens : a.o et b.o donnent le programme exécutable.

7.5.4. Directive d'édition de liens

Le langage C++ donne la possibilité d'appeler des fonctions et d'utiliser des variables qui proviennent d'un module écrit dans un autre langage. Pour permettre cela, il dispose de directives permettant d'indiquer comment l'édition de liens doit être faite. La syntaxe permettant de réaliser ceci utilise le mot-clé extern, avec le nom du langage entre guillemets. Cette directive d'édition de liens doit précéder les déclarations de variables et de données concernées. Si plusieurs variables ou fonctions utilisent la même directive, elles peuvent être regroupées dans un bloc délimité par des accolades, avec la directive d'édition de liens placée juste avant ce bloc. La syntaxe est donc la suivante :

```
extern "langage" [déclaration | {
déclaration
[...]
}]
```

Cependant, les seuls langages qu'une implémentation doit obligatoirement supporter sont les langages "C" et

"C++". Pour les autres langages, aucune norme n'est définie et les directives d'édition de liens sont dépendantes de l'implémentation.

Exemple 7-4. Déclarations utilisables en C et en C++

```
#ifdef __cplusplus
extern "C"
{
#endif
   extern int EntierC;
   int FonctionC(void);
#ifdef __cplusplus
}
#endif
```

Dans l'exemple précédent, la compilation conditionnelle est utilisée pour n'utiliser la directive d'édition de liens que si le code est compilé en C++. Si c'est le cas, la variable EntierC et la fonction FonctionC sont déclarées au compilateur C++ comme étant des objets provenant d'un module C.

Chapitre 8. C++ : la couche objet

La couche objet constitue la plus grande innovation du C++ par rapport au C. Le but de la programmation objet est de permettre une abstraction entre l'implémentation des modules et leur utilisation, apportant ainsi un plus grand confort dans la programmation. Elle s'intègre donc parfaitement dans le cadre de la modularité. Enfin, l'encapsulation des données permet une meilleure protection et donc une plus grande fiabilité des programmes.

8.1. Généralités

Théoriquement, il y a une nette distinction entre les données et les opérations qui leur sont appliquées. En tout cas, les données et le code ne se mélangent pas dans la mémoire de l'ordinateur, sauf cas très particuliers (autoprogrammation, alias pour le chargement des programmes ou des overlays, débogueurs, virus).

Cependant, l'analyse des problèmes à traiter se présente d'une manière plus naturelle si l'on considère les données avec leurs propriétés. Les données constituent les variables, et les propriétés les opérations qu'on peut leur appliquer. De ce point de vue, les données et le code sont logiquement inséparables, même s'ils sont placés en différents endroits de la mémoire de l'ordinateur.

Ces considérations conduisent à la notion d'objet. Un *objet* est un ensemble de données et sur lesquelles des procédures peuvent être appliquées. Ces procédures ou fonctions applicables aux données sont appelées *méthodes*. La programmation d'un objet se fait donc en donnant les données de l'objet et en définissant les procédures qui peuvent lui être appliquées.

Il se peut qu'il y ait plusieurs objets identiques, dont les données ont bien entendu des valeurs différentes, mais qui utilisent le même jeu de méthodes. On dit que ces différents objets appartiennent à la même *classe* d'objet. Une classe constitue donc une sorte de type, et les objets de cette classe en sont des *instances*. La classe définit donc la structure des données, alors appelées *champs* ou *variables d'instances*, que les objets correspondants auront, ainsi que les méthodes de l'objet. À chaque instanciation, une allocation de mémoire est faite pour les données du nouvel objet créé. L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le *constructeur*. Lorsque l'objet est détruit, une autre méthode est appelée : le *destructeur*. L'utilisateur peut définir ses propres constructeurs et destructeurs d'objets si nécessaire.

Comme seules les valeurs des données des différents objets d'une classe diffèrent, les méthodes sont mises en commun pour tous les objets d'une même classe (c'est à dire que les méthodes ne sont pas recopiées). Pour que les méthodes appelées pour un objet sachent quelles données elles doivent traiter, un pointeur sur les données de l'objet en question leur est passé en paramètre. Ce mécanisme est complètement invisible au programmeur en C++.

Nous voyons donc que non seulement la programmation orientée objet est plus logique, mais en plus elle est plus efficace (les méthodes sont mises en commun, les données sont séparées).

Enfin, les données des objets peuvent être *protégées* : c'est à dire que seules les méthodes de l'objet peuvent y accéder. Ce n'est pas une obligation, mais cela accroît la fiabilité des programmes. Si une erreur se produit, seules les méthodes de l'objet doivent être vérifiées. De plus, les méthodes constituent ainsi une interface entre les données de l'objet et l'utilisateur de l'objet (un autre programmeur). Cet utilisateur n'a donc pas à savoir comment les données sont gérées dans l'objet, il ne doit utiliser que les méthodes. Les avantages sont immédiats : il ne risque pas de faire des erreurs de programmation en modifiant les données lui-même, l'objet est réutilisable dans un autre programme parce qu'il a une interface standardisée, et on peut modifier l'implémentation interne de l'objet sans avoir à refaire tout le programme pourvu que les méthodes gardent le même nom et les mêmes paramètres. Cette notion de protection des données et de masquage de l'implémentation interne aux utilisateurs de l'objet constitue ce que l'on appelle l'*encapsulation*. Les avantages de l'encapsulation seront souvent mis en valeur dans la suite au travers d'exemples.

Nous allons entrer maintenant dans le vif du sujet. Ceci permettra de comprendre ces généralités.

8.2. Extension de la notion de type du C

Il faut avant tout savoir que la couche objet n'est pas un simple rajout au langage C, c'est une véritable extension. En effet, les notions qu'elle a apportées ont été intégrées au C, à tel point que le typage des données de C a fusionné avec la notion de classe. Ainsi, les types prédéfinis char, int, double, etc… représentent à présent l'ensemble des propriétés des variables ayant ce type. Ces propriétés constituent la classe de ces variables, et elles sont accessibles par les opérateurs. Par exemple, l'addition est une opération pouvant porter sur des entiers (entre autres) qui renvoie un objet de la classe entier. Par conséquent, les types de base se manipuleront exactement comme des objets. Du point de vue du C++, les utiliser revient déjà à faire de la programmation orientée objet.

De même, le programmeur peut, à l'aide de la notion de classe d'objets, définir de nouveaux types. Ces types comprennent la structure des données représentées par ces types et les opérations qui peuvent leur être appliquées. En fait, le C++ assimile complètement les classes avec les types, et la définition d'un nouveau type se fait donc en définissant la classe des variables de ce type.

8.3. Déclaration de classes en C++

Afin de permettre la définition des méthodes qui peuvent être appliquées aux structures des classes C++, la syntaxe des structures C a été étendue (et simplifiée). Il est à présent possible de définir complètement des méthodes dans la définition de la structure. Cependant il est préférable de la reporter et de ne laisser que leur déclaration dans la structure. En effet, cela accroît la lisibilité, et l'utilisateur ne peut ainsi pas voir l'implémentation de la classe, ni la modifier.

La syntaxe est la suivante :

```
struct Nom
{
  [type champs;
  [type champs;
  [...]]]

  [méthode;
  [méthode;
  [...]]]
};
```

où Nom est le nom de la classe. Elle peut contenir divers champs de divers types.

Les méthodes peuvent être des définitions de fonctions, ou seulement leurs en-têtes. Si on ne donne que leurs en-têtes, on devra les définir plus loin. Pour cela, il faudra spécifier la classe à laquelle elles appartiennent avec la syntaxe suivante :

```
type classe::nom(paramètres)
{
      /* Définition de la méthode. */
}
```

La syntaxe est donc identique à la définition d'une fonction normale, à la différence près que leur nom est précédé du nom de la classe à laquelle elles appartiennent et de deux deux-points (::). Cet opérateur :: est appelé l'*opérateur de résolution de portée*. Il permet, d'une manière générale, de spécifier le bloc auquel l'objet qui le suit appartient. Ainsi, le fait de précéder le nom de la méthode par le nom de la classe permet au compilateur de savoir de quelle classe cette méthode fait partie. Rien n'interdit, en effet, d'avoir des méthodes de mêmes signatures pourvu qu'elles soient dans des classes différentes.

Exemple 8-1. Déclaration de méthode de classe

```
struct Entier
{
  int i;              // Donnée membre de type entier.
```

```
  // Fonction définie à l'intérieur de la classe :
  int lit_i(void)
  {
    return i;
  }

  // Fonction définie à l'extérieur de la classe :
  void ecrit_i(int valeur);
};

void Entier::ecrit_i(int valeur)
{
  i=valeur;
  return ;
}
```

Note: Si la liste des paramètres de la définition de la fonction contient des initialisations supplémentaires à celles qui ont été spécifiées dans la déclaration de la fonction, les deux jeux d'initialisations sont fusionnées et utilisées dans le fichier où la définition de la fonction est placée. Si les initialisations sont redondantes ou contradictoires, le compilateur génère une erreur.

Note: L'opérateur de résolution de portée permet aussi de spécifier le bloc d'instruction d'un objet qui n'appartient à aucune classe. Pour cela, on ne mettra aucun nom avant l'opérateur de résolution de portée. Ainsi, pour accéder à une fonction globale à l'intérieur d'une classe contenant une fonction de même signature, on fera précéder le nom de la fonction globale de cet opérateur.

Exemple 8-2. Opérateur de résolution de portée

```
int valeur(void)        // Fonction globale.
{
  return 0;
}

struct A
{
  int i;

  void fixe(int a)
  {
    i=a;
    return;
  }

  int valeur(void)      // Même signature que la fonction globale.
  {
    return i;
  }

  int global_valeur(void)
  {
    return ::valeur(); // Accède à la fonction globale.
  }
};
```

De même, l'opérateur de résolution de portée permettra d'accéder à une variable globale lorsqu'une autre variable homonyme aura été définie dans le bloc en cours. Par exemple :

```
int i=1;              // Première variable de portée globale
```

```
int main(void)
{
  if (test())
  {
    int i=3;          // Variable homonyme de portée locale.
    int j=2*::i;      // j vaut à présent 2, et non pas 6.
    /* Suite … */
  }

  /* Suite … */

  return 0;
}
```

Les champs d'une classe peuvent être accédés comme des variables normales dans les méthodes de cette classe.

Exemple 8-3. Utilisation des champs d'une classe dans une de ses méthodes

```
struct client
{
  char Nom[21], Prenom[21];     // Définit le client.
  unsigned int Date_Entree;     // Date d'entrée du client
                                // dans la base de donnée.
  int Solde;

  int dans_le_rouge(void)
  {
    return (Solde<0);
  }

  int bon_client(void)          // Le bon client est
                                // un ancien client.
  {
    return (Date_Entree<1993); // Date limite : 1993.
  }
};
```

Dans cet exemple, le client est défini par certaines données. Plusieurs méthodes sont définies dans la classe même.

L'instanciation d'un objet se fait comme celle d'une simple variable :

```
classe objet;
```

Par exemple, si on a une base de données devant contenir 100 clients, on peut faire :

```
client clientele[100];  /* Instancie 100 clients. */
```

On remarquera qu'il est à présent inutile d'utiliser le mot-clé `struct` pour déclarer une variable, contrairement à ce que la syntaxe du C exigeait.

L'accès aux méthodes de la classe se fait comme pour accéder aux champs des structures. On donne le nom de l'objet et le nom du champ ou de la méthode, séparés par un point. Par exemple :

```
/* Relance de tous les mauvais payeurs. */
int i;
for (i=0; i<100; i++)
  if (clientele[i].dans_le_rouge()) relance(i);
```

Dans le cas où les fonctions membres d'une classe sont définies dans la déclaration de cette classe, elles seront implémentées en inline (à moins qu'elles ne soient récursives ou qu'il existe un pointeur sur elles).

Dans le cas où les méthodes ne sont pas définies dans la classe, la déclaration de la classe sera mise dans un fichier d'en-tête, et la définition des méthodes sera reportée dans un fichier C++, qui sera compilé et lié aux autres fichiers utilisant la classe client. Bien entendu, il est toujours possible de déclarer les fonctions membres comme étant des fonctions inline même lorsqu'elles sont définies en dehors de la déclaration de la classe. Pour cela, il faut utiliser le mot-clé inline, et placer le code de ces fonctions dans le fichier d'en-tête ou dans un fichier .inl.

Sans fonctions inline, notre exemple devient :

Fichier client.h :

```
struct client
{
  char Nom[21], Prenom[21];
  unsigned int Date_Entree;
  int Solde;

  int dans_le_rouge(void);
  int bon_client(void);
};

/*
Attention à ne pas oublier le ; à la fin de la classe dans un
fichier .h ! L'erreur apparaîtrait dans tous les fichiers ayant
une ligne #include "client.h" , parce que la compilation a lieu
après l'appel au préprocesseur.
*/
```

Fichier client.cc :

```
/* Inclut la déclaration de la classe : */
#include "client.h"

/* Définit les méthodes de la classe : */

int client::dans_le_rouge(void)
{
  return (Solde<0);
}

int client::bon_client(void)
{
  return (Date_Entree<1993);
}
```

8.4. Encapsulation des données

Les divers champs d'une structure sont accessibles en n'importe quel endroit du programme. Une opération telle que celle-ci est donc faisable :

```
clientele[0].Solde = 25000;
```

Le solde d'un client peut donc être modifié sans passer par une méthode dont ce serait le but. Elle pourrait par exemple vérifier que l'on n'affecte pas un solde supérieur au solde maximal autorisé par le programme (la borne supérieure des valeurs des entiers signés : 32767). Un programme qui ferait :

```
clientele[0].Solde = 32800;
```

espérerait obtenir un solde positif, or il obtiendrait un solde de -12 (valeur en nombre signé du nombre non signé 32800) !

Il est possible d'empêcher l'accès des champs ou de certaines méthodes à toute fonction autre que celles de la classe. Cette opération s'appelle l'encapsulation. Pour la réaliser, utiliser les mots-clés suivants :

- public : les accès sont libres ;
- private : les accès sont autorisés dans les fonctions de la classe seulement ;
- protected : les accès sont autorisés dans les fonctions de la classe et de ses descendantes (voir le paragraphe suivant) seulement. Le mot-clé protected n'est utilisé que dans le cadre de l'héritage des classes. Le paragraphe suivant détaillera ce point.

Pour changer les droits d'accès des champs et des méthodes d'une classe, il faut faire précéder ceux-ci du mot-clé indiquant les droits d'accès suivi de deux points (':'). Par exemple, pour protéger les données relatives au client, on changera simplement la déclaration de la classe en :

```
struct client
{
private:   // Données privées :

  char Nom[21], Prenom[21];
  unsigned int Date_Entree;
  int Solde;
  // Il n'y a pas de méthodes privées.

public:    // Les données et les méthodes publiques :

  // Il n'y a pas de données publiques.
  int dans_le_rouge(void);
  int bon_client(void)
};
```

Par défaut, les classes construites avec struct ont tous leurs membres publics. Il est possible de déclarer une classe dont tous les éléments sont par défaut privés. Pour cela, il suffit d'utiliser le mot-clé class à la place du mot-clé struct.

Exemple 8-4. Utilisation du mot-clé class

```
class client
{
  // private est à présent inutile.

  char Nom[21], Prenom[21];
  unsigned int Date_Entree;
  int Solde;

public:    // Les données et les méthodes publiques.

  int dans_le_rouge(void);
  int bon_client(void);
};
```

Enfin, il existe un dernier type de classe, que je me contenterai de mentionner : les classes *union*. Elles se déclarent avec le mot-clé union comme les classes struct et class. Les données sont, comme pour les unions du C, situées toutes au même emplacement, ce qui fait qu'écrire dans l'une d'entre elle provoque la destruction des autres. Les unions sont très souvent utilisées en programmation système, lorsqu'un polymorphisme physique des

données est nécessaires (c'est à dire lorsqu'elles doivent être interprétées de différentes façons selon le contexte).

> **Note:** Les classes de type union ne peuvent pas avoir de méthodes virtuelles et de membres statiques. Elles ne peuvent pas avoir de classes de base, ni servir de classe de base. Enfin, les unions ne peuvent pas contenir des références, ni des objets dont la classe a un constructeur non trivial, un constructeur de copie non trivial ou un destructeur non trivial. Pour toutes ces notions, voir la suite du chapitre.

8.5. Héritage

L'*héritage* permet de donner à une classe toutes les caractéristiques d'une ou de plusieurs autres classes. Les classes dont elle hérite sont appelées *classes mères*, *classes de base* ou *classes antécédentes*. La classe elle-même est appelée *classe fille*, *classe dérivée* ou *classe descendante*.

Les *propriétés héritées* sont les champs et les méthodes des classes de base.

Pour faire un héritage en C++, il faut faire suivre le nom de la classe fille par la liste des classes mères dans la déclaration avec les restrictions d'accès aux données, chaque élément étant séparé des autres par une virgule. La syntaxe (donnée pour class, identique pour struct et union) est la suivante :

```
class Classe_mere1
{
	/* Contenu de la classe mère 1. */
};

[class Classe_mere2
{
	/* Contenu de la classe mère 2. */
};]

[…]

class Classe_fille : public|protected|private Classe_mere1
[, public|protected|private Classe_mere2 […]]
{
	/* Définition de la classe fille. */
};
```

Dans cette syntaxe, Classe_fille hérite de la Classe_mere1, et des Classe_mere2, etc… si elles sont présentes.

La signification des mots-clés private, protected et public dans l'héritage est récapitulée dans le tableau suivant :

Tableau 8-1. Droits d'accès sur les membres hérités

		mot-clé utilisé pour l'héritage		
Accès aux données		**public**	**protected**	**private**
mot-clé utilisé	public	public	protected	private
pour les champs	protected	protected	protected	private

et les méthodes	private	interdit	interdit	interdit

Ainsi, les données publiques d'une classe mère deviennent soit publiques, soit protégées, soit privées selon que la classe fille hérite en public, protégé ou en privé. Les données privées de la classe mère sont toujours inaccessibles, et les données protégées deviennent soit protégées, soit privées.

Il est possible d'omettre les mots-clés public, protected et private dans la syntaxe de l'héritage. Le compilateur utilise un type d'héritage par défaut dans ce cas. Les classes de type struct utilisent l'héritage public par défaut et les classes de type class utilisent le mot-clé private par défaut.

Exemple 8-5. Héritage public, privé et protégé

```
class Emplacement
{
protected:
  int x, y;              // Données ne pouvant être accédées
                         // que par les classes filles.

public:
  void Change(int, int);  // Méthode toujours accessible.
};

void Emplacement::Change(int i, int j)
{
  x = i;
  y = j;
  return;
}

class Point : public Emplacement
{
protected:
  unsigned int couleur;   // Donnée accessible
                          // aux classes filles.

public:
  void SetColor(unsigned int);
};

void Point::SetColor(unsigned int NewColor)
{
  couleur = NewColor;     // Définit la couleur.
  return;
}
```

Si une classe Cercle doit hériter de deux classes mères, par exemple Emplacement et Forme, sa déclaration aura la forme suivante :

```
class Cercle : public Emplacement, public Forme
{
  /*
     Définition de la classe Cercle. Cette classe hérite
     des données publiques et protégées des classes Emplacement
     et Forme.
   */
};
```

Il est possible de redéfinir les fonctions et les données des classes de base dans une classe dérivée. Par exemple, si une classe B dérive de la classe A, et que toutes deux contiennent une donnée *d*, les instances de la classe B utiliseront la donnée *d* de la classe B et les instances de la classe A utiliseront la donnée *d* de la classe A. Cependant, les objets de classe B contiendront également un sous-objet, lui-même instance de la classe de base

A. Par conséquent, ils contiendront la donnée *d* de la classe A, mais cette dernière sera cachée par la donnée *d* de la classe la plus dérivée, à savoir la classe B.

Ce mécanisme est général : quand une classe dérivée redéfinit un membre d'une classe de base, ce membre est caché et on ne peut plus accéder qu'au membre de la redéfinition (celui de la classe dérivée). Cependant, il est possible d'accéder aux données cachées si l'on connaît leur classe, pour cela, il faut nommer le membre complètement à l'aide de l'opérateur de résolution de portée (::). Le nom complet d'un membre est constitué du nom de sa classe suivi de l'opérateur de résolution de portée, suivis du nom du membre :

classe::membre

Exemple 8-6. Opérateur de résolution de portée et membre de classes de base

```
struct Base
{
  int i;
};

struct Derivee : public Base
{
  int i;
  int LitBase(void);
};

int Derivee::LitBase(void)
{
  return Base::i; // Renvoie la valeur i de la classe de base.
}

int main(void)
{
  Derivee D;
  D.i=1;          // Accède à l'entier i de la classe Derivee.
  D.Base::i=2;    // Accède à l'entier i de la classe Base.
  return 0;
}
```

8.6. Classes virtuelles

Supposons à présent qu'une classe D hérite de deux classes mères, les classes B et C. Supposons également que ces deux classes héritent d'une classe mère commune appelée classe A. On a l'arbre " généalogique " suivant :

On sait que B et C héritent des données et des méthodes publiques et protégées de A. De même, D hérite des données de B et C, et par leur intermédiaire des données de A. Il se pose donc le problème suivant : quelles sont les données que l'on doit utiliser quand on référence les champs de A ? Celles de B ou celles de C ? On peut accéder aux deux sous-objets de classe A en spécifiant le chemin à suivre dans l'arbre généalogique à l'aide de l'opérateur de résolution de portée. Cependant, ceci n'est ni pratique ni efficace, et en général, on s'attend à ce

qu'une seule copie de A apparaisse dans D.

Le problème est résolu en déclarant *virtuelle* la classe de base commune dans la donnée de l'héritage pour les classes filles. Les données de la classe de base ne seront alors plus dupliquées. Pour déclarer une classe mère comme une classe virtuelle, il faut faire précéder son nom du mot-clé virtual dans l'héritage des classes filles.

Exemple 8-7. Classes virtuelles

```
class A
{
protected:
  int Donnee;        // La donnée de la classe de base.
};

// Héritage de la classe A, virtuelle :
class B : virtual public A
{
protected:
  int Valeur_B;      // Autre donnée que "Donnee" (héritée).
};

// A est toujours virtuelle :
class C : virtual public A
{
protected:
  int valeur_C;      // Autre donnée
                     // ("Donnee" est acquise par héritage).
};

class D : public B, public C   // Ici, Donnee n'est pas dupliqué.
{
  /* Définition de la classe D. */
};
```

> **Note:** Normalement, l'héritage est réalisé par le compilateur par aggrégation de la structure de données des classes de base dans la structure de données de la classe dérivée. Pour les classes virtuelles, ce n'est en général pas le cas, puisque le compilateur doit assurer l'unicité des données héritées de ces classes, même en cas d'héritage multiple. Par conséquent, certaines restrictions d'usage s'appliquent sur les classes virtuelles.
>
> Premièrement, il est impossible de transtyper directement un pointeur sur un objet d'une classe de base virtuelle en un pointeur sur un objet d'une de ses classes dérivées. Il faut impérativement utiliser l'opérateur de transtypage dynamique dynamic_cast. Cet opérateur sera décrit à le Chapitre 10.
>
> Deuxièmement, chaque classe dérivée directement ou indirectement d'une classe virtuelle doit en appeler le constructeur explicitement dans son constructeur, si elle ne désire pas que le constructeur par défaut soit appelé. En effet, elle ne peut pas se fier au fait qu'une autre de ses classes de base, elle-même dérivée de la classe de base virtuelle, appelle un constructeur spécifique, car il est possible que plusieurs classes de base cherchent à initialiser chacune un objet commun hérité de la classe virtuelle. Pour reprendre l'exemple donné ci-dessus, si les classes B et C appellaient toutes les deux le constructeur de la classe virtuelle A, et que la classe D appellait elle-même les constructeurs de B et C, le sous-objet hérité de A serait construit plusieurs fois. Pour éviter ceci, le compilateur ignore purement et simplement les appels au constructeur des classes de bases virtuelles dans les classes de base dérivées. Il faut donc systématiquement le spécifier, à chaque niveau de la hiérarchie de classe. La notion de constructeur sera vue dans la Section 8.8

8.7. Fonctions et classes amies

Il est parfois nécessaire d'avoir des fonctions qui ont un accès illimité aux champs d'une classe. En général, l'emploi de telles fonctions traduit un manque d'analyse dans la hiérarchie des classes, mais pas toujours. Elles restent donc nécessaires malgré tout.

De telles fonctions sont appelées des *fonctions amies*. Pour qu'une fonction soit amie d'une classe, il faut qu'elle soit déclarée dans la classe avec le mot-clé friend.

Il est également possible de faire une classe amie d'une autre classe, mais dans ce cas, cette classe devrait peut être être une classe fille. L'utilisation des classes amies peut traduire un défaut de conception.

8.7.1. Fonctions amies

Les fonctions amies se déclarent en faisant précéder la déclaration de la fonction classique du mot-clé friend à l'intérieur de la déclaration de la classe. Les fonctions amies ne sont pas des méthodes de la classe cependant (ceci n'aurait pas de sens puisque les méthodes ont déjà accès aux membres de la classe).

Exemple 8-8. Fonctions amies

```
class A
{
  int a;                    // Une donnée privée.
  friend void ecrit_a(int i);   // Une fonction amie.
};

A essai;

void ecrit_a(int i)
{
  essai.a=i;          // Initialise a.
  return;
}
```

Il est possible de déclarer amie une fonction d'une autre classe, en précisant son nom complet à l'aide de l'opérateur de résolution de portée.

8.7.2. Classes amies

Pour rendre toutes les méthodes d'une classe amies d'une autre classe, il suffit de déclarer la classe complète comme étant amie. Pour cela, il faut encore une fois utiliser le mot-clé friend avant la déclaration de la classe, à l'intérieur de la classe cible. Cette fois encore, la classe amie déclarée ne sera pas une sous-classe de la classe cible, mais bien une classe de portée globale. On peut cependant déclarer une classe amie qui fait partie de la classe dont elle est amie, en précisant le nom complet à l'aide de l'opérateur de résolution de portée.

> **Note:** Le fait, pour une classe, d'appartenir à une autre classe, ne lui donne aucun droit particulier sur les donnée de la classe hôte. Si la classe contenue doit utiliser les données private ou protected de la classe hôte, elle doit être déclarée amie de celle-ci.

Exemple 8-9. Classe amie

```
#include <stdio.h>

class Hote
{
  friend class Amie;  // Toutes les méthodes de Amie sont amies.

  int i;              // Donnée privée de la classe Hote.

public:
  Hote(void)
  {
    i=0;
    return ;
  }
};
```

```
Hote h;

class Amie
{
public:
  void print_hote(void)
  {
    printf("%d\n", h.i); // Accède à la donnée privée de h.
    return ;
  }
};

int main(void)
{
  Amie a;
  a.print_hote();
  return 0;
}
```

On remarquera plusieurs choses importantes. Premièrement, l'amitié n'est pas transitive. Cela signifie que les amis des amis ne sont pas des amis. Une classe A amie d'une classe B, elle-même amie d'une classe C, n'est pas amie de la classe C par défaut. Il faut la déclarer amie explicitement si on désire qu'elle le soit. Deuxièmement, les amis ne sont pas hérités. Ainsi, si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut. Encore une fois, il faut la déclarer amie explicitement. Ces remarques s'appliquent également aux fonctions amies (une fonction amie d'une classe A amie d'une classe B n'est pas amie de la classe B, ni des classes dérivées de A).

8.8. Constructeurs et destructeurs

Le *constructeur* et le *destructeur* sont deux méthodes particulières qui sont appelées à la création et à la destruction d'un objet respectivement. Toute classe a un constructeur et un destructeur par défaut, fournis par le compilateur (ils ne font absolument rien). Il est souvent nécessaire de les redéfinir afin de gérer certaines actions qui doivent avoir lieu lors de la création d'un objet et de leur destruction. Par exemple, si l'objet doit contenir des variables allouées dynamiquement, il faut leur réserver de la mémoire à la création de l'objet, ou au moins mettre les pointeurs correspondant à NULL. À la destruction de l'objet, il convient de restituer la mémoire allouée, s'il en a été allouée. On peut trouver bien d'autre situation où une phase d'initialisation et une phase de terminaison sont nécessaires.

8.8.1. Déclaration des constructeurs et des destructeurs

Le constructeur se définit comme une méthode normale. Cependant, pour que le compilateur puisse la reconnaître en tant que constructeur, les deux conditions suivantes doivent être vérifiées :

- elle doit porter le même nom que la classe ;
- elle ne doit avoir aucun type, *pas même le type void*.

Le destructeur doit également respecter ces règles. Pour le différencier du constructeur, son nom sera toujours précédé du signe tilde ('~').

Un constructeur est appelé automatiquement lors de l'instanciation de l'objet. Le destructeur est appelé automatiquement lors de sa destruction. Cette destruction a lieu lors de la sortie du bloc de portée courante pour les objets de classe de stockage auto. Pour les objets alloués dynamiquement, le constructeur et le destructeur sont appelés automatiquement par les expressions qui utilisent les opérateurs new, new[], delete et delete[]. C'est pour cela qu'il est recommandé de les utiliser à la place des fonctions malloc et free du C pour faire une création dynamique d'objets. De plus, il ne faut pas utiliser delete ou delete[] sur des pointeurs de type void, car il n'existe pas d'objets de type void. Le compilateur ne peut donc pas déterminer quel est le destructeur à appeler avec ce type de pointeurs.

Le constructeur est appelé après l'allocation de la mémoire de l'objet et le destructeur est appelé avant la libération de cette mémoire. La gestion de l'allocation dynamique de mémoire avec les classes est ainsi simplifiée.

Dans le cas des tableaux, l'ordre de construction est celui des adresses croissantes, et l'ordre de destruction est celui des adresses décroissantes. C'est dans cet ordre que les constructeurs et destructeurs de chaque élément du tableau sont appelés.

Les constructeurs pourront avoir des paramètres. Ils peuvent donc être surchargés, mais pas les destructeurs (ceci signifie qu'en pratique, on connaît le contexte dans lequel un objet est créé, mais qu'on ne peut pas connaître le contexte dans lequel il est détruit : il ne peut donc y avoir qu'un seul destructeur).

Exemple 8-10. Constructeurs et destructeurs

```
class chaine   // Implémente une chaîne de caractère.
{
  char * s;   // Le pointeur sur la chaîne de caractère.

public:
  chaine(void);          // Le constructeur par défaut.
  chaine(unsigned int);  // Le constructeur. Il n'a pas de type.
  ~chaine(void);         // Le destructeur.
};

chaine::chaine(void)
{
  s=NULL;               // La chaîne est initialisée avec
  return ;              // le pointeur nul.
}

chaine::chaine(unsigned int Taille)
{
  s = new char[Taille+1]; // Alloue de la mémoire pour la chaîne.
  s[0]='\0';              // Initialise la chaîne à "".
  return;
}

chaine::~chaine(void)
{
  if (s!=NULL) delete s;   // Restitue la mémoire utilisée si
                           // nécessaire.
  return;
}
```

Pour passer les paramètres au constructeur, on donne la liste des paramètres entre parenthèses juste après le nom de l'objet lors de son instanciation :

```
chaine s1;        // Instancie une chaîne de caractères
                  // non initialisée.
chaine s2(200);   // Instancie une chaîne de caractères
                  // de 200 caractères.
```

Les constructeurs devront parfois effectuer des tâches plus compliquées que celles données dans cet exemple. En général, ils peuvent faire toutes les opérations faisables dans une méthode normale, sauf utiliser les données non initialisées bien entendu. En particulier, les données des sous-objets d'un objet ne sont pas initialisées tant que les constructeurs des classes de base ne sont pas appelés. C'est pour cela qu'il faut toujours appeler les constructeurs des classes de base avant d'exécuter le constructeur de la classe en cours d'instanciation.

Il faut spécifier ces constructeurs, sinon le compilateur appellera, par défaut, les constructeurs des classes mères qui prennent void pour paramètre (et si ceux-ci ne sont pas définis, les constructeurs par défaut, qui ne font

rien).

Comment appeler les constructeurs et les destructeurs des classes mères lors de l'instanciation et de la destruction d'une classe dérivée ? Le compilateur ne peut en effet pas savoir quel constructeur il faut appeler parmi les différents constructeurs surchargés potentiellement présents… Pour appeler un autre constructeur d'une classe mère que le constructeur ne prenant pas de paramètres, il suffit de donner le nom de ce constructeur avec ses paramètres après le nom du constructeur de la classe fille, séparés par deux points (':').

En revanche, il est inutile de préciser le destructeur à appeler, puisque celui-ci est unique. Le programmeur ne doit donc pas appeler lui-même les destructeurs des classes mères, le langage s'en charge.

Exemple 8-11. Appel du constructeur des classes de base

```
/* Déclaration de la classe mère. */

class Mere
{
  int m_i;
public:
  Mere(int);
  ~Mere(void);
};

/* Définition du constructeur de la classe mère. */

Mere::Mere(int i)
{
  m_i=i;
  printf("Exécution du constructeur de la classe mère.\n");
  return;
}

/* Définition du destructeur de la classe mère. */

Mere::~Mere(void)
{
  printf("Exécution du destructeur de la classe mère.\n");
  return;
}

/* Déclaration de la classe fille. */

class Fille : public Mere
{
public:
  Fille(void);
  ~Fille(void);
};

/* Définition du constructeur de la classe fille
  avec appel du constructeur de la classe mère. */

Fille::Fille(void) : Mere(2)
{
  printf("Exécution du constructeur de la classe fille.\n");
  return;
}

/* Définition du destructeur de la classe fille
  avec appel automatique du destructeur de la classe mère. */

Fille::~Fille(void)
{
  printf("Exécution du destructeur de la classe fille.\n");
  return;
}
```

Lors de l'instanciation d'un objet de la classe fille, le programme affichera dans l'ordre les messages suivants :

```
Exécution du constructeur de la classe mère.
Exécution du constructeur de la classe fille.
```

et lors de la destruction de l'objet :

```
Exécution du destructeur de la classe fille.
Exécution du destructeur de la classe mère.
```

Si l'on n'avait pas précisé que le constructeur à appeler pour la classe Mere était le constructeur prenant un entier en paramètre, le constructeur par défaut aurait été appelé et seul le message de construction de la classe Fille aurait été affiché. Par ailleurs, on notera que l'ordre d'appel est important.

> **Note:** Afin d'éviter l'utilisation des données non initialisées de l'objet le plus dérivé dans une hiérarchie pendant la construction de ses sous-objets par l'intermédiaire des fonctions virtuelles, le mécanisme des fonctions virtuelles est désactivé dans les constructeurs (voyez la Section 8.13 pour plus de détails sur les fonctions virtuelles). Ce problème survient parce que pendant l'exécution des constructeurs des classes de base, l'objet de la classe en cours d'instanciation n'a pas encore été initialisé, et malgré cela, une fonction virtuelle aurait pu utiliser une donnée de cet objet.
>
> Une fonction virtuelle peut donc toujours être appelée dans un constructeur, mais la fonction effectivement appelée est celle de la classe du sous-objet en cours de construction : pas celle de la classe de l'objet complet. Ainsi, si une classe A hérite d'une classe B et qu'elles ont toutes les deux une fonction virtuelle `f`, l'appel de `f` dans le constructeur de B utilisera la fonction `f` de B, pas celle de A (même si l'objet que l'on instancie est de classe A).
>
> Les constructeurs des classes de base virtuelles doivent être appelés par chaque classe qui en dérive, que cette dérivation soit directe ou indirecte. En effet, les classes de base virtuelles subissent un traitement particulier qui assure l'unicité de leurs données dans toutes leurs classes dérivées. Les classes dérivées ne peuvent donc pas se reposer sur leurs classes de base pour appeler le constructeur des classes virtuelles, car il peut y avoir plusieurs classes de bases qui dérivent d'une même classe virtuelle, et ceci supposerait que le constructeur de cette dernière classe serait appelé plusieurs fois. Chaque classe doit donc prendre en charge la construction des sous-objets des classes de base virtuelles dont il hérite.

8.8.2. Constructeurs de copie

Il faudra parfois créer un constructeur de copie. Le but de ce type de constructeur est d'initialiser un objet lors de son instanciation à partir d'un autre objet. Toute classe dispose d'un constructeur de copie par défaut, dont le seul but est de recopier les champs de l'objet à recopier un à un dans les champs de l'objet à instancier.

Le constructeur par défaut ne suffira pas toujours, c'est pour cela que le programmeur devra parfois en fournir un. Ce sera notamment le cas lorsque certaines données des objets auront été allouées. Une copie brutale des champs d'un objet dans un autre ne ferait que recopier les pointeurs, pas les données pointées. Ainsi, la modification de ces données pour un objet entraînera la modification pour les données de l'autre objet.

La définition des constructeurs de copie se fait comme celle des constructeurs normaux. Le nom doit être celui de la classe, et il ne doit y avoir aucun type. Dans la liste des paramètres cependant, il devra toujours y avoir une référence sur l'objet à copier.

Pour la classe chaine définie ci-dessus, il faut un constructeur de copie. Celui-ci pourra être déclaré de la façon suivante :

```
chaine(const chaine &Source);
```

où Source est l'objet à copier.

Si l'on rajoute la donnée membre *Taille* dans la déclaration de la classe, la définition de ce constructeur peut être :

```
chaine::chaine(const chaine &Source)
{
  int i = 0;                  // Compteur de caractères.
  Taille = Source.Taille;
  s = new char[Taille + 1];    // Effectue l'allocation.
  while ((s[i]=Source.s[i])!='\0') i=i+1;   // Recopie
                              // la chaîne de caractère source
  return;
}
```

Le constructeur de copie est appelé dans toute instanciation avec initialisation, comme celles qui suivent :

```
chaine s2(s1);
chaine s2 = s1;
```

Dans les deux exemples, c'est le constructeur de copie qui est appelé. En particulier, à la deuxième ligne, le constructeur normal n'est pas appelé et aucune affectation entre objets n'a lieu.

8.8.3. Utilisation des constructeurs dans les transtypages

Les constructeurs sont utilisés dans les conversions de type dans lesquelles le type cible est celui de la classe du constructeur. Ces conversions peuvent être soit implicites (dans une expression), soit explicite (à l'aide d'un transtypage). Par défaut, les conversions implicites sont légales, pourvu qu'il existe un constructeur dont le premier paramètre a le même type que l'objet source. Par exemple, la classe Entier suivante :

```
class Entier
{
  int i;
public:
  Entier(int j)
  {
    i=j;
    return ;
  }
};
```

dispose d'un constructeur de transtypage pour les entiers. Les expressions suivantes :

```
int j=2;
Entier e1, e2=j;
e1=j;
```

sont donc légales, la valeur entière située à la droite de l'expression étant convertie implicitement en un objet du type de la classe Entier.

Si, pour une raison quelconque, ce comportement n'est pas souhaitable, on peut forcer le compilateur à n'accepter que les conversions explicites (à l'aide de transtypage). Pour cela, il suffit de placer le mot-clé explicit avant la déclaration du constructeur.

Exemple 8-12. Mot-clé explicit

```
class Entier
{
  int i;
public:
  explicit Entier(int j)
  {
    i=j;
```

```
    return ;
  }
};
```

À présent, l'expression donnée ci-dessus n'est plus valide. Si l'on veut convertir l'entier en objet de classe Entier, on est maintenant forcé d'utiliser un transtypage explicite (ce qui donne l'origine du mot-clé). L'exemple précédent donne alors :

```
int j=2;
Entier e1, e2=(Entier) j;
e1=(Entier) j;
```

8.9. Pointeur this

Nous allons à présent voir comment les fonctions membres, qui appartiennent à la classe, peuvent accéder aux données d'un objet, qui est une instance de cette classe. Ceci est indispensable pour bien comprendre les paragraphes suivants.

À chaque appel d'une fonction membre, le compilateur passe en paramètre un pointeur sur les données de l'objet implicitement. Ce paramètre est le premier paramètre de la fonction. Ce mécanisme est complètement invisible au programmeur, et nous ne nous attarderons pas dessus.

En revanche, il faut savoir que le pointeur sur l'objet est accessible à l'intérieur de la fonction membre. Il porte le nom " this ". Par conséquent, *this représente l'objet lui-même. Nous verrons une utilisation de this dans le paragraphe suivant (redéfinition des opérateurs).

this est un pointeur constant, c'est à dire qu'on ne peut pas le modifier (il est donc impossible de faire des opérations arithmétiques dessus). Ceci est tout à fait normal, puisque le faire reviendrait à sortir de l'objet en cours (celui pour lequel la méthode en cours d'exécution travaille).

Il est possible de transformer ce pointeur constant en un pointeur constant sur des données constantes pour chaque fonction membre. Le pointeur ne peut toujours pas être modifié, et les données de l'objet ne peuvent pas être modifiées non plus. L'objet est donc considéré par la fonction membre concernée comme un objet constant. Ceci revient à dire que la fonction membre s'interdit la modification des données de l'objet. On parvient à ce résultat en ajoutant le mot-clé const à la suite de l'en-tête de la fonction membre. Par exemple :

```
class Entier
{
  int i;
public:
  int lit(void) const;
};

int Entier::lit(void) const
{
  return i;
}
```

Dans la fonction membre lit, il est impossible de modifier l'objet. On ne peut donc accéder qu'en lecture seule à i. Nous verrons une application de cette possibilité dans la Section 8.15.

Il est à noter qu'une méthode qui n'est pas déclarée comme étant const modifie a priori les données de l'objet sur lequel elle travaille. Si elle est appelée sur un objet déclaré const, une erreur de compilation se produit donc. Ce comportement est normal. Si la méthode incriminée ne modifie pas réellement l'objet, on devra donc toujours

la déclarer const pour pouvoir laisser le choix de déclarer const ou non un objet.

> **Note:** Le mot-clé const n'intervient pas dans la signature des fonctions en général lorsqu'il s'applique aux paramètres (tout paramètre déclaré const perd sa qualification dans la signature). En revanche, il intervient dans la signature d'une fonction membre quand il s'applique à cette fonction (ou, plus précisément, à l'objet pointé par this). Il est donc possible de déclarer deux fonctions membres acceptant les mêmes paramètres, dont une seule est const. Lors de l'appel, la détermination de la fonction à utiliser dépendra de la nature de l'objet sur lequel elle doit s'appliquer. Si l'objet est const, la méthode appelée sera celle qui est const.

8.10. Données et fonctions membres statiques

Nous allons voir dans ce paragraphe l'emploi du mot-clé static dans les classes. Ce mot-clé intervient pour caractériser les données membres statiques des classes, les fonctions membres statiques des classes, et les données statiques des fonctions membres.

8.10.1. Données membres statiques

Une classe peut contenir des données membres statiques. Ces données sont soit des données membres propres à la classe, soit des données locales aux fonctions membres de la classe qui les ont déclarées avec le mot-clé static. Dans tous les cas, elles appartiennent à la classe, et non pas aux objets de cette classe. Elles sont donc communes à tous ces objets.

Il est impossible d'initialiser les données d'une classe dans le constructeur de la classe, car le constructeur initialise les données des nouveaux objets, et les données statiques ne sont pas spécifiques à un objet particulier. L'initialisation des données statiques doit donc se faire lors de leur définition, qui se fait en dehors de la déclaration de la classe. Pour préciser la classe à laquelle les données ainsi définies appartiennent, on devra utiliser l'opérateur de résolution de portée (::).

Exemple 8-13. Donnée membre statique

```
class test
{
  static int i;       // Déclaration dans la classe.
  …
};

int test::i=3;        // Initialisation en dehors de la classe.
```

La variable a::i sera partagée par tous les objets de classe test, et sa valeur initiale est 3.

Les variables statiques des fonctions membres doivent être initialisées à l'intérieur des fonctions membres. Elles appartiennent également à la classe, et non pas aux objets, de plus, leur portée est réduite à celle du bloc dans lequel elles ont été déclarées. Ainsi, le code suivant :

```
#include <stdio.h>

class test
{
public:
  int n(void);
};

int test::n(void)
{
  static int compte=0;
  return compte++;
}

int main(void)
{
  test objet1, objet2;
  printf("%d ", objet1.n());   // Affiche 0
  printf("%d\n", objet2.n());  // Affiche 1
  return 0;
```

```
}
```

affichera 0 et 1, parce que la variable statique compte est la même pour les deux objets.

8.10.2. Fonctions membres statiques

Les classes peuvent également contenir des fonctions membres statiques. Ceci peut surprendre à première vue, puisque les fonctions membres appartiennent déjà à la classe, c'est à dire à tous les objets. En fait, cela signifie que ces fonctions membres ne recevront pas le pointeur sur l'objet this, comme c'est le cas pour les autres fonctions membres. Par conséquent, elles ne pourront accéder qu'aux données statiques de l'objet.

Exemple 8-14. Fonction membre statique

```
class Entier
{
  int i;
  static int j;
public:
  static int get_value(void);
};

int Entier::j=0;

int Entier::get_value(void)
{
  j=1;        // Légal.
  return i;   // ERREUR ! get_value ne peut pas accéder à i.
}
```

La fonction get_value de l'exemple ci-dessus ne peut pas accéder à la donnée membre non statique *i*, parce qu'elle ne travaille sur aucun objet. Son champ d'action est uniquement la classe Entier. En revanche, elle peut modifier la variable statique *j*, puisque celle-ci appartient à la classe Entier et non aux objets de cette classe.

L'appel des fonctions membre statiques se fait exactement comme celui des fonctions membres non statiques, en spécifiant l'identificateur d'un des objets de la classe et le nom de la fonction membre, séparés par un point. Cependant, comme les fonctions membres ne travaillent pas sur les objets des classes mais plutôt sur les classes elles-mêmes, la présence de l'objet lors de l'appel est facultatif. On peut donc se contenter d'appeler une fonction statique en qualifiant son nom du nom de la classe à laquelle elle appartient à l'aide de l'opérateur de résolution de portée.

Exemple 8-15. Appel de fonction membre statique

```
class Entier
{
  static int i;
public:
  static int get_value(void);
};

int Entier::i=3;

int Entier::get_value(void)
{
  return i;
}

int main(void)
{
```

```
  // Appelle la fonction statique get_value :
  int resultat=Entier::get_value();
  return 0;
}
```

8.11. Redéfinition des opérateurs

On a vu précédemment que les opérateurs ne se différencient des fonctions que syntaxiquement, pas logiquement. D'ailleurs, le compilateur traite un appel à un opérateur comme un appel à une fonction. Le C++ dispose donc d'une syntaxe qui permet de définir les opérateurs pour les classes comme des fonctions membres classiques.

De plus, il est possible de surcharger des fonctions pourvu que toutes les fonctions portant le même nom aient une signature différente. Comme les opérateurs sont des fonctions, il est possible de les surcharger. Le C++ permet donc de surcharger les opérateurs du langage comme on surcharge les fonctions, même en dehors d'une classe.

Nous allons voir ces deux syntaxes dans les sections suivantes.

8.11.1. Définition des opérateurs interne

Une première méthode pour redéfinir les opérateurs consiste à les considérer comme des méthodes normales de la classe sur laquelle ils s'appliquent. Le nom de ces méthodes est donné par le mot-clé operator, suivi de l'opérateur à redéfinir. Le type de la fonction de l'opérateur est le type du résultat donné par l'opération, et les paramètres, donnés entre parenthèses, sont les opérandes. Les opérateurs de ce type sont appelés opérateurs internes, parce qu'ils sont déclarés à l'intérieur de la classe.

Voici la syntaxe :

type operatorOp(paramètres)

l'écriture

A Op B

se traduisant par :

A.operatorOp(B)

Avec cette syntaxe, le premier opérande est toujours l'objet auquel cette fonction s'applique. Ainsi, les paramètres de la fonction opérateur sont le deuxième opérande et les suivants.

Les fonctions opérateurs devront souvent renvoyer une valeur du type de la classe des opérandes (ce n'est pas une nécessité cependant). Il faudra donc soit renvoyer un objet qui aura été créé temporairement dans la fonction opérateur, soit l'objet caractérisé par le premier opérande de l'opérateur (c'est à dire l'objet lui-même). Ceci est faisable grâce au pointeur this.

Dans le cas où la fonction renvoie un objet créé temporairement (c'est à dire avec la classe de stockage auto), cet objet est détruit à la sortie de la fonction. Cependant, l'objet retourné par la fonction existe : il a été recopié dans la valeur de retour de la fonction. Cet objet recopié sera détruit par le compilateur une fois qu'il aura été utilisé par l'instruction qui a appelé la fonction. Le programmeur n'a pas à s'en occuper. Les bons compilateurs sont même capables d'éviter cette copie, ce qui accroît sérieusement les performances avec les objets volumineux.

Par exemple, la classe suivante implémente les nombres complexes avec leurs opérations de base :

Exemple 8-16. Redéfinition des opérateurs

```
class complexe
{
  float x, y;   // Les parties réelles et imaginaires.

public:
  void fait(float, float);    // Fonction permettant de créer
                              // un complexe.
```

```
  float re(void) const;       // Fonctions permettant de lire les
  float im(void) const;       // parties réelles et imaginaires.
  // Les opérations de base :
  complexe operator+(const complexe &) const;
  complexe operator-(const complexe &) const;
  complexe operator*(const complexe &) const;
  complexe operator/(const complexe &) const;
};

void complexe::fait(float a, float b)
{
  x = a;
  y = b;
  return;
}

float complexe::re(void) const
{
  return x;
}

float complexe::im(void) const
{
  return y;
}

complexe complexe::operator+(const complexe &c) const
{
  complexe tmp=c;    // Construit un complexe temporaire.
                     // Le constructeur de copie par défaut
                     // est appelé.
  tmp.x = tmp.x + x; // Ajoute les parties réelles et imaginaires
  tmp.y = tmp.y + y; // du complexe en cours de traitement
                     // à celles du complexe tmp.
  return tmp;
}

complexe complexe::operator-(const complexe &c) const
{
  complexe tmp=c;
  tmp.x = tmp.x - x;
  tmp.y = tmp.y - y;
  return tmp;
}

complexe complexe::operator*(const complexe &c) const
{
  complexe tmp=c;
  tmp.x = tmp.x * x - tmp.y * y;
  tmp.y = tmp.x * y + tmp.y *x;
  return tmp;
}

complexe complexe::operator/(const complexe &c) const
{
  complexe tmp=c;
  float inv = tmp.x * tmp.x + tmp.y * tmp.y;
```

```
  tmp.x = tmp.x / inv;
  tmp.y = - tmp.y / inv;  // tmp contient l'inverse de c.
  return (*this) * tmp;   // Calcule x * (1/y).
}
```

Le dernier opérateur fournit un exemple d'utilisation du pointeur this. Il sera également employé lors de la redéfinition des opérateurs =, +=, -=, etc… qui renvoient toujours l'objet en cours. Ils devront donc tous se terminer par :

```
return *this;
```

Par exemple, l'opérateur = pour les complexes se définit comme suit :

```
complexe &complexe::operator=(const complexe &c)
{
  x = c.x;
  y = c.y;
  return *this;
}
```

8.11.2. Surcharge des opérateurs externes

Une deuxième possibilité nous est offerte par le langage pour redéfinir les opérateurs. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci, par surcharge d'un opérateur prédéfini. Il s'agit donc d'opérateurs externes cette fois.

Cette redéfinition se fait donc par surcharge des opérateurs standards, comme on surcharge les fonctions normales. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètre : il n'y aura pas de paramètres implicite (le pointeur this n'est pas passé en paramètre).

La syntaxe est la suivante :

```
type operatorOp(opérandes)
```

où opérandes est la liste complète des opérandes.

Cette syntaxe permet d'implémenter les opérateurs pour lesquels l'opérande de gauche n'est pas une classe définie par l'utilisateur (par exemple si c'est un type prédéfini). En effet, on ne peut pas définir l'opérateur à l'intérieur de la classe dans ce cas, puisque la classe du premier opérande est déjà définie.

Par exemple, si l'on veut implémenter la multiplication par un scalaire à gauche pour la classe complexe, on devra procéder comme suit :

```
complexe operator*(float k, const complexe &c)
{
  complexe tmp;
  tmp.fait(c.re()*k,c.im()*k);
  return tmp;
}
```

ce qui permettra d'écrire des expressions du type :

```
complexe c1, c2;
float r;
…
c1 = r*c2;
```

La première syntaxe ne permettait pas de le faire, car il aurait fallu surcharger l'opérateur de multiplication de la classe float, mais celle ci est définie par le langage. La deuxième syntaxe est utilisable partout où la première l'est, mais elle est différente. En effet, les opérateurs ainsi définis ne font pas partie de la classe que l'on est en train de faire. Ceci a une conséquence majeure : on ne peut pas accéder aux données non publiques de la classe

dans la définition de ces opérateurs. La solution consiste à fournir toutes les fonctions membres permettant l'accès à ces données, ou bien à déclarer l'opérateur comme une fonction amie de la classe.

Exemple 8-17. Surcharge d'opérateur externe

```
// Opérateur appartenant à la classe complexe :

complexe complexe::operator*(float k) const
{
  complexe tmp=*this;
  tmp.x=tmp.x*k;
  tmp.y=tmp.y*k;
  return tmp;
}

// Opérateur externe, il n'appartient pas à la classe complexe :
complexe operator*(const complexe &c, float k)
{
  complexe tmp;
  tmp.fait(c.re()*k,c.im()*k);
  return tmp;
}
```

On dispose donc de deux possibilités pour redéfinir les opérateurs classiques.

Les seuls opérateurs qui ne peuvent pas être redéfinis sont les suivants :

::
.
.*
?:
sizeof
typeid
static_cast
dynamic_cast
const_cast
reinterpret_cast

Tous les autres opérateurs sont redéfinissables. En général, ils suivent les règles énoncées dans les paragraphes précédents. Cependant, un certain nombre d'entre eux demande des explications complémentaires.

8.11.3. Opérateurs d'incrémentation et de décrémentation

Commençons par les plus simples : les opérateurs ++ et —. Ces opérateurs sont tout les deux doubles, c'est à dire que la même notation représente deux opérateurs en réalité. En effet, ils n'ont pas la même signification, selon qu'ils sont placés avant ou après leur opérande. Le problème est que comme ces opérateurs ne prennent pas de paramètres (ils ne travaillent que sur l'objet), il est impossible de les différencier par surcharge. La solution qui a été adoptée est de les différencier en donnant un paramètre fictif de type int à l'un d'entre eux. Ainsi, les opérateurs ++ et — ne prennent pas de paramètre lorsqu'il s'agit des opérateurs préfixes, et un ont argument fictif (que l'on ne doit pas utiliser) lorsqu'ils sont suffixes. Les versions préfixées des opérateurs doivent renvoyer une référence sur l'objet lui-même, en revanche, les versions suffixées peuvent se contenter de renvoyer la valeur de l'objet.

Exemple 8-18. Opérateurs d'incrémentation et de décrémentation

```
class Entier
```

```
{
  int i;

public:
  Entier(int j)
  {
    i=j;
    return;
  }

  Entier operator++(int)   // Opérateur suffixe :
  {                        // retourne la valeur et incrémente
    Entier tmp(i);         // la variable.
    i++;
    return tmp;
  }

  Entier &operator++(void) // Opérateur préfixe : incrémente
  {                        // la variable et la retourne.
    i++;
    return *this;
  }
};
```

8.11.4. Opérateurs d'allocation dynamique de mémoire

Passons maintenant aux opérateurs d'allocation dynamique de mémoire.

Ces opérateurs prennent un nombre variable de paramètres, parce qu'ils sont surchargeables. Il est donc possible de définir plusieurs opérateur new ou new[], et plusieurs opérateurs delete ou delete[]. Cependant, les premiers paramètres de ces opérateurs doivent toujours être la taille de la zone de la mémoire à allouer dans le cas des opérateurs new et new[], et le pointeur sur la zone de la mémoire à restituer dans le cas des opérateurs delete et delete[].

La forme la plus simple de new ne prend qu'un paramètre : le nombre d'octets à allouer, qui vaut toujours la taille de l'objet à construire. Il doit renvoyer un pointeur du type void. L'opérateur delete correspondant peut prendre, quant à lui, soit un, soit deux paramètres. Comme on l'a déjà dit, le premier paramètre est toujours un pointeur du type void sur l'objet à détruire. Le deuxième paramètre, s'il existe, est du type size_t et contient la taille de l'objet à détruire. Les mêmes règles s'appliquent pour les opérateurs new[] et delete[], utilisés pour les tableaux.

Dans le cas où les opérateurs delete et delete[] prennent deux paramètres, le deuxième paramètre est la taille de la zone de la mémoire à restituer. Ceci signifie que le compilateur se charge de mémoriser cette information. Pour les opérateurs new et delete, ceci ne cause pas de problème, puisque la taille de cette zone est fixée par le type de l'objet. En revanche, pour les tableaux, la taille du tableau doit être stockée avec le tableau. En général, le compilateur utilise un en-tête devant le tableau d'objet. C'est pour cela que la taille à allouer, passée à new[], qui est la même que la taille à désallouer, passée en paramètre à delete[], n'est pas égale à la taille d'un objet multipliée par le nombre d'objets du tableau. Le compilateur demande un peu plus de mémoire, pour mémoriser la taille du tableau. On ne peut donc pas, dans ce cas, faire d'hypothèses quant à la structure que le compilateur donnera à la mémoire allouée pour stocker le tableau.

En revanche, si delete[] ne prend en paramètre que le pointeur sur le tableau, la mémorisation de la taille du tableau est à la charge du programmeur. Dans ce cas, le compilateur donne à new[] la valeur exacte de la taille du tableau, à savoir la taille d'un objet multipliée par le nombre d'objets dans le tableau.

Exemple 8-19. Détermination de la taille de l'en-tête des tableaux

```
#include <stdio.h>

int buffer[256];     // Buffer servant à stocker le tableau.

class Temp
{
  char i[13];        // sizeof(Temp) doit être premier.
```

```
public:
  static void *operator new[](size_t taille)
  {
    return buffer;
  }

  static void operator delete[](void *p, size_t taille)
  {
    printf("Taille de l'en-tête : %d\n",
          taille-(taille/sizeof(Temp))*sizeof(Temp));
    return ;
  }
};

int main(void)
{
  delete[] new Temp[1];
  return 0;
}
```

Il est à noter qu'aucun des opérateurs new, delete, new[] et delete[] ne reçoit le pointeur this en paramètre : ce sont des opérateurs statiques. Ceci est normal, puisque lorsqu'ils s'exécutent, soit l'objet n'est pas encore créé, soit il est déjà détruit. Le pointeur this n'existe donc pas encore (ou n'est plus valide) lors de l'appel de ces opérateurs.

Les opérateurs new et new[] peuvent avoir une forme encore un peu plus compliquée, qui permet de leur passer des paramètres lors de l'allocation de la mémoire. Les paramètres supplémentaires doivent impérativement être les paramètres deux et suivants, puisque le premier paramètre indique toujours la taille de la zone de mémoire à allouer.

Comme le premier paramètre est calculé par le compilateur, il n'y a pas de syntaxe permettant de le passer aux opérateurs new et new[]. En revanche, une syntaxe spéciale est nécessaire pour passer les paramètres supplémentaires. Cette syntaxe est détaillée ci-dessous.

Si l'opérateur new est déclaré de la manière suivante dans la classe classe :

```
static void *operator new(size_t taille, paramètres);
```

où taille est la taille de la zone de mémoire à allouer et paramètres la liste des paramètres additionnels, alors on doit l'appeler avec la syntaxe suivante :

```
new(paramètres) classe;
```

Les paramètres sont donc passés entre parenthèses comme pour une fonction normale. Le nom de la fonction est new, et le nom de la classe suit l'expression new comme dans la syntaxe sans paramètres. Cette utilisation de new est appelée *new avec placement*.

Le placement est souvent utilisé afin de réaliser des réallocations de mémoire d'un objet à un autre. Par exemple, si l'on doit détruire un objet alloué dynamiquement et en reconstruire immédiatement un autre du même type, les opérations suivantes se déroulent :

1. appel du destructeur de l'objet (réalisé par l'expression delete) ;
2. appel de l'opérateur delete ;
3. appel de l'opérateur new ;
4. appel du constructeur du nouvel objet (réalisé par l'expression new).

Ceci n'est pas très efficace, puisque la mémoire est restituée pour être allouée de nouveau immédiatement après. Il est beaucoup plus logique de réutiliser la mémoire de l'objet à détruire pour le nouvel objet, et de reconstruire ce dernier dans cette mémoire. Ceci peut se faire comme suit :

1. appel explicite du destructeur de l'objet à détruire ;
2. appel de new avec comme paramètre supplémentaire le pointeur sur l'objet détruit ;
3. appel du constructeur du deuxième objet (réalisé par l'expression new).

L'appel de new ne fait alors aucune allocation : on gagne ainsi beaucoup de temps.

Exemple 8-20. Opérateurs new avec placement

```
class A
{
public:
  A(void)          // Constructeur.
  {
    return ;
  }

  ~A(void)         // Destructeur.
  {
    return ;
  }

  // L'opérateur new suivant utilise le placement.
  // Il reçoit en paramètre le pointeur sur le bloc
  // à utiliser pour la requête d'allocation dynamique
  // de mémoire.
  static void *operator new (size_t taille, A *bloc)
  {
    return (void *) bloc;
  }

  // Opérateur new normal :
  static void *operator new(size_t taille)
  {
    // Implémentation.
    return pBlock;
  }
};

int main(void)
{
  A *pA=new A;      // Création d'un objet de classe A.
                    // L'opérateur new global du C++ est utilisé.
  pA->~A();         // Appel explicite du destructeur de A.
  A *pB=new(&A) A;  // Réutilisation de la mémoire de A.
  delete pA;        // Destruction de l'objet.
  return 0;
}
```

Dans cet exemple, la gestion de la mémoire est réalisée par les opérateurs prédéfinis du C++. Cependant, la réutilisation de la mémoire allouée se fait grâce à un opérateur new avec placement défini pour l'occasion. Ce dernier ne fait strictement rien d'autre que de renvoyer le pointeur qu'on lui a passé en paramètre. On notera qu'il est nécessaire d'appeler explicitement le destructeur de la classe A avant de réutiliser la mémoire de l'objet, car aucune expression delete ne s'en charge avant la réutilisation de la mémoire.

Il est impossible de passer des paramètres à l'opérateur delete dans une expression delete. Ceci est dû au fait qu'en général, on ne connaît pas le contexte de la destruction d'un objet (alors qu'à l'allocation, on connaît le contexte de création de l'objet). Normalement, il ne peut donc y avoir qu'un seul opérateur delete. Cependant, il existe un cas où l'on connaît le contexte de l'appel de l'opérateur delete : c'est le cas où le constructeur de la

classe lance une exception (voir le Chapitre 9 pour plus de détails à ce sujet). Dans ce cas, la mémoire allouée par l'opérateur new doit être restituée et l'opérateur delete est automatiquement appelé, puisque l'objet n'a pas pu être construit. Afin d'obtenir un comportement symétrique, il est permis de donner des paramètres additionnels à l'opérateur delete. Lorsqu'une exception est lancée dans le constructeur de l'objet alloué, l'opérateur delete appelé est l'opérateur dont la liste des paramètres correspond à celle de l'opérateur new qui a été utilisé pour créer l'objet. Les paramètres passés à l'opérateur delete prennent alors exactement les mêmes valeurs que celles qui ont été données aux paramètres de l'opérateur new lors de l'allocation de la mémoire de l'objet. Ainsi, si l'opérateur new a été utilisé sans placement, l'opérateur delete sans placement sera appelé. En revanche, si l'opérateur new a été appelé avec des paramètres, l'opérateur delete qui a les mêmes paramètres sera appelé. Si aucun opérateur delete ne correspond, aucun opérateur delete n'est appelé (si l'opérateur new n'a pas alloué de mémoire, ceci n'est pas grave, en revanche, si de la mémoire a été allouée, elle ne sera pas restituée). Il est donc important de définir un opérateur delete avec placement pour chaque opérateur new avec placement défini. L'exemple précédent doit donc être réécrit de la manière suivante :

```
class A
{
public:
  A(void)          // Constructeur.
  {
    // Le constructeur est susceptible
    // de lancer une exception.
    return ;
  }

  ~A(void)         // Destructeur.
  {
    return ;
  }

  // L'opérateur new suivant utilise le placement.
  // Il reçoit en paramètre le pointeur sur le bloc
  // à utiliser pour la requête d'allocation dynamique
  // de mémoire.
  static void *operator new (size_t taille, A *bloc)
  {
    return (void *) bloc;
  }

  // Opérateur new normal :
  static void operator new(size_t taille)
  {
    // Implémentation.
    return pBlock;
  }

  // L'opérateur delete suivant est utilisé dans les expressions
  // qui utilisent l'opérateur new avec placement ci-dessus,
  // si une exception se produit.
  static void operator delete(void *p, A *bloc)
  {
    // On ne fait rien, parce que l'opérateur new correspondant
    // n'a pas alloué de mémoire.
    return ;
  }
};
```

```
int main(void)
{
  A *pA=new A;      // Création d'un objet de classe A.
                    // L'opérateur new global du C++ est utilisé.
                    // Si une exception a lieu lors de la
                    // construction, l'opérateur delete global
                    // du C++ est appelé.
  pA->~A();         // Appel explicite du destructeur de A.
  A *pB=new (&A) A; // Réutilisation de la mémoire de A.
                    // Si une exception a lieu, l'opérateur
                    // delete(void *, A *) avec placement
                    // est utilisé.
  delete pA;        // Destruction de l'objet.
  return ;
}
```

> **Note:** Il est possible d'utiliser le placement avec les opérateurs new[] et delete[] exactement de la même manière qu'avec les opérateurs new et delete.
>
> On notera que dans le cas où l'opérateur new est utilisé avec placement, si le deuxième argument est de type size_t, l'opérateur delete à deux arguments peut être interprété soit comme un opérateur delete classique sans placement mais avec deux paramètres, soit comme l'opérateur delete avec placement correspondant à l'opérateur new avec placement. Afin de résoudre cette ambiguïté, le compilateur interprète systématiquement l'opérateur delete avec un deuxième paramètre de type size_t comme étant l'opérateur à deux paramètres sans placement. Il est donc impossible de définir un opérateur delete avec placement s'il a deux paramètres, le deuxième étant de type size_t. Il en est de même avec les opérateurs new[] et delete[].

Quelle que soit la syntaxe que vous désirez utiliser, les opérateurs new, new[], delete et delete[] doivent avoir un comportement bien déterminé. En particulier, les opérateurs delete et delete[] doivent pouvoir accepter un pointeur nul en paramètre. Lorsqu'un tel pointeur est utilisé dans une expression delete, aucun traitement ne doit être fait.

Enfin, vos opérateurs new et new[] doivent, en cas de manque de mémoire, appeler un gestionnaire d'erreur. Le gestionnaire d'erreur fourni par défaut lance une exception de classe std::bad_alloc (voir le Chapitre 9 pour plus de détails sur les exceptions). Cette classe est définie comme suit dans le fichier d'en-tête new :

```
class bad_alloc : public exception
{
public:
  bad_alloc(void) throw();
  bad_alloc(const bad_alloc &) throw();
  bad_alloc &operator=(const bad_alloc &) throw();
  virtual ~bad_alloc(void) throw();
  virtual const char *what(void) const throw();
};
```

> **Note:** Comme son nom l'indique, cette classe est définie dans l'espace de nommage std::. Si vous ne voulez pas utiliser les notions des espaces de nommage, vous devrez inclure le fichier d'en-tête new.h au lieu de new. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 11.

La classe exception dont bad_alloc hérite est déclarée comme suit dans le fichier d'en-tête exception :

```
class exception
{
public:
  exception (void) throw();
  exception(const exception &) throw();
  exception &operator=(const exception &) throw();
  virtual ~exception(void) throw();
  virtual const char *what(void) const throw();
```

```
};
```

Note: Vous trouverez plus d'informations sur les exceptions dans le Chapitre 9.

Si vous désirez remplacer le gestionnaire par défaut, vous pouvez utiliser la fonction std::set_new_handler. Cette fonction attend en paramètre le pointeur sur le gestionnaire d'erreur à installer et renvoie le pointeur sur le gestionnaire d'erreur précédemment installé. Les gestionnaires d'erreur ne prennent aucun paramètre et ne renvoient aucune valeur. Leur comportement doit être le suivant :

- soit ils prennent les mesures nécessaires pour permettre l'allocation du bloc de mémoire demandé et rendent la main à l'opérateur new. Ce dernier refait alors une tentative pour allouer le bloc de mémoire. Si cette tentative échoue à nouveau, le gestionnaire d'erreur est rappelé. Cette boucle se poursuit jusqu'à ce que l'opération se déroule correctement ou qu'une exception std::bad_alloc soit lancée ;
- soit ils lancent une exception de classe std::bad_alloc ;
- soit ils terminent l'exécution du programme en cours.

La librairie standard définit une version avec placement des opérateurs new qui renvoient le pointeur nul au lieu de lancer une exception en cas de manque de mémoire. Ces opérateurs prennent un deuxième paramètre, de type std::nothrow_t, qui doit être spécifié lors de l'appel. La librairie standard définit un objet constant de ce type afin que les programmes puissent l'utiliser sans avoir à le définir eux-même. Cet objet se nomme std::nothrow

Exemple 8-21. Utilisation de new sans exception

```
char *data = new(std::nothrow) char[25];
if (data == NULL)
{
   // Traitement de l'erreur…
   ⋮
}
```

Note: La plupart des compilateurs ne respecte pas les règles dictées par la norme C++. En effet, ils préfèrent retourner la valeur nulle en cas de manque de mémoire au lieu de lancer une exception. On peut rendre ces implémentations compatibles avec la norme en installant un gestionnaire d'erreur qui lance lui-même l'exception std::bad_alloc.

8.11.5. Opérateurs de transtypage

La redéfinition des opérateurs de transtypage permet de faire aisément des conversions entre des classes n'ayant pas la même structure mais ayant la même sémantique. Prenons l'exemple de la classe chaine, qui permet de faire des chaînes de caractères dynamiques (de longueur variable). Il est possible de les convertir en chaîne C classiques (c'est à dire en tableau de caractères) si l'opérateur (char const *) a été redéfini :

```
chaine::operator char const *(void) const;
```

On constatera que cet opérateur n'attend aucun paramètre, puisqu'il s'applique à l'objet qui l'appelle, mais surtout il n'a pas de type. En effet, puisque c'est un opérateur de transtypage, son type est nécessairement celui qui lui correspond (dans le cas présent, char const *).

8.11.6. Opérateurs de comparaison

Les opérateurs de comparaison sont très simples à redéfinir. La seule chose essentielle à retenir est qu'ils

renvoient une valeur booléenne. Ainsi, pour la classe chaine, on peut déclarer les opérateurs d'égalité et d'infériorité (dans l'ordre lexicographique par exemple) de deux chaînes de caractères comme suit :

```
bool chaine::operator==(const chaine &) const;
bool chaine::operator<(const chaine &) const;
```

8.11.7. Opérateur fonctionnel

Enfin, l'opérateur d'appel de fonctions () peut également être redéfini. Il est très utile en raison de son n-arité (+, -, etc… sont des opérateurs binaires car ils ont deux opérandes, ?: est un opérateur ternaire car il a trois opérandes, () est n-aire car il peut avoir n opérandes). Il est utilisé couramment pour les classes de matrices, afin d'autoriser l'écriture " matrice(i,j,k) ".

Exemple 8-22. Implémentation de la classe matrice

```
class matrice
{
  typedef double *ligne;
  ligne *lignes;
  unsigned short int n;   // Nombre de lignes (1er paramètre).
  unsigned short int m;   // Nombre de colonnes (2ème paramètre).

public:
  matrice(unsigned short int nl, unsigned short int nc);
  matrice(const matrice &source);
  ~matrice(void);
  matrice &operator=(const matrice &m1);
  double &operator()(unsigned short int i, unsigned short int j);
  matrice operator+(const matrice &m1) const;
  matrice operator-(const matrice &m1) const;
  matrice operator*(const matrice &m1) const;
};

// Le constructeur :
matrice::matrice(unsigned short int nl, unsigned short int nc)
{
  n = nl;
  m = nc;
  lignes = new ligne[n];
  for (unsigned short int i=0; i<n; i++)
     lignes[i] = new double[m];
  return;
}

// Le constructeur de copie :
matrice::matrice(const matrice &source)
{
  m = source.m;
  n = source.n;
  lignes = new ligne[n];   // Alloue.
  for (unsigned short int i=0; i<n; i++)
  {
    lignes[i] = new double[m];
    for (unsigned short int j=0; j<m; j++)  // Copie.
       lignes[i][j] = source.lignes[i][j];
  }
  return;
}

// Le destructeur :
matrice::~matrice(void)
{
  for (unsigned short int i=0; i<n; i++)
```

```
      delete[] lignes[i];
   delete[] lignes;
   return;
}

// L'opérateur d'affectation :
matrice &matrice::operator=(const matrice &source)
{
   if (source.n!=n || source.m!=m)   // Vérifie les dimensions.
   {
      for (unsigned short int i=0; i<n; i++)
         delete[] lignes[i];
      delete[] lignes;               // Détruit…
      m = source.m;
      n = source.n;
      lignes = new ligne[n];         // et réalloue.
      for (i=0; i<n; i++) lignes[i] = new double[m];
   }
   for (unsigned short int i=0; i<n; i++) // Copie.
       for (unsigned short int j=0; j<m; j++)
           lignes[i][j] = source.lignes[i][j];
   return *this;
}

// Opérateur d'accès :
double &matrice::operator()(unsigned short int i,
                            unsigned short int j)
{
   return lignes[i][j];
}

// Addition :
matrice matrice::operator+(const matrice &m1) const
{
   matrice tmp(n,m);
   for (unsigned short int i=0; i<n; i++)   // Double boucle.
       for (unsigned short int j=0; j<m; j++)
           tmp.lignes[i][j] = lignes[i][j]+m1.lignes[i][j];
   return tmp;
}

// Soustraction :
matrice matrice::operator-(const matrice &m1) const
{
   matrice tmp(n,m);
   for (unsigned short int i=0; i<n; i++)  // Double boucle.
       for (unsigned short int j=0; j<m; j++)
           tmp.lignes[i][j]=lignes[i][j]-m1.lignes[i][j];
        return tmp;
}

// Multiplication :
matrice matrice::operator*(const matrice &m1) const
{
   matrice tmp(n,m1.m);
   for (unsigned short int i=0; i<n; i++)  // Double boucle.
       for (unsigned short int j=0; j<m1.m; j++)
```

```
    {
      tmp.lignes[i][j]=0.;            // Produit scalaire.
      for (unsigned short int k=0; k<m; k++)
         tmp.lignes[i][j] += lignes[i][k]*m1.lignes[k][j];
    }
  return tmp;
}
```

Ainsi, on pourra effectuer la déclaration d'une matrice avec :

```
matrice m(2,3);
```

On pourra accéder aux éléments par exemple avec :

```
m(i,j)=6;
```

Et les opérations de bases telles que a=b+c, où a, b et c sont des matrices seront autorisées.

Les opérations d'inversion, transposition, etc... n'ont pas été reportées par souci de clarté.

8.11.8. Opérateurs d'indirection et de déréférencement

L'opérateur de déréférencement * permet l'écriture de classes dont les objets peuvent être utilisés dans des expressions manipulant des pointeurs. L'opérateur d'indirection & quant à lui, permet de renvoyer une adresse autre que celle de l'objet sur lequel il s'applique. Enfin, l'opérateur de déréférencement et de sélection de membres de structures -> permet de réaliser des classes qui encapsulent d'autres classes.

Si les opérateurs de déréférencement et d'indirection & et * peuvent renvoyer une valeur de type quelconque, ce n'est pas le cas de l'opérateur de déréférencement et de sélection de membre ->. Cet opérateur doit nécessairement renvoyer un type pour lequel il doit encore être applicable. Ce type doit donc soit redéfinir l'opérateur ->, soit être un pointeur sur une structure, union ou classe.

Exemple 8-23. Opérateur de déréférencement et d'indirection

```
// Cette classe est encapsulée par une autre classe :
struct Encapsulee
{
  int i;       // Donnée à accéder.
};

Encapsulee o;   // Objet à manipuler.

// Cette classe est la classe encapsulante :
struct Encapsulante
{
  Encapsulee *operator->(void) const
  {
    return &o;
  }

  Encapsulee *operator&(void) const
  {
    return &o;
  }

  Encapsulee &operator*(void) const
  {
    return o;
  }
};

// Exemple d'utilisation :
void f(int i)
```

```
{
  Encapsulante e;
  e->i=2;      // Enregistre 2 dans o.i.
  (*e).i = 3;     // Enregistre 3 dans o.i.
  Encapsulee *p = &e;
  p->i = 4;       // Enregistre 4 dans o.i.
  return ;
}
```

8.12. Des entrées - sorties simplifiées

La librairie C++ définit dans l'en-tête iostream des classes qui permettent de manipuler les flux d'entrée - sortie. Un flux est une notion informatique qui représente le flux des données dans un programme. Les flux d'entrées - sortie du C++ permettent donc de réaliser l'entrée des données et leur sortie sur les périphériques standards, tout comme les fonctions de la librairie C le permettent. Cependant, l'utilisation des flux C++ est, comme nous allons le voir, beaucoup plus aisée.

Trois objets particuliers sont instanciés dans la librairie : cin, cout et cerr. Ces objets sont des instances des flux d'entrée istream et de sortie ostream. Les opérateurs << et >> ont été surchargés pour ces flux afin de réaliser les entrées - sorties plus facilement. Ils permettent respectivement de passer une valeur à un flux ostream et de récupérer une donnée à partir d'un flux istream. Ils renvoient tous les deux le flux utilisé afin de réaliser des opérations multiples sur les flux d'entrée - sortie.

Les opérateurs << et >> permettent donc de réaliser des entrées - sorties sur les flux standards cin, cout et cerr. cin est le flux d'entrée des données, cout le flux de sortie et cerr le flux des erreurs. Leur utilisation est résumée dans la syntaxe suivante :

```
cin >> variable;
cout << valeur [<< valeur [...]];
```

On constate qu'il est possible d'envoyer plusieurs valeurs au flux de sortie, puisque l'opérateur << renvoie cout.

Exemple 8-24. Flux d'entrée / sortie cin et cout

```
#include <iostream>

using namespace std;

int main(void)
{
  int i;
  cin >> i;          // Entre un entier.
  cout << i << i+1;   // Affiche cet entier et le suivant.
  return 0;
}
```

> **Note:** Les flux d'entrée / sortie cin, cout et cerr sont déclarés dans l'espace de nommage std:: de la librairie standard C++. On devra donc faire précéder leur nom du préfixe std:: pour y accéder, ou utiliser un directive using pour importer les symboles de la librairie standard C++ dans l'espace de nommage global. Vous trouverez de plus amples renseignements sur les espaces de nommages dans le Chapitre 11.

Les avantages de cin et cout sont nombreux, on notera en particulier ceux-ci :

- le type des données est automatiquement pris en compte par les opérateurs << et >> (ils sont surchargés pour tous les types prédéfinis) ;

- ils travaillent par référence (on ne risque plus d'omettre l'opérateur & dans scanf) ;
- ils sont plus simple d'emploi.

Les flux d'entrée / sortie définis par la librairie C++ sont donc d'une extrême souplesse. Nous les détaillerons plus dans la Section 13.3, où nous verrons comment personnaliser le format des sorties et comment réaliser des entrées / sorties dans des fichiers.

8.13. Méthodes virtuelles

Les *méthodes virtuelles* n'ont strictement rien à voir avec les classes virtuelles, bien qu'elles utilisent le même mot-clé virtual. Ce mot-clé est utilisé ici dans un contexte et dans un sens différent.

Nous savons qu'il est possible de redéfinir les méthodes d'une classe mère dans une classe fille. Lors de l'appel d'une fonction ainsi redéfinie, la fonction appelée est la dernière fonction définie dans la hiérarchie de classe. Pour appeler la fonction de la classe mère alors qu'elle a été redéfinie, il faut préciser le nom de la classe à laquelle elle appartient avec l'opérateur de résolution de portée (::).

Bien que simple, cette utilisation de la redéfinition des méthodes peut poser des problèmes. Supposons qu'une classe B hérite de sa classe mère A. Si A possède une méthode x appelant une autre méthode y redéfinie dans la classe fille B, que se passe-t-il lorsqu'un objet de classe B appelle la méthode x ? La méthode appelée étant celle de la classe A, elle appellera la méthode y de la classe A. Par conséquent, la redéfinition de y ne sert à rien dès qu'on l'appelle à partir d'une des fonctions d'une des classes mère.

Une première solution consisterait à redéfinir la méthode x dans la classe B. Mais ce n'est ni élégant, ni efficace. Il faut en fait forcer le compilateur à ne pas faire le lien dans la fonction x de la classe A avec la fonction y de la classe A. Il faut que x appelle soit la fonction y de la classe A si elle est appelée par un objet de la classe A, soit la fonction y de la classe B si elle est appelée pour un objet de la classe B. Le lien avec l'une des méthodes y ne doit être fait qu'au moment de l'exécution, c'est à dire qu'on doit faire une édition de liens dynamique.

Le C++ permet de faire cela. Pour cela, il suffit de déclarer virtuelle la fonction de la classe de base qui est redéfinie dans la classe fille, c'est à dire la fonction y. Ceci se fait en faisant précéder par le mot-clé virtual dans la classe de base.

Exemple 8-25. Surcharge de méthode de classe de base

```
#include <iostream>

using namespace std;

// Définit la classe de base des données.

class DonneeBase
{
protected:
  int Numero;   // Les données sont numérotées.
  int Valeur;   // et sont constituées d'une valeur entière
                // pour les données de base.
public:
  void Entre(void);       // Entre une donnée.
  void MiseAJour(void);   // Met à jour la donnée.
};

void DonneeBase::Entre(void)
{
  cin >> Numero;          // Entre le numéro de la donnée.
  cout << '\n';
  cin >> Valeur;          // Entre sa valeur.
  cout << '\n';
  return;
}

void DonneeBase::MiseAJour(void)
{
```

```
  Entre();              // Entre une nouvelle donnée
                        // à la place de la donnée en cours.
  return;
}

/* Définit la classe des données détaillées. */

class DonneeDetaillee : private DonneeBase
{
  int ValeurEtendue;    // Les données détaillées ont en plus
                        // une valeur étendue.

public:
  void Entre(void);     // Redéfinition de la méthode d'entrée.
};

void DonneeDetaillee::Entre(void)
{
  DonneeBase::Entre();  // Appelle la méthode de base.
  cin >> ValeurEtendue; // Entre la valeur étendue.
  cout << '\n';
  return;
}
```

Si d est un objet de la classe DonneeDetaillee, l'appel de d.Entre ne causera pas de problème. En revanche, l'appel de d.MiseAJour ne fonctionnera pas correctement, car la fonction Entre appelée dans MiseAJour est la fonction de la classe DonneeBase, et non la fonction redéfinie dans DonneeDetaille.

Il fallait déclarer la fonction Entre comme une fonction virtuelle. Il n'est nécessaire de le faire que dans la classe de base. Celle-ci doit donc être déclarée comme suit :

```
class DonneeBase
{
protected:

  int Numero;
  int Valeur;

public:
  virtual void Entre(void);  // Fonction virtuelle.
  void MiseAJour(void);
};
```

Cette fois, la fonction Entre appelée dans MiseAJour est soit la fonction de la classe DonneeBase, si MiseAJour est appelée pour un objet de classe DonneeBase, soit celle de la classe DonneeDetaille si MiseAJour est appelée pour un objet de la classe DonneeDetaillee.

En résumé, les méthodes virtuelles sont des méthodes qui sont appelées selon la vraie classe de l'objet qui l'appelle. Les objets qui contiennent des méthodes virtuelles peuvent être manipulés en tant qu'objets des classes de base, tout en effectuant les bonnes opérations en fonction de leur type. Ils apparaissent donc comme étant des objets de la classe de base et des objets de leur classe complète indifféremment, et on peut les considérer soit comme les uns, soit comme les autres. Un tel comportement est appelé *polymorphisme* (c'est à dire qui peut avoir plusieurs aspects différents). Nous verrons une application du polymorphisme dans le cas des pointeurs sur les objets.

8.14. Dérivation

Nous allons voir ici les *règles de dérivation*. Ces règles permettent de savoir ce qui est autorisé et ce qui ne l'est pas lorsque l'on travaille avec des classes de base et leurs classes filles (ou classes dérivées).

La première règle, qui est aussi la plus simple, indique qu'il est possible d'utiliser un objet d'une classe dérivée partout où l'on peut utiliser un objet d'une de ses classes mères. Les méthodes et données des classes mères appartiennent en effet par héritage aux classes filles. Bien entendu, on doit avoir les droits d'accès sur les membres de la classe de base que l'on utilise (l'accès peut être restreint lors de l'héritage).

La deuxième règle indique qu'il est possible de faire une affectation d'une classe dérivée vers une classe mère. Les données qui ne servent pas à l'initialisation sont perdues, puisque la classe mère ne possède pas les champs correspondants. En revanche, l'inverse est strictement interdit. En effet, les données de la classe fille qui n'existent pas dans la classe mère ne pourraient pas recevoir de valeur, et l'initialisation ne se ferait pas correctement.

Enfin, la troisième règle dit que les pointeurs des classes dérivées sont compatibles avec les pointeurs des classes mères. Cela signifie qu'il est possible d'affecter un pointeur de classe dérivée à un pointeur d'une de ses classes de base. Il faut bien entendu que l'on ait en outre le droit d'accéder à la classe de base, c'est à dire qu'au moins un de ses membres puisse être utilisé. Cette condition n'est pas toujours vérifiée, en particulier pour les classes de base dont l'héritage est private.

Un objet dérivé pointé par un pointeur d'une des classes mères de sa classe est considéré comme un objet de la classe du pointeur qui le pointe. Les données spécifiques à sa classe ne sont pas supprimées, elles sont seulement momentanément inaccessibles. Cependant, le mécanisme des méthodes virtuelles continue de fonctionner correctement. En particulier, le destructeur de la classe de base doit être déclaré en tant que méthode virtuelle. Ceci permet d'appeler le bon destructeur en cas de destruction de l'objet.

Il est possible de convertir un pointeur de classe de base en un pointeur de classe dérivée si la classe de base n'est pas virtuelle. Cependant, même lorsque la classe de base n'est pas virtuelle, ceci est dangereux, car la classe dérivée peut avoir des membres qui ne sont pas présents dans la classe de base, et l'utilisation de ce pointeur peut conduire à des erreurs très graves. C'est pour cette raison qu'un transtypage est nécessaire dans ce type de conversion.

Soient par exemple les deux classes définies comme suit :

```
#include <iostream>

using namespace std;

class Mere
{
public:
  Mere(void);
  ~Mere(void);
};

Mere::Mere(void)
{
  cout << "Constructeur de la classe mère.\n";
  return;
}

Mere::~Mere(void)
{
  cout << "Destructeur de la classe mère.\n";
  return;
}

class Fille : public Mere
{
  public:
  Fille(void);
  ~Fille(void);
};
```

```
Fille::Fille(void) : Mere()
{
  cout << "Constructeur de la classe fille.\n";
  return;
}

Fille::~Fille(void)
{
  cout << "Destructeur de la classe fille.\n";
  return;
}
```

Avec ces définitions, seule la première des deux affectations suivantes est autorisée :

```
Mere m;   // Instanciation de deux objets.
Fille f;

m=f;      // Ceci est autorisé, mais l'inverse ne le serait pas :
f=m;      // ERREUR !! (ne compile pas).
```

Les mêmes règles sont applicables pour les pointeurs d'objets :

```
Mere *pm, m;
Fille *pf, f;
pf=&f;    // Autorisé.
pm=pf;    // Autorisé. Les données et les méthodes
          // de la classe fille ne sont plus accessibles
          // avec ce pointeur : *pm est un objet
          // de la classe mère.
pf=&m;    // ILLÉGAL : il faut faire un transtypage :
pf=(Fille *) &m;  // Cette fois, c'est légal, mais DANGEREUX !
          // En effet, les méthodes de la classe filles
          // ne sont pas définies, puisque m est une classe mère.
```

L'utilisation d'un pointeur sur la classe de base pour accéder à une classe dérivée nécessite d'utiliser des méthodes virtuelles. En particulier, il est nécessaire de rendre virtuels les destructeurs. Par exemple, avec la définition donnée ci-dessus pour les deux classes, le code suivant est faux :

```
Mere *pm;
Fille *pf = new Fille;
pm = pf;
delete pm; // Appel du destructeur de la classe mère !
```

Pour résoudre le problème, il faut que le destructeur de la classe mère soit virtuel (il est inutile de déclarer virtuel le destructeur des classes filles) :

```
class Mere
{
public:
  Mere(void);
```

```
  virtual ~Mere(void);
};
```

On notera que bien que l'opérateur delete soit une fonction statique, le bon destructeur est appelé, car le destructeur est déclaré virtual. En effet, l'opérateur delete recherche le destructeur à appeler dans la classe de l'objet le plus dérivé. De plus, l'opérateur delete restitue la mémoire de l'objet complet, et pas seulement celle du sous-objet référencé par le pointeur utilisé dans l'expression delete. Lorsqu'on utilise la dérivation, il est donc très important de déclarer les destructeurs virtuels pour que l'opérateur delete utilise le vrai type de l'objet à détruire.

8.15. Méthodes virtuelles pures - Classes abstraites

Une *méthode virtuelle pure* est une méthode qui est déclarée mais non définie dans une classe. Elle est définie dans une des classes dérivée de cette classe.

Une *classe abstraite* est une classe comportant au moins une méthode virtuelle pure.

Étant donné que les classes abstraites ont des méthodes non définies, il est impossible d'instancier des objets pour ces classes. En revanche, on pourra les référencer avec des pointeurs.

Le mécanisme des méthodes virtuelles pures et des classes abstraites permet de créer des classes de bases contenant toutes les caractéristiques d'un ensemble de classes dérivées, pour pouvoir les manipuler avec un unique type de pointeurs. En effet, les pointeurs des classes dérivées sont compatibles avec les pointeurs des classes de base, on pourra donc référencer les classes dérivées avec des pointeurs sur les classes de base, donc avec un unique type sous-jacent : celui de la classe de base. Cependant, les méthodes des classes dérivées doivent exister dans la classe de base pour pouvoir être accessibles à travers le pointeur sur la classe de base. C'est ici que les méthodes virtuelles pures apparaissent. Elles forment un moule pour les méthodes des classes dérivées, qui les définissent. Bien entendu, il faut que ces méthodes soient déclarées virtuelles, puisque l'accès se fait avec un pointeur de classe de base et qu'il faut que ce soit la méthode de la classe réelle de l'objet (c'est à dire la classe dérivée) qui soit appelée.

Pour déclarer une méthode virtuelle pure dans une classe, il suffit de faire suivre sa déclaration de " =0 ". La fonction doit également être déclarée virtuelle :

```
virtual type nom(paramètres) =0;
```

=0 signifie ici simplement qu'il n'y a pas d'instance de cette méthode dans cette classe.

> **Note:** =0 doit être placé complètement en fin de déclaration, c'est à dire après le mot-clé const pour les méthodes const et après la déclaration de la liste des exceptions autorisées (voir le Chapitre 9 pour plus de détails à ce sujet).

Un exemple vaut mieux qu'un long discours. Soit donc, par exemple, à construire une structure de données pouvant contenir d'autres structures de données, quel que soit leur type. Cette structure de données est appelée un conteneur, parce qu'elle contient d'autres structures de données. Il est possible de définir différents types de conteneurs. Dans cet exemple, on ne s'intéressera qu'au conteneur de type sac.

Un *sac* est un conteneur pouvant contenir zéro ou plusieurs objets, chaque objet n'étant pas forcément unique. Un objet peut donc être placé plusieurs fois dans le sac. Un sac dispose de deux fonctions permettant d'y mettre et d'en retirer un objet. Il a aussi une fonction permettant de dire si un objet se trouve dans le sac.

Nous allons déclarer une classe abstraite qui servira de classe de base pour tous les objets utilisables. Le sac ne manipulera que des pointeurs sur la classe abstraite, ce qui permettra son utilisation pour toute classe dérivant de cette classe. Afin de différencier deux objets égaux, un numéro unique devra être attribué à chaque objet manipulé. Le choix de ce numéro est à la charge des objets, la classe abstraite dont ils dérivent devra donc avoir une méthode renvoyant ce numéro. Les objets devront tous pouvoir être affichés dans un format qui leur est propre. La fonction à utiliser pour cela sera print. Cette fonction sera une méthode virtuelle pure de la classe abstraite, puisqu'elle devra être définie pour chaque objet.

Passons maintenant au programme…

Exemple 8-26. Conteneur d'objets polymorphiques

```
#include <iostream>
```

```
using namespace std;

/*************  LA CLASSE DE ABSTRAITE DE BASE   *****************/

class Object
{
  unsigned long int h;         // Handle de l'objet.
  unsigned long int new_handle(void);

public:
  Object(void);                // Le constructeur.
  virtual ~Object(void);       // Le destructeur virtuel.
  virtual void print(void) =0; // Fonction virtuelle pure.
  unsigned long int handle(void) const;  // Fonction renvoyant
                               // le numéro d'identification
                               // de l'objet.
};

// Cette fonction n'est appelable que par la classe Object :

unsigned long int Object::new_handle(void)
{
  static unsigned long int hc = 0;
  return hc = hc + 1;          // hc est le handle courant.
                               // Il est incrémenté
}                              // à chaque appel de new_handle.

// Le constructeur de Object doit être appelé par les classes dérivées :

Object::Object(void)
{
  h = new_handle();            // Trouve un nouveau handle.
  return;
}

Object::~Object(void)
{
  return ;
}

unsigned long int Object::handle(void) const
{
  return h;                    // Renvoie le numéro de l'objet.
}

/******************** LA CLASSE SAC   ******************/

class Bag : public Object      // La classe sac. Elle hérite
                               // de Object, car un sac peut
                               // en contenir un autre. Le sac
                               // est implémenté sous la forme
                               // d'une liste chaînée.
{
  typedef struct baglist
  {
    baglist *next;
```

```
    Object *ptr;
  } BagList;

  BagList *head;              // La tête de liste.

public:
  Bag(void);        // Le constructeur : appel celui de Object.
  ~Bag(void);       // Le destructeur.
  void print(void); // Fonction d'affichage du sac.
  bool has(unsigned long int) const;
                  // true si le sac contient l'objet.
  bool is_empty(void) const;   // true si le sac est vide.
  void add(Object &);          // Ajoute un objet.
  void remove(Object &);       // Retire un objet.
};

Bag::Bag(void) : Object()
{
  return;  // Ne fait rien d'autre qu'appeler Object::Object().
}

Bag::~Bag(void)
{
  BagList *tmp = head;   // Détruit la liste d'objet.
  while (tmp != NULL)
  {
    tmp = tmp->next;
    delete head;
    head = tmp;
  }
  return;
}

void Bag::print(void)
{
  BagList *tmp = head;
  cout << "Sac n° " << handle() << ".\n Contenu : \n";

  while (head != NULL)
  {
    cout << "\t";        // Indente la sortie des objets.
    tmp->ptr->print();   // Affiche la liste objets.
    tmp = tmp->next;
  }
  return;
}

bool Bag::has(unsigned long int h) const
{
  BagList *tmp = head;
  while (tmp != NULL && tmp->ptr->handle() != h)
       tmp = tmp->next;     // Cherche l'objet.
  return (tmp != NULL);
}

bool Bag::is_empty(void) const
{
  return (head==NULL);
}

void Bag::add(Object &o)
{
  BagList *tmp = new BagList;   // Ajoute un objet à la liste.
  tmp->ptr = &o;
  tmp->next = head;
```

```
  head = tmp;
  return;
}

void Bag::remove(Object &o)
{
  BagList *tmp1 = head, *tmp2 = NULL;
  while (tmp1 != NULL && tmp1->ptr->handle() != o.handle())
  {
    tmp2 = tmp1;        // Cherche l'objet…
    tmp1 = tmp1->next;
  }
  if (tmp1!=NULL)       // et le supprime de la liste.
  {
    if (tmp2!=NULL) tmp2->next = tmp1->next;
    else head = tmp1->next;
    delete tmp1;
  }
  return;
}
```

Avec la classe Bag définie telle quelle, il est à présent possible de stocker des objets dérivant de la classe Object avec les fonctions add et remove :

```
class MonObjet : public Object
{
  /*  etc…  */
};

Bag MonSac;

int main(void)
{
  MonObjet a, b, c;    // Effectue quelques opérations
                       // avec le sac :
  MonSac.add(a);
  MonSac.add(b);
  MonSac.add(c);
  MonSac.print();
  MonSac.remove(b);
  MonSac.add(MonSac);  // Un sac peut contenir un sac !
  MonSac.print();      // Attention ! Cet appel est récursif !
  return 0;
}
```

Nous avons vu que la classe de base servait de moule aux classes dérivées. Le droit d'empêcher une fonction membre virtuelle pure définie dans une classe dérivée d'accéder en écriture non seulement aux données de la classe de base, mais aussi aux données de la classe dérivée, peut donc faire partie de ses prérogatives. Ceci est faisable en déclarant le pointeur this comme étant un pointeur constant sur objet constant. Nous avons vu que cela pouvait se faire en rajoutant le mot-clé const après la déclaration de la fonction membre. Par exemple, si le handle de l'objet de base est placé en protected au lieu d'être en public, la classe Object peut empêcher la fonction print de le modifier en la déclarant const :

```
class Object
{
```

```
  unsigned long int new_handle(void);

protected:
  unsigned long int h;

public:
  Object(void);                    // Le constructeur.
  virtual void print(void) const=0;  // Fonction virtuelle pure.
  unsigned long int handle(void);    // Fonction renvoyant
                                     // le numéro d'identification
                                     // de l'objet.
};
```

Dans l'exemple donné ci-dessus, la fonction print peut accéder en lecture à *h*, mais pas en écriture. En revanche, les autres fonctions membres ont un accès total. Ceci est en particulier vrai pour les fonctions membres des classes dérivées qui ne surchargent pas une des fonctions de la classe de base. Cette méthode d'encapsulation est donc coopérative (elle requiert la bonne volonté des fonctions membres), tout comme la méthode qui consistait en C à déclarer une variable constante. Cependant, elle permettra de détecter des erreurs à la compilation.

Bien entendu, ceci fonctionne également avec les fonctions membres virtuelles non pures, et même avec les fonctions non virtuelles.

8.16. Pointeurs sur les membres d'une classe

Nous avons déjà vu les pointeurs sur les objets. Il nous reste à voir les pointeurs sur les membres des classes.

Les classes regroupent les caractéristiques des données et des fonctions des objets. Les membres des classes ne peuvent donc pas être manipulés sans passer par la classe à laquelle ils appartiennent. Par conséquent, il faut, lorsque l'on veut faire un pointeur sur un membre, indiquer le nom de sa classe. Pour cela, la syntaxe suivante est utilisée :

```
définition classe::* pointeur
```

Par exemple, si une classe test contient des entiers, le type de pointeurs à utiliser pour stocker leur adresse est :

```
int test::*
```

Si on veut déclarer un pointeur p de ce type, on écrira donc :

```
int test::*p1;   // Construit le pointeur sur entier
                 // de la classe test.
```

Une fois le pointeur déclaré, on pourra l'initialiser en prenant l'adresse du membre de la classe du type correspondant. Pour cela, il faudra encore spécifier le nom de la classe avec l'opérateur de résolution de portée :

```
p1 = &test::i;   // Récupère l'adresse de i.
```

La même syntaxe est utilisable pour les fonctions. L'emploi d'un typedef est dans ce cas fortement recommandé. Par exemple, si la classe test dispose d'une fonction membre appelée lit, qui n'attend aucun paramètre et qui renvoie un entier, on pourra récupérer son adresse ainsi :

```
typedef int (test::* pf)(void);  // Définit le type de pointeur.
pf p2=&test::lit;                // Construit le pointeur et
                                 // lit l'adresse de la fonction.
```

Cependant, ces pointeurs ne sont pas utilisables directement. En effet, les données d'une classe sont instanciées pour chaque objet, et les fonctions membres reçoivent de manière implicite systématiquement le pointeur this sur l'objet. On ne peut donc pas faire un déréférencement direct de ces pointeurs. Il faut spécifier l'objet pour lequel le pointeur va être utilisé. Ceci se fait avec la syntaxe suivante :

```
objet.*pointeur
```

Pour les pointeurs d'objet, on pourra utiliser l'opérateur ->* à la place de l'opérateur .* (appelé pointeur sur *opérateur de sélection de membre*).

Ainsi, si a est un objet de classe test, on pourra accéder à la donnée *i* de cet objet à travers le pointeur p1 avec la syntaxe suivante :

```
a.*p1 = 3;  // Initialise la donnée membre i de a avec la valeur 3.
```

Pour les fonctions membres, on mettra des parenthèses à cause des priorités des opérateurs :

```
int i = (a.*p2)();   // Appelle la fonction lit() pour l'objet a.
```

Pour les données et les fonctions membres statiques, cependant, la syntaxe est différente. En effet, les données n'appartiennent plus aux objets de la classe, mais à la classe elle-même, et il n'est plus nécessaire de connaître l'objet auquel le pointeur s'applique pour les utiliser. De même, les fonctions membres ne reçoivent plus le pointeur sur l'objet, et on peut donc les appeler sans référencer ce dernier.

La syntaxe s'en trouve donc modifiée. Les pointeurs sur les membres statiques des classes sont compatibles avec les pointeurs sur les objets et les fonctions non-membres. Par conséquent, si une classe contient une donnée statique entière, on pourra récupérer son adresse directement et la mettre dans un pointeur d'entier :

```
int *p3 = &test::entier_statique;   // Récupère l'adresse
                                    // de la donnée membre
                                    // statique.
```

La même syntaxe s'appliquera pour les fonctions :

```
typedef int (*pg)(void);
pg p4 = &test::fonction_statique;   // Récupère l'adresse
                                    // d'une fonction membre
                                    // statique.
```

Enfin, l'utilisation des ces pointeurs est identique à celle des pointeurs classiques, puisqu'il n'est pas nécessaire de fournir le pointeur this. Il est donc impossible de spécifier le pointeur sur l'objet sur lequel la fonction doit travailler aux fonctions membres statiques. Ceci est naturel, puisque les fonctions membres statiques ne peuvent pas accéder aux données non statiques d'une classe.

Exemple 8-27. Pointeurs sur membres statiques

```
#include <iostream>

using namespace std;
```

```
class test
{
  int i;
  static int j;

public:
  test(int j)
  {
    i=j;
    return ;
  }

  static int get(void)
  {
    /* return i ;  INTERDIT : i est non statique
                et get l'est ! n */
    return j; // Autorisé.
  }
};

int test::j=5;            // Initialise la variable statique.

typedef int (*pf)(void);   // Pointeur de fonction renvoyant
                      // un entier.
pf p=&test::get;          // Initialisation licite, car get
                      // est statique.

int main(void)
{
  cout << (*p)();             // Affiche 5. On ne spécifie pas
                      // l'objet.
  return 0;
}
```

Chapitre 9. Les exceptions en C++

Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend. Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.

Le C++ supporte les exceptions logicielles, dont le but est de gérer les erreurs qui surviennent lors de l'exécution des programmes. Lorsqu'une telle erreur survient, le programme doit *lancer une exception*. L'exécution normale du programme s'arrête dès que l'exception est lancée, et le contrôle est passé à un *gestionnaire d'exception*. Lorsqu'un gestionnaire d'exception s'exécute, on dit qu'il a *attrapé l'exception*.

Les exceptions permettent une gestion simplifiée des erreurs, parce qu'elles en reportent le traitement. Le code peut alors être écrit sans se soucier des cas particuliers, ce qui le simplifie grandement. Les cas particuliers sont traités dans les gestionnaires d'exception.

En général, une fonction qui détecte une erreur d'exécution ne peut pas se terminer normalement. Comme son traitement n'a pas pu se dérouler normalement, il est probable que la fonction qui l'a appelée considère elle aussi qu'une erreur a eu lieu et termine son exécution. L'erreur remonte ainsi la liste des appelants de la fonction qui a généré l'erreur. Ce processus continue, de fonction en fonction, jusqu'à ce que l'erreur soit complètement gérée, ou jusqu'à ce que le programme se termine (ce cas survient lorsque la fonction principale ne peut pas gérer l'erreur).

Traditionnellement, ce mécanisme est implémenté à l'aide de codes de retour des fonctions. Chaque fonction doit renvoyer une valeur spécifique à l'issue de son exécution, permettant d'indiquer si elle s'est correctement déroulée ou non. La valeur renvoyée est donc utilisée par l'appelant pour déterminer la nature de l'erreur, et, si erreur il y a, prendre les mesures nécessaires. Cette méthode permet à chaque fonction de libérer les ressources qu'elle a allouée lors de la remontée des erreurs, et d'effectuer ainsi sa part du traitement d'erreur.

Malheureusement, cette technique nécessite de tester les codes de retour de chaque fonction appelée, et la logique d'erreur développée finit par devenir très lourde, puisque ces tests s'imbriquent les uns à la suite des autres et que le code du traitement des erreurs se trouve mélangé avec le code du fonctionnement normal de l'algorithme. Cette complication peut devenir ingérable lorsque plusieurs valeurs de codes de retour peuvent être renvoyés afin de distinguer les différents cas d'erreur possible, car il peut en découler un grand nombre de tests et beaucoup de cas particuliers à gérer dans les fonctions appelantes.

Certains programmes utilisent donc une solution astucieuse, qui consiste à déporter le traitement des erreurs en dehors de l'algorithme à effectuer par des sauts vers la fin de la fonction. Le code de nettoyage, qui se trouve alors après l'algorithme, est exécuté complètement si tout se passe correctement. En revanche, si la moindre erreur est détectée en cours d'exécution, un saut est réalisé vers la partie du code de nettoyage correspondante au traitement qui a déjà été effectué. Ainsi, ce code n'est écrit qu'une seule fois, et le traitement des erreurs est situé en dehors du traitement normal.

La solution précédente est tout à fait valable (en fait, c'est même la solution la plus simple), mais elle souffre d'un inconvénient. Elle rend le programme moins structuré, car toutes les ressources utilisées par l'algorithme doivent être accessibles depuis le code de traitement des erreurs. Ces ressources doivent donc être placée dans une portée relativement globale, voire déclarées en tête de fonction. De plus, le traitement des codes d'erreurs multiples pose toujours les mêmes problèmes de complication des tests.

La solution qui met en ½uvre les exceptions est beaucoup plus simple, puisque la fonction qui détecte une erreur peut se contenter de lancer une exception. Cette exception interrompt l'exécution de la fonction, et un gestionnaire d'exception approprié est recherché. La recherche du gestionnaire suit le même chemin que celui utilisé lors de la remontée des erreurs : à savoir la liste des appelants. La première fonction appelante qui contient un gestionnaire d'exception approprié prend donc le contrôle, et effectue le traitement de l'erreur. Si le

traitement est complet, le programme reprend son exécution normale. Dans le cas contraire, le gestionnaire d'exception peut relancer l'exception (auquel cas le gestionnaire d'exception suivant est recherché) ou terminer le programme.

Le mécanisme des exceptions du C++ garantie que tous les objets de classe de stockage automatique sont détruits lorsque l'exception qui remonte sort de leur portée. Ainsi, si toutes les ressources sont encapsulées dans des classes disposant d'un destructeur capable de les détruire ou de les ramener dans un état cohérent, la remontée des exceptions effectue automatiquement le ménage. De plus, les exceptions peuvent être typées, et caractériser ainsi la nature de l'erreur qui s'est produite. Ce mécanisme est donc strictement équivalent en termes de fonctionnalités aux codes d'erreurs utilisés précédemment.

Comme on le voit, les exceptions permettent de simplifier le code, en reportant en dehors de l'algorithme normal le traitement des erreurs. Par ailleurs, la logique d'erreur est complètement prise en charge par le langage, et le programmeur n'a plus à faire les tests qui permettent de déterminer le traitement approprié pour chaque type d'erreur. Les mécanismes de gestion des exceptions du C++ sont décrits dans les paragraphes suivants.

9.1. Lancement et récupération d'une exception

En C++, lorsqu'il faut lancer une exception, on doit créer un objet dont la classe caractérise cette exception, et utiliser le mot-clé throw. Sa syntaxe est la suivante :

```
throw objet;
```

où objet est l'objet correspondant à l'exception. Cet objet peut être de n'importe quel type, et pourra ainsi caractériser pleinement l'exception.

L'exception doit alors être traitée par la routine d'exception correspondante. On ne peut attraper que les exceptions qui sont apparues dans une zone de code limitée (cette zone est dite *protégée* contre les erreurs d'exécution), pas sur tout un programme. On doit donc placer le code susceptible de lancer une exception d'un bloc d'instructions particulier. Ce bloc est introduit avec le mot-clé try :

```
try
{
  // Code susceptible de générer des exceptions…
}
```

Les gestionnaires d'exceptions doivent suivre le bloc try. Ils sont introduits avec le mot-clé catch :

```
catch (classe [&][temp])
{
  // Traitement de l'exception associée à la classe
}
```

Notez que les objets de classe de stockage automatique définis dans le bloc try sont automatiquement détruits lorsqu'une exception fait sortir le contrôle du programme de leur portée. C'est également le cas de l'objet construit pour lancer l'exception. Le compilateur effectue donc une copie de cet objet pour le transférer au premier bloc catch capable de le recevoir. Ceci implique qu'il y ait un constructeur de copie pour les classes d'exceptions non triviales.

De même, les blocs catch peuvent recevoir leur paramètre par valeur ou par référence, comme le montre la syntaxe indiquée ci-dessus. En général, il est préférable d'utiliser une référence, afin d'éviter une nouvelle copie de l'objet de l'exception pour le bloc catch. Toutefois, on prendra garde au fait que dans ce cas, les modifications effectuées sur le paramètre seront effectuées dans la copie de travail du compilateur et seront donc également visibles dans les blocs catch des fonctions appelantes ou de portée supérieure, si l'exception est relancée après traitement.

Il peut y avoir plusieurs gestionnaires d'exceptions. Chacun traitera les exceptions qui ont été générées dans le bloc try et dont l'objet est de la classe indiquée par son paramètre. Il n'est pas nécessaire de donner un nom à l'objet (temp) dans l'expression catch. Cependant, ceci permet de le récupérer, ce qui peut être nécessaire si l'on doit récupérer des informations sur la nature de l'erreur.

Enfin, il est possible de définir un gestionnaire d'exceptions universel, qui récupérera toutes les exceptions

possibles, quel que soient leurs types. Ce gestionnaire d'exception doit prendre comme paramètre trois points de suspension entre parenthèses dans sa clause catch. Bien entendu, dans ce cas, il est impossible de spécifier une variable qui contient l'exception, puisque son type est indéfini.

Exemple 9-1. Utilisation des exceptions

```
#include <iostream>

using namespace std;

class erreur   // Première exception possible, associée
            // à l'objet erreur.
{
public:
   int cause;  // Entier spécifiant la cause de l'exception.
   // Le constructeur. Il appelle le constructeur de cause.
   erreur(int c) : cause(c) {}
   // Le constructeur de copie. Il est utilisé par le mécanisme
   // des exceptions :
   erreur(const erreur &source) : cause(source.cause) {}
};

class other {};   // Objet correspondant à toutes
                  // les autres exceptions.

int main(void)
{
   int i;            // Type de l'exception à générer.
   cout << "Tapez 0 pour générer une exception Erreur, "
      "1 pour une Entière :";
   cin >> i;   // On va générer une des trois exceptions
                  // possibles.
   cout << endl;
   try               // Bloc où les exceptions sont prises en charge.
   {
      switch (i)    // Selon le type d'exception désirée,
      {
      case 0:
         {
            erreur a(0);
            throw (a);   // on lance l'objet correspondant
                        // (ici, de classe erreur).
                        // Ceci interrompt le code. break est
                        // donc inutile ici.
         }
      case 1:
         {
            int a=1;
            throw (a);   // Exception de type entier.
         }
      default:       // Si l'utilisateur n'a pas tapé 0 ou 1,
         {
            other c;     // on crée l'objet c (type d'exception
            throw (c);   // other) et on le lance.
         }
      }
   }              // fin du bloc try. Les blocs catch suivent :
```

```
   catch (erreur &tmp) // Traitement de l'exception erreur …
   {              // (avec récupération de la cause).
      cout << "Erreur erreur ! (cause " << tmp.cause << ")\n";
   }
   catch (int tmp)   // Traitement de l'exception int…
   {
      cout << "Erreur int ! (cause " << tmp << ")\n";
   }
   catch (...)       // Traitement de toutes les autres
   {               // exceptions (...).
                   // On ne peut pas récupérer l'objet ici.
      cout << "Exception inattendue !\n";
   }
   return 0;
}
```

Selon ce qu'entre l'utilisateur, une exception du type erreur, int ou other est générée.

9.2. Remontée des exceptions

Les fonctions intéressées par les exceptions doivent les capter avec le mot-clé catch comme on l'a vu ci-dessus. Elles peuvent alors effectuer tous les traitements d'erreurs que le C++ ne fera pas automatiquement. Ces traitements comprennent généralement le rétablissement de l'état des données manipulées par la fonction (dont, pour les fonctions membres d'une classe, les données membres de l'objet courant), ainsi que la libération des ressources non encapsulées dans des objets de classe de stockage automatique (par exemple, les fichiers ouverts, les connexions réseau, etc…).

Une fois ce travail effectué, elles peuvent, si elles le désirent, relancer l'exception, afin de permettre un traitement supérieur par leur fonction appelante. Le parcours de l'exception s'arrêtera donc dès que l'erreur aura été complètement traitée. Bien entendu, il est également possible de lancer une autre exception que celle que l'on a reçu, comme ce peut être par exemple le cas si le traitement de l'erreur provoque lui-même une erreur.

Pour relancer l'exception en cours de traitement dans un gestionnaire d'exception, il faut utiliser le mot-clé throw. La syntaxe est la suivante :

```
throw ;
```

L'exception est alors relancée, avec comme valeur l'objet que le compilateur a construit en interne pour propager l'exception. Un gestionnaire d'exception peut donc modifier les paramètres de l'exception, s'il l'attrape avec une référence.

Si, lorsqu'une exception se produit dans un bloc try, il est impossible de trouver le bloc catch correspondant à la classe de cette exception, il se produit une erreur d'exécution. La fonction prédéfinie std::terminate est alors appelée. Elle se contente d'appeler une fonction de traitement de l'erreur, qui elle-même appelle la fonction abort de la librairie C. Cette fonction termine en catastrophe l'exécution du programme fautif en générant une faute (les ressources allouées par le programme ne sont donc pas libérées, et des données peuvent être perdues). Ce n'est généralement pas le comportement désiré, aussi est-il est possible de le modifier en changeant la fonction appelée par std::terminate.

Pour cela, il faut utiliser la fonction std::set_terminate, qui attend en paramètre un pointeur sur la fonction de traitement d'erreur, qui ne prend aucun paramètre et renvoie void. La valeur renvoyée par std::set_terminate est le pointeur sur la fonction de traitement d'erreur précédente. std::terminate et std::set_terminate sont déclaréee dans le fichier d'en-tête exception.

> **Note:** Comme leur nom l'indique, std::terminate et std::set_terminate sont déclarées dans l'espace de nommage std::, qui est réservé pour toutes les objets de la librairie standard C++. Si vous ne voulez pas à avoir à utiliser systématiquement le préfixe std:: devant ces noms, vous devrez ajouter la ligne " using namespace std; " après avoir inclus l'en-tête exception. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 11.

Exemple 9-2. Installation d'un gestionnaire d'exception avec set_terminate

```
#include <iostream>
#include <exception>
```

```
using namespace std;

void mon_gestionnaire(void)
{
   cout << "Exception non gérée reçue !" << endl;
   cout << "Je termine le programme proprement..."
      << endl;
   exit(-1);
}

int lance_exception(void)
{
   throw 2;
}

int main(void)
{
   set_terminate(&mon_gestionnaire);
   try
   {
      lance_exception();
   }
   catch (double d)
   {
      cout << "Exception de type double reçue : " <<
         d << endl;
   }
   return 0;
}
```

9.3. Liste des exceptions autorisées pour une fonction

Il est possible de spécifier les exceptions qui peuvent apparaître dans une fonction. Pour cela, il faut faire suivre son en-tête du mot-clé throw, avec entre parenthèses, et séparées par des virgules, les classes des exceptions envisageables. Par exemple, la fonction suivante :

```
int fonction_sensible(void)
throw (int, double, erreur)
{
   ...
}
```

n'a le droit de lancer que des exceptions du type int, double ou erreur. Si une autre exception est lancée, par exemple une exception du type char *, il se produit encore une fois une erreur à l'exécution.

En fait, la fonction std::unexpected est appelée. Cette fonction se comporte de manière similaire à std::terminate, puisqu'elle appelle par défaut une fonction de traitement de l'erreur, qui elle-même appelle la fonction std::terminate (et donc abort en fin de compte). Ceci conduit à la terminaison du programme. On peut encore une fois changer ce comportement par défaut en remplaçant la fonction appelée par std::unexpected par une autre fonction à l'aide de std::set_unexpected, qui est déclarée dans le fichier d'en-tête exception. Cette dernière attend en paramètre un pointeur sur la fonction de traitement d'erreur, qui ne prend aucun paramètre et qui renvoie void. std::set_unexpected renvoie le pointeur sur la fonction de traitement d'erreur précédemment appelée par std::unexpected.

> **Note:** Comme leur nom l'indique, std::unexpected et std::set_unexpected sont déclarées dans l'espace de nommage std::, qui est réservé pour les objets de la librairie standard C++. Si vous

ne voulez pas avoir à utiliser systématiquement le préfixe std:: pour ces noms, vous devrez ajouter la ligne " using namespace std; " après avoir inclus l'en-tête exception. Vous obtiendrez de plus amples renseignements sur les espaces de nommage dans le Chapitre 11.

Il est possible de relancer une autre exception à l'intérieur de la fonction de traitement d'erreur. Si cette exception satisfait la liste des exceptions autorisées, le programme reprend son cours normalement dans le gestionnaire correspondant. C'est généralement ce que l'on cherche à faire. Le gestionnaire peut également lancer une exception de type std::bad_exception, déclarée comme suit dans le fichier d'en-tête exception :

```
class bad_exception : public exception
{
public:
   bad_exception(void) throw();
   bad_exception(const bad_exception &) throw();
   bad_exception &operator=(const bad_exception &) throw();
   virtual ~bad_exception(void) throw();
   virtual const char *what(void) const throw();
};
```

Ceci a pour conséquence de terminer le programme.

Enfin, le gestionnaire d'exceptions non autorisées peut directement mettre fin à l'exécution du programme en appelant std::terminate. C'est le comportement utilisé par la fonction std::unexpected définie par défaut.

Exemple 9-3. Gestion de la liste des exceptions autorisées

```
#include <iostream>
#include <exception>

using namespace std;

void mon_gestionnaire(void)
{
   cout << "Une exception illégale a été lancée." << endl;
   cout << "Je relance une exception de type int." << endl;
   throw 2;
}

int f(void) throw (int)
{
   throw "5.35";
}

int main(void)
{
   set_unexpected(&mon_gestionnaire);
   try
   {
      f();
   }
   catch (int i)
   {
      cout << "Exception de type int reçue : " <<
         i << endl;
   }
   return 0;
}
```

Note: La liste des exceptions autorisées dans une fonction ne fait pas partie de sa signature. Elle n'intervient donc pas dans les mécanismes de surcharge des fonctions. De plus, elle doit se placer après le mot-clé const dans les déclarations de fonctions membres const (en revanche, elle doit se placer avant =0 dans les déclarations des fonctions virtuelles pures).

On prendra garde au fait que les exceptions ne sont pas générées par le mécanisme de gestion des erreurs du C++ (ni du C). Cela signifie que pour avoir une exception, il faut la lancer, le compilateur ne fera pas les tests pour vous (tests de débordements numériques dans les calculs par exemple). Cela supposerait de prédéfinir un ensemble de classes pour les erreurs

génériques. Les tests de validité d'une opération doivent donc être faits malgré tout, et le cas échéant, il faut lancer une exception pour reporter le traitement en cas d'échec. De même, les exceptions générées par la machine hôte du programme ne sont en général pas récupérées par les implémentations, et si elles le sont, les programmes qui les utilisent ne sont pas portables.

9.4. Hiérarchie des exceptions

Le mécanisme des exceptions du C++ se base sur le typage des objets, puisqu'il le lancement d'une exception nécessite la construction d'un objet qui la caractérise, et le bloc catch destination de cette exception sera sélectionné en fonction du type de cet objet. Bien entendu, les objets utilisés pour lancer les exceptions peuvent contenir des informations concernant la nature des erreurs qui se produisent, mais il est également possible de classifier ces erreurs par catégories en se basant sur leurs type.

En effet, les objets exceptions peuvent être des instances de classes disposant de relations d'héritages. Comme les objets des classes dérivées peuvent être considérés comme des instances de leurs classes de base, les gestionnaire d'exceptions peuvent récupérer les exceptions de ces classes dérivées en récupérant un objet du type de leur classe de base. Ainsi, il est possible de classifier les différents cas d'erreurs en définissant une hiérarchie de classe d'exceptions, et d'écrire des traitements génériques en n'utilisant que les objets d'un certain niveau dans cette hiérarchie.

Le mécanisme des exceptions se montre donc plus puissant que toutes les autres méthodes de traitement d'erreurs à ce niveau, puisque la séléction du gestionnaire d'erreur est automatiquement réalisée par le langage. Ceci peut être très pratique pour peu que l'on ait défini correctement sa hiérarchie de classes d'exceptions.

Exemple 9-4. Classification des exceptions

```
#include <iostream>

using namespace std;

// Classe de base de toutes les exceptions :
class ExRuntimeError
{
};

// Classe de base des exceptions pouvant se produire
// lors de manipulations de fichiers :
class ExFileError : public ExRuntimeError
{
};

// Classes des erreurs de manipulation des fichiers :
class ExInvalidName : public ExFileError
{
};

class ExEndOfFile : public ExFileError
{
};

class ExNoSpace : public ExFileError
{
};

class ExMediumFull : public ExNoSpace
{
};
```

```
class ExFileSizeMaxLimit : public ExNoSpace
{
};

// Fonction faisant un travail quelconque sur un fichier :
void WriteData(const char *szFileName)
{
   // Exemple d'erreur :
   if (szFileName == NULL) throw ExInvalidName();
   else
   {
      // Traitement de la fonction
      // etc...

      // Lancement d'une exception :
      throw ExMediumFull();
   }
}

void Save(const char *szFileName)
{
   try
   {
      WriteData(szFileName);
   }
   // Traitement d'un erreur spécifique :
   catch (ExInvalidName &)
   {
      cout << "Impossible de faire la sauvegarde" << endl;
   }
   // Traitement de toutes les autres erreurs en groupe :
   catch (ExFileError &)
   {
      cout << "Erreur d'entrée / sortie" << endl;
   }
}

int main(void)
{
   Save(NULL);
   Save("data.dat");
   return 0;
}
```

La librairie standard C++ définit elle-même un certain nombre d'exceptions standard, qui sont utilisées pour signaler les erreurs qui se produisent à l'exécution des programmes. Quelques-unes de ces exceptions ont déjà été présentées avec les fonctionnalités qui sont susceptibles de les lancer. Vous trouverez une liste complète des exceptions de la librairie standard du C++ dans la Section 13.1.2.

9.5. Exceptions dans les constructeurs

Les constructeurs des objets globaux peuvent générer des exceptions. Dans ce cas cependant, comme ces objets sont construits avant l'entrée dans la fonction main, il est impossible de récupérer les exceptions lancées. Afin de résoudre ce problème, il est possible d'utiliser un bloc try pour le corps de fonction des constructeurs. Les blocs catch suivent alors la définition du constructeur, pour le traitement des exceptions lancées.

Exemple 9-5. Exceptions dans les constructeurs

```
// Classe implémentant un gestionnaire de périphérique :
class Driver
{
   Driver(void);      // Constructeur du driver
};
```

```
Driver::Driver(void)
try
{
   // Initialisation du périphérique.
   return ;
}
catch (...)
{
   // Erreur : le périphérique n'a pas pu être initialisé.
}
```

Dans cet exemple, lors de l'instanciation d'un objet de classe Driver, une erreur d'initialisation peut se produire. Cette erreur peut lancer une exception, qui est alors traitée dans le bloc catch qui suit la définition du constructeur de la classe Driver.

En général, si une classe hérite de une ou plusieurs classes de base, l'appel aux constructeurs des classes de base doit se faire entre le mot-clé try et la première accolade. En effet, les constructeurs des classes de base sont susceptibles, eux aussi, de lancer des exceptions. La syntaxe est alors la suivante :

```
Classe::Classe
  try : Base(paramètres) [, Base(paramètres) […]]
{
}
catch …
```

Il se peut, lors d'une allocation dynamique, que le constructeur de l'objet alloué lance une exception. Si une telle situation se produit, le compilateur appelle automatiquement l'opérateur delete afin de restituer la mémoire allouée, puisque l'objet n'a pas pu être construit. Il est donc inutile de restituer la mémoire de l'objet alloué dans le traitement de l'exception qui suit l'allocation dynamique de mémoire.

Chapitre 10. Identification dynamique des types

Le C++ est un langage fortement typé. Malgré cela, il se peut que le type exact d'un objet soit inconnu à cause de l'héritage. Par exemple, si un objet est considéré comme un objet d'une classe de base de sa véritable classe, on ne peut pas déterminer a priori quelle est sa véritable nature.

Cependant, les objets polymorphiques (qui, rappelons-le, sont des objets disposant de méthodes virtuelles) conservent des informations sur leur *type dynamique*, à savoir leur véritable nature. En effet, lors de l'appel des méthodes virtuelles, la méthode appelée est la méthode de la véritable classe de l'objet.

Il est possible d'utiliser cette propriété pour mettre en place un mécanisme permettant d'identifier le type dynamique des objets, mais cette manière de procéder n'est pas portable. Le C++ fournit donc un mécanisme standard permettant de manipuler les informations de type des objets polymorphiques. Ce mécanisme prend en charge l'*identification dynamique* des types et la *vérification de la validité des transtypages* dans le cadre de la dérivation.

10.1. Identification dynamique des types

10.1.1. L'opérateur typeid

Le C++ fournit l'opérateur typeid afin de récupérer les informations de type des expressions. Sa syntaxe est la suivante :

typeid(expression)

où expression est l'expression dont il faut déterminer le type.

Le résultat de l'opérateur typeid est une référence sur un objet constant de classe type_info. Cette classe sera décrite dans la Section 10.1.2.

Les informations de type récupérées sont les informations de type statique pour les types non polymorphiques. Ceci signifie que l'objet renvoyé par typeid caractérisera le type de l'expression fournie en paramètre, que cette expression soit un sous-objet d'un objet plus dérivé ou non. En revanche, pour les types polymorphiques, si le type ne peut pas être déterminé statiquement (c'est à dire à la compilation), une détermination dynamique (c'est à dire à l'exécution) du type a lieu, et l'objet de classe type_info renvoyé décrit le vrai type de l'expression (même si elle représente un sous-objet d'un objet d'une classe dérivée). Cette situation peut arriver lorsqu'on manipule un objet à l'aide d'un pointeur ou d'une référence sur une classe de base de la classe de cet objet.

Exemple 10-1. Opérateur typeid

```
#include <typeinfo>

using namespace std;

class Base
{
public:
  virtual ~Base(void);   // Il faut une fonction virtuelle
                         // pour avoir du polymorphisme.
};

class Derivee : public Base
{
public:
```

```
  virtual ~Derivee(void);
};

int main(void)
{
  Derivee* pd = new Derivee;
  Base* pb = pd;
  const type_info &t1=typeid(*pd);  // t1 qualifie le type de *pd.
  const type_info &t2=typeid(*pb);  // t2 qualifie le type de *pb.
  return 0 ;
}
```

Les objets t1 et t2 sont égaux, puisqu'ils qualifient tous les deux le même type (à savoir, la classe Derivee). t2 ne contient pas les informations de type de la classe Base, parce que le vrai type de l'objet pointé par pb est la classe Derivee.

> **Note:** Notez que la classe type_info est définie dans l'espace de nommage std::, réservé à la librairie standard C++, dans l'en-tête typeinfo. Par conséquent, son nom doit être précédé du préfixe std::. Vous pouvez vous passer de ce préfixe en important les définitions de l'espace de nommage de la librairie standard à l'aide d'une directive using. Vous trouverez de plus amples renseignements sur les espaces de nommage dans le Chapitre 11.

On fera bien attention à déréférencer les pointeurs, car sinon, on obtient les informations de type sur ce pointeur, pas sur l'objet pointé. Si le pointeur déréférencé est le pointeur nul, l'opérateur typeid lance une exception dont l'objet est une instance de la classe bad_typeid. Cette classe est définie comme suit dans l'en-tête typeinfo :

```
class bad_typeid : public logic
{
public:
  bad_typeid(const char * what_arg) : logic(what_arg)
  {
    return ;
  }

  void raise(void)
  {
    handle_raise();
    throw *this;
  }
};
```

10.1.2. La classe type_info

Les informations de type sont enregistrées dans des objets de la classe type_info, prédéfinie par le langage. Cette classe est déclarée dans l'en-tête typeinfo de la manière suivante :

```
class type_info
{
public:
  virtual ~type_info();
  bool operator==(const type_info &rhs) const;
  bool operator!=(const type_info &rhs) const;
  bool before(const type_info &rhs) const;
  const char *name() const;
private:
  type_info(const type_info &rhs);
  type_info &operator=(const type_info &rhs);
};
```

Les objets de la classe type_info ne peuvent pas être copiés, puisque l'opérateur d'affectation et le constructeur

de copie sont tous les deux déclarés private. Par conséquent, le seul moyen de générer un objet de la classe type_info est d'utiliser l'opérateur typeid.

Les opérateurs de comparaison permettent de tester l'égalité et la différence de deux objets type_info, ce qui revient exactement à comparer les types des expressions.

Les objets type_info contiennent des informations sur les types sous la forme de chaînes de caractères. Une de ces chaînes représente le type sous une forme lisible par un être humain, et une autre sous une forme plus appropriée pour le traitement des types. Le format de ces chaînes de caractères n'est pas précisé et peut varier d'une implémentation à une autre. Il est possible de récupérer le nom lisible du type à l'aide de la méthode name. La valeur renvoyée est un pointeur sur une chaîne de caractères. On ne doit pas libérer la mémoire utilisée pour stocker cette chaîne de caractères.

La méthode before permet de déterminer un ordre dans les différents types appartenant à la même hiérarchie de classes, en se basant sur les propriétés d'héritage. L'utilisation de cette méthode est toutefois difficile, puisque l'ordre entre les différentes classes n'est pas fixé et peut dépendre de l'implémentation.

10.2. Transtypages C++

Les règles de dérivation permettent d'assurer le fait que lorsqu'on utilise un pointeur sur une classe, l'objet pointé existe bien et est bien de la classe sur laquelle le pointeur est basé. En particulier, il est possible de convertir un pointeur sur un objet en un pointeur sur un sous-objet.

En revanche, il est interdit d'utiliser un pointeur sur une classe de base pour initialiser un pointeur sur une classe dérivée. Pourtant, cette opération peut être légale, si le programmeur sait que le pointeur pointe bien sur un objet de la classe dérivée. Le langage exige cependant un transtypage explicite. Une telle situation demande l'analyse du programme afin de savoir si elle est légale ou non.

Parfois, il est impossible de faire cette analyse. Ceci signifie que le programmeur ne peut pas certifier que le pointeur dont il dispose est un pointeur sur un sous-objet. Le mécanisme d'identification dynamique des types peut être alors utilisé pour vérifier, à l'exécution, si le transtypage est légal. S'il ne l'est pas, un traitement particulier doit être effectué, mais s'il l'est, le programme peut se poursuivre normalement.

Le C++ fournit un jeu d'opérateurs de transtypage qui permettent de faire ces vérifications dynamiques, et qui donc sont nettement plus sûrs que le transtypage tout puissant du C que l'on a utilisé jusqu'ici. Ces opérateurs sont capables de faire un *transtypage dynamique*, un *transtypage statique*, un *transtypage de constance* et un *transtypage de réinterprétation* des données. Nous allons voir les différents opérateurs permettant de faire ces transtypages, ainsi que leur signification.

10.2.1. Transtypage dynamique

Le *transtypage dynamique* permet de convertir une expression en un pointeur ou une référence d'une classe, ou un pointeur sur void. Il est réalisé à l'aide de l'opérateur dynamic_cast. Cet opérateur impose des restrictions lors des transtypages afin de garantir une plus grande fiabilité :

- il effectue une vérification de la validité du transtypage ;
- il n'est pas possible d'éliminer les qualifications de constance (pour cela, il faut utiliser l'opérateur const_cast, que l'on verra plus loin).

En revanche, l'opérateur dynamic_cast permet parfaitement d'accroître la constance d'un type complexe, comme le font les conversions implicites du langage vues dans la Section 3.3 et dans la Section 4.9.

Il ne peut pas travailler sur les types de base du langage, sauf void *.

La syntaxe de l'opérateur dynamic_cast est donnée ci-dessous :

```
dynamic_cast<type>(expression)
```

où type désigne le type cible du transtypage, et expression l'expression à transtyper.

Le transtypage d'un pointeur ou d'une référence d'une classe dérivée en classe de base se fait donc directement, sans vérification dynamique, puisque cette opération est toujours valide. Les lignes suivantes :

```
// La classe B hérite de la classe A :

B *pb;
A *pA=dynamic_cast<A *>(pB);
```

sont donc strictement équivalentes à celles-ci :

```
// La classe B hérite de la classe A :

B *pb;
A *pA=pB;
```

Tout autre transtypage doit se faire à partir d'un type polymorphique, afin que le compilateur puisse utiliser l'identification dynamique des types lors du transtypage. Le transtypage d'un pointeur d'un objet vers un pointeur de type void renvoie l'adresse du début de l'objet le plus dérivé, c'est à dire l'adresse de l'objet complet. Le transtypage d'un pointeur ou d'une référence sur un sous-objet d'un objet vers un pointeur ou une référence de l'objet complet est effectué après vérification du type dynamique. Si l'objet pointé ou référencé est bien du type indiqué pour le transtypage, l'opération se déroule correctement. En revanche, s'il n'est pas du bon type, dynamic_cast n'effectue pas le transtypage. Si le type cible est un pointeur, le pointeur nul est renvoyé. Si en revanche l'expression caractérise un objet ou une référence d'objet, une exception de type bad_cast est lancée.

La classe bad_cast est définie comme suit dans l'en-tête typeinfo :

```
class bad_cast : public exception
{
public:
  bad_cast(void) throw();
  bad_cast(const bad_cast&) throw();
  bad_cast &operator=(const bad_cast&) throw();
  virtual ~bad_cast(void) throw();
  virtual const char* what(void) const throw();
};
```

Lors d'un transtypage, aucune ambiguïté ne doit avoir lieu pendant la recherche dynamique du type. De telles ambiguïtés peuvent apparaître dans les cas d'héritage multiple, où plusieurs objets de même type peuvent coexister dans le même objet. Cette restriction mise à part, l'opérateur dynamic_cast est capable de parcourir une hiérarchie de classe aussi bien verticalement (convertir un pointeur de sous-objet vers un pointeur d'objet complet) que transversalement (convertir un pointeur d'objet vers un pointeur d'un autre objet frère dans la hiérarchie de classes).

L'opérateur dynamic_cast peut être utilisé dans le but de convertir un pointeur sur une classe de base virtuelle vers une des ses classes filles, ce que ne pouvaient pas faire les transtypages classiques du C. En revanche, il ne peut pas être utilisé afin d'accéder à des classes de base qui ne sont pas visibles (en particulier, les classes de base héritées en private).

Exemple 10-2. Opérateur dynamic_cast

```
struct A
{
  virtual void f(void)
  {
    return ;
  };
};

struct B : virtual public A
```

```
{
};

struct C : virtual public A, public B
{
};

struct D
{
  virtual void g(void)
  {
    return ;
  };
};

struct E : public B, public C, public D
{
};

int main(void)
{
  E e;        // e contient deux sous-objets de classe B
              // (mais un seul sous-objet de classe A).
              // Les sous-objets de classe C et D sont
              // frères.
  A *pA=&e;   // Dérivation légale : le sous-objet
              // de classe A est unique.
  // C *pC=(C *) pA;// Illégal : A est une classe de base
                    // virtuelle (erreur de compilation).
  C *pC=dynamic_cast<C *>(pA);  // Légal. Transtypage
                                // dynamique vertical.
  D *pD=dynamic_cast<D *>(pC);  // Légal. Transtypage
                                // dynamique horizontal.
  B *pB=dynamic_cast<B *>(pA);  // Légal, mais échouera
                                // à l'exécution (ambiguïté).
  return 0 ;
}
```

10.2.2. Transtypage statique

Contrairement au transtypage dynamique, le transtypage statique n'effectue aucune vérification des types dynamiques lors du transtypage. Il est donc nettement plus dangereux que le transtypage dynamique. Cependant, contrairement au transtypage C classique, il ne permet toujours pas de supprimer les qualifications de constance.

Le transtypage statique s'effectue à l'aide de l'opérateur static_cast, dont la syntaxe est exactement la même que celle de l'opérateur dynamic_cast :

```
static_cast<type>(expression)
```

où type et expression ont les mêmes signification que pour l'opérateur dynamic_cast.

Essentiellement, l'opérateur static_cast n'effectue l'opération de transtypage que si l'expression suivante est valide :

```
type temporaire(expression);
```

Cette expression construit un objet temporaire quelconque de type type et l'initialise avec la valeur de expression. Contrairement à l'opérateur dynamic_cast, l'opérateur static_cast permet donc d'effectuer les conversions entre les types autres que les classes définies par l'utilisateur. Aucune vérification de la validité de la conversion n'a lieu cependant (comme pour le transtypage C classique).

Si une telle expression n'est pas valide, le transtypage ne peut avoir lieu qu'entre classes dérivées et classes de base. L'opérateur static_cast permet d'effectuer ces transtypages dans les deux sens (classe de base vers classe dérivée et classe dérivée vers classe de base). Le transtypage d'une classe de base vers une classe dérivée ne doit être fait que lorsque l'on est sûr qu'il n'y a pas de danger, puisqu'aucune vérification dynamique n'a lieu avec static_cast.

Enfin, toutes les expressions peuvent être converties en void avec des qualifications de constance et de volatilité. Cette opération a simplement pour but de supprimer la valeur de l'expression (puisque void représente le type vide).

10.2.3. Transtypage de constance et de volatilité

La suppression des attributs de constance et de volatilité peut être réalisée grâce à l'opérateur const_cast. Cet opérateur suit exactement la même syntaxe que les opérateurs dynamic_cast et static_cast :

```
const_cast<type>(expression)
```

L'opérateur const_cast peut travailler essentiellement avec des références et des pointeurs. Il permet de réaliser les transtypages dont le type destination est moins contraint que le type source vis à vis des mots-clés const et volatile.

En revanche, l'opérateur const_cast ne permet pas d'effectuer d'autres conversions que les autres opérateurs de transtypage (ou simplement les transtypages C classiques) peuvent réaliser. Par exemple, il est impossible de l'utiliser pour convertir un flottant en entier. Lorsqu'il travaille avec des références, l'opérateur const_cast vérifie que le transtypage est légal en convertissant les références en pointeurs et en regardant si le transtypage n'implique que les attributs const et volatile. const_cast ne permet pas de convertir les pointeurs de fonctions.

10.2.4. Réinterprétation des données

L'opérateur de transtypage le plus dangereux est reinterpret_cast. Sa syntaxe est la même que celle des autres opérateurs de transtypage dynamic_cast, static_cast et const_cast :

```
reinterpret_cast<type>(expression)
```

Cet opérateur permet de réinterpréter les données d'un type en un autre type. Aucune vérification de la validité de cette opération n'est faite. Ainsi, les lignes suivantes :

```
double f=2.3;
int i=1;
const_cast<int &>(f)=i;
```

sont strictement équivalentes aux lignes suivantes :

```
double f=2.3;
int i=1;
*((int *) &f)=i;
```

L'opérateur reinterpret_cast doit cependant respecter les règles suivantes :

- il ne doit pas permettre la suppression des attributs de constance et de volatilité ;
- il doit être symétrique (c'est à dire que la réinterprétation d'un type T1 en tant que type T2, puis la réinterprétation du résultat en type T1 doit donner l'objet initial).

Chapitre 11. Espaces de nommage

Les *espaces de nommage* sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur. Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet. Par exemple, si deux programmeurs définissent différemment une même structure dans deux fichiers différents, un conflit entre ces deux structures aura lieu au mieux à l'édition de lien, et au pire lors de l'utilisation commune des sources de ces deux programmeurs. Ce type de conflit provient du fait que le C++ ne fournit qu'un seul espace de nommage de portée globale, dans lequel il ne doit y avoir aucun conflit de nom. Grâce aux espaces de nommage non globaux, ce type de problème peut être plus facilement évité, parce que l'on peut éviter de définir les objets globaux dans la portée globale.

11.1. Définition des espaces de nommage

11.1.1. Espaces de nommage nommées

Lorsque le programmeur donne un nom à un espace de nommage, celui-ci est appelé un *espace de nommage nommé*. La syntaxe de ce type d'espace de nommage est la suivante :

```
namespace nom
{
  déclarations | définitions
}
```

nom est le nom de l'espace de nommage, et déclarations et définitions sont les déclarations et les définitions des identificateurs qui lui appartiennent.

Contrairement aux régions déclaratives classiques du langage (comme par exemple les classes), un namespace peut être découpé en plusieurs morceaux. Le premier morceaux sert de déclaration, et les suivants d'extensions. La syntaxe pour une extension d'espace de nommage est exactement la même que celle de la partie de déclaration.

Exemple 11-1. Extension de namespace

```
namespace A   // Déclaration de l'espace de nommage A.
{
  int i;
}

namespace B   // Déclaration de l'espace de nommage B.
{
  int i;
}

namespace A   // Extension de l'espace de nommage A.
{
  int j;
}
```

Les identificateurs déclarés ou définis à l'intérieur d'un même espace de nommage ne doivent pas entrer en conflit. Ils peuvent avoir les mêmes noms, mais seulement dans le cadre de la surcharge. Un espace de nommage se comporte donc exactement comme aux zones de déclaration des classes et de la portée globale.

L'accès aux identificateurs des espaces de nommage se fait par défaut grâce à l'opérateur de résolution de portée (::), et en qualifiant le nom de l'identificateur à utiliser du nom de son espace de nommage. Cependant, cette qualification est inutile à l'intérieur de l'espace de nommage lui-même, exactement comme pour les membres des classes.

Exemple 11-2. Accès aux membres d'un namespace

```
int i=1;    // i est global.

namespace A
{
  int i=2; // i de l'espace de nommage A.
  int j=i; // Utilise A::i.
}

int main(void)
{
  i=1;     // Utilise ::i.
  A::i=3;  // Utilise A::i.
  return 0;
}
```

Les fonctions membres d'un espace de nommage peuvent être définies à l'intérieur de cet espace, exactement comme les fonctions membres de classes. Elles peuvent également être définies en dehors de cet espace, si l'on utilise l'opérateur de résolution de portée. Les fonctions ainsi définies doivent apparaître après leur déclaration dans l'espace de nommage.

Exemple 11-3. Définition externe d'une fonction de namespace

```
namespace A
{
  int f(void);   // Déclaration de A::f.
}

int A::f(void)    // Définition de A::f.
{
  return 0;
}
```

Il est possible de définir un espace de nommage à l'intérieur d'un autre espace de nommage. Cependant, cette déclaration doit obligatoirement avoir lieu au niveau déclaratif le plus externe de l'espace de nommage qui contient le sous-espace de nommage. On ne peut donc pas déclarer d'espaces de nommage à l'intérieur d'une fonction ou à l'intérieur d'une classe.

Exemple 11-4. Définition de namespace dans un namespace

```
namespace Conteneur
{
  int i;              // Conteneur::i.
  namespace Contenu
  {
    int j;            // Conteneur::Contenu::j.
  }
}
```

11.1.2. Espaces de nommage anonymes

Lorsque, lors de la déclaration d'un espace de nommage, aucun nom n'est donné, un *espace de nommage anonyme* est créé. Ce type d'espace de nommage permet d'assurer l'unicité du nom de l'espace de nommage ainsi déclaré. Les espaces de nommage anonymes peuvent donc remplacer efficacement le mot-clé static pour rendre unique des identificateurs dans un fichier. Cependant, elles sont plus puissantes, parce que l'on peut également déclarer des espaces de nommage anonymes à l'intérieur d'autres espaces de nommage.

Exemple 11-5. Définition de namespace anonyme

```
namespace
```

```
{
  int i;        // Équivalent à unique::i;
}
```

Dans l'exemple précédent, la déclaration de i se fait dans un espace de nommage dont le nom est choisi par le compilateur de manière unique. Cependant, comme on ne connaît pas ce nom, le compilateur utilise une directive using (voir plus loin) afin de pouvoir utiliser les identificateurs de cet espace de nommage anonyme sans préciser leur nom complet avec l'opérateur de résolution de portée.

Si, dans un espace de nommage, un identificateur est déclaré avec le même nom qu'un autre identificateur déclaré dans un espace de nommage plus global, l'identificateur global est masqué. Dans le cas des espaces de nommage nommés, l'accès peut être réalisé à l'aide de l'opérateur de résolution de portée. En revanche, il est impossible d'y accéder avec les espaces de nommage anonymes, puisqu'on ne peut pas préciser le nom de ces derniers.

Exemple 11-6. Ambiguïtés entre namespaces

```
namespace
{
  int i;         // Déclare unique::i.
}

void f(void)
{
  i++;           // Utilise unique::i.
}

namespace A
{
  namespace
  {
    int i;      // Définit A::unique::i.
    int j;      // Définit A::unique::j.
  }

  void g(void)
  {
    i++;        // Erreur : ambiguïté entre unique::i
                // et A::unique::i.
    A::i++;     // Erreur : A::i n'est pas défini
                // (seul A::unique::i l'est).
    j++;        // Correct : A::unique::j++.
  }
}
```

11.1.3. Alias d'espaces de nommage

Lorsqu'un espace de nommage porte un nom très compliqué, il peut être avantageux de définir un *alias* pour ce nom. L'alias aura alors un nom plus simple.

Cette opération peut être réalisée à l'aide de la syntaxe suivante :

```
namespace nom_alias = nom;
```

nom_alias est ici le nom de l'alias de l'espace de nommage, et nom est le nom de l'espace de nommage lui-même.

Les noms donnés aux alias d'espaces de nommage ne doivent pas entrer en conflit avec les noms des autres

identificateurs du même espace de nommage, que celui-ci soit l'espace de nommage de portée globale ou non.

11.2. Déclaration using

Les *déclarations using* permettent d'utiliser un identificateur d'un espace de nommage de manière simplifiée, sans avoir à spécifier son nom complet (c'est à dire le nom de l'espace de nommage suivi du nom de l'identificateur).

11.2.1. Syntaxe des déclarations using

La syntaxe des déclarations using est la suivante :

using identificateur;

où identificateur est le nom complet de l'identificateur à utiliser, avec qualification d'espace de nommage.

Exemple 11-7. Déclaration using

```
namespace A
{
  int i;       // Déclare A::i.
  int j;       // Déclare A::j.
}

void f(void)
{
  using A::i;   // A::i peut être utilisé sous le nom i.
  i=1;          // Équivalent à A::i=1.
  j=1;          // Erreur ! j n'est pas défini !
  return ;
}
```

Les déclarations using permettent en fait de déclarer des alias des identificateurs. Ces alias doivent être considérés exactement comme des déclarations normales. Ceci signifie qu'ils ne peuvent être déclarés plusieurs fois que lorsque les déclarations multiples sont autorisées (déclarations de variables ou de fonctions en dehors des classes), et de plus ils appartiennent à l'espace de nommage dans lequel ils sont définis.

Exemple 11-8. Déclarations using multiples

```
namespace A
{
  int i;
  void f(void);
}

namespace B
{
  using A::i;  // Déclaration de l'alias B::i, qui représente A::i.
  using A::i;  // Légal : double déclaration de A::i.

  using A::f;  // Déclare void B::f(void),
               // fonction identique à A::f.
}

int main(void)
{
  B::f();      // Appelle A::f.
  return 0;
}
```

L'alias créé par une déclaration using permet de référencer uniquement les identificateurs qui sont visibles au moment où la déclaration using est faite. Si l'espace de nommage concerné par la déclaration using est étendu après cette dernière, les nouveaux identificateurs de même nom que celui de l'alias ne seront pas pris en compte.

Exemple 11-9. Extension de namespace après une déclaration using

```
namespace A
{
  void f(int);
}

using A::f;          // f est synonyme de A::f(int).

namespace A
{
  void f(char);      // f est toujours synonyme de A::f(int),
                     // mais pas de A::f(char).
}

void g()
{
  f('a');            // Appelle A::f(int), même si A::f(char)
                     // existe.
}
```

Si plusieurs déclarations locales et using déclarent des identificateurs de même nom, ou bien ces identificateurs doivent tous se rapporter au même objet, ou bien ils doivent représenter des fonctions ayant des signatures différentes (les fonctions déclarées sont donc surchargées). Dans le cas contraire, des ambiguïtés peuvent apparaître et le compilateur signale une erreur lors de la déclaration using.

Exemple 11-10. Conflit entre déclarations using et identificateurs locaux

```
namespace A
{
  int i;
  void f(int);
}

void g(void)
{
  int i;        // Déclaration locale de i.
  using A::i;   // Erreur : i est déjà déclaré.
  void f(char); // Déclaration locale de f(char).
  using A::f;   // Pas d'erreur, il y a surcharge de f.
  return ;
}
```

> **Note:** Ce comportement diffère de celui des directives using. En effet, les directives using reportent la détection des erreurs à la première utilisation des identificateurs ambigus.

11.2.2. Utilisation des déclarations using dans les classes

Une déclaration using peut être utilisée dans la définition d'une classe. Dans ce cas, elle doit se rapporter à une classe de base de la classe dans laquelle elle est utilisée. De plus, l'identificateur donné à la déclaration using doit être accessible dans la classe de base (c'est à dire de type protected ou public).

Exemple 11-11. Déclaration using dans une classe

```
namespace A
{
  float f;
}
```

```
class Base
{
  int i;
public:
  int j;
};

class Derivee : public Base
{
  using A::f;      // Illégal : f n'est pas dans une classe
                   // de base.
  using Base::i;   // Interdit : Derivee n'a pas le droit
                   // d'utiliser Base::i.
public:
  using Base::j;   // Légal.
};
```

Dans l'exemple précédent, seule la troisième déclaration est valide, parce que c'est la seule qui se réfère à un membre accessible de la classe de base. Le membre *j* déclaré sera donc un synonyme de Base::j dans la classe Derivee.

En général, les membres des classes de base sont accessibles directement. Quelle est donc l'utilité des déclarations using dans les classes ? En fait, elles peuvent être utilisées pour rétablir les droits d'accès, modifiés par un héritage, à des membres de classes de base. Pour cela, il suffit de placer la déclaration using dans une zone de déclaration public, protected ou private dans laquelle le membre se trouvait dans la classe de base. Cependant, comme on l'a vu ci-dessus, une classe ne peut pas rétablir les droits d'accès d'un membre private des classes de base.

Exemple 11-12. Rétablissement de droits d'accès à l'aide d'une directive using

```
class Base
{
public:
  int i;
  int j;
};

class Derivee : private Base
{
public:
  using Base::i;  // Rétablit l'accessibilité sur Base::i.
protected:
  using Base::i;  // Interdit : restreint l'accessibilité
                  // sur Base::i autrement que par héritage.
};
```

> **Note:** Certains compilateurs interprètent différemment le paragraphe 11.3 des Draft Papers, qui concerne l'accessibilité des membres introduits avec une déclaration using. Selon eux, les déclarations using permettent de restreindre l'accessibilité des droits et non pas de les rétablir. Ceci implique qu'il est impossible de redonner l'accessibilité à des données pour lesquelles l'héritage a restreint l'accès. Par conséquent, l'héritage doit être fait de la manière la plus permissive possible, et les accès doivent être ajustés au cas par cas. Bien que cette interprétation soit tout à fait valable, l'exemple donné dans les Draft Papers semble indiquer qu'elle n'est pas correcte.

Quand une fonction d'une classe de base est introduite dans une classe dérivée à l'aide d'une déclaration using, et qu'une fonction de même nom et de même signature est définie dans la classe dérivée, cette dernière fonction surcharge la fonction de la classe de base. Il n'y a pas d'ambiguïté dans ce cas.

11.3. Directive using

La *directive using* permet d'utiliser, sans spécification d'espace de nommage, non pas un identificateur, comme dans le cas de la déclaration using, mais tous les identificateurs de cet espace de nommage.

La syntaxe de la directive using est la suivante :

```
using namespace nom;
```

où nom est le nom de l'espace de nommage dont les identificateurs doivent être utilisés sans qualification complète.

Exemple 11-13. Directive using

```
namespace A
{
  int i;       // Déclare A::i.
  int j;       // Déclare A::j.
}

void f(void)
{
  using namespace A; // On utilise les identificateurs de A.
  i=1;         // Équivalent à A::i=1.
  j=1;         // Équivalent à A::j=1.
  return ;
}
```

Après une directive using, il est toujours possible d'utiliser les noms complets des identificateurs de l'espace de nommage, mais ce n'est plus nécessaire. Les directives using sont valides à partir de la ligne où elles sont déclarées jusqu'à la fin du bloc de portée courante. Si un espace de nommage est étendu après une directive using, les identificateurs définis dans l'extension de l'espace de nommage peuvent être utilisés exactement comme les identificateurs définis avant la directive using (c'est à dire sans qualification complète de leurs noms).

Exemple 11-14. Extension de namespace après une directive using

```
namespace A
{
  int i;
}

using namespace A;

namespace A
{
  int j;
}

void f(void)
{
  i=0;    // Initialise A::i.
  j=0;    // Initialise A::j.
  return ;
}
```

Il se peut que lors de l'introduction des identificateurs d'un espace de nommage par une directive using, des conflits de noms apparaissent. Dans ce cas, aucune erreur n'est signalée lors de la directive using. En revanche, une erreur se produit si un des identificateurs pour lesquels il y a conflit est utilisé.

Exemple 11-15. Conflit entre directive using et identificateurs locaux

```
namespace A
{
```

```
  int i;  // Définit A::i.
}

namespace B
{
  int i;  // Définit B::i.
  using namespace A;   // A::i et B::i sont en conflit.
                       // Cependant, aucune erreur n'apparaît.
}

void f(void)
{
  using namespace B;
  i=2;    // Erreur : il y a ambiguïté.
  return ;
}
```

Chapitre 12. Les template

12.1. Généralités

Nous avons vu précédemment comment réaliser des structures de données relativement indépendantes de la classe de leurs données (c'est à dire de leur type) avec les classes abstraites. Par ailleurs, il est faisable de faire des fonctions travaillant sur de nombreux types grâce à la surcharge. Je rappelle qu'en C++, tous les types sont en fait des classes.

Cependant, l'emploi des classes abstraites est assez fastidieux, et la surcharge n'est pas généralisable pour tous les types de données. Il serait possible d'utiliser des macros pour faire des fonctions atypiques mais cela serait au détriment de la taille du code.

Le C++ permet de résoudre ces problèmes grâce aux paramètres génériques, que l'on appelle encore *paramètres template*. Un paramètre template est soit un *type générique*, soit une *constante* dont le type est assimilable à un type intégral. Comme leur nom l'indique, les paramètres template permettent de paramétrer la définition des fonctions et des classes. Les fonctions et les classes ainsi paramétrées sont appelées respectivement *fonctions template* et *classes template*.

Les *fonctions template* sont donc des fonctions qui peuvent travailler sur des objets dont le type est un type générique (c'est à dire un type quelconque), ou qui peuvent êtres paramétrés par une constante de type intégral. Les *classes template* sont des classes qui contiennent des membres dont le type est générique ou qui dépendent d'un paramètre intégral.

En général, la génération du code a lieu lors d'une opération au cours de laquelle les types génériques sont remplacés par des vrais types et les paramètres de type intégral prennent leur valeur. Cette opération s'appelle l'*instanciation des template*. Elle a lieu lorsque l'on utilise la fonction ou la classe template pour la première fois. Les types réels à utiliser à la place des types génériques sont déterminés lors de cette première utilisation par le compilateur, soit implicitement à partir du contexte d'utilisation du template, soit par les paramètres donnés explicitement par le programmeur.

12.2. Déclaration des paramètres template

Les paramètres template sont, comme on l'a vu plus haut, soit des types génériques, soit des constantes dont le type peut être assimilé à un type intégral.

12.2.1. Déclaration des types template

Les template qui sont des types génériques sont déclarés par la syntaxe suivante :

```
template <class|typename nom[=type]
        [, class|typename nom[=type]
        […]>
```

où nom est le nom que l'on donne au type générique dans cette déclaration. Le mot-clé class a ici exactement la signification de " type ". Il peut d'ailleurs être remplacé indifféremment dans cette syntaxe par le mot-clé typename. La même déclaration peut être utilisé pour déclarer un nombre arbitraire de types génériques, en les séparant par des virgules. Les paramètres template qui sont des types peuvent prendre des valeurs par défaut, en faisant suivre le nom du paramètre d'un signe égal et de la valeur. Ici, la valeur par défaut doit évidemment être un type déjà déclaré.

Exemple 12-1. Déclaration de paramètres template

```
template <class T, typename U, class V=int>
```

Dans cet exemple, T, U et V sont des types génériques. Ils peuvent remplacer n'importe quel type du langage déjà déclaré au moment où la déclaration template est faite. De plus, le type générique V a pour valeur par défaut le type entier int. On voit bien dans cet exemple que les mots-clés typename et class peuvent être utilisés indifféremment.

Lorsque l'on donne des valeurs par défaut à un type générique, on doit donner des valeurs par défaut à tous les types génériques qui le suivent dans la déclaration template. La ligne suivante provoquera donc une erreur de compilation :

```
template <class T=int, class V>
```

Il est possible d'utiliser une classe template en tant que type générique. Dans ce cas, la classe doit être déclarée comme étant template à l'intérieur même de la déclaration template. La syntaxe est donc la suivante :

```
template <template <class Type> class Classe [,...]>
```

où Type est le type générique utilisé dans la déclaration de la classe template Classe. On appelle les paramètres template qui sont des classes template des paramètres *template template*. Rien n'interdit de donner une valeur par défaut à un paramètre template template : le type utilisé doit alors être une classe template déclarée avant la déclaration template.

Exemple 12-2. Déclaration de paramètre template tempalte

```
template <class T>
class Tableau
{
  // Définition de la classe template Tableau.
};

template <class U, class V, template <class T> class C=Tableau>
class Dictionnaire
{
  C<U> Clef;
  C<V> Valeur;
  // Reste de la définition de la classe Dictionnaire.
};
```

Dans cet exemple, la classe template Dictionnaire permet de relier des clés à leurs éléments. Ces clés et ces valeurs peuvent prendre n'importe quel type. Les clés et les valeurs sont stockées parallèlement dans les membres *Clef* et *Valeur*. Ces membres sont en fait des conteneurs template, dont la classe est générique et désignée par le paramètre template template C. Le paramètre template de C est utilisé pour donner le type des données stockées, à savoir les types génériques U et V dans le cas de la classe Dictionnaire. Enfin, la classe Dictionnaire peut utiliser un conteneur par défaut, qui est la classe template Tableau.

Pour plus de détails sur la déclaration des classes template, voir la Section 12.3.2.

12.2.2. Déclaration des constantes template

La déclaration des paramètres template de type constante se fait de la manière suivante :

```
template <type paramètre[=valeur][, ...]>
```

où type est le type du paramètre constant, paramètre est le nom du paramètre et valeur est sa valeur par défaut. Il est possible de donner des paramètres template qui sont des types génériques et des paramètres template qui sont des constantes dans la même déclaration.

Le type des constantes template doit être obligatoirement l'un des types suivants :

- type intégral (char, wchar_t, int, long, short et leurs versions signées et non signées) ou énuméré ;
- pointeur ou références d'objets ;

- pointeurs ou références de fonctions ;
- pointeurs sur membres.

Ce sont donc tous les types qui peuvent être assimilées à des valeurs entières (entiers, énumérés ou adresses).

Exemple 12-3. Déclaration de paramètres template de type constante

```
template <class T, int i, void (*f)(int)>
```

Cette déclaration template comprend un type générique T, une constante template i de type int, et une constante template f de type pointeur sur fonction prenant un entier en paramètre et ne renvoyant rien.

> **Note:** Les paramètres constants de type référence ne peuvent pas être initialisés avec une donnée immédiate ou une donnée temporaire lors de l'instanciation du template. Voir la Section 12.4 pour plus de détails sur l'instanciation des template.

12.3. Fonctions et classes template

Après la déclaration d'un ou de plusieurs paramètres template suit en général la définition d'une fonction ou d'une classe template. Dans cette définition, les types génériques peuvent être utilisés exactement comme s'il s'agissait de types normaux. Les constantes template peuvent être utilisées dans la fonction ou la classe template comme des constantes locales.

12.3.1. Fonctions template

La déclaration et la définition des fonctions template se fait exactement comme si la fonction était une fonction normale, à ceci près qu'elle doit être précédée de la déclaration des paramètres template. La syntaxe d'une déclaration de fonction template est donc la suivante :

```
template <paramètres_template>
type fonction(paramètres_fonction);
```

où paramètre_template est la liste des paramètres template et paramètres_fonction est la liste des paramètres de la fonction fonction. type est le type de la valeur de retour de la fonction, ce peut être un des types génériques de la liste des paramètres template.

Tous les paramètres template qui sont des types doivent être utilisés dans la liste des paramètres de la fonction, à moins qu'une instanciation explicite de la fonction ne soit utilisée. Ceci permet au compilateur de réaliser l'identification des types génériques avec les types à utiliser lors de l'instanciation de la fonction. Voir la Section 12.4 pour plus de détails à ce sujet.

La définition d'une fonction template se fait comme une déclaration avec le corps de la fonction. Il est alors possible d'y utiliser les paramètres template comme s'ils étaient normaux : des variables peuvent être déclarés avec un type générique, et les constantes template peuvent être utilisées comme des variables définies localement avec la classe de stockage const. Les fonctions template s'écrivent donc exactement comme des fonctions classiques.

Exemple 12-4. Définition de fonction template

```
template <class T>
T Min(T x, T y)
{
  return x<y ? x : y;
}
```

La fonction Min ainsi définie fonctionnera parfaitement pour toute classe pour laquelle l'opérateur < est défini. Le compilateur déterminera automatiquement quel est l'opérateur à employer pour chaque fonction Min qu'il rencontrera.

Les fonctions template peuvent être surchargées, aussi bien par des fonctions classiques que par d'autres fonctions template. Lorsqu'il y a ambiguïté entre une fonction template et une fonction normale qui la surcharge, toutes les références sur le nom commun à ces fonctions se rapporteront à la fonction classique.

Une fonction template peut être déclarée amie de toute classe, template ou non, pourvu que cette classe ne soit pas locale. Toutes les instances générées à partir d'une fonction amie template sont amies de la classe donnant l'amitié, et ont donc libre accès sur toutes les données de cette classe.

12.3.2. Les classes template

La déclaration et la définition d'une classe template se font comme celles d'une fonction template : elles doivent être précédées de la déclaration template des types génériques. La déclaration suit donc la syntaxe suivante :

```
template <paramètres_template>
class|struct|union nom;
```

où paramètres_template est la liste des paramètres template utilisés par la classe template nom.

La seule particularité dans la définition des classes template est que si les méthodes de la classe ne sont pas définies dans la déclaration de la classe, elles devront elles aussi être déclarées template :

```
template <paramètres_template>
type classe<paramètres>::nom(paramètres_méthode)
{
  ...
}
```

où paramètre_template représente la liste des paramètres template de la classe template classe, nom représente le nom de la méthode à définir, et paramètres_méthode ses paramètres.

Il est absolument nécessaire dans ce cas de spécifier tous les paramètres template de la liste paramètres_template dans paramètres, séparés par des virgules, afin de caractériser le fait que c'est la classe classe qui est template et qu'il ne s'agit pas d'une méthode template d'une classe normale. D'une manière générale, il faudra toujours spécifier les types génériques de la classe entre crochets, juste après son nom, à chaque fois qu'on voudra la référencer. Cette règle est cependant facultative lorsque la classe est référencée à l'intérieur d'une fonction membre.

Contrairement aux fonctions template non membres, les méthodes des classes template peuvent utiliser des types génériques de leur classe sans pour autant qu'ils soient utilisés dans la liste de leurs paramètres. En effet, le compilateur détermine quels sont les types à identifier aux types génériques lors de l'instanciation de la classe template, et n'a donc pas besoin d'effectuer cette identification avec les types des paramètres utilisés. Voir la Section 12.3.3 pour plus de détails à ce sujet.

Exemple 12-5. Définition d'une pile template

```
template <class T>
class Stack
{
  typedef struct stackitem
  {
    T Item;               // On utilise le type T comme
    struct stackitem *Next; // si c'était un type normal.
  } StackItem;

  StackItem *Tete;

public:        // Les fonctions de la pile :
  Stack(void);
  Stack(const Stack<T> &);
            // La classe est référencée en indiquant
            // son type entre crochets ("Stack<T>").
```

```
            // Ici, ce n'est pas une nécessité
            // cependant.
  ~Stack(void);
  Stack<T> &operator=(const Stack<T> &);
  void push(T);
  T pop(void);
  bool is_empty(void) const;
  void flush(void);
};

// Pour les fonctions membres définies en dehors de la déclaration
// de la classe, il faut une déclaration de type générique :

template <class T>
Stack<T>::Stack(void) // La classe est référencée en indiquant
                      // son type entre crochets ("Stack<T>").
                      // C'est impératif en dehors de la
                      // déclaration de la classe.
{
  Tete = NULL;
  return;
}

template <class T>
Stack<T>::Stack(const Stack<T> &Init)
{
  Tete = NULL;
  StackItem *tmp1 = Init.Tete, *tmp2 = NULL;
  while (tmp1!=NULL)
  {
    if (tmp2==NULL)
    {
      Tete= new StackItem;
      tmp2 = Tete;
    }
    else
    {
      tmp2->Next = new StackItem;
      tmp2 = tmp2->Next;
    }
    tmp2->Item = tmp1->Item;
    tmp1 = tmp1->Next;
  }
  if (tmp2!=NULL) tmp2->Next = NULL;
  return;
}

template <class T>
Stack<T>::~Stack(void)
{
  flush();
  return;
}

template <class T>
Stack<T> &Stack<T>::operator=(const Stack<T> &Init)
{
```

```
  flush();
  StackItem *tmp1 = Init.Tete, *tmp2 = NULL;

  while (tmp1!=NULL)
  {
    if (tmp2==NULL)
    {
      Tete = new StackItem;
      tmp2 = Tete;
    }
    else
    {
      tmp2->Next = new StackItem;
      tmp2 = tmp2->Next;
    }
    tmp2->Item = tmp1->Item;
    tmp1 = tmp1->Next;
  }
  if (tmp2!=NULL) tmp2->Next = NULL;
  return *this;
}

template <class T>
void Stack<T>::push(T Item)
{
  StackItem *tmp = new StackItem;
  tmp->Item = Item;
  tmp->Next = Tete;
  Tete = tmp;
  return;
}

template <class T>
T Stack<T>::pop(void)
{
  T tmp;
  StackItem *ptmp = Tete;

  if (Tete!=NULL)
  {
    tmp = Tete->Item;
    Tete = Tete->Next;
    delete ptmp;
  }
  return tmp;
}

template <class T>
bool Stack<T>::is_empty(void) const
{
  return (Tete==NULL);
}

template <class T>
void Stack<T>::flush(void)
{
  while (Tete!=NULL) pop();
  return;
}
```

Les classes template peuvent parfaitement avoir des fonctions amies, que ces fonctions soient elles-mêmes template ou non.

12.3.3. Fonctions membres template

Les destructeurs mis à part, les méthodes d'une classe peuvent être template, que la classe elle-même soit template ou non, pourvu que la classe ne soit pas une classe locale.

Les fonctions membres template peuvent appartenir à une classe template ou à une classe normale.

Lorsque la classe à laquelle elles appartiennent n'est pas template, leur syntaxe est exactement la même que pour les fonctions template non membre.

Exemple 12-6. Fonction membre template

```
class A
{
  int i;   // Valeur de la classe.
public:
  template <class T>
  void add(T valeur);
};

template <class T>
void A::add(T valeur)
{
  i=i+((int) valeur);   // Ajoute valeur à A::i.
  return ;
}
```

Si, en revanche, la classe dont la fonction membre fait partie est elle aussi template, il faut spécifier deux fois la syntaxe template : une fois pour la classe, et une fois pour la fonction. Si la fonction membre template est définie à l'intérieur de la classe, il n'est pas nécessaire de donner les paramètres template de la classe, et la définition de la fonction membre template se fait donc exactement comme celle d'une fonction template classique.

Exemple 12-7. Fonction membre template d'une classe template

```
template<class T>
class string
{
public:
  // Fonction membre template définie
  // à l'extérieur de la classe template ::

  template<class T2> int compare(const T2 &);

  // Fonction membre template définie
  // à l'intérieur de la classe template :

  template<class T2>
  string(const string<T2> &s)
  {
    // …
  }
};

// À l'extérieur de la classe template, on doit donner
// les déclarations template pour la classe
// et pour la fonction membre template :
```

```
template<class T> template<class T2>
int string<T>::compare(const T2 &s)
{
  // ...
}
```

Les fonctions membres virtuelles ne peuvent pas être template. Si une fonction membre template a le même nom qu'une fonction membre virtuelle d'une classe de base, elle ne surcharge pas cette fonction. Par conséquent, les mécanismes de virtualité sont inutilisables avec les fonctions membres template. On peut contourner ce problème de la manière suivante : on définira une fonction membre virtuelle non template qui appellera la fonction membre template.

Exemple 12-8. Fonction membre template et fonction membre virtuelle

```
class B
{
  virtual void f(int);
};

class D : public B
{
  template <class T>
  void f(T);     // Cette fonction ne surcharge pas B::f(int).

  void f(int i)  // Cette fonction surcharge B::f(int).
  {
    f<>(i);     // Elle appelle de la fonction template.
    return ;
  }
};
```

Dans l'exemple précédent, on est obligé de préciser que la fonction à appeler dans la fonction virtuelle est la fonction template, et qu'il ne s'agit donc pas d'un appel récursif de la fonction virtuelle. Pour cela, on fait suivre le nom de la fonction template d'une paire de signes inférieur et supérieur.

Plus généralement, si une fonction membre template d'une classe peut être spécialisée en une fonction qui a la même signature qu'une autre fonction membre de la même classe, et que ces deux fonctions ont le même nom, toute référence à ce nom utilisera la fonction non template. Il est possible de passer outre cette règle, à condition de donner explicitement la liste des paramètres template entre les signes inférieurs et supérieurs lors de l'appel de la fonction.

Exemple 12-9. Surcharge de fonction membre par une fonction membre template

```
struct A
{
  void f(int);

  template <class T>
  void f(T);
};

// Fonction non template :
void A::f(int)
{
}

// Fonction template :
template <>
void A::f<int>(int)
{
}

int main(void)
{
  A a;
  a.f(1);    // Appel de la version non-template de f.
```

```
  a.f('c');  // Appel de la version template de f.
  a.f<>(1);  // Appel de la version template spécialisée de f.
}
```

Pour plus de détails sur la spécialisation des template, voir la Section 12.5.

12.4. Instanciation des template

La définition des fonctions et des classes template ne génère aucun code tant que tous les paramètres template n'ont pas pris chacun une valeur spécifique. Il faut donc, lors de l'utilisation d'une fonction ou d'une classe template, fournir les valeurs pour tous les paramètres qui n'ont pas de valeur par défaut. Lorsque suffisamment de valeurs sont données, le code est généré pour ce jeu de valeurs. On appelle cette opération l'*instanciation des template*.

Plusieurs possibilités sont offertes pour parvenir à ce résultat : l'*instanciation implicite* et l'*instanciation explicite*.

12.4.1. Instanciation implicite

L'*instanciation implicite* est utilisée par le compilateur lorsqu'il rencontre une expression qui utilise pour la première fois une fonction ou une classe template, et qu'il doit l'instancier pour continuer son travail. Le compilateur se base alors sur le contexte courant pour déterminer les types des paramètres template à utiliser. Si aucune ambiguïté n'a lieu, il génère le code pour ce jeu de paramètres.

La détermination des types des paramètres template peut se faire simplement, ou être déduite de l'expression à compiler. Par exemple, les fonctions membres template sont instanciées en fonction du type de leurs paramètres. Si l'on reprend l'exemple de la fonction template Min définie dans l'Exemple 12-4, c'est son utilisation directe qui provoque une instanciation implicite.

Exemple 12-10. Instanciation implicite de fonction template

```
int i=Min(2,3);
```

Dans cet exemple, la fonction Min est appelée avec les paramètres 2 et 3. Comme ces entiers sont tous les deux de type int, la fonction template Min est instanciée pour le type int. Partout dans la définition de Min, le type générique T est donc remplacé par le type int.

Si l'on appelle une fonction template avec un jeu de paramètres qui provoque une ambiguïté, le compilateur signale une erreur. Cette erreur peut être levée en surchargeant la fonction template par une fonction qui accepte les mêmes paramètres. Par example, la fonction template Min ne peut pas être instanciée dans le code suivant :

```
int i=Min(2,3.0);
```

parce que le compilateur ne peut pas déterminer si le type générique T doit prendre la valeur int ou double. Il y a donc une erreur, sauf si une fonction Min(int, double) est définie quelque part. Pour résoudre ce type de problème, on devra spécifier manuellement les paramètres template de la fonction, lors de l'appel. Ainsi, la ligne précédente compile si on la réécrit comme suit :

```
int i=Min<int>(2,3.0);
```

dans cet exemple, le paramètre template est forcé à int, et 3.0 est converti en entier.

On prendra garde au fait que le compilateur utilise une politique minimaliste pour l'instanciation implicite des template. Ceci signifie qu'il ne créera que le code nécessaire pour compiler l'expression qui exige une instanciation implicite. Par exemple, la définition d'un objet d'une classe template dont tous les types définis provoque l'instanciation de cette classe, mais la définition d'un pointeur sur cette classe ne le fait pas. L'instanciation aura lieu lorsqu'un déréférencement sera fait par l'intermédiaire de ce pointeur. De même, seules

les fonctionnalités utilisées de la classe template seront effectivement définies dans le programme final.

Par exemple, dans le programme suivant :

```
#include <iostream>

using namespace std;

template <class T>
class A
{
public:
   void f(void);
   void g(void);
};

// Définition de la méthode A<char>::f() :
template <class T>
void A<T>::f(void)
{
   cout << "A<T>::f() appelée" << endl;
}

// On ne définit pas la méthode A<char>::g()...

int main(void)
{
   A<char> a;  // Instanciation de A<char>.
   a.f();           // Instanciation de A<char>::f().
   return 0;
}
```

seule la méthode f de la classe template A est instanciée, car c'est la seule méthode utilisée à cet endroit. Ce programme pourra donc parfaitement être compilé, même si la méthode g n'a pas été définie.

12.4.2. Instanciation explicite

L'instanciation explicite des template est une technique permettant au programmeur de forcer l'instanciation des template dans son programme. Pour réaliser une instanciation explicite, il faut spécifier explicitement tous les paramètres template à utiliser. Ceci ce fait simplement en donnant la déclaration du template, précédée par le mot-clé template :

```
template nom<valeur[, valeur[…]]>
```

Par exemple, pour forcer l'instanciation d'une pile telle que celle définie dans l'Exemple 12-5, il faudra préciser le type des éléments entre crochets après le nom de la classe :

```
template Stack<int>;  // Instancie la classe Stack<int>.
```

Cette syntaxe peut être simplifiée pour les fonctions template, à condition que tous les paramètres template puissent être déduits par le compilateur des types des paramètres utilisés dans la déclaration de la fonction. Ainsi, il est possible de forcer l'instanciation de la fonction template Min de la manière suivante :

```
template int Min(int, int);
```

Dans cet exemple, la fonction template Min est instanciée pour le type int, puisque ses paramètres sont de ce type.

Lorsqu'une fonction ou une classe template a des valeurs par défaut pour ses paramètres template, il n'est pas nécessaire de donner une valeur pour ces paramètres. Si toutes les valeurs par défaut sont utilisées, la liste des valeurs est vide (mais les signes d'infériorité et de supériorité doivent malgré tout être présents).

Exemple 12-11. Instanciation explicite de classe template

```
template<class T = char>
class Chaine;

template Chaine<>;  // Instanciation explicite de Chaine<char>.
```

12.4.3. Problèmes soulevés par l'instanciation des template

Les template doivent impérativement être définis lors de leur instanciation, pour que le compilateur puisse générer le code de l'instance. Ceci signifie que les fichiers d'en-tête doivent contenir non seulement la déclaration, mais également la définition complète des template. Ceci a plusieurs inconvénients. Le premier est bien entendu que l'on ne peut pas considérer les template comme les fonctions et les classes normales du langage, pour lesquels il est possible de séparer la définition de la déclaration dans des fichiers séparés. Le deuxième inconvénient est que les instances des template sont compilées plusieurs fois, ce qui diminue d'autant plus les performances des compilateurs. Enfin, ce qui est le plus grave, c'est que les instances des template sont en multiples exemplaires dans les fichiers objets générés par le compilateur, et accroissent donc la taille des fichiers exécutables à l'issue de l'édition de liens. Ceci n'est pas gênant pour les petits programmes, mais peut devenir rédhibitoire pour les programmes assez gros.

Le premier problème n'est pas trop gênant, car il réduit le nombre de fichiers sources, ce qui n'est en général pas une mauvaise chose. Notez également que les template ne peuvent pas être considérés comme les fichiers sources classiques, puisque sans instanciation, ils ne génèrent aucun code machine (ce sont des *classes de classes*, ou " *métaclasses* "). Mais ce problème peut devenir ennuyant dans le cas de librairies template écrites et vendues par des sociétés désireuses de conserver leur savoir faire. Pour résoudre ce problème, le langage donne la possibilité d'exporter les définitions des template dans des fichiers complémentaires. Nous verrons la manière de procéder dans la Section 12.7.

Le deuxième problème peut être résolu avec l'exportation des template, ou par tout autre technique d'optimisation des compilateurs. Actuellement, la plupart des compilateurs sont capables de générer des fichiers d'en-têtes *précompilés*, qui contiennent le résultat de l'analyse des fichiers d'en-tête déjà lus. Cette technique permet de diminuer considérablement les temps de compilation, mais nécessite souvent d'utiliser toujours le même fichier d'en-tête au début des fichiers sources.

Le troisième problème est en général résolu par des techniques variées, qui nécessitent des traitements complexes dans l'éditeur de liens ou le compilateur. La technique la plus simple, utilisée par la plupart des compilateurs actuels, passe par une modification de l'éditeur de liens, pour qu'il regroupe les différentes instances des même template. D'autres compilateurs, plus rares, gèrent une base de données dans laquelle les instances de template générées lors de la compilation sont stockées. Lors de l'édition de liens, les instances de cette base sont ajoutées à la ligne de commande de l'éditeur de liens, afin de résoudre les symboles non définis. Enfin, certains compilateurs permettent de désactiver les instanciations implicites des template. Ceci permet de laisser au programmeur la responsabilité de les instancier manuellement, à l'aide d'instanciations explicites. Ainsi, les template peuvent n'être définies que dans un seul fichier source, réservé à cet effet. Cette dernière solution est de loin la plus sûre, et il est donc recommandé d'écrire un tel fichier pour chaque programme.

Ce paragraphe vous a présenté trois des principaux problèmes soulevés par l'utilisation des template, ainsi que les solutions les plus courantes qui y ont été apportées. Il est vivement recommandé de consulter la documentation fournie avec l'environnement de développement utilisé, afin à la fois de réduire les temps de compilation et d'optimiser les exécutables générés.

12.5. Spécialisation des template

Jusqu'à présent, nous avons défini les classes et les fonctions template d'une manière unique, pour tous les types et toutes les valeurs des paramètres template. Cependant, il peut être intéressant de définir une version particulière d'une classe ou d'une fonction pour un jeu particulier de paramètres template.

Par exemple, la pile de l'Exemple 12-5 peut être implémentée beaucoup plus efficacement si elle stocke des pointeurs plutôt que des objets, sauf si les objets sont petits (ou appartiennent à un des types prédéfinis du langage). Il peut être intéressant de manipuler les pointeurs de manière transparente au niveau de la pile, pour que la méthode pop renvoie toujours un objet, que la pile stocke des pointeurs ou des objets. Afin de réaliser ceci, il faut donner une deuxième version de la pile pour les pointeurs.

Le C++ permet tout ceci : lorsqu'une fonction ou une classe template a été définie, il est possible de la *spécialiser* pour un certain jeu de paramètres template. Il existe deux types de spécialisation : les *spécialisations totales*, qui sont les spécialisations pour lesquelles il n'y a plus aucun paramètre template (ils ont tous une valeur bien déterminée), et les *spécialisations partielles*, pour lesquelles seuls quelques paramètres template ont une valeur fixée.

12.5.1. Spécialisation totale

Les *spécialisations totales* nécessitent de fournir les valeurs des paramètres template séparées par des virgules et entre les signes d'infériorité et de supériorité, après le nom de la fonction ou de la classe template. Il faut faire précéder la définition de cette fonction ou de cette classe par la ligne suivante :

```
template <>
```

qui permet de signaler que la liste des paramètres template pour cette spécialisation est vide (et donc que la spécialisation est totale).

Par exemple, si la fonction Min définie dans l'Exemple 12-4 doit être utilisée sur une structure Structure et se baser sur un des champs de cette structure pour effectuer les comparaisons, elle pourra être spécialisée de la manière suivante :

Exemple 12-12. Spécialisation totale

```
struct Structure
{
  int Clef;     // Clef permettant de retrouver des données.
  void *pData;  // Pointeur sur les données.
};

template <>
Structure Min<Structure>(Structure s1, Structure s2)
{
  if (s1.Clef>s2.Clef)
    return s1;
  else
    return s2;
}
```

> **Note:** Pour quelques compilateurs, la ligne déclarant la liste vide des paramètres template ne doit pas être écrite. On doit donc faire des spécialisations totale sans le mot-clé template. Ce comportement n'est pas celui spécifié par la norme, et le code écrit pour ces compilateurs n'est donc pas portable.

12.5.2. Spécialisation partielle

Les *spécialisations partielles* permettent de définir l'implémentation d'une fonction ou d'une classe template pour certaines valeurs de leurs paramètres template et de garder d'autres paramètres indéfinis. Il est même possible de changer la nature d'un paramètre template (c'est à dire préciser s'il s'agit d'un pointeur ou non) et de forcer le compilateur à prendre une implémentation plutôt qu'une autre selon que la valeur utilisée pour ce paramètre est elle-même un pointeur ou non.

Comme pour les spécialisations totales, il est nécessaire de déclarer la liste des paramètres template utilisés par la spécialisation. Cependant, à la différence des spécialisations totales, cette liste ne peut plus être vide.

Comme pour les spécialisations totales, la définition de la classe ou de la fonction template doit utiliser les signes d'infériorité et de supériorité pour donner la liste des valeurs des paramètres template pour la spécialisation.

Exemple 12-13. Spécialisation partielle

```
// Définition d'une classe template :
template <class T1, class T2, int I>
class A
{
};

// Spécialisation n°1 de la classe :
template <class T, int I>
class A<T, T*, I>
{
};

// Spécialisation n°2 de la classe :
template <class T1, class T2, int I>
class A<T1*, T2, I>
{
};

// Spécialisation n°3 de la classe :
template <class T>
class A<int, T*, 5>
{
};

// Spécialisation n°4 de la classe :
template <class T1, class T2, int I>
class A<T1, T2*, I>
{
};
```

On notera que le nombre des paramètres template déclarés à la suite du mot-clé template peut varier, mais que le nombre de valeurs fournies pour la spécialisation est toujours constant (dans l'exemple précédent, il y en a trois).

Les valeurs utilisées dans les identificateurs template des spécialisations doivent respecter les règles suivantes :

- une valeur ne peut pas être exprimée en fonction d'un paramètre template de la spécialisation ;

```
template <int I, int J>
struct B
{
};

template <int I>
struct B<I, I*2>    // Erreur !
{                   // Spécialisation incorrecte !
};
```

- le type d'une des valeurs de la spécialisation ne peut pas dépendre d'un autre paramètre ;

```
template <class T, T t>
struct C
{
};
```

```
template <class T>
struct C<T, 1>;     // Erreur !
                    // Spécialisation incorrecte !
```

- la liste des arguments de la spécialisation ne doit pas être identique à la liste implicite de la déclaration template correspondante.

Enfin, la liste des paramètres template de la déclaration d'une spécialisation ne doit pas contenir des valeurs par défaut. On ne pourrait d'ailleurs les utiliser en aucune manière.

12.5.3. Spécialisation d'une méthode d'une classe template

La spécialisation partielle d'une classe peut parfois être assez lourde à employer, en particulier si la structure de données qu'elle contient ne change pas entre les versions spécialisées. Dans ce cas, il peut être plus simple de ne spécialiser que certaines méthodes de la classe et non la classe complète. Ceci permet de conserver la définition des méthodes qui n'ont pas lieu d'être modifiées pour les différents type, et d'éviter d'avoir à redéfinir les données membres de la classe à l'identique.

La syntaxe permettant de spécialiser une méthode d'une classe template est très simple. Il suffit en effet de considérer la méthode comme une fonction template normale, et de la spécialiser en précisant les paramètres template à utiliser pour cette spécialisation.

Exemple 12-14. Spécialisation de fonction membre de classe template

```
#include <iostream>

using namespace std;

template <class T>
class Item
{
  T item;
public:
  Item(T);
  void set(T);
  T get(void) const;
  void print(void) const;
};

template <class T>
Item<T>::Item(T i)              // Constructeur
{
  item = i;
}

// Accesseurs :

template <class T>
void Item<T>::set(T i)
{
  item = i;
}

template <class T>
T Item<T>::get(void) const
{
  return item;
}

// Fonction d'affichage générique :

template <class T>
void Item<T>::print(void) const
```

```
{
  cout << item << endl;
}

// Fonction d'affichage spécialisée explicitement pour le type int *
// et la méthode print :
template <>
void Item<int *>::print(void) const
{
  cout << *item << endl;
}
```

12.6. Mot-clé typename

Nous avons déjà vu que le mot-clé typename pouvait être utilisé pour introduire les types génériques dans les déclarations template. Cependant, il peut être utilisé dans un autre contexte pour introduire les identificateurs de types inconnus dans les template. En effet, un type générique peut très bien être une classe définie par l'utilisateur, à l'intérieur de laquelle des types sont définis. Afin de pouvoir utiliser ces types dans les définitions des template, il est nécessaire d'utiliser le mot-clé typename pour les introduire, car a priori le compilateur ne sait pas que le type générique contient la définition d'un autre type. Ce mot-clé doit être placé avant le nom complet du type :

```
typename identificateur
```

Le mot-clé typename est donc utilisé pour signaler au compilateur qu'un identificateur identificateur inconnu est un type.

Exemple 12-15. Mot-clé typename

```
class A
{
public:
  typedef int Y;    // Y est un type défini dans la classe A.
};

template <class T>
class X
{
  typename T::Y i;  // La classe template X suppose que le
                    // type générique T définisse un type Y.
};

X<A> x;             // A peut servir à instancier une classe
                    // à partir de la classe template X.
```

12.7. Fonctions exportées

Comme on l'a vu, les fonctions et classes template sont toutes instanciées lorsqu'elles sont rencontrées pour la première fois par le compilateur ou lorsque la liste de leurs paramètres est fournie explicitement.

Cette règle a une conséquence majeure : la définition complète des fonctions et des classes template doit être incluse dans chacun des fichiers dans lequel elles sont utilisées. En général, les déclarations et les définitions des fonctions et des classes template sont donc regroupées ensemble dans les fichiers d'en-tête (et le code ne se trouve pas dans un fichier C++). Ceci est à la fois très lent (la définition doit être relue par le compilateur à

chaque fois qu'un template est utilisé) et ne permet pas de protéger le savoir faire des entreprises qui éditent des librairies template, puisque leur code est accessible à tout le monde.

Afin de résoudre ces problèmes, le C++ permet de " compiler " les fonctions et les classes template, et ainsi d'éviter l'inclusion systématique de leur définition dans les fichiers sources. Cette " compilation " se fait à l'aide du mot-clé export.

Pour parvenir à ce résultat, vous devez déclarer " export " les fonctions et les classes template concernées. La déclaration d'une classe template export revient à déclarer export toutes ses fonctions membres non inline, toutes ses données statiques, toutes ses classes membres et toutes ses fonctions membres template non statiques. Si une fonction template est déclarée comme étant inline, elle ne peut pas être de type export.

Les fonctions et les classes template qui sont définies dans un espace de nommage anonyme ne peuvent pas être déclarées export. Voir le Chapitre 11 plus de détails sur les espaces de nommage.

Exemple 12-16. Mot-clé export

```
export template <class T>
void f(T);          // Fonction dont le code n'est pas fourni
                    // dans les fichiers qui l'utilisent.
```

Dans cet exemple, la fonction f est déclarée export. Sa définition est fournie dans un autre fichier, et n'a pas besoin d'être fournie pour que f soit utilisable.

Les définitions des fonctions et des classes déclarées export doivent elles aussi utiliser le mot-clé export. Ainsi, la définition de f pourra ressembler aux lignes suivantes :

```
export template <class T>
void f(T p)
{
  // Corps de la fonction.
  return ;
}
```

Note: Aucun compilateur ne gère le mot-clé export à ce jour.

Chapitre 13. La librairie standard C++

> **Note:** Ce chapitre en est à une version préliminaire. Les informations données sont incomplètes et peuvent ne pas avoir été vérifiées. Tout commentaire ou toute demande de correction sont les bienvenus.

Tout comme pour le langage C, pour lequel un certain nombre de fonctions ont été définies et standardisées par la librairie C, une librairie de classes et de fonctions ont été définies pour le langage C++. Cette librairie est le résultat de l'évolution de plusieurs librairies, parfois développées indépendamment par plusieurs fournisseurs d'environnement C++, qui ont été fusionnées et normalisées afin de garantir la portabilité des programmes qui les utilisent. Une des principales briques de cette librairie est sans aucun doute la STL (abréviation de " Standard Template Library "), à tel point qu'il y a souvent confusion entre les deux.

La librairie standard C++ reprend la plupart des fonctionnalités déjà disponibles avec la librairie C, et les complète avec les fonctionnalités dont tout programmeur a ou aura un jour besoin. Elle couvre donc les entrées / sorties à l'aide de la notion de flux, la définition des limites d'une implémentation, la classification des caractères et la localisation. Mais les fonctionnalités les plus intéressantes sont sans doute la gestion de tous les conteneurs d'objets et les algorithmes permettant de résoudre les problèmes les plus classiques, ainsi que la définition des types complémentaires qui font cruellement défaut au langage.

Ce chapitre a pour but de présenter les principales fonctionnalités de la librairie standard C++. Bien entendu, il est hors de question de décrire complètement chaque fonction ou chaque détail du fonctionnement de la librairie standard, car cela rendrait illisibles et incompréhensibles les explications. Cependant, les informations de base vous seront données, afin de vous permettre d'utiliser efficacement la librairie standard C++ et de comprendre les fonctionnalités les plus avancées lorsque vous vous y intéresserez.

La librairie standard C++ est réellement un sujet de taille. À titre indicatif, sa description est aussi volumineuse que celle du langage lui-même dans la norme C++. Mais ce n'est pas tout, il faut impérativement avoir compris en profondeur les fonctionnalités les plus avancées du C++ pour appréhender correctement la librairie standard. En particulier, tous les algorithmes et toutes les classes fournies par la librairie sont susceptibles de travailler sur des données de type arbitraire. La librairie utilise donc pour cela complètement la notion de template, et se base sur plusieurs abstractions des données manipulées et de leurs types, afin de rendre générique l'implémentation des fonctionnalités. De plus, la librairie utilise les mécanismes d'exceptions afin de signaler les erreurs qui peuvent se produire lors de l'exécution des méthodes de ses classes et de ses fonctions. Enfin, un certain nombre de notions algorithmiques avancées sont utilisées dans toute la librairie. La présentation qui sera faite sera donc progressive, tout en essayant de conserver un ordre logique. Tout comme pour les chapitres précédents, il est probable que plusieurs lectures soient nécessaires au débutant pour assimiler toutes les subtilités de la librairie.

> **Note:** Notez bien que les informations décrites ici sont basées sur la norme ISO 14882 du langage C++, et non sur la réalité des compilateurs actuels. Il est donc fortement probable que bon nombre d'exemples fournis ici ne seront pas utilisables tels quels sur les environnement de développement existants sur le marché, bien que l'on commence à voir apparaître des compilateurs implémentant quasiment la totalité de la norme à présent. De légères différences dans l'interface des classes décrites peuvent également apparaître et nécessiter la modification de ces exemples. Cependant, à terme, tous les environnements de développement respecteront les interfaces spécifiées par la norme, et les programmes utilisant la librairie standard seront réellement portables au niveau source.

13.1. Services et notions de base de la librairie standard

La librairie standard C++ fournit un certain nombre de fonctionnalités de base, sur lesquelles toutes les autres fonctionnalités de la librairie s'appuient. Ces fonctionnalités apparaissent comme des classes d'encapsulation de la

librairie C, et des classes d'abstraction des principales constructions du langages. Ces dernières utilisent des notions très évoluées pour permettre une encapsulation réellement générique de types de base. Bien que complexes, ces notions sont omniprésentes dans toute la librairie, aussi est-il extrêmement important de les comprendre en détail.

13.1.1. Encapsulation de la librairie C standard

La librairie C définit un grand nombre de fonctions C standard, que la librairie standard C++ reprend à son compte et complète par toutes ses fonctionnalités avancées. Pour bénéficier de ces fonctions, il suffit simplement d'inclure les fichiers d'en-têtes de la librairie C, tout comme on le faisait avec les programmes C classiques.

Toutefois, les fonctions ainsi déclarées par ces en-têtes apparaissent dans l'espace de nommage global, ce qui risque de provoquer des conflits de noms avec des fonctions homonymes (rappelons que les fonctions C ne sont pas surchargeables). Par conséquent, et dans un soucis d'homogénéité avec le reste des fonctionnalités de la librairie C++, un jeu d'en-têtes complémentaires a été défini pour les fonctions de la librairie C. Ces en-têtes définissent tous leurs symboles dans l'espace de nommage `std::`, qui est réservé pour la librairie standard C++.

Ces en-têtes se distinguent des fichiers d'en-tête de la librairie C par le fait qu'ils ne portent pas d'extension `.h` et par le fait que leur nom est préfixé par la lettre 'c'. Les en-têtes utilisables ainsi sont donc les suivants :

cassert
cctype
cerrno
cfloat
ciso646
climits
clocale
cmath
csetjmp
csignal
cstdarg
cstddef
cstdio
cstdlib
cstring
ctime
cwchar
cwctype

Par exemple, on peut réécrire notre tout premier programme que l'on a fait à la Section 2.9 de la manière suivante :

```
#include <cstdio>

long double x, y;

int main(void)
{
   std::printf("Calcul de moyenne\n");
   std::printf("Entrez le premier nombre : ");
   std::scanf("%Lf", &x);
   std::printf("\nEntrez le deuxième nombre : ");
   std::scanf("%Lf", &y);
   std::printf("\nLa valeur moyenne de %Lf et de %Lf est %Lf.\n",
      x, y, (x+y)/2);
   return 0;
}
```

Note: L'utilisation systématique du préfixe `std::` peut être énervante sur les grands programmes. On aura donc intérêt soit à utiliser les fichiers d'en-têtes classiques de la librairie C, soit à inclure une directive `using namespace std;` pour intégrer les fonctionnalités de la librairie

standard dans l'espace de nommage global.

Remarquez que la norme ne suppose pas que ces en-têtes soient des fichiers physiques. Les déclarations qu'ils sont supposés faire peuvent donc être réalisées à la volée par les outils de développement, et vous ne les trouverez pas forcément sur votre disque dur.

Certaines fonctionnalités fournies par la librairie C ont été encapsulées dans des fonctionnalités équivalentes de la librairie standard C++. C'est notamment le cas pour la gestion des locales et la gestion de certains types de données complexes. C'est également le cas pour la détermination des limites de représentation que les types de base peuvent avoir. Classiquement, ces limites sont définies par des macros dans les en-têtes de la librairie C, mais elles sont également accessibles au travers de la classe template numeric_limits, définie dans l'en-tête limits :

```
// Types d'arrondis pour les flottants :
enum float_round_style
{
   round_indeterminate       = -1,
   round_toward_zero         =  0,
   round_to_nearest          =  1,
   round_toward_infinity     = 2,
   round_toward_neg_infinity = 3
};

template <class T>
class numeric_limits
{
public:
   static const bool is_specialized = false;
   static T min() throw();
   static T max() throw();
   static const int digits   = 0;
   static const int digits10 = 0;
   static const bool is_signed  = false;
   static const bool is_integer = false;
   static const bool is_exact   = false;
   static const int radix = 0;
   static T epsilon() throw();
   static T round_error() throw();
   static const int min_exponent   = 0;
   static const int min_exponent10 = 0;
   static const int max_exponent   = 0;
   static const int max_exponent10 = 0;
   static const bool has_infinity  = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const bool has_denorm        = false;
   static const bool has_denorm_loss   = false;
   static T infinity() throw();
   static T quiet_NaN() throw();
   static T signaling_NaN() throw();
   static T denorm_min() throw();
   static const bool is_iec559  = false;
   static const bool is_bounded = false;
   static const bool is_modulo  = false;
   static const bool traps      = false;
   static const bool tinyness_before = false;
   static const float_round_style
```

```
        round_style = round_toward_zero;
};
```

Cette classe template ne sert à rien en soi. En fait, elle est spécialisée pour tous les types de base du langage, et ce sont ces spécialisations qui sont réellement utilisées. Elles permettent d'obtenir toutes les informations pour chaque type grâce à leurs données membres et à leurs méthodes statiques.

Exemple 13-1. Détermination des limites d'un type

```
#include <iostream>
#include <limits>

using namespace std;

int main(void)
{
        cout << numeric_limits<int>::min() << endl;
        cout << numeric_limits<int>::max() << endl;
        cout << numeric_limits<int>::digits << endl;
        cout << numeric_limits<int>::digits10 << endl;
        return 0;
}
```

Ce programme d'exemple détermine le plus petit et le plus grand nombre représentable avec le type entier int, ainsi que le nombre de bits utilisés pour coder les chiffres et le nombre de chiffres maximum que les nombres en base 10 peuvent avoir en étant sûr de pouvoir être stocké tel quel.

Les fonctions de manipulation des types de données complexes seront présentés plus en détail dans la Section 13.2.

13.1.2. Définition des exceptions standard

La librairie standard utilise le mécanisme des exceptions du langage pour signaler les erreurs qui peuvent se produire à l'exécution au sein de ses fonctionnalités. Pour cela, elle définit un certain nombre de classes d'exceptions standard, que toutes les fonctionnalités de la librairie sont susceptibles d'utiliser. Ces classes peuvent être utilisées telles quelles, ou servir de classes de base à des classes d'exceptions personnalisées pour vos propres développements.

Ces classes d'exception sont presque toutes déclarées dans l'en-tête stdexcept, et dérivent de la classe de base exception. Cette dernière n'est pas déclarée dans le même en-tête et n'est pas utilisée directement, mais fournit les mécanismes de base de toutes les exceptions de la librairie standard. Elle est déclarée comme suit dans l'en-tête exception :

```
class exception
{
public:
   exception() throw();
   exception(const exception &) throw();
   exception &operator=(const exception &) throw();
   virtual ~exception() throw();
   virtual const char *what() const throw();
};
```

Outre les constructeurs, opérateurs d'affectation et destructeurs classiques, cette classe définit une méthode what, qui retourne une chaîne de caractères statique. Le contenu de cette chaîne de caractères n'est pas normalisé, cependant, il sert généralement à décrire la nature de l'erreur qui s'est produite. C'est une méthode virtuelle, car elle est bien entendu destinée à être redéfinie par les classes d'exception spécialisées pour les différents types d'erreurs. Notez que toutes les méthodes de la classe exception sont déclarées comme ne pouvant pas lancer d'exceptions elle-mêmes, ce qui est naturel lorsqu'on est déjà en train de traiter une exception lorsqu'on manipule des objets de cette classe.

L'en-tête exception contient également la déclaration de la classe d'exception bad_exception. Cette classe n'est, elle aussi, pas utilisée en temps normal. Le seul cas où elle peut être lancée est dans le traitement de la fonction de traitement d'erreur appelée par la fonction std::unexpected lorsqu'une exception a provoqué la sortie d'une

fonction qui n'avait pas le droit de la lancer. La classe bad_exception est déclarée comme suit dans l'en-tête exception :

```
class bad_exception : public exception
{
public:
   bad_exception() throw();
   bad_exception(const bad_exception &) throw();
   bad_exception &operator=(const bad_exception &) throw();
   virtual ~bad_exception() throw();
   virtual const char *what() const throw();
};
```

Notez que l'exception bad_alloc, lancée par les gestionnaires de manque de mémoire lorsque l'opérateur new ou l'opérateur new[] n'ont pas réussi à faire une allocation, n'est pas déclarée dans l'en-tête stdexcept non plus. Sa déclaration a été placée avec celle des opérateurs d'allocation mémoire, dans l'en-tête new. Cette classe dérive toutefois de la classe exception, comme le montre sa déclaration :

```
class bad_alloc : public exception
{
public:
   bad_alloc() throw();
   bad_alloc(const bad_alloc &) throw();
   bad_alloc &operator=(const bad_alloc &) throw();
   virtual ~bad_alloc() throw();
   virtual const char *what() const throw();
};
```

Les autres exceptions sont déclarées classées en deux grandes catégories. La première catégorie regroupe toutes les exceptions dont l'occurrence traduit sans doute une erreur de programmation dans le programme, car elles ne devraient jamais se produire à l'exécution. Il s'agit des exceptions dites " d'erreurs dans la logique du programme ", et en tant que telles, dérivent de la classe d'exception logic_error. Cette classe est déclarée comme suit dans l'en-tête stdexcept :

```
class logic_error : public exception
{
public:
   logic_error(const string &what_arg);
};
```

Elle ne contient qu'un constructeur, permettant de définir la chaîne de caractères qui sera renvoyée par la méthode virtuelle what. Ce constructeur prend en paramètre cette chaîne de caractères, sous la forme d'un objet de la classe string. Cette classe est définie par la librairie standard afin de faciliter la manipulation des chaînes de caractères, et sera décrite plus en détail dans la Section 13.2.1.

Les classes d'exception qui dérivent de la classe logic_error disposent également d'un constructeur similaire. Ces classes sont les suivantes :

- la classe domain_error, qui spécifie qu'une fonction a été appelée avec des paramètres sur lesquels elle n'est pas définie. Il faut contrôler les valeurs des paramètres utilisées lors de l'appel de la fonction qui a lancé cette exception ;
- la classe invalid_argument, qui spécifie qu'un des arguments d'une méthode ou d'une fonction n'est pas valide. Cette erreur arrive lorsque l'on utilise des valeurs de paramètres qui n'entrent pas dans le cadre de

fonctionnement normal de la méthode appelée, cela traduit souvent une mauvaise utilisation de la fonctionnalité correspondante ;

- la classe length_error, qui indique qu'un dépassement de capacité maximale d'un objet a été réalisé. Ces dépassements se produisent dans les programmes bogués, qui essaient d'utiliser une fonctionnalité au delà des limites qui avaient été prévues pour elle ;
- la classe out_of_range, qui spécifie qu'une valeur située en dehors de la plage de valeur autorisée a été utilisée. Ce type d'erreur signifie souvent que les paramètres utilisés pour un appel de fonction ne sont pas corrects ou pas initialisés, et qu'il faut vérifier leur validité.

La deuxième catégorie d'exception correspond aux erreurs qui ne peuvent pas toujours être corrigées lors de l'écriture du programme, et qui font donc partie des événements naturels qui se produisent lors de son exécution. Elles caractérisent les erreurs d'exécution, et dérivent de la classe d'exception runtime_error. Cette classe est déclarée de la manière suivante dans l'en-tête stdexcept :

```
class runtime_error : public exception
{
public:
   runtime_error(const string &what_arg);
};
```

Elle s'utilise exactement comme la classe logic_error.

Les exceptions de la catégorie des erreurs d'exécution sont les suivantes :

- la classe range_error, qui signifie qu'une valeur est sortie de la plage de valeur dans laquelle elle devait se trouver, suite à débordement interne à la librairie ;
- la classe overflow_error, qui signifie qu'un débordement par valeurs supérieures s'est produit dans un calcul interne à la librairie ;
- la classe underflow_error, qui signifie qu'un débordement par valeurs inférieures s'est produit dans un calcul interne à la librairie.

13.1.3. Abstraction des types de données : les traits

Un certain nombre de classes ou d'algorithmes peuvent manipuler des types ayant une signification particulière. Par exemple, la classe string, que nous verrons plus loin, manipule des objets de type caractère. En réalité, ces classes et ces algorithmes peuvent travailler avec n'importe quels types, pourvu que tous ces types se comportent de la même manière. La librairie standard C++ définit donc la notion de " *traits* ", qui ne sont rien d'autre que des caractéristiques de ces types, et qui sont définis dans une classe prévue à cet usage. Les classes et les algorithmes standard n'utilisent que cette classe de trait pour manipuler les objets, garantissant ainsi une abstraction totale vis à vis de leur type. Ainsi, il suffit de coder une spécialisation de la classe des traits pour un type particulier afin de permettre son utilisation dans les algorithmes génériques. La librairie standard définit bien entendu des spécialisations pour les types de base du langage.

Par exemple, la classe de définition des traits des types caractère est la classe template char_traits. Elle contient contient les définitions des types suivants :

- le type char_type, qui est le type représentant les caractères eux-mêmes ;
- le type int_type, qui est un type capable de contenir toutes les valeurs possibles pour les caractères, y compris la valeur spéciale du marqueur de fin de fichier ;
- le type off_type, qui est le type permettant de représenter les déplacements dans une séquence de caractères, ainsi que les positions absolues dans cette séquence. Ce type est signé, car les déplacements peuvent être réalisés aussi bien vers le début de la séquence que vers la fin ;
- le type pos_type, qui est un sous-type du type off_type, et qui n'est utilisé que pour les déplacements dans les fonctions de positionnement des flux de la librairie standard ;
- le type state_type, qui permet de représenter l'état courant d'une séquence de caractères dans les fonctions de conversion. Ce type est utilisé dans les fonctions de transcodage des séquences de caractères d'un encodage vers un autre.

Note: Pour comprendre l'utilité de ce dernier type, il faut savoir qu'il existe plusieurs manières de coder les caractères. La plupart des méthodes utilisent un encodage à *taille fixe*, où chaque caractère est représenté par une valeur entière et une seule. Cette technique est très pratique pour les jeux de caractères contenant moins de 256 caractères, pour lesquels un seul octet est utilisé par caractère. Elle est également utilisée pour les jeux de caractères de moins de 65536 caractères, car l'utilisation de 16 bits par caractères est encore raisonable. En revanche, les caractères des jeux de caractères orientaux sont codés avec des valeurs numériques supérieures à 65536 par les encodages standard (Unicode et ISO 10646), et ne peuvent donc pas être stockés dans les types char ou wchar_t. Pour ces jeux de caractères, on utilise donc souvent des encodages à *taille variable*, et chaque caractère peut être représenté par un ou plusieurs octets, selon sa nature et éventuellement selon sa position dans la chaîne de caractères.

Pour ces encodages à taille variable, il est évident que le positionnement dans les séquences de caractères se fait en fonction du contexte de la chaîne, à savoir en fonction de la position du caractère précédent et parfois en fonction des caractères déjà analysés. Les algorithmes de la librairie standard qui manipulent les séquences de caractères doivent donc stocker le contexte courant lors de l'analyse de ces séquences. Elles le font grâce au type state_type de la classe des traits de ces caractères.

L'exemple suivant vous permettra de vérifier que le type char_type de la classe de définition des traits pour le type char est bien entendu le type char lui-même :

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

using namespace std;

int main(void)
{
   // Récupère les informations de typage des traits :
   const type_info &ti_trait =
      typeid(char_traits<char>::char_type);
   // Récupère les informations de typage directement :
   const type_info &ti_char = typeid(char);
   // Compare les types :
   cout << "Le nom du type caractère des traits est : " <<
      ti_trait.name() << endl;
   cout << "Le nom du type char est : " <<
      ti_char.name() << endl;
   if (ti_trait == ti_char)
      cout << "Les deux types sont identiques." << endl;
   else
      cout << "Ce n'est pas le même type." << endl;
   return 0;
}
```

La classe char_traits définit également un certain nombre de méthodes travaillant sur les types de caractères et permettant de réaliser les opérations de base sur ces caractères. Ces méthodes permettent essentiellement de comparer, de copier, de déplacer et de rechercher des caractères dans des séquences de caractères, en tenant compte de toutes les caractéristiques de ces caractères. Elle contient également la définition de la valeur spéciale utilisée dans les séquences de caractères pour marquer les fin de flux (" EOF ", abréviation de l'anglais " End Of File ").

Par exemple, le programme suivant permet d'afficher la valeur utilisée pour spécifier une fin de fichier dans une séquence de caractères de type wchar_t :

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

using namespace std;

int main(void)
{
   char_traits<wchar_t>::int_type wchar_eof =
      char_traits<wchar_t>::eof();
   cout << "La valeur de fin de fichier pour wchar_t est : "
      << wchar_eof << endl;
   return 0;
}
```

Les autres méthodes de la classe de définition des traits des caractères, ainsi que les classes de définition des traits des autre types, ne seront pas décrites plus en détail ici. Elles sont essentiellement utilisées au sein des algorithmes de la librairie standard, et n'ont donc qu'un intérêt limité pour les programmeurs, mais il est important de savoir qu'elles existent.

13.1.4. Abstraction des pointeurs : les itérateurs

La librairie standard définit un certain nombre de structures de données évoluées, qui permettent de stocker et de manipuler les objets utilisateur de manière optimale, évitant ainsi au programmeur d'avoir à réinventer la roue. On appelle ces structures de données des *conteneurs*. Ces conteneurs peuvent être manipulés au travers de fonctions spéciales, selon un grand nombre d'algorithmes possible, dont la librairie dispose en standard. L'ensemble des fonctionnalités fournies par la librairie permet de subvenir au besoin des programmeurs dans la majorité des cas. Nous détaillerons la notion de conteneurs et les algorithmes disponibles plus loin dans ce chapitre.

La manière d'accéder aux données des conteneurs dépend bien entendu de leur nature et de leur structure. Ceci signifie qu'en théorie, il est nécessaire de spécialiser les fonctions permettant d'appliquer les algorithmes pour chaque type de conteneur existant. Cette technique n'est ni pratique, ni extensible, puisque les algorithmes fournis par la librairie ne pourraient dans ce cas pas travailler sur des conteneurs écrits par le programmeur. C'est pour cette raison que la librairie standard utilise une autre technique pour accéder aux données des conteneurs. Cette technique est basée sur la notion d'*itérateur*.

13.1.4.1. Notions de base et définition

Un itérateur n'est rien d'autre qu'un objet permettant d'accéder à tous les objets d'un conteneur donné, souvent séquentiellement, selon une interface standardisée. La dénomination d'itérateur provient donc du fait que les itérateurs permettent d'*itérer* sur les objets d'un conteneur, c'est à dire d'en parcourir le contenu en passant par tous ses objets.

Comme les itérateurs sont des objets permettant d'accéder à d'autres objets, ils ne représentent pas eux-mêmes ces objets, mais plutôt le moyen de les atteindre. Ils sont donc comparables aux pointeurs, dont ils reprennent exactement la même sémantique. En fait, les concepteurs de la librairie standard se sont basés sur cette propriété pour définir l'interface des itérateurs, qui sont donc une extension de la notion de pointeur. Par exemple, il est possible d'écrire des expressions telles que " *i " ou " i++ " avec un itérateur i. Tous les algorithmes de la librairie, qui travaillent normalement sur des itérateurs, sont donc susceptibles de fonctionner avec des pointeurs classiques.

Bien entendu, pour la plupart des conteneurs, les itérateurs ne sont pas de simples pointeurs, mais des objets qui se comportent comme des pointeurs, et qui sont spécifiques à chaque conteneur. Ainsi, les algorithmes sont écrits de manière uniforme, et ce sont les conteneurs qui fournissent les itérateurs qui leurs sont appropriés, afin de permettre l'accès à leurs données.

13.1.4.2. Classification des itérateurs

La librairie définit plusieurs catégories d'itérateurs, qui contiennent des itérateurs plus ou moins puissants. Le

comportement des d'itérateurs les plus puissants se rapproche beaucoup des pointeurs classiques, et quasiment toutes les opérations applicables aux pointeurs peuvent l'être à ces itérateurs. En revanche, les itérateurs des classes plus restrictives ne définissent qu'un sous-ensemble des opérations que les pointeurs supportent, et ne peuvent donc être utilisés que dans le cadre de ce jeu d'opérations réduit.

Les algorithmes de la librairie n'utilisent que les itérateurs des classes les plus faibles permettant de réaliser leur travail. Ils s'imposent ces restrictions afin de garantir leur utilisation correcte même avec les itérateurs les plus simples. Bien entendu, comme les pointeurs disposent de toutes les fonctionnalités définies par les itérateurs, même les plus puissants, les algorithmes standard fonctionnent également avec des pointeurs. Autrement dit, la librairie standard est écrite de façon à n'utilise qu'une partie des opérations applicables aux pointeurs, afin de garantir que ce qui fonctionne avec des itérateurs fonctionne avec des pointeurs.

Les itérateurs de chaque catégorie possèdent toutes les propriétés des itérateurs des catégories inférieures. Il existe donc une hiérarchie dans la classification des itérateurs. Les catégories définies par la librairie standard sont les suivantes :

- les itérateurs de la catégorie " *Output* " sont utilisés pour effectuer des affectations de valeurs aux données qu'ils référencent. Ces itérateurs ne peuvent donc être déréférencés par l'opérateur '*' que dans le cadre d'une affectation. Il est impossible de lire la valeur d'un itérateur de type Output, et on ne doit écrire dans la valeur qu'ils référencent qu'une fois au plus. Les algorithmes qui utilisent ces itérateurs doivent donc impérativement ne faire qu'une seule passe sur les données itérées ;
- les itérateurs de la catégorie " *Input* " sont similaires aux itérateurs de type Output, à ceci près qu'ils ne peuvent être déréférencés que pour lire une valeur. Contrairement aux itérateurs de type Output, il est possible de comparer deux itérateurs. Cependant, le fait que deux itérateurs soient égaux ne signifie aucunement que leurs successeurs le seront encore. Les algorithmes qui utilisent les itérateurs de type Input ne peuvent donc faire aucune hypothèse sur l'ordre de parcours utilisé par l'itérateur, et sont donc nécessairement des algorithmes en une passe ;
- les itérateurs de la catégorie " *Forward* " possèdent toutes les fonctionnalités des itérateurs de type Input et de type Output. Comme ceux-ci, ils ne peuvent passer que d'une valeur à la suivante, et jamais reculer ou revenir à une valeur déjà itérée. Les algorithmes qui utilisent des itérateurs de cette catégorie s'imposent donc de ne parcourir les données des conteneurs que dans un seul sens. Cependant, la restriction imposée sur l'égalité des opérateurs de type Input est levée, ce qui signifie que plusieurs parcours successifs se feront dans le même ordre. Les algorithmes peuvent effectuer plusieurs parcours, par exemple en copiant la valeur initiale de l'itérateur et de parcourir le conteneur plusieurs fois avec chaque copie effectuée ;
- les itérateurs de la catégorie " *Bidirectionnal* " disposent de toutes les fonctionnalités des itérateurs de type Forward, mais lèvent la restriction sur le sens de parcours. Ces itérateurs peuvent donc revenir sur les données déjà itérées, et les algorithmes qui les utilisent peuvent donc travailler en plusieurs passe, dans les deux directions ;
- enfin, les itérateurs de la catégorie " *RandomAccess* " sont les plus puissants. Ils fournissent toutes les fonctionnalités des itérateurs de type Bidirectionnal, plus la possibilité d'accéder aux éléments des conteneurs par l'intermédiaire d'un index en un temps constant. Il n'y a donc plus de notion de sens de parcours, et les données peuvent être accédées comme les données d'un tableau. Il est également possible d'effectuer les opérations classiques de l'arithmétique des pointeurs sur ces itérateurs.

Tous les itérateurs de la librairie standard dérivent de la classe de base suivante :

```
template <class Category,
   class T, class Distance = ptrdiff_t,
   class Pointer = T*, class Reference = T &>
struct iterator
{
   typedef T         value_type;
```

```
   typedef Distance  difference_type;
   typedef Pointer   pointer;
   typedef Reference reference;
   typedef Category  iterator_category;
};
```

Cette classe est déclarée dans l'en-tête iterator.

Cette classe définit les types de base des itérateurs, à savoir : le type des valeurs référencées, le type des différences entre deux itérateurs dans les calculs d'arithmétique des pointeurs, le type des pointeurs des valeurs référencées par l'itérateur, le type des références pour ces valeurs et la catégorie de l'itérateur. Ce dernier type doit être l'une des classes suivantes, également définies par la librairie standard :

- input_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Input ;
- output_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Output ;
- forward_iterator_tag, pour les itérateurs de la catégorie des itérateurs de type Forward ;
- bidirectionnal_iterator_tag, pour les itérateurs de la catégorie des itérateurs bidirectionnels ;
- random_access_iterator_tag, pour les itérateurs de la catégorie des itérateurs à accès complet.

Notez que pour les types par défaut pour la différence entre deux pointeurs est le type ptrdiff_t, qui est utilisé classiquement pour les pointeurs normaux. De même, le type pointeur et le type référence correspondent respectivement, par défaut, aux types T * et T &. Pour les itérateurs pour lesquels ces types n'ont pas de sens, le type utilisé est void, ce qui permet de provoquer une erreur de compilation si on cherche à les utiliser.

Ces types sont utilisés par les itérateurs nativement, cependant, ils ne le sont généralement pas par les algorithmes. En effet, ceux-ci sont susceptibles d'être appelées avec des pointeurs normaux, et les pointeurs ne définissent pas tous ces types. C'est pour cette raison qu'une classe de traits a été définie pour les itérateurs par la librairie standard. Cette classe est déclarée comme suit dans l'en-tête iterator :

```
template <class Iterator>
struct iterator_traits
{
   typedef Iterator::value_type        value_type;
   typedef Iterator::difference_type   difference_type;
   typedef Iterator::pointer           pointer;
   typedef Iterator::reference         reference;
   typedef Iterator::iterator_category iterator_category;
};
```

La classe des traits permet donc d'obtenir de manière indépendante de la nature de l'itérateur la valeur des types fondamentaux de l'itérateur. Comme ces types n'existent pas pour les pointeurs classiques, cette classe est spécialisée de la manière suivante :

```
template <class T>
struct iterator_traits<T *>
{
   typedef T         value_type;
   typedef ptrdiff_t difference_type;
   typedef T         *pointer;
   typedef T         &reference;
   typedef random_access_iterator_tag iterator_category;
};
```

Ainsi, le type iterator_traits<itérateur>::difference_type renverra toujours le type permettant de stocker la différence entre deux itérateurs, que ceux-ci soient des itérateurs ou des pointeurs normaux.

Pour comprendre l'importance des traits, prenons l'exemple de deux fonctions fournies par la librairie standard permettant d'avancer un itérateur d'un certain nombre d'étapes, et de calculer la différence entre deux itérateurs. Il s'agit respectivement des fonctions advance et distance. Ces méthodes devant pouvoir travailler avec n'importe quel itérateur, et n'importe quel type de donnée pour exprimer la différence entre deux itérateurs, elles utilisent la classe des traits. Elles sont déclarées de la manière suivante dans l'en-tête iterator :

```
template <class InputIterator, class Distance>
void advance(InputIterator &i, Distance n);

template <class InputIterator>
iterator_traits<InputIterator>::difference_ype
   distance(InputIterator first, InputIterator last);
```

Notez que le type de retour de la fonction distance est Iterator::difference_type pour les itérateurs normaux, et ptrdiff_t pour les pointeurs.

> **Note:** Ces deux méthodes ne sont pas très efficaces avec les itérateurs de type Forward, car elles doivent parcourir les valeurs de ces itérateurs une à une. Cependant, elles sont spécialisées pour les itérateurs de type plus évolués (en particulier les itérateurs à accès complet), et sont donc plus efficaces pour eux. Elles permettent donc de manipuler les itérateurs de manière uniforme, sans pour autant sacrifier les performances.

13.1.4.3. Itérateurs adaptateurs

Les itérateurs sont une notion extrêmement utilisée dans toute la librairie standard, car ils regroupent toutes les fonctionnalités permettant d'effectuer un traitement séquentiel de données. Cependant, il n'existe pas toujours d'itérateurs pour les sources de données que l'on manipule. La librairie standard fournit donc ce que l'on appelle des itérateurs *adaptateurs*, qui permettent de manipuler ces structures de données en utilisant la notion d'itérateur, même si ces structures ne gèrent pas elles-même la notion d'itérateur.

13.1.4.3.1. Adaptateurs pour les flux d'entrée / sortie standard

Les flux d'entrée / sortie standard de la librairie sont normalement utilisés avec les opérations '>>' et '<<' respectivement pour recevoir et pour envoyer des données sur un flux. Il n'existe pas d'itérateurs de type Input et de type Output permettant de lire et d'écrire sur ces flux. La librairie définit donc des adaptateurs permettant de construire ces itérateurs.

L'itérateur adaptateur pour les flux d'entrée est implémenté par la classe template istream_iterator. Cet adaptateur est déclaré comme suit dans l'en-tête iterator :

```
template <class T, class charT, class traits = char_traits<charT>,
   class Distance=ptrdiff_t>
class istream_iterator :
   public iterator<input_iterator_tag, T, Distance,
      const T *, const T &>
{
public:
   typedef charT  char_type;
   typedef traits trait_type;
   typedef basic_istream<char, traits> istream_type;
   istream_iterator();
   istream_iterator(istream_iterator &);
   istream_iterator(const istream_iterator<T, charT, traits,
      Distance> &);
   ~istream_iterator();
   const T &operator*() const;
   const T *operator->() const;
   istream_iterator<T, charT, traits, Distance> &operator++();
   istream_iterator<T, charT, traits, Distance> operator++(int);
};
```

Les opérateurs d'égalité et d'inégalité sont également définis pour ces itérateurs.

Comme vous pouvez le constater d'après cette déclaration, il est possible de construire un itérateur sur un flux

d'entrée, permettant de lire les données de ce flux une à une. S'il n'y a plus de données à lire sur ce flux, l'itérateur prend la valeur de l'itérateur de fin de fichier pour le flux. Cette valeur est celle qui est attribuée à tout nouvel itérateur construit sans flux d'entrée. L'exemple suivant présente comment faire la somme de plusieurs nombres lus sur le flux d'entrée, et de l'afficher lorsqu'il n'y a plus de données à lire.

Exemple 13-2. Itérateurs de flux d'entrée

```
#include <iostream>
#include <iterator>

using namespace std;

int main(void)
{
   double somme = 0;
   istream_iterator<double, char> is(cin);
   while (is != istream_iterator<double, char>())
   {
      somme = somme + *is;
      is++;
   }
   cout << "La somme des valeurs lue est : " <<
      somme << endl;
   return 0;
}
```

Vous pourrez essayer ce programme en tapant plusieurs nombres successivement, puis en envoyant un caractère de fin de fichier avec la combinaison de touches CTRL + Z. Ce caractère provoquera la sortie de la boucle while et affichera le résultat.

L'itérateur adaptateur pour les flux de sortie fonctionne de manière encore plus simple, car il n'y a pas à faire de test sur la fin de fichier. Il est déclaré comme suit dans l'en-tête iterator :

```
template <class T, class charT = char, class traits = char_traits<charT> >
class ostream_iterator :
   public iterator<output_iterator_tag, void, void, void, void>
{
public:
   typedef charT  char_type;
   typedef traits trait_type;
   typedef basic_ostream<char, traits> ostream_type;
   ostream_iterator(ostream_type &);
   ostream_iterator(ostream_type &, const char T *);
   ostream_iterator(const ostream_iterator<T, charT, traits> &);
   ~ostream_iterator();
   ostream_iterator<T, charT, traits> &operator=(const T &);
   ostream_iterator<T, charT, traits> &operator*();
   ostream_iterator<T, charT, traits> &operator++();
   ostream_iterator<T, charT, traits> &operator++(int);
};
```

Cet itérateur est de type Output, et ne peut donc être déréférencé que dans le but de faire une écriture dans l'objet ainsi obtenu. Ce déréférencement retourne en fait l'itérateur lui-même, si bien que l'écriture provoque l'appel de l'opérateur d'affectation de l'itérateur. Cet opérateur envoie simplement les données sur le flux de sortie que l'itérateur prend en charge, et renvoie sa propre valeur afin de réaliser une nouvelle écriture. Notez que les opérateurs d'incrémentation existent également, mais ne font strictement rien. Ils ne sont là que pour permettre d'utiliser ces itérateurs comme de simples pointeurs.

Exemple 13-3. Itérateur de flux de sortie

```
#include <iostream>
#include <iterator>

using namespace std;
```

```
const char *texte[6] = {
   "Bonjour", "tout", "le", "monde", "!", NULL
};

int main(void)
{
   ostream_iterator<const char *> os(cout, " ");
   int i = 0;
   while (texte[i] != NULL)
   {
      *os = texte[i];  // Le déréférencement est facultatif.
      os++;            // Cette ligne est facultative.
      i++;
   }
   cout << endl;
   return 0;
}
```

Il existe également des adaptateurs pour les tampons de flux d'entrée / sortie basic_streambuf. Le premier adaptateur est implémenté par la classe template istreambuf_iterator, et il permet de lire les données provenant d'un tampon de flux basic_streambuf aussi simplement qu'en manipulant un pointeur et en lisant la valeur de l'objet pointé. Le deuxième adaptateur, ostreambuf_iterator, permet quant à lui d'écrire dans un tampon en affectant une nouvelle valeur à l'objet référencé par l'itérateur. Ces adaptateurs fonctionnent donc exactement de la même manière que les itérateurs pour les flux d'entrée / sortie formatés. En particulier, la valeur de fin de fichier que prend l'itérateur d'entrée peut être récupérée à l'aide du constructeur par défaut de la classe istreambuf_iterator, instanciée pour le type de tampon utilisé.

> **Note:** L'opérateur de d'incrémentation suffixé des itérateurs istreambuf_iterator a un type de retour particulier, qui permet de représenter la valeur précédente de l'itérateur avant incrémentation et de s'assurer que cette valeur est toujours déréférençable à l'aide de l'opérateur '*'. La raison de cette particularité est que le contenu du tampon peut être modifié suite à l'appel de l'opérateur '++(int)', mais l'ancienne valeur de cet itérateur doit toujours permettre d'accéder à l'objet qu'il référençait. La valeur retournée par l'itérateur contient donc une sauvegarde de cet objet, et peut se voir appliquer l'opérateur de déréférencement '*' par l'appelant afin d'en récupérer la valeur.
>
> La notion de tampon de flux sera présentée en détail dans la Section 13.3.2.

13.1.4.3.2. Adaptateurs pour l'insertion d'éléments dans les conteneurs

Les itérateurs fournis par les conteneurs permettent d'en parcourir le contenu, et d'obtenir une référence sur chacun de leurs éléments. Ce comportement est tout à fait classique, et constitue même une des bases de la notion d'itérateur. Toutefois, l'insertion de nouveaux éléments dans un conteneur ne peut se faire que par l'intermédiaire des méthodes spécifiques aux conteneurs. La librairie standard C++ définit donc des adaptateurs pour des itérateurs dits d'*insertion*, qui permettent d'insérer des éléments dans des conteneurs par un simple déréférencement et une écriture. Grâce à ces adaptateurs, l'insertion des éléments dans les conteneurs peut être réalisée de manière uniforme, de la même manière qu'on écrirait dans un tableau qui se redimensionnerait automatiquement, à chaque écriture.

Il est possible d'insérer les nouveaux éléments en plusieurs endroits dans les conteneurs. Ainsi, les éléments peuvent être placés au début du conteneur, à sa fin, ou après un élément donné. Bien entendu, ces notions n'ont de sens que pour les conteneurs qui ne sont pas ordonnés, puisque dans le cas contraire, la position de l'élément inséré est déterminée par le conteneur lui-même.

La classe template back_insert_iterator est la classe de l'adaptateur d'insertion en fin de conteneur. Elle est déclarée comme suit dans l'en-tête iterator :

```
template <class Container>
class back_insert_iterator :
   public iterator<output_iterator_tag, void, void, void, void>
{
public:
   typedef Container container_type;
   explicit back_insert_iterator(Container &);
   back_insert_iterator<Container> &
      operator=(const typename Container::value_type &);
   back_insert_iterator<Container> &operator*();
   back_insert_iterator<Container> &operator++();
   back_insert_iterator<Container> operator++(int);
};
```

Comme vous pouvez le constater, les objets des instances cette classe peuvent être utilisés comme des itérateurs de type Output. L'opérateur de déréférencement '*' renvoie l'itérateur lui-même, si bien que les affectations sur les itérateurs déréférencées sont traitées par l'opérateur '=' de l'itérateur. C'est cet opérateur qui ajoute l'élément à affecter à la fin du conteneur auquel l'itérateur permet d'accéder, en utilisant la méthode push_back de ce dernier.

De même, la classe template front_insert_iterator de l'adaptateur d'insertion en tête de conteneur est déclarée comme suit dans l'en-tête iterator :

```
template <class Container>
class front_insert_iterator :
   public iterator<output_iterator_tag, void, void, void, void>
{
public:
   typedef Container container_type;
   explicit front_insert_iterator(Container &);
   front_insert_iterator<Container> &
      operator=(const typename Container::value_type &);
   front_insert_iterator<Container> &operator*();
   front_insert_iterator<Container> &operator++();
   front_insert_iterator<Container> operator++(int);
};
```

Son fonctionnement est identique à celui de back_insert_iterator, à ceci près qu'il effectue les insertions des éléments au début du conteneur, par l'intermédiaire de sa méthode push_front.

Enfin, la classe template de l'adaptateur d'itérateur d'insertion à une position donnée est déclarée comme suit :

```
template <class Container>
class insert_iterator :
   public iterator<output_iterator_tag, void, void, void, void>
{
public:
   typedef Container container_type;
   insert_iterator(Container &, typename Container::iterator);
   insert_iterator<Container> &
      operator=(const typename Container::value_type &);
   insert_iterator<Container> &operator*();
   insert_iterator<Container> &operator++();
   insert_iterator<Container> operator++(int);
};
```

Le constructeur de cette classe prend en paramètre, en plus du conteneur sur lequel l'itérateur d'insertion doit travailler, un itérateur spécifiant la position à laquelle les éléments doivent être insérés. Les éléments sont insérés juste avant l'élément référencé par l'itérateur fourni en paramètre. De plus, ils sont insérés séquentiellement, les uns après les autres, dans leur ordre d'affectation à l'itérateur.

La librairie standard C++ fournit trois fonctions template, qui permettent d'obtenir les trois types d'itérateur d'insertion pour chaque conteneur. Ces fonctions sont déclarées comme suit dans l'en-tête iterator :

```
template <class Container>
back_insert_iterator<Container>
```

```
    back_inserter(Container &);

template <class Container>
front_insert_iterator<Container>
    front_inserter(Container &);

template <class Container, class Iterator>
insert_iterator<Container>
    inserter(Container &, Iterator i);
```

Le programme suivant utilise un itérateur d'insertion pour remplir une liste d'élément, avant d'afficher le contenu de cette liste.

Exemple 13-4. Itérateur d'insertion

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

// Définit le type liste d'entier :
typedef list<int> li_t;

int main()
{
    // Crée une liste :
    li_t lst;
    // Insère deux éléments dans la liste de la manière classique :
    lst.push_back(1);
    lst.push_back(10);
    // Récupère un itérateur référençant le premier élément :
    li_t::iterator it = lst.begin();
    // Passe au deuxième élément :
    it++;
    // Construit un itérateur d'insertion pour insérer de nouveaux
    // éléments avant le deuxième élément de la liste :
    insert_iterator<li_t> ins_it = inserter(lst, it);
    // Insère les éléments avec cet itérateur :
    for (int i = 2; i < 10; i++)
    {
        *ins_it = i;
        ins_it++;
    }
    // Affiche le contenu de la liste :
    it = lst.begin();
    while (it != lst.end())
    {
        cout << *it++ << endl;
    }
    return 0;
}
```

La manière d'utiliser le conteneur de type list sera décrite en détail dans la Section 13.5.

13.1.4.3.3. Itérateur inverse pour les itérateurs bidirectionnels

Les itérateurs bidirectionnels et les itérateurs à accès aléatoire peuvent être parcourus dans les deux sens. Pour ces itérateurs, il est donc possible de définir un itérateur associé, dont le sens de parcours est inversé. Le premier élément de cet itérateur est donc le dernier élément de l'itérateur associé, et inversement.

La librairie standard C++ définit un adaptateur permettant d'obtenir un itérateur inverse facilement dans l'en-tête iterator. Il s'agit de la classe template reverse_iterator :

```
template <class Iterator>
class reverse_iterator :
   public iterator<
      iterator_traits<Iterator>::iterator_category,
      iterator_traits<Iterator>::value_type,
      iterator_traits<Iterator>::difference_type,
      iterator_traits<Iterator>::pointer,
      iterator_traits<Iterator>::reference>
{
public:
   typedef Iterator iterator_type;
   reverse_iterator();
   explicit reverse_iterator(Iterator);
   Iterator base() const;
   Reference operator*() const;
   Pointer operator->() const;
   reverse_iterator &operator++();
   reverse_iterator operator++(int);
   reverse_iterator &operator—();
   reverse_iterator operator—(int);
   reverse_iterator operator+(Distance) const;
   reverse_iterator &operator+=(Distance);
   reverse_iterator operator-(Distance) const;
   reverse_iterator &operator-=(Distance);
   Reference operator[](Distance n) const;
};
```

Les opérateurs de comparaison classiques et d'arithmétique des pointeurs externes + et - sont également définis dans cette en-tête.

Le constructeur de cet adaptateur prend en paramètre l'itérateur associé dans le sens inverse duquel le parcours doit se faire. L'itérateur inverse pointera alors automatiquement sur l'élément précédent l'élément pointé par l'itérateur passé en paramètre. Ainsi, si on initialise l'itérateur inverse avec la valeur de fin de l'itérateur direct, il référencera le dernier élément que l'itérateur direct avait référencé. La valeur de fin de l'itérateur inverse peut être obtenue en construisant un itérateur inverse à partir de la valeur de début de l'itérateur direct.

> **Note:** Notez que le principe spécifiant que l'adresse suivant celle du dernier élément d'un tableau doit toujours être une adresse valide est également en vigueur pour les itérateurs. La valeur de fin d'un itérateur est assimilable à cette adresse, pointant sur le dernier élément passé d'un tableau, et n'est pas plus déréférençable, car elle se trouve en dehors du tableau. Cependant, elle peut être utilisée dans les calculs d'arithmétique des pointeurs, et c'est exactement ce que fait l'adaptateur reverse_iterator.

En fait, les itérateurs inverses sont utilisés en interne par les conteneurs pour fournir des itérateurs permettant de parcourir leurs données dans le sens inverse. Le programmeur n'aura donc généralement pas besoin de construire des itérateurs inverses lui-même, il utilisera plutôt les itérateurs fournis par les conteneurs.

Exemple 13-5. Utilisation d'un itérateur inverse

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

// Définit le type liste d'entier :
typedef list<int> li_t;
```

```
int main(void)
{
   // Crée une nouvelle liste :
   li_t li;
   // Remplit la liste :
   for (int i = 0; i < 10; i++)
      li.push_back(i);
   // Affiche le contenu de la liste à l'envers :
   li_t::reverse_iterator rev_it = li.rbegin();
   while (rev_it != li.rend())
   {
      cout << *rev_it++ << endl;
   }
   return 0;
}
```

13.1.5. Abstraction des fonctions : les foncteurs

La plupart des algorithmes de la librairie standard, ainsi que quelques méthodes des classes qu'elle fournit, donnent la possibilité à l'utilisateur d'appliquer une fonction aux données manipulées. Ces fonctions peuvent être utilisées pour différentes tâches, comme pour comparer deux objets par exemple, ou tout simplement pour en modifier la valeur.

Cependant, la librairie standard n'utilise pas ces fonctions directement, mais a plutôt recours à une abstraction des fonctions : les *foncteurs*. Un *foncteur* n'est rien d'autre qu'un objet dont la classe définit l'opérateur fonctionnel '()'. Les foncteurs ont la particularité de pouvoir être utilisés exactement comme des fonctions, puisqu'il est possible de leur appliquer leur opérateur fonctionnel selon une écriture similaire à un appel de fonction. Cependant, ils sont un peu plus puissants que de simples fonctions, car ils permettent de transporter, en plus du code de l'opérateur fonctionnel, des paramètres additionnels dans leurs données membres. Les foncteurs constituent donc une fonctionnalité extrêmement puissante, qui peut être très pratique en de nombreux endroits. En fait, comme on le verra plus loin, toute fonction peut être transformée en foncteur. Les algorithmes de la librairie standard peuvent donc également être utilisés avec des fonctions classiques, moyennant cette petite transformation.

Les algorithmes de la librairie standard qui utilisent des foncteurs sont déclarés avec un paramètre template, dont la valeur sera celle du foncteur permettant de réaliser l'opération à appliquer sur les données en cours de traitement. Au sein de ces algorithmes, les foncteurs sont utilisés comme de simples fonctions, et la librairie standard ne fait donc pas d'autre hypothèses sur leur nature. Cependant, il est nécessaire de donner que des foncteurs en paramètres aux algorithmes de la librairie standard, pas de fonctions. C'est pour cette raison que la librairie standard définit un certain nombre de foncteurs standard afin de faciliter la tâche du programmeur.

13.1.5.1. Foncteurs prédéfinis

La librairie n'utilise, dans ses algorithmes, que des fonctions qui ne prennent un ou deux paramètres. Les fonctions qui prennent un paramètre et un seul sont dites " *unaires* ", alors que les fonctions qui prennent deux paramètres sont qualifiées de " *binaires* ". Afin de faciliter la création de foncteurs utilisables avec ses algorithmes, la librairie standard définit donc deux classes de base, qui définissent les types utilisés par ses algorithmes. Ces classes de base sont les suivantes :

```
template <class Arg, class Result>
struct unary_function
{
   typedef Arg    argument_type;
   typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function
{
   typedef Arg1   first_argument_type;
   typedef Arg2   second_argument_type;
   typedef Result result_type;
};
```

Ces classes sont définies dans l'en-tête functional.

La librairie définit également un certain nombre de foncteurs standard qui encapsulent les opérateurs du langage dans cette en-tête. Ces foncteurs sont les suivants :

```
template <class T>
struct plus : binary_function<T, T, T>
{
   T operator()(const T &, const T &) const;
};

template <class T>
struct minus : binary_function<T, T, T>
{
   T operator()(const T &, const T &) const;
};

template <class T>
struct multiplies : binary_function<T, T, T>
{
   T operator()(const T &, const T &) const;
};

template <class T>
struct divides : binary_function<T, T, T>
{
   T operator()(const T &, const T &) const;
};

template <class T>
struct modulus : binary_function<T, T, T>
{
   T operator()(const T &, const T &) const;
};

template <class T>
struct negate : unary_function<T, T>
{
   T operator()(const T &) const;
};

template <class T>
struct equal_to : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};

template <class T>
struct not_equal_to : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};

template <class T>
struct greater : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};
```

```
template <class T>
struct less : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};

template <class T>
struct greater_equal : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};

template <class T>
struct less_equal : binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};
```

Ces foncteurs permettent d'appliquer les principaux opérateurs du langage comme des fonctions classiques dans les algorithmes de la librairie standard.

Exemple 13-6. Utilisation des foncteurs prédéfinis

```
#include <iostream>
#include <functional>

using namespace std;

// Fonction template prenant en paramètre deux valeurs
// et un foncteur :
template <class T, class F>
T applique(T i, T j, F foncteur)
{
   // Applique l'opérateur fonctionnel au foncteur
   // avec comme arguments les deux premiers paramètres :
   return foncteur(i, j);
}

int main(void)
{
   // Construit le foncteur de somme :
   plus<int> foncteur_plus;
   // Utilise ce foncteur pour faire faire une addition
   // à la fonction "applique" :
   cout << applique(2, 3, foncteur_plus) << endl;
   return 0;
}
```

Dans l'exemple précédent, la fonction template applique prend en troisième paramètre un foncteur et l'utilise pour réaliser l'opération à faire avec les deux premiers paramètres. Cette fonction ne peut, théoriquement, être utilisée qu'avec des objets disposant d'un opérateur fonctionnel '()', et pas avec des fonctions normales. La librairie standard fournit donc les adaptateurs suivants, qui permettent de convertir respectivement n'importe quelle fonction unaire ou binaire en foncteurs :

```
template <class Arg, class Result>
class pointer_to_unary_function :
```

```
    public unary_function<Arg, Result>
{
public:
    explicit pointer_to_unary_function(Result (*)(Arg));
    Result operator()(Arg) const;
};

template <class Arg1, Arg2, Result>
class pointer_to_binary_function :
    public binary_function<Arg1, Arg2, Result>
{
public:
    explicit pointer_to_binary_function(Result (*)(Arg1, Arg2));
    Result operator()(Arg1, Arg2) const;
};

template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
    ptr_fun(Result (*)(Arg));

template <class Arg, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
    ptr_fun(Result (*)(Arg1, Arg2));
```

Les deux surcharges de la fonction template ptr_fun permettent de faciliter la construction d'un foncteur unaire ou binaire à partir du pointeur d'une fonction du même type.

Exemple 13-7. Adaptateurs de fonctions

```
#include <iostream>
#include <functional>

using namespace std;

template <class T, class F>
T applique(T i, T j, F foncteur)
{
    return foncteur(i, j);
}

// Fonction classique effectuant une multiplication :
int mul(int i, int j)
{
    return i * j;
}

int main(void)
{
    // Utilise un adaptateur pour transformer le pointeur
    // sur la fonction mul en foncteur :
    cout << applique(2, 3, ptr_fun(&mul)) << endl;
    return 0;
}
```

Note: En fait, le langage C++ est capable d'appeler une fonction directement à partir de son adresse, sans déréférencement. De plus, le nom d'une fonction représente toujours sont adresse, et est donc converti implicitement par le compilateur en pointeur de fonction. Par conséquent, il est tout à fait possible d'utiliser la fonction template applique avec une autre fonction à deux paramètres, comme dans l'appel suivant :

```
applique(2, 3, mul);
```

Cependant, cette écriture provoque la conversion implicite de l'identificateur mul en pointeur de fonction prenant deux entiers en paramètres et renvoyant un entier d'une part, et l'appel de la fonction mul par l'intermédiaire de son pointeur sans déréférencement dans la fonction template

applique d'autre part. Cette écriture est donc acceptée par le compilateur par tolérance, mais n'est pas rigoureusement exacte.

La librairie standard C++ définit également des adaptateurs pour les pointeurs de méthodes non statiques de classes. Ces adaptateurs se construisent comme les adaptateurs de fonctions statiques classiques, à ceci près que leur constructeur prend un pointeur de méthode de classe et non un pointeur de fonction normale. Ils sont déclarés de la manière suivante dans l'en-tête functional :

```
template <class Result, class Class>
class mem_fun_t :
   public unary_function<Class *, Result>
{
public:
   explicit mem_fun_t(Result (Class::*)());
   Result operator()(Class *);
};

template <class Result, class Class, class Arg>
class mem_fun1_t :
   public binary_function<Class *, Arg, Result>
{
public:
   explicit mem_fun1_t(Result (Class::*)(Arg));
   Result operator()(Class *, Arg);
};

template <class Result, class Class>
class mem_fun_ref_t :
   public unary_function<Class, Result>
{
public:
   explicit mem_fun_ref_t(Result (Class::*)());
   Result operator()(Class &);
};

template <class Result, class Class, class Arg>
class mem_fun1_ref_t :
   public binary_function<Class, Arg, Result>
{
public:
   explicit mem_fun1_ref_t(Result (Class::*)(Arg));
   Result operator()(Class &, Arg);
};

template <class Result, class Class>
mem_fun_t<Result, Class> mem_fun(Result (Class::*)());

template <class Result, class Class, class Arg>
mem_fun1_t<Result, Class> mem_fun(Result (Class::*)(Arg));

template <class Result, class Class>
mem_fun_ref_t<Result, Class> mem_fun_ref(Result (Class::*)());

template <class Result, class Class, class Arg>
mem_fun1_ref_t<Result, Class> mem_fun_ref(Result (Class::*)(Arg));
```

Comme vous pouvez le constater d'après leurs déclarations, les opérateurs fonctionnels de ces adaptateurs prennent en premier paramètre soit un pointeur sur l'objet sur lequel le foncteur doit travailler (adaptateurs mem_fun_t et mem_fun1_t), soit une référence sur cet objet (adaptateurs mem_fun_ref_t et mem_fun1_ref_t). Le premier paramètre de ces foncteurs est donc réservé pour l'objet sur lequel la méthode encapsulée doit être appelée.

En fait, la liste des adaptateurs présentée ci-dessus n'est pas exhaustive. En effet, chaque adaptateur présenté est doublé d'un autre adaptateur, capable de convertir les fonctions membres const. Il existe donc 8 adaptateurs au total permettant de construire des foncteurs à partir des fonctions membres de classes. Pour diminuer cette complexité, la librairie standard définit plusieurs surcharges pour les fonctions mem_fun et mem_fun_ref, qui permettent de construire tous ces foncteurs plus facilement, sans avoir à se soucier de la nature des pointeurs de fonctions membres utilisés. Il est fortement recommandé de les utiliser plutôt que de chercher à construire ces objets manuellement.

13.1.5.2. Prédicats et foncteurs d'opérateurs logiques

Les foncteurs qui peuvent être utilisés dans une expression logique constituent une classe particulière :les *prédicats*. Un *prédicat* est un foncteur dont l'opérateur fonctionnel renvoie un booléen. Les prédicats ont donc un sens logique, et caractérisent une propriété qui ne peut être que vraie ou fausse.

La librairie standard fournit des prédicats prédéfinis, qui effectuent les opérations logiques des opérateurs logiques du langage. Ces prédicats sont également déclarés dans l'en-tête functional :

```
template <class T>
struct logical_and :
   binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};

template <class T>
struct logical_or :
   binary_function<T, T, bool>
{
   bool operator()(const T &, const T &) const;
};
```

Ces foncteurs fonctionnent exactement comme les foncteurs vu dans la section précédente.

La librairie standard définit aussi deux foncteurs particuliers, qui permettent d'effectuer la négation d'un autre prédicat. Ces deux foncteurs travaillent respectivement sur les prédicats unaires et sur les prédicats binaires :

```
template <class Predicate>
class unary_negate :
   public unary_function<typename Predicate::argument_type, bool>
{
public:
   explicit unary_negate(const Predicate &);
   bool operator()(const argument_type &) const;
};

template <class Predicate>
class binary_negate :
   public binary_function<typename Predicate::first_argument_type,
      typename Predicate::second_argument_type, bool>
{
public:
   explicit binary_negate(const Predicate &);
   bool operator()(const first_argument_type &,
      const second_argument_type &) const;
};

template <class Predicate>
unary_negate<Predicate> not1(const Predicate &);

template <class Predicate>
```

```
binary_negate<Predicate> not2(const Predicate &);
```

Les fonctions not1 et not2 servent à faciliter la construction d'un prédicat inverse pour les prédicats unaires et binaires.

13.1.5.3. Foncteurs réducteurs

Nous avons vu que la librairie standard ne travaillait qu'avec des foncteurs prenant au plus deux arguments. Certains algorithmes n'utilisant que des foncteurs unaires, ils ne sont normalement pas capables de travailler avec les foncteurs binaires. Toutefois, si un des paramètres d'un foncteur binaire est fixé à une valeur donnée, celui-ci devient unaire, puisque seul le deuxième paramètre peut alors varier. Il est donc possible d'utiliser des foncteurs binaires même avec des algorithmes qui n'utilisent que des foncteurs unaires, à la condition de fixer l'un de ses paramètres.

La librairie standard définit des foncteurs spéciaux qui permettent de transformer tout foncteur binaire en foncteur unaire à partir de la valeur de l'un des paramètres. Ces foncteurs effectuent une opération dite de *réduction*, car ils réduisent le nombre de paramètres du foncteur binaire à un. Pour cela, ils définissent un opérateur fonctionnel à un argument, qui applique l'opérateur fonctionnel du foncteur binaire à cet argument et à une valeur fixe qu'ils mémorisent dans leur données membres.

Ces foncteurs réducteurs sont, comme les autres foncteurs, dans l'en-tête fonctional :

```
template <class Operation>
class binder1st :
   public unary_function<typename Operation::second_argument_type,
      typename Operation::result_type>
{
protected:
   Operation op;
   typename Operation::first_argument_type value;
public:
   binder1st(const Operation &,
      const typename Operation::first_argument_type &);
   result_type operator()(const argument_type &) const;
};
```

```
template <class Operation>
class binder2nd :
   public unary_function<typename Operation::first_argument_type,
      typename Operation::result_type>
{
protected:
   Operation op;
   typename Operation::second_argument_type value;
public:
   binder2nd(const Operation &,
      const typename Operation::second_argument_type &);
   result_type operator()(const argument_type &) const;
};

template <class Operation, class T>
binder1st<Operation> bind1st(const Operation & const T &);

template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation & const T &);
```

Il existe deux jeux de réducteurs, qui permettent de réduire les foncteurs binaires en fixant respectivement leur premier ou leur deuxième paramètre. Les réducteurs qui figent le premier paramètre peuvent être construits à l'aide de la fonction template bind1st, et ceux qui figent la valeur du deuxième paramètre peuvent l'être à l'aide de la fonction bind2nd.

Exemple 13-8. Réduction de foncteurs binaires

```
#include <iostream>
#include <functional>

using namespace std;

// Fonction template permettant d'appliquer une valeur
// à un foncteur unaire. Cette fonction ne peut pas
// être utilisée avec un foncteur binaire.
template <class Foncteur>
typename Foncteur::result_type applique(
   Foncteur f,
   typename Foncteur::argument_type valeur)
{
   return f(valeur);
}

int main(void)
{
   // Construit un foncteur binaire d'addition d'entiers :
   plus<int> plus_binaire;
   int i;
   for (i = 0; i < 10; i++)
   {
      // Réduit le foncteur plus_binaire en fixant son
      // premier paramètre à 35. Le foncteur unaire obtenu
      // est ensuite utilisé avec la fonction applique :
      cout << applique(bind1st(plus_binaire, 35), i) << endl;
   }
   return 0;
}
```

13.1.6. Gestion personnalisée de la mémoire : les allocateurs

L'une des plus grandes forces de la librairie standard est de donner aux programmeurs le contrôle total de la gestion de la mémoire pour leurs objets. En effet, les conteneurs peuvent être amenés à créer un grand nombre d'objets, dont le comportement peut être très différent selon leur type. Si, dans la majorité des cas, la gestion de la mémoire effectuée par la librairie standard convient, il peut être parfois nécessaire de prendre en charge soi-même les allocations et les libérations de la mémoire pour certains objets.

La librairie standard utilise pour cela la notion d'*allocateur*. Un allocateur est une classe C++ disposant de méthodes standard que les algorithmes de la librairie peuvent appeler lorsqu'elles désirent allouer ou libérer de la mémoire. Pour cela, les conteneurs de la librairie standard C++ prennent tous un paramètre template représentant l'allocateur mémoire qu'ils devront utiliser. Bien entendu, la librairie standard fournit un allocateur par défaut, et ce paramètre template prend par défaut la valeur de cet allocateur. Ainsi, les programmes qui ne désirent pas spécifier un allocateur spécifique pourront simplement ignorer ce paramètre template.

Les autres programmes pourront définir leur propre allocateur. Cet allocateur devra évidemment fournir toutes les fonctionnalités de l'allocateur standard, et satisfaire à quelques contraintes particulières. L'interface des allocateurs est fournie par la déclaration de l'allocateur standard, dans l'en-tête memory :

```
template <class T>
class allocator
{
public:
   typedef size_t    size_type;
   typedef ptrdiff_t difference_type;
   typedef T         *pointer;
   typedef const T   *const_pointer;
```

```
    typedef T       &reference;
    typedef const T  &const_reference;
    typedef T       value_type;
    template <class U>
    struct rebind
    {
       typedef allocator<U> other;
    };

    allocator() throw();
    allocator(const allocator &) throw();
    template <class U>
    allocator(const allocator<U> &) throw();
    ~allocator() throw();
    pointer address(reference);
    const_pointer address(const_reference) const;
    pointer allocate(size_type,
       typename allocator<void>::const_pointer);
    void deallocate(pointer, size_type);
    size_type max_size() const throw();
    void construct(pointer, const T &);
    void destroy(pointer);
};

// Spécialisation pour le type void pour éliminer les références :
template <>
class allocator<void>
{
public:
    typedef void       *pointer;
    typedef const void *const_pointer;
    typedef void       value_type;
    template <class U>
    struct rebind
    {
       typedef allocator<U> other;
    };
};
```

Vous noterez que cet allocateur est spécialisé pour le type void, car certaines méthodes et certains typedef n'ont pas de sens pour ce type de donnée.

Le rôle de chacune des méthodes des allocateurs est très clair et n'appelle pas beaucoup de commentaires. Les deux surcharges de la méthode address permettent d'obtenir l'adresse d'un objet alloué par cet allocateur à partir d'une référence. Les méthodes allocate et deallocate permettent respectivement de réaliser une allocation de mémoire et la libération du bloc correspondant. La méthode allocate prend en paramètre le nombre d'objets qui devront être stockés dans le bloc à allouer, et un pointeur indiquant l'emplacement où l'allocation doit se faire de préférence. Ce dernier paramètre peut ne pas être pris en compte par l'implémentation de la librairie standard que vous utilisez, et, s'il l'est, son rôle n'est pas spécifié. Dans tous les cas, s'il n'est pas nul, ce pointeur doit être un pointeur sur un bloc déjà alloué par cet allocateur et non encore libéré. La plupart des implémentations chercheront à allouer un bloc adjacent à celui fourni en paramètre, mais ce n'est pas toujours le cas. De même, notez le nombre d'objets spécifiée à la méthode deallocate doit être exactement que celle utilisée pour l'allocation dans l'appel correspondant à allocate. Autrement dit, l'allocateur ne mémorise pas lui-même la taille des blocs mémoire qu'il a fourni.

> **Note:** Le pointeur passé en paramètre à la méthode allocate n'est ni libéré, ni réalloué, ni réutilisé par l'allocateur. Il ne s'agit donc pas d'une modification de la taille mémoire du bloc

fourni en paramètre, et ce bloc devra toujours être libéré indépendamment de celui qui sera alloué. Ce pointeur n'est utilisé par les implémentations que dans un but d'optimisation dans les algorithmes et les conteneurs internes.

La méthode allocate peut lancer l'exception bad_alloc en cas de manque de mémoire, ou si le nombre d'objets spécifié en paramètre est trop gros. Vous pourrez obtenir le nombre maximum que la méthode allocate est capable d'accepter grâce à la méthode max_size de l'allocateur.

Les deux méthodes construct et destroy permettent respectivement de construire un nouvel objet et d'en détruire à l'adresse indiquée en paramètre. Elles doivent être utilisées lorsque l'on désire appeler le constructeur ou le destructeur d'un objet stocké dans une zone mémoire allouée par cet allocateur et non par les opérateurs new et delete du langage (rappelons que ces opérateurs effectuent ce travail automatiquement). Pour effectuer la construction d'un nouvel objet, construct utilise l'opérateur new avec placement, et pour le détruire, destroy appelle directement le destructeur de l'objet.

Note: Les méthodes construct et destroy n'effectuent pas l'allocation et la libération de la mémoire elles-mêmes. Ces opérations doivent être effectuées avec les méthodes allocate et deallocate de l'allocateur.

Exemple 13-9. Utilisation de l'allocateur standard

```
#include <iostream>
#include <memory>

using namespace std;

class A
{
public:
   A();
   A(const A &);
   ~A();
};

A::A()
{
   cout << "Constructeur de A" << endl;
}

A::A(const A &)
{
   cout << "Constructeur de copie de A" << endl;
}

A::~A()
{
   cout << "Destructeur de A" << endl;
}

int main(void)
{
   // Construit une instance de l'allocateur standard pour la classe A :
   allocator<A> A_alloc;

   // Alloue l'espace nécessaire pour stocker cinq instances de A :
   allocator<A>::pointer p = A_alloc.allocate(5);

   // Construit ces instances et les initialise :
   A init;
   int i;
   for (i=0; i<5; i++)
      A_alloc.construct(p+i, init);
   // Détruit ces instances :
   for (i=0; i<5; i++)
      A_alloc.destroy(p+i);
```

```
    // Reconstruit ces 5 instances :
    for (i=0; i<5; i++)
        A_alloc.construct(p+i, init);
    // Destruction finale :
    for (i=0; i<5; i++)
        A_alloc.destroy(p+i);

    // Libère la mémoire :
    A_alloc.deallocate(p, 5);
    return 0;
}
```

Vous voyez ici l'intérêt que peut avoir les allocateurs de la librairie standard. Les algorithmes peuvent contrôler explicitement la construction et la destruction des objets, et surtout les dissocier des opérations d'allocation et de libération de la mémoire. Ainsi, un algorithme devant effectuer beaucoup d'allocation mémoire pourra, s'il le désire, effectuer les allocations de mémoire une bonne fois pour toutes grâce à l'allocateur standard, et de n'effectuer que les opérations de construction et de destruction des objets lorsque cela est nécessaire. En procédant ainsi, le temps passé dans les routines de gestion de la mémoire est éliminé, et l'algorithme sera d'autant plus performant. Inversement, un utilisateur expérimenté pourra définir son propre allocateur mémoire, adapté aux objets qu'il voudra stocker dans un conteneur. En imposant au conteneur de la librairie standard d'utiliser cet allocateur personnalisé, il obtiendra des performances optimales.

La définition d'un allocateur maison consiste simplement à implémenter une classe template disposant des mêmes méthodes et types que ceux définis par l'allocateur allocator. Toutefois, il faut savoir que la librairie impose des contraintes sur la sémantique de ces méthodes :

- toutes les instances de la classe template de l'allocateur permettent d'accéder à la même mémoire. Ces instances sont donc interchangeables, et il est possible de passer de l'une à l'autre à l'aide de la structure template rebind et de son typedef other. Notez que le fait d'encapsuler ce typedef dans une structure template permet de simuler la définition d'un type template ;
- toutes les instances d'un allocateur d'un type donné permettent également d'accéder à la même mémoire. Ceci signifie qu'il n'est pas nécéssaire de disposer d'une instance globale pour chaque allocateur, il suffit simplement de créer un objet local d'une des instances de la classe template de l'allocateur pour allouer et libérer de la mémoire. Notez ici la différence avec la contrainte précédente : cette contrainte porte ici sur les objets instances des classes template instanciées, alors que la contrainte précédente portait sur les instances elles-mêmes de la classe template de l'allocateur ;
- toutes les méthodes de l'allocateur doivent s'exécuter dans un temps amorti constant (ceci signifie que le temps d'exécution de ces méthodes est majoré par une borne supérieure fixe, qui ne dépend pas du nombre d'allocation déjà effectuées ni de la taille du bloc de mémoire demandé) ;
- les méthodes allocate et deallocate sont susceptibles d'utiliser les opérateurs new et delete du langage. Ce n'est pas une obligation, mais cette contrainte signifie que les programmes qui redéfinissent ces deux opérateurs doivent être capable de satisfaire les demandes de l'allocateur standard ;
- les types pointer, const_pointer, size_type et difference_type doivent être égaux respectivement aux types T *, const T*, size_t et ptrdiff_t. En fait, cette contrainte n'est imposée que pour les allocateurs destinés à être utilisés par les conteneurs de la librairie standard, mais il est plus simple de la généraliser à tous les cas d'utilisation.

Pour terminer ce tour d'horizon des allocateurs, sachez que la librairie standard définit également un type itérateur spécial permettant de stocker des objets dans une zone de mémoire non initialisée. Cet itérateur, nommé raw_storage_iterator, est de type Output et n'est utilisé qu'en interne par la librairie standard. De même, la librairie définit des algorithmes permettant d'effectuer des copies brutes de blocs mémoire et d'autres manipulations sur les blocs alloués par les allocateurs. Ces algorithmes sont également utilisés en interne, et ne

seront donc pas décrits plus en détail ici.

13.2. Les types complémentaires

13.2.1. Les chaînes de caractères

dynamique (5 échecs en statique)

13.2.2. Les pointeurs auto

13.2.3. Les complexes

13.2.4. Les tableaux de valeurs

13.3. Les flux d'entrée / sortie

Dynamique (2 échecs en statique)

* Rappels sur cin et cout

* Présentation du mécanisme standard des flux

* Flux et fichiers

* Options de formatage

13.3.1. Notions de base

13.3.2. Les tampons

stream_buf

13.3.3. Les classes de base : ios_base et basic_ios

13.3.4. Flux d'entrée

13.3.5. Flux de sortie

13.3.6. Flux d'entrée / sortie

13.4. Les locales

* Définition

* Facettes

* Classification

* Conversion

* Numériques

* Monnaie

* Temps

* Messages

* etc.

13.5. Les conteneurs

Dynamique (2 échecs en statique, 2 échecs à la compilation sur map_operators, setoperators)

* Complexité algorithmique

* La paire

* Les conteneurs

* Les adaptateurs

13.6. Les algorithmes

* Les algorithmes

Chapitre 14. Conclusion

Pour terminer, je rappellerais les principales règles pour réaliser de bons programmes. Sans organisation, aucun langage, aussi puissant soit-il, ne peut garantir le succès d'un projet. Voici donc quelques conseils :

- commentez votre code, mais ne tuez pas le commentaire en en mettant là où les opérations sont vraiment très simples ou décrites dans un document externe. Marquez les références aux documents externes dans les commentaires ;
- analysez le problème avant de commencer la programmation. Ceci comprend plusieurs étapes. La première est de réfléchir aux structures de données à utiliser et aux opérations qu'on va leur appliquer (il faut donc identifier les classes). Il faut ensuite établir les relations entre les classes ainsi identifiées et leurs communications. Pour cela, on pourra faire des diagrammes d'événements qui identifient les différentes étapes du processus permettant de traiter une donnée. Enfin, on décrira chacune des méthodes des classes fonctionnellement, afin de savoir exactement quelles sont leurs entrées et les domaines de validité de celles-ci, leurs sorties, leurs effets de bords et les opérations effectuées. Enfin seulement on passera au codage. Si le codage implique de corriger les résultats des étapes précédentes, c'est que la conception a été incorrecte ou incomplète : il vaut mieux retourner en phase de conception un peu pour voir l'impact des modifications à faire. Ceci permet de ne pas passer à coté d'un effet de bord inattendu, et donc d'éviter de perdre du temps dans la phase de mise au point ;
- ne considérez aucun projet, même un petit projet ou un projet personnel, comme un projet qui échappe à ces règles. Si vous devez interrompre le développement d'un projet pour une raison quelconque, vous serez content de retrouver le maximum d'informations sur lui. Il en est de même si vous désirez améliorer un ancien projet. Et si la conception a été bien faite, cette amélioration ne sera pas une verrue sur l'ancienne version du logiciel, contrairement à ce qui se passe trop souvent.

Voilà. Vous connaissez à présent la plupart des fonctionnalités du C++. J'espère que la lecture de ce cours vous aura été utile et agréable. Si vous voulez en savoir plus, consultez les Draft Papers, mais sachez qu'ils sont réellement difficiles à lire. Ils ne peuvent vraiment pas être pris pour un support de cours. L'annexe B décrit l'organisation générale de ce document et donne quelques renseignements pour faciliter leur lecture.

Bonne lecture.

Annexe A. Priorités des opérateurs

Cette annexe donne la priorité des opérateurs du langage C++, dans l'ordre décroissant. Cette priorité intervient dans l'analyse de toute expression et dans la détermination de son sens. Cependant, l'analyse des expressions peut être modifiée en changeant les priorités, à l'aide de parenthèses.

Tableau A-1. Opérateurs du langage

Opérateur	**Nom ou signification**
::	Opérateur de résolution de portée
[]	Opérateur d'accès aux éléments de tableau
()	Opérateur d'appel de fonction
type()	Opérateur de transtypage explicite
.	Opérateur de sélection de membre
->	Opérateur de sélection de membre par déréférencement
++	Opérateur d'incrémentation post-fixe
—	Opérateur de décrémentation post-fixe
new	Opérateur de création dynamique d'objet
new[]	Opérateur de création dynamique de tableaux
delete	Opérateur de destruction d'objet créé dynamiquement
delete[]	Opérateur de destruction de tableaux créés dynamiquement
++	Opérateur d'incrémentation préfixe
—	Opérateur de décrémentation préfixe
*	Opérateur de déréférencement
&	Opérateur d'adresse
+	Opérateur plus unaire
-	Opérateur moins unaire
!	Opérateur de négation logique
~	Opérateur de complément à un
sizeof	Opérateur de taille d'objet
sizeof	Opérateur de taille de type
typeid	Opérateur d'identification de type
(type)	Opérateur de transtypage
const_cast	Opérateur de transtypage de constance

Opérateur	**Nom ou signification**
dynamic_cast	Opérateur de transtypage dynamique
reinterpret_cast	Opérateur de réinterprétation
static_cast	Opérateur de transtypage statique
.*	Opérateur de sélection de membre par pointeur sur membre
->*	Opérateur de sélection de membre par pointeur sur membre par déréférencement
*	Opérateur de multiplication
/	Opérateur de division
%	Opérateur de modulo
+	Opérateur d'addition
-	Opérateur de soustraction
<<	Opérateur de décalage à gauche
>>	Opérateur de décalage à droite
<	Opérateur d'infériorité
>	Opérateur de supériorité
<=	Opérateur d'infériorité ou d'égalité
>=	Opérateur de supériorité ou d'égalité
==	Opérateur d'égalité
!=	Opérateur d'inégalité
&	Opérateur et binaire
^	Opérateur ou exclusif binaire
|	Opérateur ou inclusif binaire
&&	Opérateur et logique
||	Opérateur ou logique
?:	Opérateur ternaire
=	Opérateur d'affectation
*=	Opérateur de multiplication et d'affectation
/=	Opérateur de division et d'affectation
%=	Opérateur de modulo et d'affectation
+=	Opérateur d'addition et d'affectation
-=	Opérateur de soustraction et d'affectation
<<=	Opérateur de décalage à gauche et d'affectation
>>=	Opérateur de décalage à droite et d'affectation
&=	Opérateur de et binaire et d'affectation
|=	Opérateur de ou inclusif binaire et d'affectation
^=	Opérateur de ou exclusif binaire et d'affectation
,	Opérateur virgule

Annexe B. Draft Papers

Les Draft Papers sont vraiment une source d'informations très précises, mais ils ne sont absolument pas structurés. En fait, ils ne sont destinés qu'aux éditeurs de logiciels désirant réaliser un compilateur, et la structure du document ressemble à un texte de loi (fortement technique en prime). Les exemples y sont rares, et quand il y en a, on ne sait pas à quel paragraphe ils se réfèrent. Enfin, nombre de termes non définis sont utilisés, et il faut lire le document pendant quelques 40 pages avant de commencer à le comprendre.

Afin de faciliter leur lecture, je donne ici quelques définitions, ainsi que la structure des Draft Papers.

Les Draft Papers sont constitués de deux grandes parties. La première traite du langage, de sa syntaxe et de sa sémantique. La deuxième partie décrit la librairie standard C++. Quasiment aucune des informations présentes dans cette deuxième partie n'a été traitée dans ce cours, seule la première partie a servi. Heureusement, cette deuxième partie est nettement plus lisible.

La syntaxe est décrite dans la première partie de la manière BNF. Il vaut mieux être familiarisé avec cette forme de description pour la comprendre. Ceci ne causera pas de problèmes cependant si l'on maîtrise déjà la syntaxe du C++.

Lors de la lecture de la deuxième partie, on ne s'attardera pas trop sur les fonctionnalités de gestion des langues et des jeux de caractères (locales). Elles ne sont pas nécessaires à la compréhension de la librairie standard template. Une fois les grands principes de la librairie assimilés, les notions de locale pourront être approfondies.

Les termes suivants sont souvent utilisés et non définis (ou définis au milieu du document d'une manière peu claire). Leurs définitions pourront être d'un grand secours lors de lecture de la première partie des Draft Papers :

- *cv*, *cv qualified* : l'abréviation *cv* signifie ici *const* ou *volatile*. Ce sont donc les propriétés de constance et de volatilité ;
- un *agrégat* est un tableau ou une classe qui n'a pas de constructeurs, pas de fonctions virtuelles, et pas de données non statiques *private* ou *protected*
- *POD* : cette abréviation signifie *plain ol' data*, ce qui n'est pas compréhensible a priori. En fait, un type *POD* est un type relativement simple, pour lequel aucun traitement particulier n'est nécessaire (pas de constructeur, pas de virtualité, etc…). La définition des types *POD* est récursive : une structure ou une union est un type *POD* si c'est un agrégat qui ne contient pas de pointeurs sur des membres non statiques, pas de références, pas de type non *POD*, pas de constructeur de copie et pas de destructeur.

Les autres termes sont définis lorsqu'ils apparaissent pour la première fois dans le document.

Appendix C. GNU Free Documentation License

Version 1.1, March 2000

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format,

LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

BIBLIOGRAPHIE

Les titres marqués d'un astérisque sont vraiment bons (au niveau pédagogique).

Langage C

** C as a Second Language For Native Speakers of Pascal*, Müldner and Steele, Addison-Wesley.

The C Programming Language, Brian W. Kernigham and Dennis M. Ritchie, Prentice Hall.

Langage C++

** L'essentiel du C++*, Stanley B. Lippman, Addison-Wesley.

The C++ Programming Language, Bjarne Stroustrup, Addison-Wesley.

Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++, ISO.

Librairie C / appels systèmes POSIX et algorithmique

** Programmation système en C sous Linux*, Christophe Blaess, Eyrolles.

Introduction à l'algorithmique, Thomas Cormen, Charles Leiserson, et Ronald Rivest, Dunod.