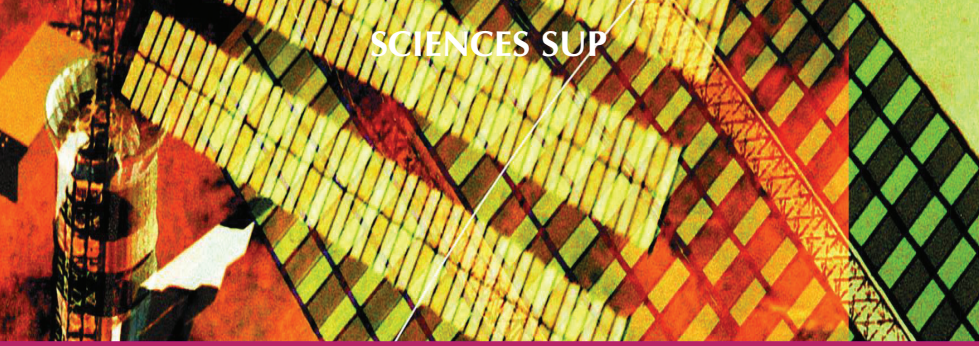


Aide-mémoire

BTS • IUT • Licence • Écoles d'ingénieurs

AIDE-MÉMOIRE DE C++

Jean-Michel Réveillac

DUNOD

AIDE-MÉMOIRE DE C++

Consultez nos catalogues sur le Web

www.dunod.com

AIDE-MÉMOIRE DE C++

Jean-Michel Réveillac
Maître de conférences à l'université de Bourgogne

DUNOD

Illustration de couverture : *Digital Vision*

ISBN 2 10 007621 3

Table des matières

Avertissement

Dans cet ouvrage, chaque chapitre commence par une liste d'opérateurs, de mots-clés et de fonctions nouvellement utilisées.

Les termes spécifiques à la manipulation ou à la description du langage sont en italique.

Les programmes d'exemples qui mettent en application un ou plusieurs principes spécifiques sont placés dans des encadrés constitués de trois parties :

– l'ensemble des lignes de code du programme ;
– l'affichage des résultats lors de son exécution ;
– des commentaires qui viennent éclaircir son fonctionnement.

Vous ne trouverez pas de commentaires à l'intérieur même du code, sous la forme dédiée au langage C++. Je n'ai pas voulu alourdir les exemples pour que le lecteur puisse les analyser le mieux possible.

En fin d'ouvrage, de nombreuses annexes viennent compléter l'ensemble des notions présentes dans chacun des chapitres.

Ce livre est une introduction au langage C++. Il essaie de présenter chacune des fonctionnalités principales du langage par le biais d'exemples de code concis que le lecteur pourra étudier de façon approfondie.

Il est impératif que le lecteur comprenne que le manque de rigueur dans l'écriture du C++ conduit au développement de programmes très vite illisibles et difficiles à maintenir.

J'ai essayé de respecter un style de programmation toujours identique sur l'ensemble des chapitres.

Les exemples ont été testés sur un micro-ordinateur de type PC, opérant avec le système d'exploitation Microsoft Windows XP Pro. L'édition du code et la compilation ont été réalisés avec Microsoft Visual C++ 6.0 sous la forme de programmes exécutables en mode console (voir annexe A).

Vous trouverez ci-dessous quelques liens de téléchargement de compilateurs C++ gratuits ou « shareware » :

– Compilateur Digital Mars C/C++ 8.29 :
  www.digitalmars.com
– Environnement de programmation et compilateur Bloodshed Dev-C++ 4.0 :
  www.bloodshed.net
– Compilateur DJGPP C/C++ pour PC sous DOS :
  www.delorie.com/djgpp/
– Compilateur Borland C++ 5.5 US :
  www.borland.com

Introduction

LE CONTENU DE CET OUVRAGE

Ce livre est une introduction au langage C++. Il essaie, en peu plus d'une centaine de pages, d'exposer de façon claire et précise les principes et concepts clés du langage.

Écrire un livre si concis sur le langage C++, comme pour tout langage de programmation est une tâche ardue et difficile, les possibilités de traitement des données étant inépuisables.

Je pense être honnête et objectif en précisant que le langage C++ est complexe mais que l'étude d'exemples basiques autorise son apprentissage de façon simple. Toutefois, seule sa pratique enrichira les connaissances que vous pourrez acquérir par la lecture de cet ouvrage.

Fixez-vous un objectif, afin d'avoir à développer une application si petite soit elle. Les erreurs et les difficultés rencontrées lors de la programmation vous permettront de parfaire vos connaissances.

J'ai essayé de rassembler ici les principes de bases en considérant que le lecteur ne connaît rien de ce langage mais possède déjà une expérience de la programmation.

UN BREF RAPPEL HISTORIQUE

Le langage C++ est né en 1983. Il a pour origine le langage C qui a été créé au début des années 1970 par Dennis MacAlistair Ritchie puis

Brian W. KERNIGHAN qui l'a rejoint, tous deux travaillant pour le laboratoire de recherche AT&T Bell.

Son but initial était la réécriture d'un nouveau système d'exploitation UNIX, devant être rapide à l'exécution et portable.

Les deux créateurs rédigeront « The C programming language », ouvrage de référence sur la programmation C.

En 1983, l'institut national américain de normalisation (ANSI, *American National Standards Institute*) commence un travail de normalisation du langage qui aboutira à l'approbation d'une norme « C ANSI » en 1988.

En 1989, L'ISO (*International Organization for Standardization,* ou Organisation Internationale de Normalisation), standardise le C sous la dénomination C89, qui sera mise à jour en 1999, pour définir C99.

Le langage C est un langage de bas niveau qui manipule des nombres, des caractères et des adresses. Son avantage réside avant tout dans le code concis et optimal généré par les compilateurs.

La communauté des développeurs trouvant le langage C limité, Bjarne STROUSTRUP eut l'idée de reprendre, dans les années 1980, ce langage pour le faire évoluer vers un langage orienté objet. En 1998, le langage C++ est standardisé (ISO/IEC 14882).

QUELQUES CONSEILS POUR LA LECTURE

Les chapitres de ce livre sont conçus de façon à suivre une progression croissante dans l'apprentissage du langage C++. Toutefois, le lecteur déjà averti pourra consulter directement les notions qui l'intéressent, à l'aide de la table des matières, sans respecter la progression, chaque exemple étant conçu indépendamment.

Bonne lecture et que le C++ soit avec vous !
Jean-Michel RÉVEILLAC

Chapitre **1**

Langage C++, les bases

Opérateurs, mots-clés et fonctions

`+, -, *, /, %, ++, --, ||, !, ?, //, /#, #/, &&, char, const, cout, double, endl, enum, float, include, int, long, main, return, short, signed, unsigned`

1.1 STRUCTURE D'UN PROGRAMME SIMPLE

```
#include <iostream.h>

main()
{
  cout << "bonjour\n";
  return 0;
}
```

```
bonjour
```

Un premier programme.

➤ Description

La première ligne de notre programme intègre une directive `#include` qui permet d'appeler le fichier d'en-tête `<iostream.h>`[1] qui fait partie des bibliothèques standards de C++ et dans lequel se trouve l'objet `cout` qui est utilisé un peu plus bas. Les symboles < et > sont là pour indiquer qu'ils encadrent un fichier de la bibliothèque.

La seconde ligne contient l'en-tête de la fonction `main()`. Elle est obligatoire dans un programme C++. C'est le début du programme pour le compilateur. Les parenthèses qui suivent `main` sont elles aussi obligatoires.

La troisième ligne est constituée d'une accolade ouvrante : `{`. Elle marque le début de la fonction `main` dont la fin, en sixième ligne, est indiquée par une accolade fermante : `}`.

La ligne suivante demande l'affichage du mot `bonjour` sur l'écran du système.

L'objet `cout` (console out) défini la sortie, en général l'écran du système. Les symboles << constituent ce que l'on appelle un opérateur de sortie (on trouvera en annexe B la liste des opérateurs de C++). Comme tous les opérateurs, il agit sur l'objet `cout` en lui envoyant l'expression placée à sa droite, c'est-à-dire le mot `bonjour`. On a pour habitude de dire que l'objet `cout` est un *flux* car il écoule ce qui est situé à sa droite à destination de l'écran. Ce flux peut être composé de plusieurs expressions comme nous le verrons plus tard.

Le mot `bonjour` et constitué d'une chaîne de caractères (ou *chaîne littérale*) et doit donc être encadré par des guillemets (ou apostrophes doubles - `" "`). Il est aussi suivi de la séquence `\n` qui indiquent que derrière le mot `bonjour`, un passage à la ligne suivante doit être effectué (une liste des séquences de ce type est située en annexe C).

---

1. En fonction des différentes versions ou implémentations de C++, on peut utiliser `iostream.h` ou `iostream`. La norme ISO/IEC 14882-1998 recommande d'utiliser iostream, cependant beaucoup d'éditeurs de compilateurs C++ conservent l'extension ou bien même, mettent à disposition les deux bibliothèques.
Dans cet ouvrage, j'ai utilisé dans la plupart des exemples, les fichiers d'en-tête ancienne norme, celle-ci étant souvent reconnue et encore très répandue.

Il faut remarquer que l'opérateur de sortie `<<` et le passage à la ligne `\n` sont des ensembles de deux caractères qui doivent être accolés sans espace.

La constante `endl` aurait pu remplacer la séquence `\n` tout en apportant le vidage du tampon de sortie si elle avait été envoyée à `cout`.

```
cout << "bonjour" << endl;
```

L'objet `cout` peut reconstituer un *flux de sortie* continu depuis un envoi morcelé. On pourrait écrire cette quatrième ligne comme ci-dessous, le résultat obtenu serait identique.

```
cout << "bon" << "jour\n";
```

ou

```
cout << "bon" << "jour" << endl;
```

Un point-virgule termine cette ligne. Ce symbole est demandé par C++ lorsqu'une instruction se termine. Plusieurs instructions peuvent se trouver sur la même ligne séparées par des points virgules mais la plupart des développeurs ne le font pas, car cette mise en forme du programme source nuit à la lisibilité. Au contraire une instruction peut s'étendre sur plusieurs lignes dont la dernière se terminera par un point-virgule, cela permet d'aérer le code source et ainsi de le rendre plus compréhensible.

La cinquième ligne qui contient `return 0;` est là pour terminer proprement le programme qui redonnera la main au système d'exploitation de la machine. Cette ligne n'est pas obligatoire mais de nombreux compilateurs envoient un message d'avertissement si elle n'existe pas.

1.2 LES COMMENTAIRES

Un programme est souvent susceptible de contenir des commentaires qui deviennent très vite indispensables pour le développeur lors de la mise au point du programme et qui facilitent la compression du code pour des tiers.

Il existe deux façons pour inclure des commentaires. Il faut savoir que le compilateur doit pouvoir ignorer ces lignes supplémentaires et donc les reconnaître pour pouvoir les différencier des lignes de code.

Une première solution est de faire précéder le commentaire de deux // (slash), mais dans ce cas, il ne peut s'étendre sur plus d'une ligne. C'est le style le plus utilisé et défini dans la norme C++.

```
//Mon premier programme avec ses commentaires
//-----------------------------------------
#include <iostream.h> //Directive de compilation
main()
{
  cout << "bonjour\n"; //envoi du mot bonjour en
sortie
  return 0; //redonne le contrôle au système
}
```

```
bonjour
```

Un programme commenté.

La seconde solution est d'encadrer le commentaire des deux caractères /* et */ (cette notation vient du langage C).

```
/*Mon premier programme avec ses commentaires
-----------------------------------------*/
#include <iostream.h> /*Directive de compilation*/
main()
{
  cout << "bonjour\n"; /*envoi du mot bonjour en
sortie*/
  return 0; /*redonne le contrôle au système*/
}
```

```
bonjour
```

Un programme commenté suivant la notation du langage C.

1.3 LES VARIABLES

Une variable est un *identificateur* qui désigne un type d'information dans le programme. Elle est placée à un endroit précis dans la mémoire de la machine. Une variable représente souvent une donnée élémentaire, c'est-à-dire une valeur numérique ou un caractère.

Le mécanisme qui consiste à associer une valeur à une variable est appelé affectation et son opérateur en langage C++ est le signe `=`.

Une variable va donc posséder un type qui va permettre au compilateur de définir l'encombrement mémoire de cette dernière. Les types de données de base sont les suivant :

- `int` : valeur entière ;
- `char` : caractère simple ;
- `float` : nombre réel en virgule flottante ;
- `double` : nombre réel en virgule flottante double précision.

Des *qualificateurs* (ou *spécificateurs*) comme `short`, `long`, `signed`, `unsigned` peuvent préciser les types de données. Un tableau résumant l'ensemble total des types est disponible en annexe D.

En langage C++, on doit informer le compilateur du type des variables qui seront utilisées dans la programmation, pour ce faire on va effectuer une opération de déclaration.

Pour déclarer une variable on précise son type, suivi de son *identificateur* (son nom).

L'*identificateur* d'une variable peut être composé de chiffres ou de lettres dans un ordre quelconque, la seule restriction est que le premier caractère soit une lettre. Les minuscules et les majuscules sont autorisées, le caractère « _ » (underscore ou blanc souligné) est admis. Il faut veiller à ne pas utiliser des mots-clés ou mots réservés du langage C++ (voir annexe E) ou des séquences d'échappement (voir annexe C).

Vous trouverez ci-dessous quelques exemples d'identificateurs autorisés.

```
X                x
x15              taux
T_V_A            Total
somme_totale     _montant
```

On peut assimiler la déclaration d'une variable à la création en mémoire d'un contenant dont le type sera la dimension et la valeur de la variable le contenu.

Lorsqu'aucune valeur n'a encore été affectée à la variable seule sa place est réservée, son contenu n'étant pas encore défini.

On peut déclarer une variable à tout moment au cours de la rédaction des lignes de code, cependant, pour des raisons de compréhension, les déclarations sont souvent regroupées.

Dans une déclaration, on peut mentionner le type une seule fois pour plusieurs variables. Il suffit simplement de les séparer par des virgules.

```
#include <iostream.h>

main()
{
  //déclarations des variables entières
  int multiplicateur, multiplicande;
  //déclarations de variables réelles
  float x, pi;
  //affectation des variables
  multiplicateur = 1234;
  multiplicande=5678;
  x=9.0976;
  pi=3.14;
  //création du flux de sortie, affichage écran
  cout << multiplicateur << endl << multiplicande
<< endl << x << endl << pi << endl;
  return 0;
}
```

```
1234
5678
9.0976
3.14
```

On déclare deux variables entières : multiplicateur, multiplicande *et deux variables réelles :* x *et* y. *On leur affecte ensuite des valeurs entières puis on compose un flux de sortie.*

Une déclaration peut être associée à une affectation ce qui permet d'initialiser la variable.

```
#include <iostream.h>

main()
{
  //déclarations et initialisations des variables
  int multiplicateur=55;
  int multiplicande=15;
  int produit;
  produit=multiplicande*multiplicateur;
  //création du flux de sortie
  cout << produit << endl ;
  return 0;
}
```

```
825
```

On déclare trois variables, dont multiplicande *et* multiplicateur *qui sont déclarées et initialisées dans les mêmes lignes. 55 et 15 sont les valeurs qui leurs sont affectées. On affecte à la variable* Produit*, le produit* multiplicande * multiplicateur.

Il existe d'autres possibilités pour la déclaration et l'affectation.

```
int x = 4 ;
int y = x*15 ;

int x=4, y=15 ;
```

1.4 LES TYPES ENTIERS

Ce sont des nombres qui peuvent être positifs ou négatifs. Ils existent des entiers signés et des entiers non signés qui sont répartis suivant neuf types différents en langage C++.

```
int, short int, long int,
unsigned int, unsigned short int, unsigned long int,
char, signed char, unsigned car
```

Ils sont tous différents car ils couvrent des plages de valeurs différentes. Ces plages varient en fonction de la machine et du compilateur. Sur un PC type Pentium 4, avec Microsoft Visual C++, les plages couvertes sont indiquées dans le tableau 1.1.

TABLEAU 1.1 PLAGES COUVERTES.

Type	Mini	Maxi	Taille (octets)
int	-2 147 483 648	2 147 483 647	4
short int	-32 768	32767	2
long int	-2 147 483 648	2 147 483 647	4
unsigned int	0	4 294 967 295	4
unsigned short int	0	65 535	2
unsigned long int	0	4 294 967 295	4
char	-128	127	1
signed char	-128	127	1
unsigned char	0	255	1

Le type caractère, `char`, comme on le voit est un type entier et cela permet de manipuler des caractères comme des entiers à l'intérieur d'un programme C++. En fait le compilateur considère toujours la valeur ASCII (voir annexe F) du caractère.

```
#include <iostream.h>

main()
{
  char a,b,c;
  a=5;
  b=10;
  c=15;
  char d=a+b+c+35;
  cout << d << endl;
  return 0;
}
```

```
A
```

On déclare trois variables : a, b, c *de type caractère et on leur affecte trois valeurs entières. On déclare ensuite une variable à laquelle on affecte calcul* a+b+c+35, *soit* 65, *c'est-à-dire la valeur du code ASCII pour la lettre A.*

1.5 LES TYPES RÉELS

Il existe trois types de nombres réels en C++, `float`, `double` et `long double`. Le tableau 1.2 présente les plages couvertes sur un PC type Pentium 4, avec Microsoft Visual C++.

TABLEAU 1.2 PLAGES COUVERTES.

Type	Mini	Maxi	Taille (octets)
`float`	1.17549e-38	3.40282e+38	4
`double`	2.22507e-308	1.79769e+308	8
`long double`	2.22507e-308	1.79769e+308	8

```
#include <iostream.h>

main()
{
  double base,hauteur;
  base=12.525;
  hauteur=10.85;
  cout << "Surface = " << base*hauteur/2 << endl;
  return 0;
}
```

```
Surface = 67.9481
```

On déclare deux variables : base *et* hauteur*, de type double et on dirige vers le flux de sortie le résultat du calcul de la surface* base * hauteur/2 *qui sera évalué lui-même comme un type double.*

Les nombres réels sont souvent exprimés en notation scientifique. Dans cette notation le nombre est suivi de la lettre `e`, elle-même suivi d'un nombre qui représente l'exponentiation. Par exemple le nombre `2.25225e-8` représente 2.25225 x 10^{-8} soit 0,0000000225225.

Quand la valeur absolue du nombre est comprise entre 0.1 et 999 999, les nombres ne sont pas affichés au format scientifique.

La manipulation de nombres réels par le langage C++ peut amener des erreurs d'arrondi, il faut donc rester vigilant, certaines imprécisions pouvant apparaître.

```
#include <iostream.h>

main()
{
  double x=10000;
  double y=x/9-1111;
  double z;
  cout << "y=" << y << endl;
  cout << "z=" << y*9 << endl;
  return 0;
}
```

```
Z=0.111111
y=1
```

La valeur de la variable z *met ici en évidence l'erreur d'arrondi, en effet on s'attendrait plutôt à obtenir 0.999999, résultat du calcul 0.111111*9 (y*9).*

1.6 LES TYPES ÉNUMÉRATIONS

En langage C++, l'utilisateur peut créer son propre type de données. Il existe plusieurs méthodes pour atteindre ce but. Elles seront toutes décrites dans cet ouvrage que ce soit les *énumérations*, les *structures* ou les *classes.*

Nous allons commencer par les *énumérations*, les autres méthodes seront décrites dans les chapitres 6 et 7.

Un type *énumération* est constitué d'un ensemble fini de valeurs ou *énumérateurs*. Il peut être utilisé comme tous les autres types, il est simplement précédé du mot-clé `enum`. Sa syntaxe est la suivante :

```
enum nomtype {énumérateur1, énumérateur2, …, énumérateurn}
```

Un type `enum` peut être défini à l'intérieur d'une déclaration de variable.

Lorsque l'on définit un type `enum`, des valeurs entières de 0 à n sont automatiquement affectées à chaque *énumérateur*. On peut modifier ces valeurs et même affecter des valeurs identiques à des énumérateurs différents.

Une variable de type `enum` peut être initialisée.

L'exemple de code ci-dessous viendra éclaircir ces différentes notions.

```
#include <iostream.h>

enum couleur {cyan, rouge, magenta, vert, jaune,
bleu};
enum test {non=0, oui=1, faux=0, vrai=1};
enum polarite {moins=-1, plus=+1} polar1=moins,
polar2;
enum jour {lundi=1, mardi=2, mercredi=3, jeudi=4,
vendredi=5, samedi=6, dimanche=7};

main()
{
  couleur color=jaune;
  if (color==4) cout << "jaune" << endl;

  test reponse1=non, reponse2=oui;
  if(reponse1==0) cout << "c'est non et faux" <<
endl;
  if(reponse2==1) cout << "c'est oui et vrai" <<
endl;

  polarite polar3;
  polar3=plus;
  polar2=moins;
  if (polar3==polar1) cout << "identique" << endl;
    else cout << "oppose" << endl;
  if (polar2==polar1) cout << "identique" << endl;
    else cout << "oppose" << endl;
  if (polar1==-1) cout << "c'est moins" << endl;
```

```
 jour jsem=vendredi;
  if(jsem==5) cout << "vendredi = " << jsem <<endl;
  if(jsem>4) cout << "vendredi > 4 " <<endl;
  if(jsem<3) cout << "vendredi = " << jsem << endl;
    else cout << "vendredi n'est pas inferieur a 3"
<< endl;

  return 0;
}
```

```
jaune
c'est non et faux
c'est oui et vrai
oppose
identique
c'est moins
vendredi = 5
vendredi > 4
vendredi n'est pas inferieur a 3
```

Dans cet exemple quatre nouveaux types sont créés, couleur, test, polarite *et* jour.

Le type couleur *est défini normalement.*

Le type test *affecte des valeurs à chaque énumérateur. Les énumérateurs portent des valeurs identiques 0 et 1.*

Le type polarite *affecte de valeurs à chaque énumérateur, déclare deux variables,* polar1 *et* polar2 *et initialise* polar1.

Le type jour *réaffecte des valeurs entières différentes à chaque énumérateur.*

*Le reste du programme montre le fonctionnement et l'utilisation des types énumération définis au travers de différentes affectations et fonctions de test (*if*).*

L'instruction if *sera présentée au chapitre 2.*

1.7 LES CONSTANTES

Nous manipulons très souvent lors de l'écriture d'un programme des valeurs que nous affectons à des variables. Cependant il existe une possibilité d'affecter une valeur définitive à une variable qui par ce fait est dénommée constante.

C++ reconnaît plusieurs types de constantes, les constantes entières, réelles, caractères, chaîne de caractères et énumérations.

Ces constantes sont très pratiques pour manipuler des valeurs comme le nombre PI, le nombre e, etc.

Tous comme les variables, elles peuvent être signées ou non signées.

Une constante caractère est composée d'un caractère unique précisé entre apostrophes.

Une constante chaîne est formée d'un nombre quelconque de caractères encadrés par des guillemets (apostrophes doubles).

Le mot-clé qui permet de définir une constante est `const`.

```
#include <iostream.h>

main()
{
  const double PI=3.14;
  double rayon=10;
  cout << "Surface = " << PI*rayon*rayon << endl;
  cout << "Circonference = << 2*PI*rayon << endl;
  return 0;
}
```

```
Surface = 314
Circonference = 62.8
```

On déclare ici l'identificateur PI *auquel on affecte la valeur constante 3.14. Cet identificateur est ensuite utilisé lors des calculs de la surface et de la circonférence d'un cercle.*

Les développeurs utilisent habituellement des lettres majuscules comme identificateur d'une constante, afin de les différencier des autres.

Nous verrons, un peu plus loin, que langage C++ définit également une notion de constante symbolique.

1.8 LES OPÉRATEURS ARITHMÉTIQUES

Les opérateurs les plus courants sont utilisés dans l'arithmétique, ce sont : +, -, *, /, % pour la somme, la différence, le produit, le quotient et le reste de la division aussi appelé modulo. L'opérateur d'exponentiation n'existe pas en C++, mais il existe une fonction externe qui remplit ce rôle.

L'opérateur – peut être également utilisé pour la négation d'un nombre. On appelle ce type d'opérateur, un *opérateur unaire*. Nous en verrons deux autres un peu plus loin dans ce chapitre.

```
#include <iostream.h>

main()
{
  int a=32, b=5;
  cout << "a=" << a << "; b=" << b << endl;
  cout << "a+b=" << a+b << endl;
  cout << "a-b=" << a-b << endl;
  cout << "a*b=" << a*b << endl;
  cout << "a/b=" << a/b << endl;
  cout << "reste de a/b : " << a%b << endl;
  cout << "-b=" << -b << endl;
  return 0;
}
```

```
a=32 ; b=5
a+b =37
a-b=27
```

```
a*b=160
a/b=6
reste de a/b : 2
-b=-5
```

*Un exemple d'utilisation des opérateurs : +, -, *, / et % sur deux nombres entiers* a=32 *et* b=5.

Il faut faire attention aux problèmes d'arrondis qui peuvent survenir lorsque les opérandes d'une division entière mettent en jeu des nombres négatifs. Cependant, dans tous les cas, le quotient multiplié par le diviseur plus le reste donnera le dividende, c'est une règle absolue quels que soient la machine et le compilateur utilisé.

```
#include <iostream.h>

main()
{
  int dividende, diviseur, quotient, reste;
  dividende=-69;
  diviseur=10;
  quotient=dividende/diviseur;
  reste=dividende%diviseur;
  cout << "Le quotient de " << dividende << " par
" << diviseur << " est " << quotient << endl;
  cout << "et le reste "<< reste << endl;
  cout << "Pour verification (quotient*diviseur
+reste) = " << quotient*diviseur+reste << ", c'est
bien le dividende!" << endl;
  return 0;
}
```

```
Le quotient de -69 par 10 est -6
et le reste -9
Pour verification (quotient*diviseur+reste) = -69,
c'est bien le dividende !
```

On traite ici le quotient, 69/10 avec des variables de types entiers. Dans ce cas, suivant le type d'ordinateur et de compilateur, le quotient peut valoir –6 ou –7 suivant l'arrondi utilisé. Toutefois le reste sera –9 ou bien 1 afin de respecter dividende = quotient * diviseur + reste.

Dans certains cas des opérandes de types différents peuvent subir des conversions de type pour que le système puisse évaluer correctement le résultat et fournir la plus grande précision possible.

Si les deux opérandes sont de types `int` et `long int`, le résultat sera de type `long int`.

Si les deux opérandes sont de types `int` et `short int`, le résultat sera de type `int`.

Si l'un des opérandes est de type `float`, `double` ou `long double` et le second de type `int`, `short int`, `long int` ou `char`, le résultat est converti en une variable de même type que l'opérande qui est virgule flottante.

Si les deux opérandes sont en virgule flottante avec un type de précision différente, celui qui a la plus faible précision prend la précision de l'autre et le résultat est exprimé dans cette même précision.

```
#include <iostream.h>

main()
{
  int x=10000;
  short int y=2;
  long int z=50000;
  float m=2.5;
  long double n=2000000.000001;
  cout << "int * short int : " << x*y << endl;
  cout << "int * long int : " << x*z << endl;
  cout << "float * int : " << x*m << endl;
  cout << "float * long double : " << m*n << endl;
  return 0;
}
```

```
int * short int : 20000
int * long int : 500000000
float * int : 25000
float * long double : 5e+006
```

Les variables x, y, z, m *et* n *sont toutes de types différents. On voit que le compilateur applique les règles citées précédemment pour chacun des calculs.*

1.9 LES OPÉRATEURS UNAIRES

Nous avons déjà présenté un *opérateur unaire*, il s'agit du signe moins qui précède une valeur numérique, constante, variable ou expression. Il en existe d'autres, notamment l'opérateur d'incrémentation ++ et l'opérateur de décrémentation -- qui augmentent ou diminuent de 1 la valeur de leur opérande.

Ces deux opérateurs peuvent suivre ou précéder leur opérande, on parle alors de *post* ou de *pré-incrémentation* ou *décrémentation*.

Dans le cas d'un traitement *post*, l'opérande est modifié avant son affectation alors qu'il est modifié après lors d'un traitement *pré*. Dans de nombreux cas, la différence est fondamentale, il faut donc utiliser ces opérateurs avec précaution.

```
#include <iostream.h>

main()
{
  int x=1, y=1;
  cout << "++x = " << ++x << endl;
  cout << "x   = " << x << endl;
  cout << endl;
  cout << "y++ = " << y++ << endl;
  cout << "y   = " << y << endl;
  return 0;
}
```

```
++x = 2
x   = 2

y++ = 1
y   = 2
```

Les variables entières x, y *sont initialisées à 1 puis pré-incrémentées pour* x *et post-incrémentées pour* y. *On voit que* x *est déjà incrémenté puis placé dans le flux de sortie alors que* y *est placé avec sa valeur initiale dans le flux de sortie puis incrémenté.*

L'opérateur `sizeof` est également un opérateur *unaire* qui retourne la taille en octets de son opérande. C'est un opérateur peu employé, cependant il peut présenter une certaine utilité lorsque l'on recherche la taille des données, notamment dans le cadre de l'adaptation d'un programme à un nouveau compilateur.

1.10 L'OPÉRATEUR CONDITIONNEL

Il existe un opérateur conditionnel assez peu utilisé mais qui cependant permet de traiter des expressions simples. Cet opérateur remplace le classique `if then else` que nous étudierons au chapitre suivant. La forme de cette expression est :

`expression1 ? expression2 : expression3.`

– `expression1` constitue la condition ;
– `expression2` est évaluée si l'expression1 est vraie ;
– `expression3` est évaluée si l'expression1 est fausse.

1.11 LES OPÉRATEURS RELATIONNELS, DE COMPARAISON ET LOGIQUES

Le tableau 1.3 précise l'ensemble des opérateurs.

TABLEAU 1.3

Relationnel	<	inférieur à
	<=	inférieur ou égal à
	>	supérieur à
	>=	supérieur ou égal à
Relationnel de comparaison	==	identique à
	!=	différent de
Logique	&&	et
	||	ou
	!	non

Les opérateurs relationnels et de comparaison sont d'une grande importance comme dans la plupart des langages et sont donc très employés. Ils établissent une relation ou une comparaison entre les instructions qu'ils relient, généralement pour répondre à une condition.

```
#include <iostream.h>

main()
{
  int x=10, y=5, z=10;
  cout << "x<y -> " << ((x<y)?0:1) << endl;
  cout << "x>y -> " << ((x>y)?0:1) << endl;
  cout << "x<=z -> " << ((x<=z)?0:1) << endl;
  cout << "x==z -> " << ((x==z)?0:1) << endl;
  cout << "x!=z -> " << ((x!=z)?0:1) << endl;
  return 0;
}
```

```
x<y -> 1
x>y -> 0
x<=z -> 0
x==z -> 0
x!=z -> 1
```

Nous trouvons ici une illustration des opérateurs relationnels, de comparaison et conditionnel avec les différentes expressions évaluées.

Les opérateurs logiques sont le `OU`, le `ET` et le `NON` logique, ils suivent les tables de vérité classiques (cf. tableau 1.4 et 1.5).

TABLEAU 1.4 TABLES DE VÉRITÉ ET ET OU (AVEC 0 → FAUX ; 1 → VRAI).

		ET	OU
a	b	a&&b	allb
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

TABLEAU 1.5 TABLE DE VÉRITÉ NON (AVEC 0 → FAUX ; 1 → VRAI).

NON	
a	!a
1	0
0	1

Ces opérateurs agissent sur des opérandes qui sont eux-mêmes des expressions logiques et combinent ainsi des conditions élémentaires afin de définir des conditions composées qui seront vraies ou fausses.

```
#include <iostream.h>

main()
{
  int a=5, b=10, c=15, x, y, z;
  x=(a<b)&&(b<c)?"VRAI": « FAUX »;
  y=(a<b)||(a>c)?1:0;
  z=(a<b)&&(a<c)||(c<a)?1:0;
  cout << x << endl;
  cout << y << endl;
  cout << z << endl;
  return 0;
}
```

```
1
1
1
```

Ici, x est égal à 1 si (a<b) et (b<c), or a<b est VRAI (1) et (b<c) est VRAI (1) donc 1 ET 1 → VRAI (1).

y est égal à 1 car (a<b) est VRAI (1) et (a>c) est FAUX (0) donc 1 ou 0 → VRAI (1).

z est égal à 1 car (a<b) est VRAI (1), (a<c) est VRAI (1), c<a est FAUX (0) donc 1 ET 1 OU 0 → VRAI (1)

1.12 LA PRIORITÉ DES OPÉRATEURS

Les opérateurs du langage C++ possèdent une priorité qui devient très importante lors de la manipulation d'expressions arithmétiques complexes. L'annexe B présente l'ensemble des opérateurs avec leurs priorités.

Quand deux opérateurs de même niveau de priorité sont présents dans une expression, on tient compte de l'associativité de chacun qui détermine dans quel sens, de gauche à droite ou de droite à gauche, seront évalués les opérateurs.

```
#include <iostream.h>

main()
{
{
  int x=1 , y=5, z=10;
  x-=(y<z)&&(z>=20)?--y:z/-y;
  cout << "si y<z ET z>=20 alors x sera egal a
y=y-1 sinon a z/-y : " << x << endl;
  return 0;
}
```

```
si y<z ET z<=10 alors x sera egal a y=y-1 sinon a
z/-y : 3
```

Le calcul de x fait appel à l'évaluation d'une expression complexe qui prend en compte la priorité des opérateurs.

On évalue tout d'abord l'expression (y<z) && (z<=10)*, puis* --y *si la précédente est vraie, sinon* z/-y*, puis l'affectation* -=*.*

(1<10) ET (10>=20) → 1 ET 0 → FAUX

donc z/-y *est évalué*
10/-5=-2

et x *est calculé*
x=x-(-2) → x=1-(-2)=1+2=3

Une des autres caractéristiques liées à la priorité des opérateurs est l'*arité* qui indique si l'opérateur manipule un ou deux opérandes.

Chapitre **2**

Entrées et structures de contrôle

Opérateurs, mots-clés et fonctions

`break, cin, continue, do, else, for, goto, if, switch, while`

2.1 ENTRÉE

En C++, les données de l'entrée sont dirigées vers un flux d'entrée, de la même manière que pour l'instruction `cout`. Les données sont récupérées par l'instruction `cin` qui appartient, elle aussi, au fichier externe `iostream.h`.

```
  #include <iostream.h>

main()
{
  float M,m;
  cout << "Entrez un nombre M : ";
  cin >> M;
  cout << "Entrez un nombre m : ";
  cin >> m;
  cout << M << "x" << m << "=" << M*m << "\n";
  return 0;
}
```

```
Entrez un nombre M : 12
Entrez un nombre m : 5
12x5=60
```

Le programme demande l'entrée d'un nombre M *puis d'un nombre* m. *Il calcule ensuite le produit* M*m *et affiche le résultat.*

À l'aide de `cin` on peut aussi définir des entrées multiples qui seront lues de gauche à droite. La syntaxe est simple il suffit de mettre à la suite les entrées dans le flux.

```
cin << a << b << c ;
```

La ligne précédente attend l'entrée de trois variables `a`, `b` et `c`. Un espace, un retour chariot (touche entrée du clavier) ou une tabulation permettront de différencier les variables. Ces caractères sont en effet ignorés par `cin`.

2.2 LE TEST CONDITIONNEL

Le test conditionnel, opération indissociable de la plupart des langages de programmation est bien sûr disponible en langage C++. Elle est

construite suivant l'expression classique si… alors… sinon… qui offre le test d'une condition et qui répond suivant deux alternatives.

En C++, nous pouvons trouver la syntaxe simple `if (condition) instruction`, ou la syntaxe évoluée `if (condition) instruction1 else instruction2` si la seconde alternative du test est explicite.

L'interprétation d'un test est vraie lorsque la valeur de l'expression interprétée dans la condition est non nulle. Une valeur nulle entraîne un test faux. Les valeurs vrai et faux sont ici prises en compte au sens booléen du terme.

```
#include <iostream.h>

main()
{
  int x;
  cout << "Entrez un nombre entier entre -10 et
10 : ";
  cin >> x;
  if (x>0) cout << "le nombre est positif \n";
  return 0;
}
```

```
Entrez un nombre entier entre -10 et 10 : 5
le nombre est positif
```

```
Entrez un nombre entier entre -10 et 10 : -3
```

En fonction du nombre entré inférieur ou supérieur à 0, on obtient le message : le nombre est positif ou bien aucun message puisque le test ne comporte ici qu'une seule alternative.

Dans l'exemple précédent, aucun traitement n'est réalisé en cas d'entrée d'un nombre négatif. Nous pouvons pallier à ce problème par l'utilisation du test classique à deux alternatives (si… alors… sinon…).

```
#include <iostream.h>

main()
{
  int x;
  cout << "Entrez un nombre entier entre -10 et
10 : ";
  cin >> x;
  if (x>0) cout << "le nombre est positif" << endl;
  else cout << "le nombre est negatif ou nul" <<
endl;
  return 0;
```

```
Entrez un nombre entier entre -10 et 10 : 5
le nombre est positif
```

```
Entrez un nombre entier entre -10 et 10 : -3
le nombre est negatif ou nul
```

Quelle que soit la valeur saisie, on obtient toujours un message par l'utilisation de la deuxième alternative du test, située derrière l'instruction else.

Chacune des alternatives qui suivent la condition peut être constituée d'un ensemble d'instructions insérées entre des accolades : `{ }`.

```
#include <iostream.h>

main()
{
  char a,b,x;
  cout << "Tapez deux lettres minuscules de
l'alphabet : ";
  cin >> a >> b;
//Condition qui teste si le code ASCII de a est
supérieur à b
```

```
  if (a>b) {
    //Intervertit a et b par l'intermédiaire de x
    x=b;
    b=a;
    a=x;
    cout << "les lettres classees dans l'ordre
alphabetique sont : " << a << b << endl;
  }
  else
    cout << "les lettres classees dans l'ordre
alphabetique sont : " << a << b << endl;
  return 0;
}
```

```
Tapez deux lettres minuscules de l'alphabet :
w   a
les lettres classees dans l'ordre alphabetique
sont : aw
```

```
Tapez deux lettres minuscules de l'alphabet :
a   w
les lettres classees dans l'ordre alphabetique
sont : aw
```

Le programme demande deux lettres minuscules qui seront ensuite classées suivant l'ordre alphabétique croissant. Dans le cas où a *à un code ASCII supérieur à* b *(les variables représentant les deux lettres étant de type char), on échange* a *et* b *par l'intermédiaire d'une variable intermédiaire* x *sinon on ne change rien et on les affiche comme elles ont été saisies.*

Les instructions `if…else` peuvent être imbriquées les unes dans les autres suivant différentes formes.

```
if condition1 if condition2 résultat1
              else résultat2
```

```
else if condition3 résultat3
     else résultat4

if condition1 résultat1
else if condition2 résultat2

if condition1 résultat1
else if condition2 résultat2
     else résultat3

if condition1 if condition2 résultat1
              else résultat2

if condition1 if condition2 résultat1
              else résultat2
else résultat3

if condition1
     {
     if condition2 résultat1
     else résultat2
     }

if condition1
     {
     if condition2 résultat1
     }
else résultat2
```

Dans tous les cas et par convention en langage C++, `else` est toujours associé à l'instruction `if` la plus proche qui ne possède pas encore de `else`.

À l'aide de l'instruction conditionnelle `if`, des expressions booléennes peuvent être manipulées de façon simple.

```
#include <iostream.h>

main()
{
  int x=0, y=2, D=3, d=2;
  if (x) cout << "x est vrai";
    else cout << "x est faux ou nul" << endl;
  if (y) cout << "y est vrai ou non nul" << endl;
  if (D%d) cout << "Le quotient D/d n'est pas
entier" << endl;  return 0;
}
```

```
x est faux ou nul
y est vrai ou non nul
Le quotient D/d n'est pas entier
```

Les trois instructions conditionnelles présentes testent si la condition est vraie ou fausse. Une valeur égale à 0 est FAUSSE et une valeur non nulle est VRAIE.

2.3 LES BOUCLES

La plupart des langages de programmation possèdent des instructions dédiées à la réalisation d'itérations. L'itération est la répétition d'une ou plusieurs instructions. Le C++ dispose de trois instructions itératives : `while`, `do…while` et `for`.

2.3.1 L'instruction `while`

Sa forme suit la construction suivante :

```
while (condition) expression
```

La ou les instructions constituant l'expression sont répétées tant que la valeur de la condition n'est pas fausse (égale à 0).

```
#include <iostream.h>

main()
{
  int i=0, n, somme=0;
  cout << "Dernier nombre : ";
  cin >> n;
  while (i<=n)
  {
    somme+=i;
    ++i;
  }
  cout << "Somme : " << somme << endl;
  return 0;
}
```

```
Dernier nombre : 5
Somme : 15
```

La boucle while *calcule la somme des nombres entiers compris entre* 0 *et* n *inclus. Tant que* i *est inférieur ou égal à* n*, la boucle se poursuit sinon la valeur de somme s'affiche.*

2.3.2 L'instruction `do…while`

La différence fondamentale de cette instruction avec la précédente réside dans la position du test. Dans l'instruction `do…while` le test s'effectue en fin d'exécution de l'expression.

Sa forme est la suivante :

```
do expression while (condition)
```

La ou les instructions constituant l'expression sont exécutées de façon répétitive tant que la valeur de la condition n'est pas fausse (égale à 0). Quoi qu'il en soit, l'expression est toujours évaluée au moins une fois puisque la condition n'est prise en compte qu'après une première exécution de ou des instructions.

```
#include <iostream.h>

main()
{
  int i=1, n, produit=1;
  cout << "Valeur finale : ";
  cin >> n;
  do
  {
    produit*=i;
    ++i;
  }
  while (i<n+1);
  cout << "Produit : " << produit << endl;
  return 0;
}
```

```
Valeur finale : 10
Produit : 3628800
```

Mise en place d'une boucle do...while *pour calculer le produit des* 1 *à* n *(valeur finale) premiers nombres.*

2.3.3 L'instruction `for`

L'instruction `for` est l'une des plus utilisées pour la réalisation de boucle d'instructions. Elle est composée de plusieurs éléments. Une expression qui gère la valeur initiale d'un index (initialisation), une expression qui teste l'index (condition) et enfin une expression qui modifie l'index. Sa forme est la suivante :

```
for (expression1 ; expression2 ; expression3) expression4
```

L'initialisation (`expression1`) correspond généralement à une affectation et la modification d'index (`expression3`) à une expression *unaire*.

Lors de l'exécution de l'instruction `for`, la condition (`expression2`) est évaluée et testée avant chaque parcours de ou des instructions qui composent `expression4`. Cette dernière est définie comme la boucle d'instructions.

```
#include <iostream.h>

main()
{
  int i, n, somme=0;
  cout << "Valeur finale : ";
  cin >> n;
  for (i=0; i<=n; ++i)
  {
    somme+=i;
  }
  cout << "Somme : " << somme << endl;
  return 0;
}
```

```
Valeur finale : 10
Somme : 5
```

Ce programme calcule, à l'aide d'une boucle for*, la somme des* 0 *à* n *(valeur finale) premiers nombres.*

Une boucle `for` peut utiliser plusieurs index initialisés différemment qui peuvent être aussi modifiés de façon individuelle.

```
#include <iostream.h>

main()
{
  int i, j;
  for (i=0, j=0; i<=100; i++, j-=5)
  {
    cout << "i = " << i << " / j = " << j << endl;
  }
  return 0;
}
```

```
i = 0 / j = 0
i = 1 / j = -5
i = 2 / j = -10
   .
   .
   .
i = 100 / j = -500
```

La boucle traite deux index i *et* j. *Le premier compte de 1 à 100 avec un incrément de 1 et le second de 5 en 500 avec un incrément égal à -5. La condition d'arrêt repose sur la valeur de* i *qui doit être inférieure ou égale à 100.*

Lors de l'écriture d'un programme en langage C++, les boucles peuvent être imbriquées les unes dans les autres avec des structures de contrôles différentes. Attention cependant, les chevauchements sont interdits et les index de chacune des boucles doivent être différents.

```
#include <iostream.h>

main()
{
  int n,exp,i,j,puiss=1;
  for (i=1;i<=4;++i){
    cout << "Entrer le nombre : ";
    cin >> n;
    cout << "Entrer l'exposant : ";
    cin >> exp;
    for (j=1;j<=exp;++j)
        puiss*=n;
    cout << n <<" exposant " << exp << " = " <<
puiss << endl;
    puiss=1;
  }
return 0;
}
```

```
Entrer le nombre : 2
Entrer l'exposant : 8
2 exposant 8 = 256
Entrer le nombre : 4
Entrer l'exposant : 2
4 exposant 2 = 16
Entrer le nombre : 5
Entrer l'exposant : 3
5 exposant 3 = 125
Entrer le nombre : 15
Entrer l'exposant : 4
15 exposant 4 = 50625
```

Deux boucles i *et* j *imbriquées. La première demande à l'utilisateur une valeur* n *et un exposant* exp*, la seconde exécute le calcul* n^{exp}*, puis la première affiche le résultat. Cette opération est répétée quatre fois. La boucle* j *est imbriquée dans la boucle* i.

2.4 LES INSTRUCTIONS BREAK ET CONTINUE

Dans certains cas il est utile de pouvoir mettre fin à une boucle, c'est le rôle de l'instruction `break` que l'on peut placer dans les boucles de type `while`, `do…while` et `for`.

Lors de son exécution la boucle est véritablement rompue et l'itération en cours se termine à cet endroit précis.

```
#include <iostream.h>
main()
{
  int n,exp,i,puiss=1;
  while (1){
   cout << "Entrer le nombre (0 pour terminer) : ";
    cin >> n;
    if (n==0) break;
```

```
    cout << "Entrer l'exposant : ";
    cin >> exp;
    for (i=1;i<=exp;++i)
         puiss*=n;
    cout << n <<" exposant " << exp << " = " <<
puiss << endl;
    puiss=1;
  }
return 0;
}
```

```
Entrer le nombre (0 pour terminer) : 2
Entrer l'exposant : 8
2 exposant 8 = 256
Entrer le nombre (0 pour terminer) : 5
Entrer l'exposant : 5
2 exposant 8 = 3125
Entrer le nombre (0 pour terminer) : 0
```

Tant que la boucle while *est vraie, le programme demande une valeur* n*, un exposant* exp*, exécute le calcul* n^{exp} *et affiche le résultat.*

Si la valeur saisie pour n *est égale à 0 le programme se termine.*

L'instruction `continue` arrête elle aussi l'exécution du bloc d'instructions en cours mais, à la différence de `break` ne rompt pas la boucle. Elle permet la reprise de cette dernière à l'itération suivante.

```
#include <iostream.h>
#include <math.h>

main()
{
  int n,puiss=1;
```

```
  while (1){
    cout << "Entrer le nombre (0 pour terminer): ";
    cin >> n;
    if (n==0) break;
    if (n<0){
      cout << "Erreur, nombre negatif!" << endl;
      continue;
    }
    cout << "Racine carree de " << n << " = " <<
sqrt(n) << endl;
  }
return 0;
}
```

```
Entrer le nombre (0 pour terminer):: 56
Racine carree de 56 = 7.48331
Entrer le nombre (0 pour terminer):: -34
Erreur, nombre negatif!
Entrer le nombre (0 pour terminer):: 16
Racine carree de 16 = 4
Entrer le nombre (0 pour terminer):: 0
```

Si le nombre n *saisi par l'utilisateur est négatif, le programme affiche un message d'erreur et ignore les instructions d'affichage pour retourner au début de la boucle* while*. Si l'entrée de l'utilisateur est égale à 0, le programme se termine.*

2.5 L'INSTRUCTION `SWITCH`

Cette instruction autorise le choix d'un groupe d'instructions parmi d'autres. La sélection est déterminée par l'évaluation d'une expression liée à l'instruction `switch`.

Sa forme est la suivante :

```
switch (expression) {
  case constante1 : instructions1 ;
```

```
    case constante2 : instructions2 ;
                      .
                      .
                      .
    case constanteN : instructionsN ;
    default : instructionsDefault ;
}
```

Derrière l'instruction `switch`, `expression` représente un caractère ou un entier. Celui-ci est ensuite comparé à chacune des constantes de cas déterminées par l'instruction `case`. En cas d'égalité la suite d'instructions correspondantes est exécutée sinon c'est le bloc d'instructions définies derrière `default` qui est pris en compte.

Chaque bloc d'instructions doit se terminer par l'instruction `break` déjà rencontrée plus haut, afin de marquer la fin du bloc.

```
#include <iostream.h>

main()
{
  double n1, n2, r;
  char operation;
  while (1){
    cout << "Entrer le nombre 1 (0 pour terminer)
: ";
    cin >> n1;
    if (n1==0) break;
    cout << "Entrer le nombre 2 : ";
    cin >> n2;
    cout << "Operation A/S/M/D : ";
    cin >> operation;
    switch (operation){
      case 'A':
        r=n1+n2;
        cout << "Resultat : " << r << endl;
        break;
      case 'S':
```

```
 r=n1-n2;
        cout << "Resultat : " << r << endl;
        break;
      case 'M':
        r=n1*n2;
        cout << "Resultat : " << r << endl;
        break;
      case 'D':
        r=n1/n2;
        cout << "Resultat : " << r << endl;
        break;
      default:
       cout << "Operation mal definie, choisissez
A/D/M/D/F" << endl;
        break;
    }
  }
  return 0;
}
```

```
Entrer le nombre 1 (0 pour terminer) : 4
Entrer le nombre 2 : 45
Operation A/S/M/D : D
Resultat : 0.0888889
Entrer le nombre 1 (0 pour terminer) : 45.89
Entrer le nombre 2 : 12.22
Operation A/S/M/D : M
Resultat : 560.776
Entrer le nombre 1 (0 pour terminer) : 23
Entrer le nombre 2 : 3
Operation A/S/M/D : H
Operation mal definie, choisissez A/D/M/D
Entrer le nombre 1 (0 pour terminer) : 16
Entrer le nombre 2 : 16.5
Operation A/S/M/D : A
Resultat : 32.5
Entrer le nombre 1 (0 pour terminer): 0
```

Le programme demande à l'utilisateur deux nombres n1 *et* n2*, puis une opération* A, S, M, D *ou* F *(Addition, Soustraction, Multiplication, Division ou Fin). Dans le cas où l'opération existe, l'instruction* switch *dirige les 2 nombres saisis vers le calcul puis l'affichage du résultat.*

Si l'opération n'existe pas (erreur de saisie), l'instruction switch *choisit le cas default et affiche un message d'erreur.*

Si l'utilisateur donne 0 comme premier nombre n1*, le programme se termine.*

Le cas `default` est optionnel. S'il n'est pas précisé aucun des traitements présents dans les blocs d'instructions n'est réalisé.

```
#include <iostream.h>

main()
{
  double n1, n2, r;
  char operation;
  while (1){
    cout << "Entrer le nombre 1 (0 pour terminer)
: ";
    cin >> n1;
    if (n1==0) break;
    cout << "Entrer le nombre 2 : ";
    cin >> n2;
    cout << "Operation A/S/M/D : ";
    cin >> operation;
    switch (operation){
      case 'A':
        r=n1+n2;
        cout << "Resultat : " << r << endl;
        break;
      case 'S':
        r=n1-n2;
```

```
        cout << "Resultat : " << r << endl;
        break;
      case 'M':
        r=n1*n2;
        cout << "Resultat : " << r << endl;
        break;
      case 'D':
        r=n1/n2;
        cout << "Resultat : " << r << endl;
        break;
    }
  return 0;
}
```

```
Entrer le nombre 1 (0 pour terminer) : 4
Entrer le nombre 2 : 45
Operation A/S/M/D : A
Resultat : 49
Entrer le nombre 1 (0 pour terminer) : 23
Entrer le nombre 2 : -34.5
Operation A/S/M/D : S
Resultat : 57.5
Entrer le nombre 1 (0 pour terminer) : 50
Entrer le nombre 2 : 5.25
Operation A/S/M/D : E
Entrer le nombre 1 (0 pour terminer) : 50
Entrer le nombre 2 : -5.25
Operation A/S/M/D : D
Resultat : -9.52381
Entrer le nombre 1 (0 pour terminer): 0
```

Le programme demande à l'utilisateur deux nombres n1 *et* n2*, puis une opération* A, S, M, D *ou* F *(Addition, Soustraction, Multiplication, Division ou Fin). Dans le cas où l'opération existe, l'instruction* switch *dirige les 2 nombres saisis vers le calcul puis l'affichage du résultat.*

Si l'opération n'existe pas (erreur de saisie), l'instruction switch *ignore la demande et n'effectue aucun calcul.*

Si l'utilisateur donne 0 comme premier nombre n1*, le programme se termine.*

2.6 L'INSTRUCTION GOTO

Les instructions `break`, `continue` et `switch` provoquent des sauts à des endroits déterminés dans l'ensemble des instructions qui composent le programme. L'instruction `goto` va elle aussi générer un saut mais, dans ce cas, la destination est précisée par une *étiquette*. Cette instruction peut permettre de bouleverser la logique d'exécution d'un programme.

L'*étiquette* qui suit `goto` est un identificateur suivi d'un point-virgule. La destination ou cible du saut est cette même *étiquette* suivie de deux-points.

La syntaxe est donc la suivante :

```
goto etiquette ;
     .
     .
     .
etiquette : instruction
```

L'étiquette doit être unique dans le programme (ou au sein de la fonction, voir plus loin dans cet ouvrage).

```
#include <iostream.h>
#include <math.h>

main()
{
  double n, r;
  saisie:
 cout << "Entrer un nombre reel (0 pour terminer) :
";
```

```
  cin >> n;
  if (n==0) goto fin3;
  if (n<0) goto fin1;
  else {
    r=sqrt(n);
    cout << "Racine carree de " << n << " = " << r
<< endl;
    goto fin2;
  }
  fin1:
  cout << "Nombre negatif - Erreur!" << endl;
  goto saisie;
  fin2:
  goto saisie;
  fin3:
  cout << "Fin de traitement" << endl;
  return 0;
}
```

```
Entrer un nombre reel (0 pour terminer) : 123
Racine carree de 123 = 11.0905
Entrer un nombre reel (0 pour terminer) : -4
Nombre negatif - Erreur!
Entrer un nombre reel (0 pour terminer) : 12
Racine carree de 123 = 3.4641
Entrer un nombre reel (0 pour terminer) : 0
Fin de traitement
```

On trouve dans ce programme trois étiquettes : fin1, fin2, fin3 *et* saisie.

Le programme va vers fin1 *si l'utilisateur entre un nombre négatif, vers* fin2 *si le nombre est positif, vers* fin3 *si le nombre est égal à 0 (pour terminer le programme).*

En cas d'accès à fin1 *ou* fin2 *le programme retourne à l'étiquette* saisie *qui redemande un nombre à l'utilisateur.*

L'utilisation de l'instruction `goto` est sujette à de nombreuses controverses dans la communauté des développeurs par le fait qu'elle peut mettre rapidement en cause l'intégrité de la logique d'un programme. Je vous conseille donc de l'utiliser avec parcimonie, celle-ci pouvant toujours être remplacée.

Chapitre **3**

Les fonctions

Opérateurs, mots-clés et fonctions

`cos, inline, pow, sin, sqrt,`
`strdate, strlen, strtime, void`

3.1 INTRODUCTION

En C++, l'utilisation de bibliothèques de fonctions est indispensable pour de nombreuses opérations. L'accès à ces bibliothèques se fait par l'intermédiaire de fichiers d'en-tête (voir chapitre 1) comme `iostream.h` que nous avons déjà utilisé.

Il en existe d'autres comme `math.h` pour les fonctions mathématiques, `time.h` pour les fonctions de gestion du temps…
Ces bibliothèques peuvent varier en fonction des compilateurs pour lesquels les éditeurs fournissent des environnements de programmation différents.

Il existe toutefois une norme qui impose aux éditeurs de fournir une bibliothèque nommée : bibliothèque C++ standard, comportant

50 fichiers d'en-tête. Ces 50 fichiers sont formés des 18 fichiers d'entête de la bibliothèque C standard plus 32 autres fichiers appelés en-têtes STL (*Standard Template Library*).

L'annexe H fournit une liste de l'ensemble des 50 fichiers retenus par la norme ISO (*International Organization for Standardization*).

```
#include <iostream.h>
#include <time.h>
#include <math.h>
#include <string.h>

const double PI=3.14;

main()
{
  char chaine[9];
  cout << "cos(PI/4) = " << cos(PI/4) << endl;
  cout << "sin(PI/2) = " <<sin(PI/2) << endl;
  cout << "2 a la puissance 8 : " << pow(2,8) <<
endl;
  cout << "Racine carree de 225 : " << sqrt(225) <<
endl;
  cout << "Longueur du mot bonjour : " << strlen
("bonjour") << endl;
  cout << "date (mm/jj:aa) : " << _strdate(chaine)
<< endl;
  cout << "heure (hh:mm:ss) : " << _strtime(chaine)
<< endl;
  return 0;
}
```

```
cos(PI/4) = 0.707388
sin(PI/2) = 1
2 a la puissance 8 : 256
Racine carree de 225 : 15
Longueur du mot bonjour : 7
```

```
date (mm/jj:aa) : 01/06/04
heure (hh:mm:ss) : 06:12:35
```

*Utilisation des fichiers d'en-tête de la bibliothèque C++ standard pour des fonctions mathématiques (*math.h*), de gestion de chaînes de caractères (*string.h*) et de temps (*time.h*).*

3.2 LES FONCTIONS PERSONNALISÉES

En marge des fichiers d'en-têtes et des fonctions qui les constituent, le langage C++ autorise la création et l'utilisation de fonctions personnalisées.

Elles vont permettre au programmeur de décomposer un programme important en un certain nombre de sous-ensembles plus petits permettant ainsi une modularité du code.

Les fonctions peuvent être compilées et testées séparément pour offrir une mise au point plus rapide et un *débogage* plus efficace dans la création et la mise au point d'un programme complexe.

La lisibilité du programme s'en trouvera également améliorée et sa portabilité s'en trouvera favorisée du fait de l'isolement des fonctionnalités spécifiques.


```
#include <iostream.h>

double surf(int x, int y, int z)
{
  return ((x+y)*z)/2;
}

main()
{
  int b, B, h;
  cout << "Petite base : ";
```

```
 cin >> b;
  cout << "Grande base : ";
  cin >> B;
  cout << "Hauteur : ";
  cin >> h;
  cout << "Surface du trapeze : " << surf(b,B,h) <<
endl;  return 0;
}
```

```
Petite base : 2
Grande base : 4
Hauteur : 3
Surface du trapeze : 9
```

Le programme principal main comporte ici l'appel d'une fonction surf*, qui porte trois arguments, permettant de calculer la surface d'un trapèze.*

Une fonction doit être composée d'un ensemble d'instructions qui vont former un sous-ensemble du programme destiné à assurer une tâche parfaitement définie.

Un programme peut être constitué de plusieurs fonctions et il en possède au moins une, c'est celle que nous avons dénommée `main` et que nous utilisons dans chaque programme depuis le début de cet ouvrage.

Quand un programme contient plusieurs fonctions, leur ordre d'écriture est indifférent.

Des fonctions ne peuvent jamais être imbriquées les unes dans les autres ou se chevaucher.

L'utilisation d'une fonction s'effectue par un appel qui peut être répété plusieurs fois au cours de l'exécution du programme.

Une fonction a pour objectif de traiter les informations qui lui sont transmises et de générer en retour un ou plusieurs résultats.

Une fonction créée possède deux parties qui sont l'*en-tête* et le *corps*. Dans l'en-tête on précise quel est le nom du résultat qui sera retourné, son type et ses arguments (paramètres). Le corps est le bloc

d'instructions qui suit l'en-tête, c'est le code même de la fonction, il se termine par `return` qui renvoie à l'emplacement de son appel.

Les arguments (paramètres) employés dans la fonction sont dénommés *arguments formels* (*paramètres formels*).

La ou les fonctions doivent être écrites avant le programme principal (fonction principale) `main` sauf si l'on utilise le *prototypage* (voir plus bas dans ce chapitre).

La syntaxe générale est la suivante :

```
type nomfonction (argumentformel1, argumentformel2,…,
argumrentformeln)
{
    .
    .
    .
    return expression ;
}

main()
{
    .
    .
    .
    nomfonction (argumenteffectif1, argumenteffectif2,…,
argumenteffectifn)
    .
    .
    .
}
```

La transmission des informations à la fonction se fait au moyen des *arguments formels* (ou *paramètres formels*) sous forme d'identificateurs spécifiques. Le retour du résultat est assuré par l'instruction `return`.

Les *arguments formels* établissent le lien entre la partie appelante du programme et la fonction. L'instruction `return` rend le contrôle à la fonction principale `main` du programme. L'expression qui la suit fait en général référence à la donnée renvoyée au programme appelant. Cette expression est facultative dans certains cas de figure et seul `return` peut

être spécifié. Le programme appelant reprend alors son cours d'exécution du fait de `return` qui lui redonne le contrôle.

Dans le cas où l'instruction `return` est seule, il faut noter qu'elle n'est pas obligatoire mais reste conseillée pour assurer une bonne traçabilité du code.

L'appel de la fonction est effectué en spécifiant son nom, suivi, entre parenthèses de la liste des arguments (ou paramètres) qui lui sont nécessaires. Ces arguments sont souvent appelés *arguments effectifs* (*paramètres effectifs*) ou *arguments réels* (*paramètres réels*).

Les *arguments formels* peuvent êtres des constantes, des variables ou des expressions. Ils mentionnent le type de ces dernières.

```
#include <iostream.h>

const double PI=3.14159;

double surf(const double pi, double x)
{
  double z;
  z=x*x*pi;
  return z;
}

main()
{
  double r;
  cout << "rayon : ";
  cin >> r;
  cout << "Surface du cercle : " << surf(PI, r) <<
endl;  return 0;
}
```

```
rayon : 10
Surface du cercle : 314.159
```

*On met ici en évidence la possibilité de passation d'arguments formels différents (*const, double*) à une fonction.*

Les arguments effectifs ou réels sont PI *et* r*, on les trouve derrière l'appel de la fonction dans* main.

La valeur z *est l'expression retournée par la fonction* surf *à* main.

Une instruction `return` ne peut contenir qu'une expression unitaire et ne renvoyer de ce fait qu'une seule valeur. Cependant une définition de fonction peut contenir plusieurs `return` avec des expressions distinctes dans le cas où des alternatives au sein même de la fonction sont nécessaires.

```
#include <iostream.h>
double ttc(double prix, int codeTVA)
{
  if (codeTVA==1) return prix*1.186 ;
  if (codeTVA==2) return prix*1.055 ;
  else return prix*1.3333 ;
}

main()
{
  int tva;
  double ht;
  cout << "Entrer un prix : ";
  cin >> ht;
  cout << "TVA 1, 2 ou 3 : ";
  cin >> tva;
  cout << "Prix TTC : " << ttc(ht,tva) << endl;
  return 0;
}
```

```
Entrer un prix : 100
TVA 1, 2 ou 3 : 2
Prix TTC : 105.5
```

On voit ici que la fonction ttc *possède plus d'une instruction* return. *Cette instruction apporte une sortie de la fonction par rapport au taux de TVA choisi.*

Quand une fonction renvoie un entier ou un caractère le type peut être omis.

```
#include <iostream.h>
puissance (int n, int e)
{
  int x=1;
  for (int i=1; i<=e;i++) x*=n;
  return x;
}

main()
{
  int nbr, exp;
  cout << "valeur entiere : ";
  cin >> nbr;
  cout << "exposant : ";
  cin >> exp;
  cout << nbr <<" puissance " << exp << " = " <<
puissance(nbr, exp) << endl;
  return 0;
}
```

```
valeur entiere : 2
Exposant : 16
2 puissance 16 = 65536
```

La fonction puissance *est présente mais son type n'a pas été spécifié, la valeur retournée étant entière.*

Si une fonction ne comporte pas d'argument, elle est suivie d'une paire de parenthèses.

```
#include <iostream.h>
int fact()
{
  int n;
  cout << "valeur entiere : ";
  cin >> n;
  int x=1;
  for (int i=1; i<=n;i++) x*=i;
  return x;
}

main()
{
  cout << fact() << endl;
  return 0;
}
```

```
valeur entiere : 8
40320
```

Dans ce programme, c'est la fonction fact *qui demande la valeur à l'utilisateur, qui calcule la factorielle et qui retourne le résultat, la fonction et son appel n'ont donc besoin d'aucun paramètre.*

3.3 FONCTION RÉCURSIVE

Une fonction peut s'appeler elle-même, c'est le principe de la *récursivité* ou *récursion*.
Ce type de programmation autorise des calculs répétitifs jusqu'à ce qu'une condition soit vérifiée.

L'écriture d'un *programme récursif* et son débogage sont souvent complexes et demandent au programmeur une certaine habitude ainsi qu'un raisonnement particulier. Toutefois de nombreux problèmes trouvent leur solution dans ce moyen qui, de plus, réduit souvent la taille du programme par rapport à son équivalent itératif.

```
#include <iostream.h>

int fact(int x)
{
  if (x==1)
    return x;
  else
    x=(x*fact(x-1));
    return x;
}

main()
{
  int n;
  cout << "valeur entiere : ";
  cin >> n;
  cout << n << " != " << fact(n) << endl;
  return 0;
}
```

```
valeur entiere : 8
8 != 40320
```

On peut remarquer dans ce programme que la fonction fact *s'appelle elle-même et met en œuvre le principe de la récursivité. Dans ce cas, les appels sont placés dans une pile jusqu'à la condition d'arrêt, ici* x==1, *puis exécutés en ordre inverse et retirés de cette même pile.*

3.4 LA FONCTION `VOID`

Une fonction qui ne retourne aucune valeur utilise le mot-clé `void` à la place de la spécification du type.

```
#include <iostream.h>
const double PI=3.14159;
```

```
void volume(double R)
{
  double v;
  v=4.0/3*PI*R*R*R;
  cout << "Volume = " << v << endl;
}

main()
{
  double rayon;
  cout << "Rayon : ";
  cin >> rayon;
  volume (rayon);
  return 0;
}
```

```
Rayon : 2
Volume = 33.5103
```

La fonction volume *ne renvoie aucune valeur. Elle est donc définie à l'aide du mot-clé* void.

On peut remarquer dans le calcul de v, *à l'intérieur de la fonction, la valeur 4/3 qui est écrite* 4.0/3, *ceci afin d'obtenir une valeur décimale (1.33…) pour le quotient, sinon il serait entier.*

3.5 DÉCLARATION, DÉFINITION ET PROTOTYPAGE DE FONCTIONS

La plupart des programmeurs donnent l'en-tête de la fonction avant le programme principal (`main`) et sa définition complète après.

En adoptant cette méthode, les fonctions peuvent être placées n'importe où dans le programme.

Cette écriture de fonctions porte le nom de *prototypage*. La déclaration de la fonction est le *prototype* et la fonction complète, en-tête plus corps est la *définition* de la fonction.

La déclaration d'une fonction est identique à la déclaration d'une variable, elle contient comme argument (paramètres) les types des différentes variables qui seront utilisées.

Sa syntaxe est la suivante :

```
type nomfonction (argument1, argument2,…, argumrentn)
```

```
#include <iostream.h>

void bougering(int, char, char, char);

main()

{
  int nbdisk;
  cout << "TOUR DE HANOI" << endl;
  cout << "Combien de disques : ";
  cin >> nbdisk;
  cout << "Les piliers sont numerotes de 1, 2, 3 de
la gauche vers la droite" << endl;
  cout << "Les disques sont au depart du jeu sur le
pilier 1 (a gauche)" << endl;
  bougering(nbdisk, '1', '3', '2');
  return 0;

}

void bougering (int n, char origine, char destina-
tion, char attente)
{
  if (n>0) {
    bougering(n-1, origine, attente, destination);
    cout << "Passer le disque " << n << " du pilier
"<< origine << " au pilier " << destination << endl;
    bougering(n-1, attente, destination, origine);
  }
  return;
}
```

```
TOUR DE HANOI
Combien de disques : 3
Les piliers sont numerotes de 1, 2, 3 de la gauche
vers la droite
Les disques sont au depart du jeu sur le pilier 1
(a gauche)
Passer le disque 1 du pilier 1 au pilier 3
Passer le disque 2 du pilier 1 au pilier 2
Passer le disque 1 du pilier 3 au pilier 2
Passer le disque 3 du pilier 1 au pilier 3
Passer le disque 1 du pilier 2 au pilier 1
Passer le disque 2 du pilier 2 au pilier 3
Passer le disque 1 du pilier 1 au pilier 3
```

Ce programme est un exemple de programmation du jeu des « Tours de Hanoï ».

On voit qu'il utilise une fonction nommée bougering *déclarée comme prototype devant la fonction* main*.*

L'algorithme repose sur un appel récursif de cette fonction.

Dans une fonction les variables déclarées dans la liste des arguments de la fonction et celle déclarées dans son corps sont dites *locales*. Cette notion de localité vient du fait que ces variables ne sont accessibles qu'à l'intérieur même du bloc constituant la fonction.

3.6 PASSAGE PAR VALEURS ET PAR RÉFÉRENCE

Les fonctions que nous avons étudiées jusqu'à présent utilisaient des arguments qui étaient transmis dans la fonction par l'intermédiaire d'un processus nommé *passage par valeur.*

Lors de l'appel de la fonction on évalue l'argument et on l'affecte ensuite à l'argument présent dans l'en-tête de la fonction avant d'en commencer l'exécution. Par ce moyen des expressions peuvent être utilisées comme argument.

```
#include <iostream.h>

double volume(double surfBase, int hauteur)
{
  double volume;
  volume=surfBase*hauteur;
  return volume;
}

main()
{
  int R, H;
  cout << "Entrez le rayon du cercle : ";
  cin >> R;
  cout << "Entrez la hauteur du cylindre : ";
  cin >> H;
  cout << "Volume du cylindre : " << volume
(R*R*3.14, H) << endl;
  return 0 ;
}
```

```
Entrez le rayon du cercle : 15
Entrez la hauteur du cylindre : 5
Volume du cylindre : 3532.5
```

Lors de l'appel de la fonction volume*, dans le programme principal* main*, on peut voir que le premier argument est une expression* R*R*3.14 *qui sera évaluée avant d'être transmise à la fonction.*

Cette méthode de travail avec les fonctions est la plus courante toutefois dans quelques cas de figure on peut être amené à travailler autrement. On utilise alors le *passage par référence*.

En effet, le *passage par valeur* ne modifie en rien la valeur de l'argument que l'on passe à la fonction, même si celui-ci est manipulé dans le corps de la fonction. Lors de l'utilisation du *passage par référence*, l'argument peut prendre une nouvelle valeur au retour de la fonction.

Pour passer un argument par *référence*, il suffit d'ajouter un `&` (et commercial) derrière le type de l'argument ou devant sa variable.

Voici sa syntaxe :

```
type nomfonction (type argument1& variableargument,…)
```

ou

```
type nomfonction(type argument1 &variableargument1,…)
```

```
#include <iostream.h>

double volume(double surfBase, int& hauteur)
{
  double volume;
  hauteur=hauteur*2;
  volume=surfBase*hauteur;
  return volume;}

main()
{
  int R, H;
  cout << "Entrez le rayon du cercle : ";
  cin >> R;
  cout << "Entrez la hauteur du cylindre : ";
  cin >> H;
  cout << "Volume du cylindre : " << volume
(R*R*3.14, H) << endl;
  cout << "Hauteur du cylindre : " << H << endl;
  return 0 ;
}
```

```
Entrez le rayon du cercle : 15
Entrez la hauteur du cylindre : 5
Volume du cylindre : 7065
Hauteur du cylindre : 10
```

La hauteur du cylindre est ici passée par référence. Dans la fonction sa valeur est multipliée par 2. Cette nouvelle valeur est retournée à H *après l'exécution de la fonction. Dans le cas où le passage aurait été effectué par valeur,* H *serait resté égal à 5.*

Il existe un autre moyen d'effectuer le passage d'un argument, c'est le *passage par référence constante*.

Une autre des caractéristiques du *passage par référence* c'est que cette technique évite la duplication de l'argument lorsqu'il est passé à la fonction.

Le fait que cette duplication n'est pas lieu allège la mémoire et rend donc plus efficace le programme.

Pour utiliser le *passage par référence* tout en maintenant l'efficacité on peut utiliser le *passage par référence constante* qui consiste à faire précéder l'argument du mot-clé `const`.

```
#include <iostream.h>

double volume(double surfBase,const int& hauteur)
{
  double volume;
  volume=surfBase*hauteur;
  return volume;}

main()
{
  int R, H;
  cout << "Entrez le rayon du cercle : ";
  cin >> R;
  cout << "Entrez la hauteur du cylindre : ";
  cin >> H;
  cout << "Volume du cylindre : " << volume
(R*R*3.14, H) << endl;
  cout << "Hauteur du cylindre : " << H << endl;
  return 0;
}
```

```
Entrez le rayon du cercle : 15
Entrez la hauteur du cylindre : 5
Volume du cylindre : 3532.5
Hauteur du cylindre : 5
```

L'argument hauteur *est ici passé par référence constante. La fonction* volume *ne peut donc pas le modifier.*

3.7 SURCHARGE D'UNE FONCTION

Le langage C++ autorise l'utilisation d'un même nom pour plusieurs fonctions qui peuvent porter des arguments différents.

Afin que le compilateur puisse les différencier, la règle qui est appliquée consiste à leurs faire porter des arguments de types différents ou en nombres différents.

```
#include <iostream.h>

double volume(double surfBase, int hauteur1);

double volume(double surfBase, int hauteur2);

double volume(double surfBase);

main()
{
  int R, H1, H2;
  cout << "Entrez le rayon du cercle : ";
  cin >> R;
  cout << "Entrez la hauteur du cylindre N.1 : ";
  cin >> H1;
  cout << "Entrez la hauteur du cylindre N.2 : ";
  cin >> H2;
  cout << "Volume du cylindre N.1 : " << volume
(R*R*3.14, H1) << endl;
  cout << "Volume du cylindre N.2 : " << volume
(R*R*3.14, H2) << endl;
  cout << "Volume du cylindre N.3 de hauteur 20 :
" << volume(R*R*3.14) << endl;
  return 0;
}

double volume(double surfBase,int hauteur1)
{
```

```
  double volume;
  volume=surfBase*hauteur1;
  return volume;}

double volume(float surfBase,int hauteur2)
{
  double volume;
  volume=surfBase*hauteur2;
  return volume;}

double volume(double surfBase)
{
  double volume;
  volume=surfBase*20;
  return volume;
}
```

```
Entrez le rayon du cercle : 15
Entrez la hauteur du cylindre N.1 : 5
Entrez la hauteur du cylindre N.2 : 10
Volume du cylindre N.1 : 3532.5
Volume du cylindre N.2 : 7065
Volume du cylindre N.3 de hauteur 20 : 14130
```

Dans ce programme on a placé trois prototypes de la fonction volume *qui sont différenciés par des types d'arguments différents ou bien un nombre d'arguments différents. Le compilateur C++ exécute les trois calculs sans faillir grâce à la surcharge possible de la fonction* volume*.*

3.8 QUELQUES PRÉCISIONS

Comme nous l'avons déjà mentionné en début de ce chapitre, la fonction `main` est obligatoire dans un programme C++.

La plupart des compilateurs acceptent le fait que `main` soit déclarée comme une fonction `void` avec la syntaxe suivante :

```
void main()
```

Dans quelques cas exceptionnels on peut être amené à vouloir terminer l'exécution d'un programme à l'intérieur d'une fonction. Pour cela C++ dispose de la fonction `exit()`.

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

void puissance8(int);

main()
{
  int n;
  cout << "Entrer un nombre entier : ";
  cin >> n;
  puissance8(n);
  return 0;
}
void puissance8(int x)
{
  cout << x << " exposant 8 : " << pow(x,8) << endl;
  exit(0);
}
```

```
Entrer un nombre entier : 2
2 exposant 8 : 256
```

La fonction puissance8 *utilise la fonction* exit *pour terminer le programme sans retourner à la fonction main.*

Au début du programme les fichiers d'en-tête stdlib.h *et* math.h *sont inclus afin de disposer de la fonction* exit *et de la fonction* pow *(élévation à la puissance).*

L'utilisation de cette fonction nécessite de faire appel au fichier d'en-tête `stdlib.h` et donc de l'inclure dans le programme.

Un autre élément important dans la manipulation des fonctions, c'est la *portée* de chacun des éléments manipulés, fonctions et variables. La

portée définie leur limite d'utilisation dans le programme. Cette notion fait appel aux deux qualificatifs *local* et *global*.

Une fonction *globale* ou une variable *globale* sont valides et accessibles dans l'ensemble du fichier programme manipulé. Elles sont déclarées à l'extérieur de toute fonction.

Une fonction *locale* ou une variable *locale* est valide et accessible dans la fonction qui la contient. Elle est déclarée à l'intérieur du bloc constituant le corps de la fonction.

```
#include <iostream.h>

double inverse (int);

main()
{
  int n;
  double carreinverse(int);
  cout << "Entrer un nombre entier : ";
  cin >> n;
  cout << inverse(n) << endl;
  cout << carreinverse(n) << endl;
  return 0;
}

double inverse(int x)
{
  return 1.0/x;
}

double carreinverse(int y)
{
  return inverse(y)*inverse(y);
}
```

```
Entrer un nombre entier : 5
0.2
0.04
```

La fonction inverse *est de type globale puisqu'elle est en dehors de toute fonction, ce qui lui confère la possibilité d'être appelée par toutes les fonctions.*

La fonction carreinverse *est locale à la fonction* main *et ne peut donc être appelée que depuis celle-ci.*

La variable n *est locale et n'est utilisable que dans la fonction* main.

Cette notion de *portée* est très sensible et souvent source d'erreur. Il faut donc veiller tout particulièrement à sa bonne utilisation.

3.9 LA FONCTION `INLINE`

Quand le temps d'exécution est de première importance dans un programme, on peut faire appel à la fonction `inline`.

Quand le compilateur rencontre cette fonction, il incorpore immédiatement les instructions de la fonction qui suit, contrairement à un traitement normal qui passe par une procédure d'appel puis une procédure de retour.

Par ce biais on va donc faire une économie substantielle de temps machine au détriment cependant d'une consommation mémoire plus importante, puisqu'à chaque appel de la fonction les instructions correspondantes sont générées.

La syntaxe est la suivante :

```
inline type nomfonction(...)
```

```
#include <iostream.h>

inline double inverse(int);

main()
{
```

```
  int n;
  double carreinverse(int);
  cout << "Entrer un nombre entier : ";
  cin >> n;
  cout << inverse(n) << endl;
  return 0;
}

double inverse(int x)
{
  return 1.0/x;
}
```

```
Entrer un nombre entier : 5
0.2
```

La fonction inverse *est déclarée comme une fonction* inline.

L'utilisation de la fonction `inline` ne doit pas être systématique, son emploi doit rester attaché à des cas de programmation bien spécifiques.

Il faut aussi remarquer qu'il n'est pas possible de compiler séparément une fonction `inline` contrairement à une fonction classique.

Chapitre **4**

Les tableaux

4.1 PREMIÈRE APPROCHE

Pour traiter un nombre de données importantes, on utilise des listes d'éléments qui sont rangés sous la forme d'un *vecteur* ou d'une *matrice*. Dans la vie courante un *vecteur* de données est simplement représenté par une liste de valeurs et une *matrice* par un tableau à plusieurs entrées, lignes et colonnes.

En langage C++, un *vecteur* ou une *matrice* sont toujours considérés comme un tableau dont la dimension va varier en fonction du nombre d'entrées. Chaque élément du tableau va être repéré par un *indice*. Il y a autant de séries d'*indices* qu'il y a de dimensions dans le tableau.

Un tableau qui possède un indice est dit *unidimensionnel* (tableau 4.1), à partir de deux, il est *multidimensionnel* (tableau 4.2).

TABLEAU 4.1 UN TABLEAU DE DIMENSION 1 (VECTEUR OU LISTE).

indices	0	1	2	3	…	n
éléments	e[0]	e[1]	e[2]	e[3]	...	e[n]

TABLEAU 4.2 UN TABLEAU DE DIMENSION 2 (MATRICE OU TABLEAU À 2 ENTRÉES).

indices	0	1	2	3	…	n
0	e[0] [0]	e[1] [0]	e[2] [0]	e[3] [0]	…	e[n] [0]
1	e[0] [0]	e[1] [1]	e[2] [1]	e[3] [1]	…	e[n] [1]
2	e[0] [0]	e[1] [2]	e[2] [2]	e[3] [2]	…	e[n] [2]
3	e[0] [0]	e[1] [3]	e[2] [3]	e[3] [3]	…	e[n] [3]
…	…	…	…	…	…	...
m	e[0] [m]	e[1] [m]	e[2] [m]	e[3] [m]	…	e[n] [m]

Les *indices* viennent indexer la position des éléments. Cette numérotation des éléments est entière et commence toujours à 0. On traitera donc du *ième* élément d'un tableau.

4.2 DÉCLARATION D'UN TABLEAU

La déclaration d'un tableau en langage C++ est identique à celle d'une variable ordinaire sauf qu'elle doit être suivie de sa taille, c'est-à-dire de son nombre d'éléments ou *indice*, entre crochets.

La syntaxe est la suivante :

```
type nomtableau[nbélément]
```

```
#include <iostream.h>

main()
{
  int i;
  char alpha[5];
  for (i=0;i<=4;++i){
    cout << "Entrer un caractere : ";
    cin >> alpha[i];
  }
  for (i=0;i<=4;++i){
   cout << "Entree "<< i << "=" << alpha[i] <<endl;
  }
  return 0;
}
```

```
Entrer un caractere : A
Entrer un caractere : f
Entrer un caractere : H
Entrer un caractere : M
Entrer un caractere : t
Entree 0=A
Entree 1=f
Entree 2=H
Entree 3=M
Entree 4=t
```

Le tableau a pour nom alpha*, il est composé de cinq caractères qui seront placés dans les emplacements 0 à 4 d'une liste (vecteur).*

La seconde boucle for *vient relire les caractères saisis.*

Si le tableau est *multidimensionnel*, c'est-à-dire s'il comporte plus d'un *indice*, il faut une paire de crochets pour chacun.

La syntaxe est la suivante :

```
type nomtableau[nbélément1][ nbélément 2]...[ nbélémentn]
```

```
#include <iostream.h>

main()
{
  int i,j;
  int pt[3][2];
  for(i=0;i<3;i++){
    for(j=0;j<2;j++){
      cout << "Emplacement affecte
["<<i<<"]["<<j<<"] - Entrer le poids : ";
      cin >> pt[i][j];
      j++;
    cout<<"Emplacement affecte ["<<i<<"]["<<j<<"]
- Entrer la taille : ";
      cin >> pt[i][j];
    }
  }
  for(i=0;i<3;i++){
    for(j=0;j<2;j++){
   cout << "ligne "<<i<<"/colonne "<<j<<" - Poids
= "<<pt[i][j];
    j++;
    cout << " : ligne "<<i<<"/colonne "<<j<<" -
Taille = "<<pt[i][j]<<endl;
    }
  }
  return 0;
}
```

```
Emplacement affecte [0][0] - Entrer le poids : 45
Emplacement affecte [0][1] - Entrer la taille : 140
Emplacement affecte [1][0] - Entrer le poids : 65
Emplacement affecte [1][1] - Entrer la taille : 160
Emplacement affecte [2][0] - Entrer le poids : 85
Emplacement affecte [2][1] - Entrer la taille : 180
Ligne 0/colonne 0 - Poids = 45 : ligne 0/colonne1 -
Taille = 140
```

```
Ligne 1/colonne 0 - Poids = 65 : ligne 1/colonne1 -
Taille = 160
Ligne 2/colonne 0 - Poids = 85 : ligne 2/colonne1 -
Taille = 180
```

Le tableau pt *est initialisé avec deux indices de valeur 3 et 2, il est donc multidimensionnel ou bidimensionnel. On peut l'assimiler à une matrice constituée de 3 lignes et 2 colonnes. La boucle d'indice* i *incrémente les lignes [3] et la boucle d'indice* j*, les colonnes [2].*

Il faut tenir compte du fait que la valeur des *indices* commence à partir de zéro. C'est-à-dire qu'un tableau dimensionné pour une taille de 5 éléments comportera un *indice* maximal égal à 4. Ce point est important car en C++, le dépassement de l'*indice* maximal du tableau n'est pas vérifié et peut donc amener des résultats pour le moins bizarres. En effet, si l'*indice* supérieur du tableau est dépassé, le compilateur ira chercher en mémoire une valeur qui ne fera pas partie du tableau et qui possédera une valeur totalement imprévisible.

```
#include <iostream.h>

main()
{
  int i;
  int nbr[5];
  for (i=0;i<=4;i++){
    cout << "Entrer un entier : ";
    cin >> nbr[i];
  }
  for (i=0;i<=7;i++){
    cout << "Entree "<< i << "=" << nbr[i] <<endl;
  }
  return 0;
}
```

```
Entrer un entier : 2
Entrer un entier : 4
Entrer un entier : 16
Entrer un entier : 3
Entrer un entier : 345
Entree 0=2
Entree 1=4
Entree 2=16
Entree 3=3
Entree 4=345
Entree 5=1245120
Entree 6=4213225
Entree 7=3456671
```

Le tableau nbr *est rempli par cinq valeurs (de 0 à 4) saisies par l'utilisateur.*

Lors de sa relecture, la boucle i *appelle 8 valeurs (de 0 à 7) et effectue ainsi une erreur de débordement, ce qui conduit à l'affichage de valeurs totalement imprévisibles pour les entrées 5, 6 et 7 dont la saisie n'a pas été réalisée.*

Ces valeurs sont celles contenues dans la mémoire de la machine lors de la lecture.

Lors de la compilation, le compilateur ne précise aucune erreur car il ne vérifie pas les valeurs d'indice.

Un tableau peut être initialisé avec une liste d'éléments prédéfinis. Les affectations se font dans leur ordre d'apparition.

La syntaxe est la suivante :

```
type nomtableau[nbélément]={élément1, élément2,...,
élémentn}
```

ou

```
type nomtableau[]={élément1, élément2,...,élémentn}
```

Si la taille n'est pas précisée, c'est le nombre d'éléments de la liste d'initialisation qui la détermine.

```
#include <iostream.h>

main()
{
  int nbr[10]={12, 14, 5, 678, 89, 56, 1, 2, 89,
54};
  int somme=0;
  for (int i=0;i<=9;i++){
    cout << "Element [" << i << "] = " << nbr[i]<<
endl;
    somme+=nbr[i];
  }
  cout << "Somme = " << somme<< endl;
  return 0;
}
```

```
Element [0] = 12
Element [1] = 14
Element [2] = 5
Element [3] = 678
Element [4] = 89
Element [5] = 56
Element [6] = 1
Element [7] = 2
Element [8] = 89
Element [9] = 54
Somme = 1000
```

Le tableau nbr *possède une liste de dix éléments prédéfinis qui sont appelés et affichés par une boucle* for*. Leur somme est affectée à la variable somme qui est ensuite dirigée, après les dix traitements de la boucle, vers un flux de sortie.*

4.3 TABLEAU ET FONCTIONS

Il arrive souvent que l'on veuille effectuer le passage d'un tableau à une ou plusieurs fonctions.

Pour effectuer cette opération il suffit de préciser le type des éléments du tableau ainsi que son nom.

```
#include <iostream.h>

float moy(float []);

main()
{
  float note[10];
  for (int i=0;i<=9;++i){
    cout << "Note /20 : ";
    cin >> note[i];
  }
  cout << "Moyenne : " << moy(note) << endl;
  return 0;
}

float moy(float note[])
{
  float somme=0;
  for(int i=0;i<=9;++i){
    somme+=note[i];
  }
  return somme/10;
}
```

```
Note /20 : 15
Note /20 : 12
Note /20 : 5
Note /20 : 10
Note /20 : 5
Note /20 : 18
Note /20 : 12
Note /20 : 13.5
Note /20 : 6.5
Note /20 : 19
Moyenne : 11.6
```

On déclare ici une fonction prototype moy *qui travaille sur un tableau* note *contenant dix valeurs dont elle calcule la moyenne.*

Le fonctionnement du programme précédent met en évidence que le passage du tableau est effectué par valeur, cependant, ses éléments sont modifiés par la fonction comme avec un passage par référence. Cela est dû au fonctionnement même du compilateur.

4.4 QUELQUES EXEMPLES ET CAS PARTICULIERS

Vous trouverez dans ce paragraphe quelques exemples d'utilisation de fonctions avec différents types de données manipulées.

4.4.1 Tri de valeurs entières

Les algorithmes de tri sont classiques en programmation. Ils mettent en œuvre des mécanismes de manipulations de variables bien connus et sont bien adaptés à la programmation par l'intermédiaire de fonctions.


```
#include <iostream.h>

const int NBVAL=100;

void tri(int, double tab[]);
void affich(int, double tab[]);

main()
{
  int nbr;
  double tab[NBVAL];
  cout << "Nombre de valeurs a trier : ";
```

```
  cin >> nbr;
  for (int i=0;i<nbr;++i){
    cout << "valeur " << i+1 <<" : ";
    cin >> tab[i];
  }
  tri(nbr, tab);
  affich (nbr, tab);
  return 0;
}
void tri(int nb, double tab[])
{
  for (int i=0;i<=nb;++i)
    for (int j=i+1;j<nb;++j)
        if (tab[j]<tab[i]){
            double temp=tab[j];
            tab[j]=tab[i];
            tab[i]=temp;
        }
  return ;
}

void affich(int n, double tab[])
{
  for (int i=0;i<n;++i)
    cout << tab[i] << endl;
  return;
}
```

```
Nombre de valeurs a trier : 8
valeur 1 : 45
valeur 2 : 8.9
valeur 3 : -23.12
valeur 4 : 0
valeur 5 : 34
valeur 6 : 120
valeur 7 : 121
valeur 8 : -120
```

```
-120
-23.12
0
8.9
34
45
120
121
```

Ce programme trie des nombres réels.

La taille maximum du tableau tab *est fixée à 100. Elle est définie à l'aide d'une constante* NBVAL *déclarée en début de programme.*

Ce type de déclaration est classiquement utilisé par les développeurs C++.

Deux fonctions sont définies :
– une fonction tri*, pour le tri, qui intervertit les 2 valeurs* tab[j] *et* tab[i] *sélectionnées dans le tableau si la première est inférieure à la seconde ;*
– une fonction affich *qui s'occupe de l'affichage des données triées.*

Nota : Ce programme aurait pu s'écrire de manière plus simple mais cet exemple permet de montrer la manipulation d'un tableau au travers de deux fonctions.

4.4.2 Produit des éléments de deux tableaux bidimensionnels

De nombreux traitements de données amènent souvent le programmeur à manipuler plusieurs tableaux dont il va devoir croiser les données.

```
#include <iostream.h>
const int NBLIG=100;
const int NBCOL=100;
```

```
double tab3[NBLIG][NBCOL];

void lectab(int, int, double tab[NBLIG][NBCOL]);
void produit(int, int, double tab1[NBLIG][NBCOL],
double tab2[NBLIG][NBCOL]);
void affich(int, int, double tab3[NBLIG][NBCOL]);

main()
{
  cout<<"PRODUIT de 2 TABLEAUX"<<endl;
  int nbl,nbc;
  double tab1[NBLIG][NBCOL], tab2[NBLIG][NBCOL];
  cout << "Nombre de lignes : ";
  cin >> nbl;
  cout << "Nombre de colonnes : ";
  cin >> nbc;
  cout<<"Saisissez le premier tableau"<<endl;
  lectab(nbl, nbc, tab1);
  cout<<"Saisissez le second tableau"<<endl;
  lectab(nbl, nbc, tab2);
  produit(nbl, nbc, tab1, tab2);
  affich(nbl, nbc, tab3);
  return 0;
}

void lectab(int nl, int nc, double tab[NBLIG][NB-
COL])
{
  for (int i=0;i<nl;++i)
    for (int j=0;j<nc;++j){
    cout<<"ligne/colonne -> "<<i+1<<"/"<<j+1<<" :
";
    cin >> tab[i][j];
    }
  return;
}
```

```
void produit(int nl, int nc, double tab1[NBLIG][NB-
COL], double tab2[NBLIG][NBCOL])
{
  for (int i=0;i<nl;++i)
    for (int j=0;j<nc;++j)
        tab3[i][j]=tab1[i][j]*tab2[i][j];
  return ;
}

void affich(int nl, int nc, double tab3[NBLIG][NB-
COL])
{
  cout<<"Produit des 2 tableaux"<<endl;
  for (int i=0;i<nl;++i)
    for(int j=0;j<nc;++j)
    cout<<"ligne/colonne -> "<<i+1<<"/"<<j+1<<" =
"<<tab3[i][j]<<endl;
  return;
}
```

```
PRODUIT de 2 TABLEAUX
Nombre de lignes : 2
Nombre de colonnes : 3
Saisissez le premier tableau
ligne/colonne -> 1/1 : 1.5
ligne/colonne -> 1/1 : 2.5
ligne/colonne -> 1/1 : 3.5
ligne/colonne -> 1/1 : 4.5
ligne/colonne -> 1/1 : 5.5
ligne/colonne -> 1/1 : 6.5
Saisissez le second tableau
ligne/colonne -> 1/1 : 1
ligne/colonne -> 1/1 : 2
ligne/colonne -> 1/1 : 3
ligne/colonne -> 1/1 : 4
ligne/colonne -> 1/1 : 5
ligne/colonne -> 1/1 : 6
```

```
Produit des 2 tableaux
ligne/colonne -> 1/1 : 1.5
ligne/colonne -> 1/1 : 5
ligne/colonne -> 1/1 : 10.5
ligne/colonne -> 1/1 : 18
ligne/colonne -> 1/1 : 27.5
ligne/colonne -> 1/1 : 39
```

Le programme ci-dessus effectue le produit de deux tableaux bidimensionnels tab1 *et* tab2 *d'une taille maximum de 100 lignes (*NBLIG*) par 100 colonnes (*NBLOG*).*

La fonction lectab *permet la saisie de l'ensemble des valeurs pour chacun des tableaux suivant un nombre de lignes* nbl *et de colonnes* nbc *précisé auparavant par l'utilisateur dans la fonction* main*.*

Le résultat est calculé par la fonction produit *qui range les valeurs obtenues dans le tableau* tab3*.*

La fonction affich *lit le tableau* tab3 *et affiche les valeurs qu'il contient.*

4.4.3 Traitement de chaînes de caractères

Une chaîne de caractère peut être considérée comme un tableau de caractères. Considérant cette situation, il devient facile de manipuler celles-ci en associant au traitement, des fonctions spécifiques empruntées au fichier d'en-tête `string.h`.

```
#include <iostream.h>
#include <string.h>

const int NBMOT=100;
const int LGMOT=30;

void tri(int, char tab[NBMOT][LGMOT]);
void affich(int, char tab[NBMOT][LGMOT]);
```

```
main()
{
  char tab[NBMOT][LGMOT];
  for (int i=0;i<100;++i){
    cout << "MOT (STOP pour terminer) " << i+1 <<"
: ";
    cin >> tab[i];
    if (!(strcmpi(tab[i],"STOP"))){
      tri(i, tab);
      break;
    }
  }
  affich (i, tab);
  return 0;
}

void tri(int nb, char tab[NBMOT][LGMOT])
{
 char temp[30];
  for (int i=0;i<nb;++i)
    for (int j=i+1;j<nb;++j){
      cout<<"avant:"<<tab[i]<<"/"<<tab[j]<<endl;
      if (strcmpi(tab[j],tab[i])<0){
        strcpy(temp,tab[j]);
        strcpy(tab[j],tab[i]);
        strcpy(tab[i],temp);
       cout<<"après:"<<tab[i]<<"/"<<tab[j]<<endl;
      }
    }
  return ;
}

void affich(int n, char tab[NBMOT][LGMOT])
{
  for (int i=0;i<n;++i)
    cout << tab[i] << endl;
  return;
}
```

```
MOT (STOP pour terminer) 1 : jean
MOT (STOP pour terminer) 1 : Pierre
MOT (STOP pour terminer) 1 : paul
MOT (STOP pour terminer) 1 : anne
MOT (STOP pour terminer) 1 : marcel
MOT (STOP pour terminer) 1 : simone
MOT (STOP pour terminer) 1 : alice
MOT (STOP pour terminer) 1 : anna
MOT (STOP pour terminer) 1 : PATRICK
MOT (STOP pour terminer) 1 : Jacques
MOT (STOP pour terminer) 1 : STOP
Alice
Anna
Anne
Jacques
Jean
Marcel
PATRICK
Paul
Pierre
simone
```

Ce programme de tri de chaînes de caractères reprend le principe du tri des nombres présenté précédemment. Il utilise les fonctions strcmpi *et* strcpy *du fichier d'en-tête* string.h.

La fonction strcmpi *travaille sur deux chaînes suivant la syntaxe* strcmpi(chaîne1, chaîne2). *Elle retourne une valeur positive, négative ou nulle suivant que* chaîne1 *est après* chaîne2, *que* chaîne1 *est avant* chaîne2 *ou que les deux chaînes sont identiques dans l'ordre alphabétique.*

La fonction strcpy *travaille également sur deux chaînes suivant une syntaxe identique. Elle affecte la valeur contenue dans* chaîne1 *à* chaîne2.

La saisie par l'utilisateur du mot STOP *termine la saisie.*

Chapitre **5**

Les pointeurs

Opérateurs, mots-clés et fonctions

`&, *, new, delete, pow, sqrt`

5.1 LE CONCEPT

Lors de la programmation dans un langage évolué, nous manipulons des variables qui s'adaptent au système en occupant un nombre d'octets en mémoire dépendant de leurs types. Le langage C++ n'échappe pas à cette règle et utilise des variables qui représentent l'adresse mémoire de la donnée manipulée et non sa valeur.

Cette possibilité est un des atouts majeurs de C et C++ qui apporte une souplesse et une puissance nouvelle dans un langage de programmation.

Chaque donnée manipulée par l'ordinateur occupe un emplacement mémoire particulier représenté par une ou plusieurs cellules mémoires contiguës d'une taille déterminée (un ou plusieurs octets en fonction du type).

L'adresse de l'emplacement d'une variable en mémoire s'obtient en faisant précéder la variable du symbole `&` (« et » commercial). L'opérateur `&` est appelé *opérateur d'adresse*, celui-ci a d'autres utilisations possibles comme nous l'avons vu dans le chapitre 4 (passage par référence) ou comme nous le verrons un peu plus loin.

Pour atteindre le contenu, on fera précéder la variable de l'opérateur *, qui est un *opérateur unaire d'indirection*.

La variable qui contient l'adresse est appelée *pointeur* car elle pointe sur l'adresse mémoire ou se situe la valeur considérée.

L'opération qui consiste à retrouver la valeur d'une variable depuis son pointeur s'appelle *déférencement*.

5.2 DÉCLARATION ET UTILISATION

La déclaration d'un pointeur nécessite que le nom de la variable soit précédé du symbole * (astérisque).

La syntaxe est la suivante :

```
type *pointeur
```

```
#include <iostream.h>

main()
{
  int x,y, *px, *py;
  x=100;
  cout<<"valeur de x : "<<x<<endl;
  cout<<"adresse de x : "<<&x<<endl;
  px=&x;
  *px=999;
  cout<<"pointeur vers x : "<<*px<<endl;
  cout<<"valeur de x : "<<x<<endl;
  cout<<"adresse de x : "<<px<<endl;
  y=*px;
  py=&y;
  cout<<"valeur de y : "<<y<<endl;
```

```
  cout<<"adresse de y : "<<py<<endl;
  cout<<"pointeur vers y : "<<*py<<endl;
  return 0;
}
```

```
Valeur de x : 100
Adresse de x : 0x0012FF7C
Pointeur vers x : 999
Valeur de x : 999
Adresse de x : 0x0012FF7C
Valeur de y : 999
Adresse de y : 0x0012FF78
Pointeur vers y : 999
```

Dans ce programme *px *et* *py *sont des pointeurs vers un entier.*

L'affectation px=&x *assigne l'adresse de* x *à* px, *soit* 0x0012FF7C.

L'affectation *px=999 *stocke la valeur 999 dans la case mémoire désignée par* 0x0012FF7C *en remplacement de la valeur 100.*

L'affectation y=*px *assigne la valeur de* x *à* y, *soit 999.*

L'affectation py=&y *assigne l'adresse de* y *à* py, *soit* 0x0012FF78.

Le pointeur *py *contient la valeur 999.*

5.3 POINTEUR ET FONCTION

Nous allons souvent être amenés à passer un pointeur comme argument d'une fonction. La technique la plus souvent retenue est un *passage par référence* comme nous l'avons expliqué au chapitre 3.

C'est en effet compréhensible puisque la donnée (le contenu de l'adresse) sera modifiée de façon globale, dans la fonction et le programme qui l'appelle.

```
#include <iostream.h>

main()
{
  int x,y;

  void fctentier(int x, int y);
  void fctpointeur(int *x, int *y);

  cout<<"Valeur entiere pour x : ";
  cin>>x;
  cout<<"Valeur entiere pour y : ";
  cin>>y;
  cout<<"Avant l'appel de fctentier x="<<x<<" et
y="<<y<<endl;
  fctentier(x, y);
  cout<<"Apres l'appel de fctentier x="<<x<<" et
y="<<y<<endl;
 cout<<"Avant l'appel de fctpointeur x="<<x<<" et
y="<<y<<endl;
  fctpointeur(&x, &y);
 cout<<"Apres l'appel de fctpointeur x="<<x<<" et
y="<<y<<endl;
  return 0;
}

void fctentier(int i, int j)
{
 i=99;
  j=100;
 cout<<"A l'interieur de fctentier i=x="<<i<<" et
j=y="<<j<<endl;
  return;
}

void fctpointeur(int *m, int *n)
{
```

```
  *m=99;
  *n=100;
  cout<<"A l'interieur de fctpointeur
*m=*x="<<*m<<" et *n=*y="<<*n<<endl;
  return;
}
```

```
Valeur entiere pour x : 5
Valeur entiere pour y : 6
Avant l'appel de fctentier x=5 et y=6
A l'interieur de fctentier i=x=9999 et j=y=1111
Apres l'appel de fctentier x=5 et y=6
Avant l'appel de fctpointeur x=5 et y=6
A l'interieur de fctpointeur *m=*x=9999 et
*n=*y=1111
Apres l'appel de fctpointeur x=9999 et y=1111
```

La fonction fctentier *utilise des arguments standards* i, j *et la fonction* fctpointeur, *deux pointeurs* *m *et* *n.

5.4 POINTEUR ET TABLEAU

Pour le compilateur, le nom d'un tableau est un pointeur vers le premier élément. En prenant en compte cette constatation nous pouvons dire qu'une expression `&nomtableau[0]` est équivalente à `nomtableau` en terme d'adresse. En extrapolant on arrive ainsi à `nomtableau[i]` équivalent à `*[nomtableau+i]` si on considère les contenus.

```
#include <iostream.h>

const int NBVAL=100;

main()
{
  int  nb, tab[NBVAL], vmax=0, vmin=0;
```

```
  cout<<"Nombre de valeurs a traiter : ";
  cin>>nb;
  for(int i=0;i<nb;++i){
    cout<<"Nombre entier : ";
    cin>>tab[i];
  }
  for(int j=0;j<nb;++j){
    if (*(tab+j)>vmax) vmax=*(tab+j);
    if (*(tab+j)<vmin) vmin=*(tab+j);
  }
  cout<<"Valeur maxi : "<<vmax<<endl;
  cout<<"Valeur mini : "<<vmin<<endl;
  return 0;
}
```

```
Nombre de valeurs a traiter : 6
Nombre entier : 0
Nombre entier : 9
Nombre entier : -34
Nombre entier : 45
Nombre entier : 6
Nombre entier : -2
Valeur maxi : 45
Valeur mini : -34
```

Ce programme recherche la valeur maxi et la valeur mini d'un tableau tab *d'entiers.*

*Deux tests (*if*) placés à l'intérieur d'une boucle comparent successivement la valeur en cours, à deux entiers* vmax *et* vmin *dans lesquels sont rangés chacun des entiers du tableau* tab *à tour de rôle.*

Les tests utilisent le pointeur *(tab+j) *pour aller chercher les entiers dans le tableau* tab *et faire les comparaisons.*

Il est bien sûr possible de parcourir un tableau à l'aide d'un pointeur. Il faut remarquer que l'adresse n'étant pas de type entier, nous pouvons

tout de même l'incrémenter comme si elle l'était. Le compilateur se charge de dimensionner correctement la taille de l'incrément qui dépend de l'objet auquel le pointeur est attaché.

On peut dire que l'incrément est plutôt un décalage (*offset*) depuis une adresse de base qui est définie par l'emplacement du premier élément du tableau.

```
#include <iostream.h>

main()
{
  int tab[10]={1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
  int j=0;
  for(int *n=tab; n<tab+10; ++n){
    cout <<"n="<<n<<endl;
    cout << "tab["<<j<<"]="<<*n<<" - adresse :
"<<n<<endl;
    ++j;
  }
return 0;
}
```

```
n=0x0012FF58
tab[0]=1 - adresse : 0x0012FF58
n=0x0012FF58
tab[1]=2 - adresse : 0x0012FF5C
n=0x0012FF58
tab[2]=3 - adresse : 0x0012FF60
n=0x0012FF58
tab[3]=5 - adresse : 0x0012FF64
n=0x0012FF58
tab[4]=7 - adresse : 0x0012FF68
n=0x0012FF58
tab[5]=11 - adresse : 0x0012FF6C
n=0x0012FF58
tab[6]=13 - adresse : 0x0012FF70
```

```
n=0x0012FF58
tab[7]=17 - adresse : 0x0012FF74
n=0x0012FF58
tab[8]=19 - adresse : 0x0012FF78
n=0x0012FF58
tab[9]=23 - adresse : 0x0012FF7C
```

Lors de la lecture du tableau tab, n *est un pointeur sur* int, *il avance donc de 4 octets à chaque incrémentation de la boucle et lit chacune des valeurs entières contenues dans le tableau* tab.

L'instruction int *n=tab *initialise le pointeur* n *à l'adresse de départ du tableau* tab.

5.5 DE NOUVEAUX OPÉRATEURS

En langage C++ , deux opérateurs, `new` et `delete`, de gestion de la mémoire sont présents aux côtés de `malloc` et `free` existant en C.

Quand un pointeur est déclaré, on n'est pas certain que l'adresse qu'il va utiliser n'est pas déjà allouée à une autre variable. Si c'est le cas, une erreur est générée et le pointeur n'est pas initialisé.

Pour ne pas avoir ce problème, l'opérateur *unaire* `new` vient à notre secours.

Cet opérateur va allouer de la mémoire au pointeur concerné.

Voici quelques exemples d'utilisation de cet opérateur.

```
float *x;
x=new float;
*x = 2.718;
```

ou

```
float *x=new float;
*x = 2.718;
```

La variable pointeur `x` est déclarée de type *float*. Une zone mémoire est allouée pour le flottant `x` puis, `*x` qui a été alloué, reçoit 2.718.

```
float *tab[10];
tab[1] = new float(2.718);
```

Dans le cas où les éléments d'un tableau sont des pointeurs, nous pouvons allouer une zone mémoire à chacun des éléments comme lors d'une utilisation classique.

Par ce moyen nous pouvons manipuler un tableau constitué de pointeurs.

```
#include <iostream.h>

const int NBVAL=100;

main()
{
  void tri(int, int *tab[NBVAL]);
  void affich(int, int *tab[NBVAL]);
  void saisie(int, int *tab[NBVAL]);

  int nbr;
  int *tab[NBVAL];
  cout << "Nombre de valeurs a trier : ";
  cin >> nbr;

  saisie(nbr, tab);
  tri(nbr, tab);
  affich (nbr, tab);
  return 0;
}

void saisie(int nb, int *tab[NBVAL])
{
  int v;
  for (int i=0;i<nb;++i){
    cout << "valeur " << i+1 <<" : ";
    cin >> v;
    tab[i]=new int(v);
```

```
  }
}

void tri(int nb, int *tab[])
{
  int* temp;
  for (int i=0;i<=nb;++i)
    for (int j=i+1;j<nb;++j)
      if (*tab[j]<*tab[i]){
        temp=tab[j];
        tab[j]=tab[i];
        tab[i]=temp;
      }
  return ;
}

void affich(int n, int *tab[])
{
  for (int i=0;i<n;++i)
    cout << *tab[i] << endl;
  return;
}
```

```
Nombre de valeurs a trier : 6
Valeur 1 : -34
Valeur 2 : 3
Valeur 3 : 100
Valeur 4 : 0
Valeur 5 : 2
Valeur 6 : -3
-34
-3
0
2
3
100
```

Dans ce programme de tri, le programme utilise un tableau de pointeurs tab[NBVAL] *à une dimension (vecteur).*
Lors de la saisie, on alloue un emplacement mémoire à chaque élément : tab[i]=new int(v).

L'opérateur `delete` a la fonction inverse de l'opérateur `new`, il libère la mémoire allouée. Il n'est applicable que sur des pointeurs déjà alloués.

Cette opération permet de libérer la zone mémoire.

Quand un pointeur est désalloué, il n'est plus initialisé et ne pointe sur plus rien.

5.6 QUELQUES REMARQUES

Nous pouvons nous demander ce qui se passe quand une déclaration de fonction est présente à l'intérieur d'une autre fonction. On cherche là, à définir une fonction de fonction.

Dans ce cas, nous allons être amenés à spécifier comme argument d'une fonction le nom d'une autre fonction.

La syntaxe adoptée est la suivante :

```
type (* nomfonction)(argument1, argument2,…, argumentn)
```

L'exemple qui suit montre les principes qui viennent d'être exposés.

```
#include <iostream.h>
#include <math.h>

double add(double (*)(double),int);
double racine(double);
double carre(double);

main()
{
  cout<<"Somme des racines : "<<add(racine,
10)<<endl;
```

```
  cout<<"Somme des carres : "<<add(carre, 10)<<
endl;
  return 0;
}

double add(double (*pt)(double k), int x)
{
  int i=0;
  double somme=0;
  for(i=1;i<=x;++i)
    somme+=(pt)(i);
  return somme;
}

double racine(double y)
{
  double rac;
  rac=sqrt(y);
  cout<<"Racine de "<<y<<" = "<<rac<<endl;
  return rac;
}
double carre(double z)
{
  double car;
  car=pow(z, 2);
  cout<<"carre de "<<z<<" = "<<car<<endl;
  return car;
}
```

```
Racine de 1 = 1
Racine de 2 = 1.41421
Racine de 3 = 1.73205
Racine de 4 = 2
Racine de 5 = 2.23607
Racine de 6 = 2.44949
Racine de 7 = 2.64575
Racine de 8 = 2.82843
```

```
Racine de 9 = 3
Racine de 10 = 3.16228
Somme des racines : 22.4683
Carre de 1 = 1
Carre de 2 = 4
Carre de 3 = 9
Carre de 4 = 16
Carre de 5 = 25
Carre de 6 = 36
Carre de 7 = 49
Carre de 8 = 64
Carre de 9 = 81
Carre de 10 = 100
Somme des carres : 385
```

La fonction add *évalue la fonction sur laquelle pointe* pt.

La boucle i *calcule ensuite la somme des racines carrées des nombres de 1 à 10, puis la somme des carrés des nombres de 1 à 10.*

Le fichier d'en-tête math.h *est présent pour fournir les fonctions* sqrt *(racine carrée) et* pow *(puissance d'un nombre).*

Une possibilité qui peut également survenir lors d'un programme, est la définition d'un pointeur de pointeur. Ce traitement est tout à fait possible.

```
#include <iostream.h>

main()
{
  int x=5;
  cout<<"x = "<<x<<endl;
  int *ptx=&x;
  cout<<"*ptx = "<<*ptx<<endl;
  cout<<"&x : "<<&x<<endl;
```

```
  int **ptptx=&ptx;
  cout<<"**ptptx = "<<x<<endl;
  cout<<"&ptx : "<<&ptx<<endl;
  **ptptx=7;
  cout<<"x = "<<x<<endl;
  return 0;
}
```

```
x = 5
*ptx = 5
&x : 0x0012FF7C
**ptptx = 5
&ptx : 0x0012FF78
x = 7
```

L'affectation **ptptx=7 *change la valeur entière contenue dans* x *en 7.*

Cette affectation **ptptx=7 *retourne au contenu de l'adresse* pt *pointée par l'adresse* ptptx.

Vous trouverez ci-dessous quelques exemples commentés de déclarations utilisant des pointeurs :

– `int *pt`
  `pt` est un pointeur sur un entier.
– `int *pt[100]`
  `pt` est un tableau de 100 pointeurs sur des entiers.
– `int (*pt)[100]`
  `pt` est un pointeur vers un tableau de 100 entiers.
– `int pt(char *x)`
  `pt` est une fonction ayant pour argument un pointeur de type `char`. Elle renvoie un entier.
– `int (*pt)(char *x)`
  `pt` est un pointeur de fonction ayant pour argument un pointeur de type `char`.
  Elle renvoie un entier.

– `int *pt(char *x)`
  `pt` est une fonction ayant pour argument un pointeur de type `char`.
  Elle renvoie un pointeur sur un entier.
– `int *(*pt)(char *a[])`
  `pt` est un pointeur vers une fonction ayant pour argument un tableau de pointeurs de type `char`.
  Elle retourne un pointeur sur un entier.
– `int *(*pt)(char (*a)[])`
  `pt` est un pointeur vers une fonction ayant pour argument un pointeur vers un tableau de type `char`.
  Elle renvoie un entier.

Chapitre **6**

Structures et définitions de type

Opérateurs, mots-clés et fonctions

`., ->, atoi, getline, struct, typedef`

6.1 DÉCLARATION ET DÉFINITION

Pour déclarer une structure, on va devoir préciser le type et le nom des variables qu'elle va manipuler.

Le mot-clé retenu pour la déclaration d'une structure est `struct`.

La syntaxe est la suivante :

```
struct nomstructure {
  typechamp1 nomchamp1 ;
  typechamp2 nomchamp2 ;
  .
  .
  .
```

```
  typechampn nomchampn ;
} ;
```

Le mot-clé `struct` est optionnel et deux champs de la structure ne peuvent pas posséder un même nom.

Tous les types de données sont admis.

Lorsque la structure a été déclarée on peut préciser les variables qui seront du type spécifié. On désigne ces variables par le terme *variables structurées*.

Le nom des *variables structurés* peut suivre la déclaration de la structure, suivant la syntaxe suivante :

```
struct nomstructure {
  typechamp1 nomchamp1 ;
  typechamp2 nomchamp2 ;
  .
  .
  .
  typechampn nomchampn ;
} nomvariable1, nomvariable2,…, nom variable3;
```

6.2 ACCÈS

Pour accéder aux champs d'une variable structurée, on utilise l'opérateur . (point) qui sera placé entre le nom de la variable et le nom du champ choisi.

La syntaxe est la suivante :

```
nomvariable.nomchamp
```

```
#include <iostream.h>

struct fichier{
  int ref;
  int nb;
  float longueur;
  float diametre;
}article1, article2;
```

```
main()
{
  article1.ref=999;
  article1.nb=50;
  article1.longueur=200.5;
  article1.diametre=10.75;
  article2.ref=111;
  article2.nb=25;
  article2.longueur=152.5;
  article2.diametre=6.5;
  cout<<"Ref : "<<article1.ref<<" Nb :
"<<article1.nb<<" Long : "<<article1.longueur<<"
Diam : "<<article1.diametre<<endl;
  cout<<"Ref : "<<article2.ref<<" Nb :
"<<article2.nb<<" Long : "<<article2.longueur<<"
Diam : "<<article2.diametre<<endl;
  return 0;
}
```

```
Ref : 999 Nb : 50 Long : 200.5 Diam : 10.75
Ref : 111 Nb : 25 Long : 152.5 Diam : 6.5
```

*La structure fichier est composée des membres de type entier (*int*)* ref, nb *et de type flottant (*float*)* longueur, diametre.

Deux variables structurées de type fichier, article1 *et* article2 *sont définies et suivent la définition de la structure.*

Par l'intermédiaire de l'opérateur point . (point), le programme accède aux différents champs des variables article1 *et* article2 *pour leur assigner des valeurs.*

6.3 TABLEAU ET STRUCTURE

Un tableau peut contenir des éléments construits sur un type de structure déjà défini.

```
#include <iostream.h>
struct personne{
  char nom[25];
  char prenom[25];
  int age;
  float taille;
  float poids;
}patient[100];

main()
{
  void saisie(int i);
  void lecture(int i);

  int i,nb;

  cout<<"Nombre de patients : ";
  cin>>nb;
  for(i=0; i<nb; ++i)
    saisie(i);
  for(i=0; i<nb; ++i)
    lecture(i);
  return 0;
}
 void saisie(int i)
  {
    cout<<"Nom : ";
    cin>>patient[i].nom;
    cout<<"Prenom : ";
    cin>>patient[i].prenom;
    cout<<"Age : ";
    cin>>patient[i].age;
    cout<<"Taille : ";
    cin>>patient[i].taille;
    cout<<"Poids : ";
    cin>>patient[i].poids;
    return;
```

```
  }

  void lecture(int i)
  {
    cout<<"Patient : "<<patient[i].nom<<" "<<
patient[i].prenom<<" "<<patient[i].age<<"ans "<<
patient[i].taille<<"m "<<pa-
tient[i].poids<<"kg"<<endl;
}
```

```
Nombre de patients : 3
Nom : DUPONT
Prenom : Marie
Age : 30
Taille : 1.65
Poids : 60
Nom : DURAND
Prenom : Pierre
Age : 35
Taille : 1.80
Poids : 82
Nom : MARTIN
Prenom : Annie
Age : 25
Taille : 1.71
Poids : 68
Patient : DUPONT Marie 30ans 1.65m 60kg
Patient : DURAND Pierre 35ans 1.8m 82kg
Patient : MARTIN Annie 25ans 1.71m 68kg
```

Le tableau patient *est construit sur le type de structure* personne *et peut contenir 100 éléments. La fonction* saisie *permet de le remplir suivant un nombre déterminé de patients* nb.

La fonction lecture *lit le tableau* patient *et affiche son contenu.*

6.4 STRUCTURES ET POINTEURS

L'adresse de départ de rangement des variables d'une structure s'obtient classiquement à l'aide de l'opérateur `&` (adresse). On peut déclarer un pointeur sur cette variable comme pour toute autre.

On peut accéder à un membre d'une structure par l'intermédiaire de sa variable pointeur *via* la syntaxe suivante :

```
pointeurvarstruct->membrestruct
```

où `pointeurvarstruct` est une variable pointeur associée à la structure et `->` un opérateur d'un type équivalent à `.` (`point`), présenté précédemment.

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

struct film{
  char titre[40];
  int annee;
}cine1, *cine2;

main()
{
  char tampon[40];
  cine2=&cine1;
  cout<<"Titre du film : ";
  cin.getline(cine2->titre,40);
  cout<<"Annee : ";
  cin.getline(tampon,40);
  cine2->annee=atoi(tampon);
  cout<<cine2->titre;
  cout<<" - Annee : "<<cine2->annee<<endl;
  return 0;
}
```

```
Titre du film : Le docteur Jivago
Annee : 1965
Le docteur Jivago - Annee : 1965
```

La variable cine1 *est de type* film *et la variable* cine2 *est un pointeur sur la structure* film*. On affecte à ce pointeur l'adresse de départ de la variable structurée* film par cine2=&cine1.

La fonction cin.getline(cine2->titre, 40) *lit les caractères du titre du film sur le flot qui l'a appelé et les place dans* titre *suivant une longueur de 40 caractères. Il est en de même pour l'année du film.*

La fonction atoi *retourne l'entier représenté par tampon.*

L'expression précédente est équivalente à :

```
(*pointeurvarstruct).membrestruct
```

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

struct film{
  char titre[40];
  int annee;
}cine1, *cine2;

main()
{
  char tampon[40];
  cine2=&cine1;
  cout<<"Titre du film : ";
  cin.getline(*cine2).titre,40);
  cout<<"Annee : ";
  cin.getline(tampon,40);
  cine2->annee=atoi(tampon);
  cout<<(*cine2).titre;
  cout<<" - Annee : "<<cine2->annee<<endl;
  return 0;
}
```

```
Titre du film : Le docteur Jivago
Annee : 1965
Le docteur Jivago - Annee : 1965
```

Idem à l'exemple précédent avec le remplacement de cine2->titre *par* (*cine2).titre.

Il faut bien distinguer `(*pointeurvarstruct).membrestruct` qui désigne le membre pointé par la structure de l'expression `*(pointeurvarstruct.membrestruct)` qui pointe le membre de la structure.

L'opérateur `->` peut aussi être employé pour accéder à un élément d'un tableau qui est membre d'une structure suivant la syntaxe :

```
pointeurvariable->membrestruct[expression]
```

où expression est une valeur positive ou nulle qui désigne un élément du tableau.

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

struct film{
  char titre[40];
  int annee;
}cine1,*cine2;

main()
{
  char tampon[40];
  cine2=&cine1;
  cout<<"Titre du film : ";
  cin.getline(cine2->titre,40);
  cout<<"Annee : ";
  cin.getline(tampon,40);
  cout<<"La 6eme lettre du titre est :
"<<cine2->titre[5]<<endl;
```

```
  cout<<"La 6eme lettre du titre est :
"<<cine1.titre[5]<<endl;
  cout<<"La 6eme lettre du titre est :
"<<(*cine2).titre[5]<<endl;
  cout<<"La 6eme lettre du titre est :
"<<*(cine1.titre+5)<<endl;
  return 0;
}
```

```
Titre du film : Le docteur Jivago
Annee : 1065
La 6eme lettre du titre est : c
La 6eme lettre du titre est : c
La 6eme lettre du titre est : c
La 6eme lettre du titre est : c
```

Les expressions cine2->titre[5], cine1.titre[5], (*cine2).titre[5], *(cine1.titre+5) *affichent la 6ème lettre du titre du film et sont toutes équivalentes.*

6.4 STRUCTURES IMBRIQUÉES

Une structure peut être membre d'une autre structure.

```
#include <iostream.h>
struct film{
  char titre[40];
  int annee;
};

struct cinefil{
  char nom[30];
  char prenom[30];
  struct film filmpref;
```

```
}membreclub[100];

void saisie(int i);
void lecture(int i);

main()
{
  int i, nbm;
  cout<<"Nombre de membres a traiter : ";
  cin>>nbm;
  for (i=0; i<nbm;++i)
    saisie(i);
  for (i=0; i<nbm;++i)
    lecture(i);
  return 0;
}

void saisie(int i)
{
 cout<<"Nom : ";
  cin>>membreclub[i].nom;
  cout<<"Prenom : ";
  cin>>membreclub[i].prenom;
  cout<<"Film prefere : ";
  cin>>membreclub[i].filmpref.titre;
  cout<<"Annee : ";
  cin>>membreclub[i].filmpref.annee;
  return;
}

void lecture(int i)
{
  cout << membreclub[i].nom << " " << membre-
club[i].prenom << " - film prefere : " << membre-
club[i].filmpref.titre << " - "<<
membreclub[i].filmpref.annee <<endl;
  return;
}
```

```
Nombre de membres a traiter : 2
Nom : DUPONT
Prenom : Marie
Film prefere : Ricochet
Annee : 1991
Nom : DUMAS
Prenom : Paul
Film prefere : Solaris
Annee : 1972
DUPONT Marie - film prefere : Ricochet - 1991
DUMAS Paul - film prefere : Solaris - 1972
```

La structure film *est imbriquée dans la structure* cinefil.

6.5 DÉFINITIONS DE TYPE

Le langage C++ offre la possibilité de définir ses propres types comme nous l'avons déjà vu au chapitre 1 avec les types `enum` mais aussi des types qui sont des *alias* de types déjà existants.

C'est la directive utilisant le mot-clé `typedef` qui se charge de cette opération.

La syntaxe est la suivante :

```
typedef type typealias
```

```
#include <iostream.h>

typedef double reel;
typedef int entier;

const reel PI=3.14159;

main()
{
```

```
  entier i;
  for(i=1; i<=4
    ;++i)
    cout<<"Rayon : "<<i<<" - Circonference :
"<<2*PI*i<<endl;  return 0;
}
```

```
Rayon : 1 - Circonference : 6.28318
Rayon : 2 - Circonference : 12.5664
Rayon : 3 - Circonference : 18.8495
Rayon : 4 - Circonference : 25.1327
```

Le type double *prend pour synonyme* reel.

Le type int *prend pour synonyme* entier.

La constante PI *utilise le type alias* reel *et la variable* i*, le type alias* entier.

Les définitions de type sont utilisables dans la déclaration de tableaux, comme dans l'exemple suivant :

```
typedef float degre
degre celsius[50], fahrenheit[50]
```

Cette forme est équivalente à celle-ci :

```
typedef float degre[50]
degre celsius, fahrenheit
```

La directive `typedef` est applicable sur les structures, où elle simplifie leur utilisation, lorsque le programmeur doit manipuler plusieurs structures identiques de noms différents.

```
typedef struct{
  int ref;
  int qte;
  char designation[50];
  int taille;
  float cout;
```

```
}article;

article vetFemme, vetHomme;
```

Dans l'exemple ci-dessus `vetFemme` et `vetHomme` sont définis comme des variables structurées ayant le type `article`.

Chapitre 7

Les classes

Opérateurs, mots-clés et fonctions

```
~, ::, class, private, protected, public, static
```

7.1 RAPPELS SUR LA PROGRAMMATION OBJET

La programmation orientée objet (POO) apporte une amélioration de la fiabilité du code et une réutilisation possible des éléments d'un développement.

Elle repose sur quatre grands concepts ou principes :

- l'encapsulation ;
- les classes ;
- l'héritage ;
- le polymorphisme.

Un objet est un composant logiciel avec lequel on va manipuler des données et des fonctions qui vont apporter des fonctionnalités au programme.

Par l'intermédiaire de l'*encapsulation*, un objet va pouvoir être utilisé tout en masquant son fonctionnement interne.

À l'intérieur d'un objet on trouve des informations concernant les données qui sont définies en C++, comme étant des *attributs* et des fonctions de traitement des données dénommées *méthodes*.

Les *attributs* et les *méthodes* sont appelés des membres de l'objet.

Une *classe* peut être vue comme une *structure* (voir chapitre 6) possédant des fonctions en plus des champs habituels.

Un objet est obtenu par *l'instanciation* d'une *classe*. Cette *instanciation* n'est autre que sa déclaration.

L'héritage ou *dérivation* autorise la création de *sous-classe* à partir d'une autre *classe* (dite aussi *super-classe* ou *sur-classe*).

Le *polymorphisme* permet à tout objet *instancié* d'une classe ancêtre d'être remplacé par un objet d'une *classe* descendante de la *classe* ancêtre. Par exemple, si nous possédons deux objets *instanciés* des *classes* ancêtres rectangle et triangle, nous pouvons définir un objet (*instance*) d'une *classe* polygone qui pourra manipuler des rectangles et des triangles. Nous avons donc défini un *polymorphisme* d'objets.

Note : ces notions sont loin d'être complètes, j'invite le lecteur qui voudrait approfondir ces notions à consulter la bibliographie de cet ouvrage.

7.2 LES CLASSES

La manipulation d'objets va nécessiter la création de *classes*. Une *classe* définie la structure d'un objet.

Le mot-clé retenu est `class`.

Une *classe* va contenir des données, appelées *données membres* et des traitements appelés *traitements membres.*

Les *données membres* sont des variables qui sont rangées à l'intérieur d'une *classe*. Comme toutes variables, elles doivent être précédées de leur type et posséder un nom. On regroupe ensuite ces variables sous une étiquette qui mentionne la *portée*, c'est-à-dire l'accès que le programme pourra avoir vis-à-vis de cette donnée.

```
class nomclasse{
...
} ;
```

Il y a trois étiquettes possibles que l'on appelle aussi des *qualificatifs d'accès* :

– `private` : elle définit un accès aux membres seulement depuis l'intérieur de la *classe* ;
– `public` : elle définit un accès aux membres seulement depuis l'extérieur de la *classe* ;
– `protected` : elle définit un accès aux membres seulement aux *classes dérivées* ou *amies* (voir chapitre 8).

L'exemple suivant va nous permettre de présenter plusieurs notions liées à la *classe*.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
  void dimensions(double, double, double);
  double surface (void){
    return ((b+B)*h/2);
  }
};

void trapeze::dimensions(double x, double y,
double z){
  b=x/2.54;
  B=y/2.54;
  h=z/2.54;
}
main(){
  double pb, gb, ht;
  cout<<"Petite base (en pouces) : ";
  cin>>pb;
```

```
  cout<<"Grande base (en pouces) : ";
  cin>>gb;
  cout<<"Hauteur (en pouces) : ";
  cin>>ht;
  trapeze T;
  T.dimensions(pb, gb, ht);
  cout<<"Surface (cm2) : "<<T.surface()<<endl;
  return 0;
}
```

```
Petite base (en pouces) : 10
Grande base (en pouces) : 20
Hauteur (en pouces) : 5
Surface (cm2) : 11.625
```

*Ce programme calcule la surface en cm² d'un trapèze dont les dimensions, petite base (*b*), grande base (*B*) et hauteur (*h*) sont données en pouces.*

Rappel : un pouce vaut 2,54 cm.

On peut tout d'abord voir la déclaration de la *classe* qui se nomme `trapeze` et qui présente ses *membres privés*, `b`, `B`, `h` et ses *membres publics*, la fonction `dimensions` et la fonction `surface`.

Les variables `b`, `B`, et `h` étant privés, elles ne sont accessibles que depuis l'intérieur de la *classe* c'est la mise en pratique de la notion d'*encapsulation* déjà présentée.

L'objet `T` (*instance*) fait partie de la *classe* `trapeze` : `trapeze T`.

L'appel d'une fonction membre se fait par l'intermédiaire de l'opérateur . (point). On précise tout d'abord le nom de son propriétaire puis son nom : `T.dimensions`.

L'opérateur de *résolution de portée* `::` précise que la fonction `dimensions` est attachée à la *classe* `trapeze`. C'est par ce moyen que le compilateur peut savoir que la fonction qui va être définie est membre de la *classe* précisée.

Très souvent les fonctions membres sont définies hors de la déclaration de la *classe* comme pour la fonction dimensions. Elles peuvent cependant être définies à l'intérieur comme `surface`, bien que

ce soit assez rare. En effet, le but est de séparer les déclarations de fonctions, des définitions de fonctions afin de mettre en application le principe général de l'*encapsulation*.

Le choix de « *privatiser* » les variables `b`, `B`, et `h` est dû au fait que le reste du programme n'a pas du tout besoin de leur accéder. Les initialisations, les affectations et la conversion restent internes à la fonction `dimensions`.

Un des avantages principal des *classes* est la possibilité de déclarer ou d'*instancier* plusieurs objets différents de la même *classe*.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public :
  void dimensions(double, double, double);
  double surface (void){
    return ((b+B)*h/2);
  }
};

void trapeze::dimensions(double x, double y,
double z){
  b=x/2.54;
  B=y/2.54;
  h=z/2.54;
}

main(){
  trapeze T1, T2, T3;
  T1.dimensions(10, 20, 2.5);
  T2.dimensions(15, 20, 3.5);
  T3.dimensions(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 5.81251
Surface : 9.49377
Surface : 6.35501
```

On définit ici trois objets (instances) de la classe dimensions, T1, T2 *et* T3 *dont on calcule la surface en cm² en appelant la fonction membre* dimensions.

7.3 CONSTRUCTEURS ET DESTRUCTEURS

Dans l'exemple précédent l'initialisation des objets se fait par l'intermédiaire de la fonction `dimensions`. Cette étape pourrait se faire lors de la déclaration.

Le langage C++ utilise à cette fin la notion de `constructeur`.

Un *constructeur* est une fonction membre qui sera appelée systématiquement lors de la déclaration d'un objet.

Pour exister, cette fonction doit posséder le même nom que la *classe* et ne pas avoir de type de retour.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
  trapeze(double, double, double);
  double surface (void){
    return ((b+B)*h/2);
  }
};

trapeze::trapeze(double x, double y, double z){
  b=x/2.54;
  B=y/2.54;
  h=z/2.54;
}
```

```
main(){
  trapeze T1(10, 20, 2.5);
  trapeze T2(15, 20, 3.5);
  trapeze T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 5.81251
Surface : 9.49377
Surface : 6.35501
```

La fonction trapeze *est ici un constructeur de la classe* trapeze.

Elle est présente dans la déclaration de la classe et sa définition est extérieure.

On peut placer la définition de la fonction *constructeur* dans la déclaration de la *classe*.

C'est souvent le cas si la fonction ne fait qu'initialiser les variables.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
  trapeze(double x, double y, double z){
    b=x; B=y; h=z;}
  double surface (void){
    return ((b+B)*h/2);
  }
};
```

```
main(){
  trapeze T1(10, 20, 2.5);
  trapeze T2(15, 20, 3.5);
  trapeze T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 37.5
Surface : 61.25
Surface : 41
```

Ici, la définition de la fonction constructeur trapeze *suit sa déclaration.*

C++ fournit aussi le moyen d'initialiser les données membres de l'objet de façon simple par l'intermédiaire des listes d'initialisations de *constructeurs*.

Cette liste permet la gestion de valeurs par défaut si aucun paramètre n'est passé lors de l'appel de la fonction.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
  trapeze(double x=1, double y=2, double
z=1):b(x),B(y),h(z) {}
```

```
  double surface (void){
    return ((b+B)*h/2);
  }
};

main(){
  trapeze T1, T2(15, 20, 3.5), T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 1.5
Surface : 61.25
Surface : 41
```

Dans la déclaration de la classe, on trouve une liste d'initialisation pour le constructeur trapeze*. Par défaut les valeurs affectées à* x, y *et* z *sont égales à 1, 2 et 1.*

Lors de l'appel, T1 *utilise les valeurs par défaut puisqu'aucun paramètre n'est passé.*

T2 *et* T3 *utilisent les paramètres qui leurs sont passés.*

Lorsqu'un objet n'est plus utilisé, d'une manière analogue à sa construction on va pouvoir le supprimer ou le détruire par l'appel d'une fonction de destruction nommée *destructeur.*

Une *classe* ne peut comporter qu'un seul *destructeur* qui existe automatiquement par défaut s'il n'est pas déclaré. Il est tout de même conseillé de le déclarer.

Un *destructeur* doit porter le même nom que la classe et doit être précédé d'un tilde ~.

Il ne retourne aucune valeur.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
 trapeze(double x, double y, double z):b(x), B(y),
h(z) {}
  ~trapeze (){}
  double surface (void){
    return ((b+B)*h/2);
  }
};


main(){
  trapeze T1(10, 20, 2.5);
  trapeze T2(15, 20, 3.5);
  trapeze T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 37.5
Surface : 61.25
Surface : 41
```

Dans ce programme, le destructeur de la classe trapeze *a été défini.*

Lorsqu'aucun *constructeur* n'est défini, le compilateur défini automatiquement deux *constructeurs* : le *constructeur* par défaut et le *constructeur* par copie.

Le *constructeur par défaut* est appelé chaque fois qu'un objet est déclaré. Il ne comporte pas de paramètres.

Quand un objet est dupliqué c'est le *constructeur par copie* qui intervient. Il comporte un paramètre qui est l'objet qui doit être copié.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public:
 trapeze(double x, double y, double z):b(x), B(y),
h(z) {}
  trapeze(const trapeze&);
  ~trapeze (){}
  double surface (void){
    return ((b+B)*h/2);
  }
};

int i=1;

trapeze::trapeze(const trapeze& T){
  b=T.b;
  B=T.B;
  h=T.h;
  cout<<"Appel "<<i<<" du constructeur par
copie"<<endl;
  i=i+1;
}

trapeze Ttrapeze(trapeze T)          //Appel 2
{
  trapeze T5=T;                      //Appel 3
  return T5;                         //Appel 4
}
```

```
main(){
  trapeze T1(10, 20, 2.5);
  cout<<"Surface T1 : "<<T1.surface()<<endl;
  trapeze T2(15, 20, 3.5);
  cout<<"Surface T2 : "<<T2.surface()<<endl;
  trapeze T3(5.5, 15, 4);
  cout<<"Surface T3 : "<<T3.surface()<<endl;
  trapeze T4(T2);                    //Appel 1
  cout<<"Surface T4 : "<<T4.surface()<<endl;
  Ttrapeze(T4);
  return 0;
}
```

```
Surface T1 : 37.5
Surface T2 : 61.25
Surface T3 : 41
Appel 1 du constructeur par copie
Surface T4 : 61.25
Appel 2 du constructeur par copie
Appel 3 du constructeur par copie
Appel 4 du constructeur par copie
```

Dans ce programme, le destructeur de la classe trapeze *a été défini.*

Les commentaires mentionnent les différents appels au constructeur par copie.

Le *constructeur par copie* n'est pas obligatoire, il faut cependant le déclarer lorsque les données des membres sont manipulées par l'intermédiaire d'un pointeur.

Il est tout à possible d'avoir un pointeur sur une *classe*. Pour accéder au membre d'un objet géré par un pointeur on utilise l'opérateur `->`.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
public :
  void dimensions(double, double, double);
  double surface (void){
    return ((b+B)*h/2);
  }
};

void trapeze::dimensions(double x, double y,
double z){
  b=x/2.54;
  B=y/2.54;
  h=z/2.54;
}

main()
{
  trapeze T1, *T2, *T3;
  trapeze *T4=new trapeze[3];
  T2=new trapeze;
  T3=&T1;
  T1.dimensions (10, 20, 5);
  T2->dimensions(15, 20, 3.5);
  T4->dimensions(5.5, 15, 4);
  T4[1].dimensions(5, 10, 5);
  T4[2].dimensions(20, 40, 10);
  cout<<"Surface T1 : "<<T1.surface()<<endl;
  cout<<"Surface *T2 : "<<T2->surface()<<endl;
  cout<<"Surface *T3 : "<<T3->surface()<<endl;
  cout<<"Surface T4[0] : "<<T4[0].surface()<<endl;
  cout<<"Surface T4[1] : "<<T4[1].surface()<<endl;
  cout<<"Surface T4[2] : "<<T4[2].surface()<<endl;
vreturn 0;
}
```

```
Surface T1 : 11.625
Surface *T2 : 9.49377
Surface *T3 : 11.625
Surface T4[0] : 6.35501
Surface T4[1] : 5.81251
Surface T4[2] : 46.5001
```

Dans ce programme *T2, *T3 *et* *T4 *sont des pointeurs sur la classe* trapeze.

Lorsque la valeur d'une donnée s'applique à plusieurs membres d'une *classe*, elle peut être déclarée comme un membre spécifique. Ce type de donnée membre est nommé *donnée membre statique* et le mot-clé qui lui confère cette propriété est `static`.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
  static int mult;
public:
 trapeze(double x, double y, double z):b(x), B(y),
h(z) {}
  ~trapeze (){}
  double surface (void){
    return ((b+B)*h/mult);
  }
  double perimetre(void){
    return (((b+B)*mult)+(h*mult))/mult;
  }
};

int trapeze::mult=2;
```

```
main(){
  trapeze T1(10, 20, 2.5);
  trapeze T2(15, 20, 3.5);
  trapeze T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Périmetre : "<<T1.perimetre()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Périmetre : "<<T2.perimetre()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  cout<<"Périmetre : "<<T3.perimetre()<<endl;
  return 0;
}
```

```
Surface : 37.5
Périmetre : 32.5
Surface : 61.25
Périmetre : 38.5
Surface : 41
Périmetre : 24.5
```

Dans ce programme mult *est déclarée comme une donnée membre statique affectée à la valeur 2.*

Une *donnée membre statique* est une donnée membre globale pour la classe qui peut être déclarée comme étant *privée* (`private`) ou *publique* (`public`).

Chapitre **8**

Fonctions amies et surcharge des opérateurs

Opérateurs, mots-clés et fonctions

`friend, operator, this`

8.1 FONCTION AMIE

Comme il a été expliqué au chapitre 7 les fonctions membres d'une classe peuvent accéder à tous les membres public et privés de n'importe quel objet de la classe. Il n'en est pas de même pour les fonctions classiques ou les fonctions membres d'une autre classe.

Une possibilité existe en langage C++, c'est la *fonction amie* qui va autoriser l'accès aux membres privés.

Le mot-clé pour déclarer une *fonction amie* est `friend`.

```
#include <iostream.h>

int const TRUE=1, FALSE=0

class rectangle {
private:
  double L, l;
public:
  rectangle(double x, double y){
    L=x; l=y;}
  double surface (void){
    return L*l;
  }
  friend double SurfEgal(const rectangle&, const
rectangle&);
};

double SurfEgal(const rectangle& R1, const
rectangle& R2)
{
  if (R1.L*R1.l== R2.L*R2.l) return TRUE;
  else return FALSE;
}

main()
{
double L1, l1, L2, l2;
  cout<<"Longueur rectangle 1 : ";
  cin>>L1;
  cout<<"Largeur rectangle 1 : ";
  cin>>l1;
  cout<<"Longueur rectangle 2 : ";
  cin>>L2;
  cout<<"Largeur rectangle 2 : ";
  cin>>l2;
  rectangle R1(L1, l1);
  rectangle R2(L2, l2);
```

```
 if (SurfEgal (R1, R2)){
    cout<<"Les surfaces sont égales"<<endl;
    cout<<"Surface R1 : "<<R1.surface()<<endl;
    cout<<"Surface R2 : "<<R2.surface()<<endl;
  }
  else
  {
    cout<<"Les surfaces sont différentes"<<endl;
    cout<<"Surface R1 : "<<R1.surface()<<endl;
    cout<<"Surface R2 : "<<R2.surface()<<endl;
  }
  return 0;
}
```

```
Longueur rectangle 1 : 10.5
Largeur rectangle 1 : 12.5
Longueur rectangle 2 : 5
Largeur rectangle 2 : 26.25
Les surfaces sont égales
Surface R1 : 131.25
Surface R2 : 131.25
```

La fonction SurfEgal *est une fonction amie de la classe rectangle. Elle possède tous les privilèges d'une fonction membre de la classe rectangle sans en être membre.*

8.2 LE MOT-CLÉ POINTEUR `THIS`

Le mot-clé `this` référence à l'aide d'un pointeur l'objet sur lequel un opérateur ou une méthode porte ou travaille.

```
#include <iostream.h>

class trapeze {
private:
  double b, B, h;
```

```
public:
  trapeze(double, double, double);
  double surface (void){
 return ((b+B)*h/2);
  }
};

trapeze::trapeze(double x, double y, double z){
  this->b=x;
  this->B=y;
  this->h=z;
}

main(){
  trapeze T1(10, 20, 2.5);
  trapeze T2(15, 20, 3.5);
  trapeze T3(5.5, 15, 4);
  cout<<"Surface : "<<T1.surface()<<endl;
  cout<<"Surface : "<<T2.surface()<<endl;
  cout<<"Surface : "<<T3.surface()<<endl;
  return 0;
}
```

```
Surface : 37.5
Surface : 61.25
Surface : 41
```

Dans ce programme l'écriture du constructeur utilise le pointeur this.

8.3 SURCHARGE DES OPÉRATEURS

Le langage C++ comporte de nombreux opérateurs présentés dans l'annexe B qui sont définis pour les types standards `int`, `float`, `double`,…

Lors de la création d'une classe, un nouveau type est défini par le programmeur. Nous pouvons créer pour celui-ci, des fonctions qui

simulent les calculs spécifiques de chacun des opérateurs et un appel à ces dernières nous permettra de traiter les calculs à réaliser. Il serait cependant plus simple de pouvoir les utiliser comme des opérateurs normaux.

Dans le cas de l'addition, par exemple, un appel de fonction serait du type :

```
s=somme(x, y)
```

Il paraîtrait plus simple d'écrire :

```
s=x+y
```

La *surcharge* ou *surdéfinition* des opérateurs possible en C++ va réaliser ce traitement pour une classe.

Le mot-clé qui permet la surcharge d'un opérateur est `operator`. Il est suivi de l'opérateur lui-même.

8.4 SURCHARGE D'OPÉRATEURS ARITHMÉTIQUES

La *surcharge* des opérateurs arithmétiques +, -, *, / est très pratique et souvent utilisée.

Il faut toutefois prendre en compte que ces opérateurs arithmétiques ne font partie des fonctions membres de la classe et ne peuvent donc pas accéder aux données *membres privées*.

Pour résoudre ce problème, nous allons faire intervenir la notion de fonction amie déjà présentée au début de ce chapitre.

```
#include <iostream.h>

class rectangle{

private:
  double L, l;
public:
  rectangle(double x=0, double y=0){
    L=x; l=y;}
```

```
  friend rectangle operator +(rectangle&,
rectangle&);
  void affiche(){
    cout<<"Longueur r3 : "<<L<<endl;
    cout<<"Largeur r3 : "<<l<<endl;
  }
};

rectangle operator +(rectangle& m, rectangle& n)
{
  rectangle R(m.L+n.L, m.l+n.l);
  return R;
}

main()
{
 double L1, l1, L2, l2;
  cout<<"Longueur rectangle r1 : ";
  cin>>L1;
  cout<<"Largeur rectangle r1 : ";
  cin>>l1;
  cout<<"Longueur rectangle r2 : ";
  cin>>L2;
  cout<<"Largeur rectangle r2 : ";
  cin>>l2;
  rectangle r1(L1, l1);
  rectangle r2(L2, l2);
  rectangle r3=r1+r2;
  r3.affiche();
  return 0;
}
```

```
Longueur rectangle r1 : 5.5
Largeur rectangle r1 : 12
Longueur rectangle r2 : 4.5
Largeur rectangle r2 : 2.4
Longueur r3 : 10
Largeur r3 : 14.4
```

*Pour surcharger l'opérateur +, une fonction amie (*friend*) est définie. On peut donc ensuite utiliser l'opérateur + naturellement :* r3=r1+r2.

8.5 SURCHARGE D'OPÉRATEURS RELATIONNELS

Les opérateurs relationnels ==, !=, <, <=, > >= peuvent être aussi *surchargés* comme l'ensemble des opérateurs arithmétiques.

```
#include <iostream.h>

int const TRUE=1, FALSE=0;

class rectangle {

private:
  double L, l;
public:
  rectangle(double x, double y){
    L=x; l=y;}
  double operator !=(rectangle R){
    if (R.L*R.l== L*l) return TRUE;
    else return FALSE;
  }
};

main()
{
  double L1, l1, L2, l2, L3, l3;
  cout<<"Longueur rectangle 1 : ";
  cin>>L1;
  cout<<"Largeur rectangle 1 : ";
  cin>>l1;
  cout<<"Longueur rectangle 2 : ";
  cin>>L2;
```

```
  cout<<"Largeur rectangle 2 : ";
  cin>>l2;
  cout<<"Longueur rectangle 3 : ";
  cin>>L3;
  cout<<"Largeur rectangle 3 : ";
  cin>>l3;
  rectangle R1(L1, l1);
  rectangle R2(L2, l2);
  rectangle R3(L3, l3);
  cout<<"Égalite R1, R2 : "<<(R1!=R2)<<endl;
  cout<<"Égalite R2, R3 : "<<(R2!=R3)<<endl;
  cout<<"Égalite R1, R3 : "<<(R1!=R3)<<endl;
  return 0;
}
```

```
Longueur rectangle 1 : 6.5
Largeur rectangle 1 : 5
Longueur rectangle 2 : 5.5
Largeur rectangle 2 : 6
Longueur rectangle 3 : 8.125
Largeur rectangle 2 : 4
Égalite R1, R2 : 0
Égalite R2, R3 : 0
Égalite R1, R3 : 1
```

L'opérateur relationnel != *est surchargé au sein de la classe rectangle. Il permet de comparer de façon simple l'ensemble des surfaces des rectangles* R1, R2 *et* R3. *En cas d'égalité, le programme renvoie la valeur 1 (*TRUE*).*

8.6 SURCHARGE DE L'OPÉRATEUR D'AFFECTATION

La *surcharge* de l'opérateur d'affectation = est un peu plus délicate car il nous faut tester lors de cette opération que les objets ne sont pas identiques ce qui conduirait l'affectation d'un objet à lui-même.

```
#include <iostream.h>

class rectangle {
private:
  double L, l;
public:
  rectangle(double x=2, double y=2.5):L(x),l(y){};
  rectangle(const rectangle&);
  rectangle& operator =(const rectangle& R){
    if (&R != this){
      this->L=R.L;
      this->l=R.l;
    }
    return *this;
  }
  double surface (void){
    return L*l;
  }
};

main()
{
  rectangle R1, R2(2, 5), R3(2.5, 6);
  cout<<"Avant affectation : "<<endl;
  cout<<"R1 = "<<R1.surface()<<endl;
  cout<<"R2 = "<<R2.surface()<<endl;
  cout<<"R3 = "<<R3.surface()<<endl;
  R1=R2;
  R2=R3;
  R3=R1 ;
  cout<<"Apres affectation : "<<endl;
  cout<<"R1 = "<<R1.surface()<<endl;
  cout<<"R2 = "<<R2.surface()<<endl;
  cout<<"R3 = "<<R3.surface()<<endl;
  return 0;
}
```

```
Avant affectation :
R1 = 5
R2 = 10
R3 = 15
Apres affectation :
R1 = 10
R2 = 15
R3 = 10
```

L'opérateur = est ici surdéfini. C'est une fonction membre de la classe rectangle comme nous l'impose le langage C++. Le test if (&R != this) *vérifie que les objets ne sont pas identiques afin d'effectuer correctement l'affectation.*

8.8 SURCHARGE DES OPÉRATEURS D'ENTRÉE-SORTIE

Les deux opérateurs de flux << et >> sont souvent *surchargés* (*surdéfinis*) afin d'avoir des entrées-sorties personnalisées et adaptées au traitement en cours.

Pour effectue cette surcharge, nous allons utiliser des classes déjà existantes au sein du fichier d'en-tête `iostream.h`, ce sont `ostream` pour l'extraction d'un flux et `istream` pour son insertion.

```
#include <iostream.h>

class rectangle{

private:
  double L, l;
public:
  rectangle(double x=0, double y=0){
    L=x; l=y;}
  friend ostream& operator <<(ostream&, const
rectangle&);
```

```
};

ostream& operator <<(ostream& ostr, const
rectangle& r)
{
  cout<<"/Appel de l'operateur << surdefini/"<<
endl;
  return ostr<<(r.L)*(r.l)<<" m2";
}

main()
{
  double L1, l1, L2, l2;
  cout<<"Longueur rectangle r1 : ";
  cin>>L1;
  cout<<"Largeur rectangle r1 : ";
  cin>>l1;
  cout<<"Longueur rectangle r2 : ";
  cin>>L2;
  cout<<"Largeur rectangle r2 : ";
  cin>>l2;
  rectangle r1(L1, l1);
  rectangle r2(L2, l2);
  cout<<r1<<" est la surface de r1"<<endl;
  cout<<r2<<" est la surface de r2"<<endl;
  return 0;
}
```

```
Longueur rectangle r1 : 12.5
Largeur rectangle r1 : 3
Longueur rectangle r2 : 45
Largeur rectangle r2 : 3.5
/Appel de l'operateur << surdefini/
37.5 m2 est la surface de r1
/Appel de l'operateur << surdefini/
157.5 m2 est la surface de r2
```

Par l'intermédiaire d'une fonction amie, l'opérateur de sortie << *est surdéfini au sein de la classe* rectangle. *Il permet de retourner la surface calculée du rectangle et intègre l'affichage de l'unité m2.*

L'exemple suivant montre la surcharge de l'opérateur d'insertion de flux >>.

Nous pouvons remarquer la personnalisation qui est apportée à la saisie des données par cette opération de *surcharge*.

```
#include <iostream.h>

class rectangle{

private:
  double L, l;
public:
  rectangle(double x=0, double y=0){
    L=x; l=y;}
  friend ostream& operator <<(ostream&, const
rectangle&);
  friend istream& operator >>(istream&,
rectangle&);
};

ostream& operator <<(ostream& ostr, const
rectangle& r)
{
  cout<<"/Appel de l'operateur << surdefini/"<<
endl;
  return ostr<<(r.L)*(r.l)<<" m2";
}

istream& operator >>(istream& istr, rectangle& r)
{
  cout<<"/Appel de l'operateur >> surdefini/"<<
endl;
```

```
  cout<<"Longueur : ";
  istr>>r.L;
  cout<<"Largeur : ";
  istr>>r.l;
  return istr;
}

main()
{
  rectangle r1;
  cin>>r1;
  rectangle r2;
  cin>>r2;
  cout<<r1<<" est la surface de r1"<<endl;
  cout<<r2<<" est la surface de r2"<<endl;
  return 0;
}
```

```
/Appel de l'operateur >> surdefini/
Longueur : 12.5
Largeur : 3
/Appel de l'operateur >> surdefini/
Longueur : 45
Largeur : 3.5
/Appel de l'operateur << surdefini/
37.5 m2 est la surface de r1
/Appel de l'operateur << surdefini/
157.5 m2 est la surface de r2
```

Les opérateurs de sortie << *et d'entrée* >> *sont surdéfinis au sein de la classe rectangle.*

Pour l'opérateur de sortie <<, *on retrouve un traitement analogue à celui de l'exemple précédent.*

Une autre fonction amie est définie pour l'opérateur d'entrée >>.

Ce dernier bénéficie d'une invite de saisie pour la longueur et la largeur du rectangle r1 *et* r2.

Comme il a été précisé plus haut tous les opérateurs de C++ peuvent être *surchargés* ou *surdéfinis*. Les quelques exemples présentés donnent une illustration de la méthode à employer.

Chapitre **9**

Héritage, polymorphisme et patrons

Opérateurs, mots-clés et fonctions

`class, template, virtual`

9.1 HÉRITAGE

Afin de réutiliser des composants logiciels pour créer de nouveaux programmes, le langage C++ met à la disposition du programmeur la technique de l'*héritage* aussi appelée *dérivation*.

Une classe d'origine appelée *super-classe* (voir début du chapitre 7 § 7.1) va donner naissance à une *classe dérivée*.

Le mot-clé `public` présent dans la déclaration de la *classe dérivée* précise que les membres publics de la classe d'origine vont devenir des membres publics de la *classe dérivée.*

Pour pouvoir accéder aux membres privés d'une classe d'origine depuis une classe dérivée, nous allons transformer l'accès de type privé (`private`) en type protégé (`protected`).

Le *qualificatif d'accès* (voir chapitre 7 § 7.2) `protected` autorise l'accès aux membres privées pour toute *classe dérivée*.

```
#include <iostream.h>
#include <math.h>

const double PI=3.14;

class cercle
{
public:
  cercle(double ray=5):r(ray){}
  cercle(const cercle& rc):r(rc.r){}
  double srayon();
  void rayon(double r){
    if (r<=0) r=1;
    else r;
  }
  double diametre(){
    return r*2;
  }
  double circonf(){
    return PI*diametre();
  }
  double surf(){
    return PI*pow(r,2);
  }
  void affcercle();
protected:
  double r;
};
class sphere:public cercle
{
public:
  double vol();
  void affsphere();
};
```

```
double cercle::srayon()
{
  cout<<"Rayon : ";
  cin>>r;
  return r;
  }

  void cercle::affcercle()
  {
    cout<<"le cercle a pour :"<<endl;
    cout<<"Diametre : "<<diametre()<<endl;
    cout<<"Circonference : "<<circonf()<<endl;
    cout<<"Surface : "<<surf()<<endl<<endl;
  }

  double sphere::vol()
  {
    return 4.0/3.0*PI*pow(r,3);
  }

  void sphere::affsphere()
  {
    cout<<"la sphere a pour :"<<endl;
    cout<<"Diametre : "<<diametre()<<endl;
    cout<<"Circonference : "<<circonf()<<endl;
    cout<<"Surface : "<<surf()<<endl;
    cout<<"Volume : "<<vol()<<endl;
  }

  main()
  {
    cercle c1;
    cout<<"Par defaut, ";
    c1.affcercle();
    cercle c2;
    c2.srayon();
```

```
    c2.affcercle();
    sphere s1;
    s1.srayon();
    s1.affsphere();
    return 0;
  }
```

```
Par defaut, le cercle a pour :
Diametre : 10
Circonference : 31.4
Surface : 78.5

Rayon : 10
le cercle a pour :
Diametre : 20
Circonference : 62.8
Surface : 314

Rayon : 2.5
La sphere a pour :
Diametre : 5
Circonference : 15.7
Surface : 19.625
Volume : 65.4167
```

La classe sphere *est une classe dérivée de la classe cercle.*

Le membre r *de la classe d'origine cercle possède un qualificatif d'accès* protected *pour pouvoir être manipulé par la classe dérivée* sphere.

La classe cercle *calcule le diamètre, la circonférence et la surface d'un cercle.*

La classe sphere *réutilise les membres de la classe* cercle *et calcule le volume d'une sphère.*

9.2 HÉRITAGE MULTIPLE

L'héritage multiple offre la possibilité pour une classe d'hériter d'une ou plusieurs autres classes.

La syntaxe utilisée est la suivante :

```
class NomClasse : qualifAccès1 NomClasse1, qualifAccès2
NomClasse2,…
```

Le qualificatif d'accès peut être `public`, `private` ou `protected`.

```
#include <iostream.h>

class polygone{
protected:
  double l, h;
public:
  void saisie(double a, double b){
    l=a; h=b;
  }
};

class affiche{
public:
  void print(double s){
    cout<<"Surface : "<<s<<endl;
  }
};

class rectangle:public polygone, public affiche{
public:
  double surf(void){
    return l*h;
  }
};

class triangle:public polygone, public affiche{
public:
```

```
  double surf(void){
    return l*h/2;
  }
};

main()
{
  double largeur, hauteur;
  cout<<"Largeur : ";
  cin>>largeur;
  cout<<"Hauteur : ";
  cin>>hauteur;
  rectangle R;
  triangle T;
  R.saisie(largeur,hauteur);
  T.saisie(largeur,hauteur);
  cout<<"Rectangle - ";
  R.print(R.surf());
  cout<<"Triangle - ";
  T.print(T.surf());
  return 0;
}
```

```
Largeur : 12.5
Hauteur : 5
Rectangle - Surface : 62.5
Triangle - Surface : 31.25
```

Les classes rectangle *et* triangle *héritent des membres de la classe* polygone *et utilisent aussi le membre* print *de la classe* affiche.

9.3 POLYMORPHISME

Lorsque nous créons des objets qui sont des instances de classes dérivées, elles-mêmes instances d'une classe de base, on peut être

amené à vouloir leur appliquer un traitement défini dans un membre de la classe de base.

Cette caractéristique qui permet à des instances d'objets de types différents de répondre de façon différente à un même appel de fonction est le *polymorphisme*.

En C++ un pointeur sur une instance d'une classe de base peut pointer sur toute instance de classe dérivée. Ce sont les *fonctions virtuelles* qui vont nous permettre de réaliser ce traitement. Elles vont réaliser un *lien dynamique*, c'est-à-dire que le type de l'objet ne sera pris en compte qu'au moment de l'exécution et non pas au moment de la compilation comme c'est le cas classiquement (*lien statique*).

Le mot-clé `virtual` est utilisé pour déclarer la fonction membre qui sera gérée par un *lien dynamique*.

```
#include <iostream.h>

class polygone
{
public :
  void saisie(double a, double b){
    l=a; h=b;
  }
  virtual double surf(void){
    cout<<"Appel de la fonction surf de la classe
de base qui renvoie : ";
    return (0);
  }
protected:
  double l, h;
};

class rectangle:public polygone
{
public:
  double surf(void){
    cout<<"Appel de la fonction surf de la classe
de base qui renvoie : ";
```

```
    return l*h;  }
};

class triangle:public polygone
{
public:
  double surf(void){
    cout<<"Appel de la fonction surf de la classe
de base qui renvoie : ";
 return l*h/2;
  }
};

main()
{
  rectangle R;
  triangle T;
  polygone P;
  polygone *ptP1=&R;
  polygone *ptP2=&T;
  polygone *ptP3=&P;
  ptP1->saisie(4,5);
  ptP2->saisie(5,2.5);
  ptP3->saisie(3.5, 2.5);
  cout<<ptP1->surf()<<endl;
  cout<<ptP2->surf()<<endl;
  cout<<ptP3->surf()<<endl;
  return 0;
}
```

```
Appel de la fonction surf de la classe de base qui
renvoie : 20
Appel de la fonction surf de la classe de base qui
renvoie : 6.25
Appel de la fonction surf de la classe de base qui
renvoie : 0
```

La fonction surf *définie dans la classe de base est virtuelle.*

Les appels ptP1->surf(), ptP2->surf() *et* ptP3->surf() *appellent dans l'ordre, les fonctions* rectangle::surf(void), triangle::surf(void) *et* polygone::surf(void).

Les pointeurs sont liés dynamiquement à la fonction surf.

Les appels sont polymorphes car ils fournissent un résultat différent suivant les instances qu'ils manipulent.

Une classe de base qui possède comme membre, au moins une méthode virtuelle est dite *polymorphe*.

9.4 PATRONS

Les *patrons* ou *modèles* sont utilisables sur les fonctions et sur les classes. Ils permettent de condenser le code en offrant la possibilité d'écrire une seule fois la définition d'une fonction ou d'une classe.

9.4.1 Patrons de fonctions

Les *patrons de fonctions* permettent de créer des fonctions génériques qui vont supporter plusieurs types de données différents.

Le mot-clé retenu pour ces *patrons* est `template` suivi de `class` qui spécifie le type.

La syntaxe utilisée est la suivante :

```
template <class T>
```

Le paramètre T est le paramètre de type qui va venir remplacer les types classiques présents dans la définition de la fonction.

```
#include <iostream.h>

template <class T>
void affiche(T *tableau, int n) {
```

```
  for(int i=0;i<n;i++){
    cout<<"Element "<<i<<" : "<<tableau[i]<<endl;
  }
  cout << endl;
}

main()
{
  int entier[6] = {25, 4, 52, 18, 6, 55};
  affiche(entier, 6);
  double decimal[3] = {12.3, 23.4, 34.5};
  affiche(decimal, 3);
  char *chaine[] = {"Tim", "Berners", "Lee"};
  affiche(chaine, 3);
  return 0;
}
```

```
Element 0 : 25
Element 1 : 4
Element 2 : 52
Element 3 : 18
Element 4 : 6
Element 5 : 55

Element 0 : 12.3
Element 1 : 23.4
Element 2 : 34.5

Element 0 : Tim
Element 1 : Berners
Element 2 : Lee
```

Dans la fonction affiche*, le paramètre de type* T *est générique pour* *tableau. *À chaque appel de la fonction affiche le compilateur génère une fonction qui tient compte du type de l'argument passé, entier (*int*), décimal (*double*) ou chaîne de caractères (*char*).*

9.4.2 Patrons de classes

Les *patrons de clas*se sont identiques aux *patrons de fon*ctions. Les fonctions membres d'un *patron de classe* sont aussi des patrons de fonctions qui ont un en-tête de patron identique au *patron de classe*.

Nous allons donc obtenir de cette façon des classes génériques qui vont pouvoir, comme les *patrons de fonctions*, traiter des types de données différents.

```
#include <iostream.h>

template <class T>class rectangle
{
private:
  T L,l;
public:
  rectangle();
  rectangle(T,T);
  void affiche(void);
};

template <class T>rectangle<T>::rectangle(){
  L=0; l=0;
}

template <class T>rectangle<T>::rectangle(T Lg,T
lg){
  L=Lg; l=lg;
}

template <class T>void rectangle<T>::affiche(){
  cout<<"Surface : "<<L*l<<endl;
}

int main()
{
```

```
  float L1=2.5, l1=4.5;
  double L2=5.5, l2=7.25;
  int L3=5, l3=3;
  rectangle<float>s1(L1,l1);
  rectangle<double>s2(L2,l2);
  rectangle<int>s3(L3,l3);
  s1.affiche();
  s2.affiche();
  s3.affiche();
  return 0;
}
```

```
Surface : 11.25
Surface : 39.875
Surface : 15
```

Pour la classe rectangle*, le paramètre de type* T *est générique.*

Lors de l'appel des membres de la classe rectangle*, les types des arguments passés,* float*,* double *et* int *sont pris en compte.*

Conclusion

Comme je l'ai dit au début de cet ouvrage, le contenu de ce livre n'est qu'une introduction au langage C++, de nombreuses notions n'ont pas été évoquées ou décrites.

Vous trouverez dans la bibliographie un ensemble d'ouvrages qui vont bien au-delà des techniques présentées ici et qui vous permettront d'aller plus loin.

Toutefois je vous invite à déjà bien explorer les exemples présentés ici car ils restent une solide approche à l'apprentissage de notions plus complexes.

Les listes suivantes donne quelques adresses[1] de sites Internet dédiés au langage C++.

➤ Sites en langue française

http://www-igm.univ-mlv.fr/~dr/C_CPP_index.html

http://www.iut-bethune.univ-artois.fr/~caron/courscpp/courscpp.html

http://ltiwww.epfl.ch/Cxx/index.html#c2_1

http://www.emse.fr/~boissier/enseignement/c/LATEX-WWW/SEMESTER-II/COURS-CPP/

1. Sous réserve que les liens soient toujours valides.

http://docs.mandragor.org/files/Programming_languages/Cpp/Cours_de_c-cpp_par_Christian_Casteyde_%5Bfr%5D/p144.html

http://www-ipst.u-strasbg.fr/pat/program/cpp.htm

➤ Sites en langue anglaise

http://www.cprogramming.com/tutorial.html

http://www.zib.de/Visual/people/mueller/Course/Tutorial/tutorial.html

http://www.intap.net/~drw/cpp/index.htm

http://www.acm.org/crossroads/xrds1-1/ovp.html

http://www.glenmccl.com/tutor.htm

http://appsrv.cse.cuhk.edu.hk/~csc4510/cxx/tutorial.2/1.htm

http://www.cs.wustl.edu/~schmidt/C++/index.html

http://www.troubleshooters.com/codecorn/crashprf.htm

http://www.research.att.com/~bs/C++.html

Bibliographie

➤ Ouvrage en français

Le langage C++ – BJARNE STROUSTRUP, Pearson Education, 2003.

Visual C++ – Davis Chapman, CampusPress, 2003.

Programmation en C++ – Jean-Paul BODEVEIX, Mamoun FILALI, Amal SAYAH, Dunod, 1997.

Programmation Internet en C et C++ – Kris JAMSA, Ken COPE, International Thomson publishing France, 1996.

Langage C++ – Bruno DUBOIS, ENI, 2000.

Mieux programmer en C++ : 47 problèmes pratiques résolus – Herb SUTTER, Eyrolles, 2000.

Langage C++ – Nino SILVERIO, Eyrolles, 1998.

L'essentiel du C++ – Stanley B. LIPPMAN, Josée LAJOIE, Vuibert, 2000.

Visual C++ 6 – Ivor HORTON, Eyrolles, 1999.

Visual C++ 6 – Davis CHAPMAN, Campus Press, 2004.

C++ – Gerhard WILLMS, Micro Application.

Comment programmer en C++ – Harvey M. DEITEL, Paul J. DEITEL, Reynald GOULET, 2004.

Exercices en langage C++ – Claude DELANNOY, Eyrolles, 2002.

➤ Ouvrages en anglais

The C++ Programming language – Bjarne STROUSTRUP, Addison-Wesley Professional, 2000.

C++, The Complete Reference – Herbert SCHILDT, Osborne/McGraw-Hill, 2002.

Sams Teach Yourself C++ in 21 Days – Jesse LIBERTY, SAMS, 2001.

Object-Oriented Programming in C++, 4th ed. – Robert Lafore, SAMS, 2001.

The C++ Standard Library : A Tutorial and Reference – Nicolai M. JOSUTTIS, Addison-Wesley Professional, 1999.

Learn to Program with C++ – John SMILEY, Bruce NEIGER, Osborne/McGraw-Hill; 2002.

Practical C++ Programming, 2nd ed. – Steve OUALLINE, O'Reilly, 2003.

C++, Pocket Reference – Kyle LOUDON, O'Reilly, 2003.

C/C++, Programmer's Reference, 3rd ed. – Herbert SCHILDT, Osborne/McGraw-Hill, 2002.

Accelerated C++ : Practical Programming by Example – Andrew KOENIG, Barbara E. MOO, Addison-Wesley Professional, 2000.

Data Structures and Algorithms in C++, 2nd ed. – Adam DROZDEK, Brooks/Cole Publishing Company, 2000.

ANNEXES

Annexe **A**

Création d'un projet en mode console avec Visual C++ 6.0

Microsoft Visual C++ utilise pour fonctionner la notion de *project* (projet) et de *workspace* (espace de travail).

Après avoir lancé l'application, allez dans le menu FILE, puis choisissez l'option NEW.

Dans l'onglet PROJECTS de la fenêtre qui apparaît, sélectionnez l'option WIN32 CONSOLE APPLICATION (figure A.1). Puis saisissez dans la zone PROJECT NAME un nom de projet. Choisissez ensuite le répertoire de destination de ce projet en cliquant sur le bouton situé à droite du champ LOCATION. Laissez cocher CREATENEW WORKSPACE et WIN32, puis cliquez sur le bouton OK.

Une fenêtre assistant s'ouvre, choisissez AN EMPTY PROJECT, cliquez sur le bouton FINISH, puis sur le bouton OK de la fenêtre d'avertissement.

Votre environnement de travail doit ressembler à celui de la figure A.2.

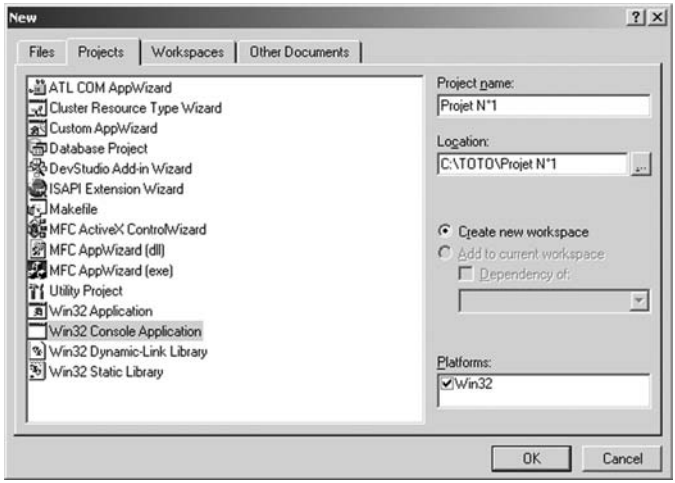


Figure A.1 La fenêtre de l'onglet PROJECTS pour la création d'un projet.

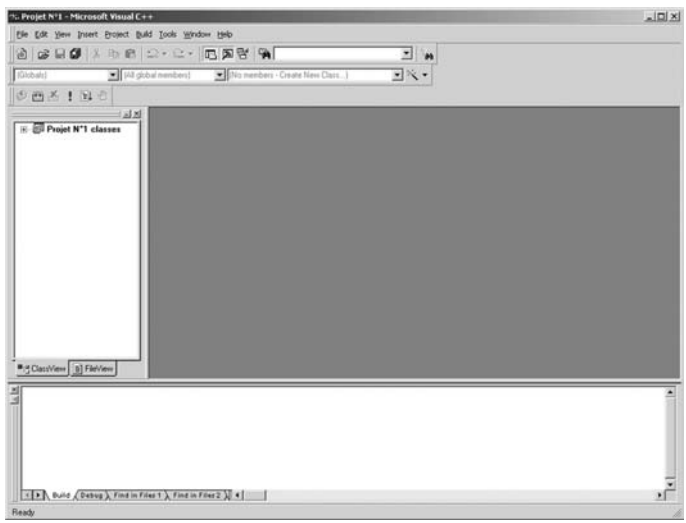


Figure A.2 L'environnement de travail du nouveau projet.

Allez dans le menu FILE, choisissez de nouveau le sous-menu NEW. La fenêtre déjà utilisée tout à l'heure s'ouvre à nouveau mais placée sur l'onglet FILES. Choisissez C++ SOURCE FILE, puis saisissez dans la zone FILE NAME le nom de votre programme (figure A.3). Cliquez ensuite sur le bouton OK.

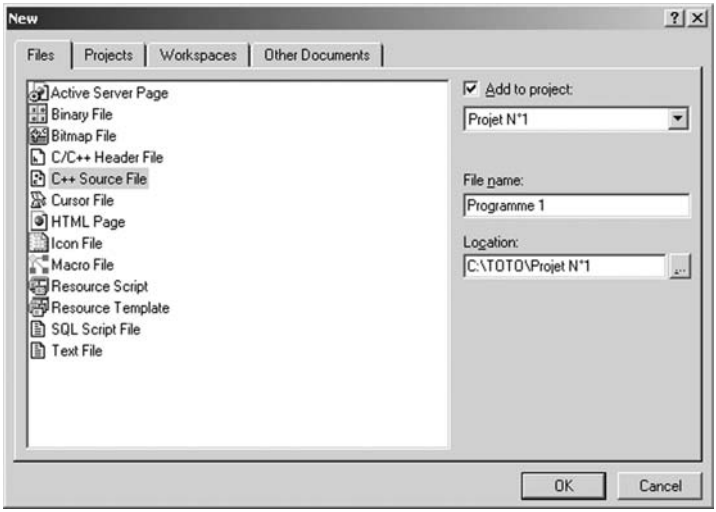


Figure A.3 La fenêtre de l'onglet FILES pour spécifier le programme à créer.

Une fenêtre au nom de votre programme suivi de l'extension cpp s'ouvre. C'est ici que vous allez pouvoir saisir vos lignes de code (figure A.4).

Une fois votre code saisi, appuyez sur la touche de fonction F7 de votre clavier ou allez dans le menu BUILD et choisissez le sous-menu BUILD xxx.EXE.

La compilation démarre, en cas d'erreurs ou d'avertissements la fenêtre inférieure les signale (figure A.5).

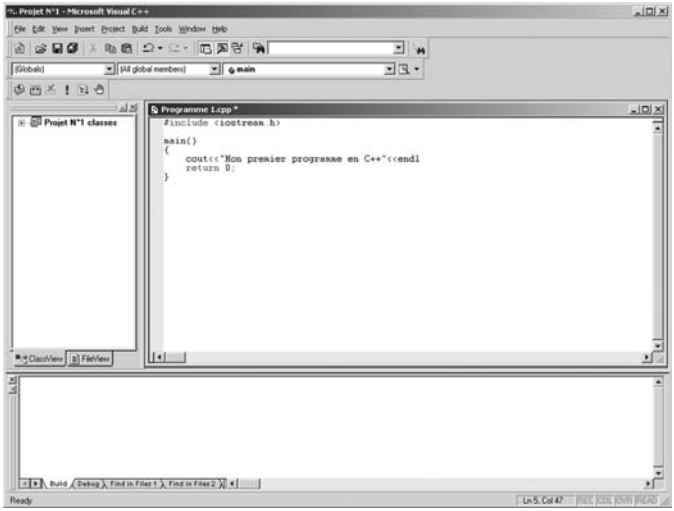


Figure A.4 La fenêtre de saisie du code du programme C++.

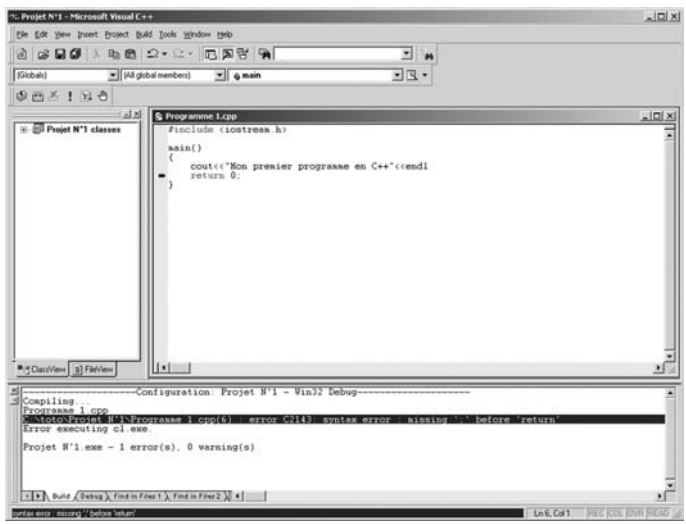


Figure A.5 La fenêtre inférieure indique les erreurs.

Un clic sur la ligne signalant l'erreur ou l'avertissement vous indique la ligne contenant l'erreur dans la fenêtre qui contient le code (*attention l'erreur peut être située dans les lignes de part et d'autre dans certains cas de figure*).

Corrigez la ou les erreurs et relancez la compilation et la construction, à nouveau par la touche F7.

Si plus aucune erreur n'est présente, le *linking* (éditions des liens) se fait et la fenêtre inférieure mentionne : `xxx.exe - 0 error(s), 0 warning(s)` (figure A.6).

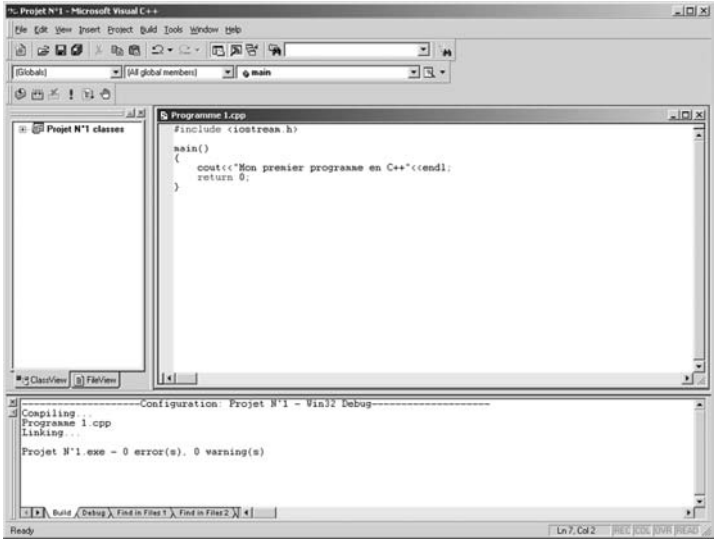


Figure A.6 Le résultat d'une compilation sans erreur.

Appuyez sur les touches CTRL+F5 ou allez dans le menu BUILD et choisissez l'option EXECUTE xxx.EXE, votre programme se lance et s'exécute dans une fenêtre DOS (*Mode console*) (figure A.7).

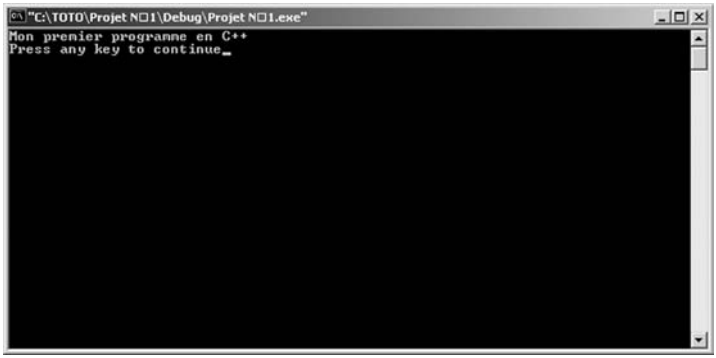


Figure A.7 La fenêtre d'exécution en mode console.

Pour enregistrer vos travaux, fermez la fenêtre DOS d'exécution du programme, allez dans le menu FILE et choisissez CLOSE WORKSPACE, répondez OUI en cliquant sur le bouton de la fenêtre d'avertissement, votre projet et votre espace de travail sont sauvegardés.

Pour en savoir plus sur le fonctionnement de Microsoft Visual C++, consultez la documentation fournie avec le logiciel.

Vous trouverez également dans la bibliographie de cet ouvrage, quelques références de livres dédiés à ce compilateur.

Annexe **B**

Les opérateurs du langage C++

Opérateur	Nom	Priorité*	Surcharge
,	Virgule	0	OUI
++	Post et pré-incrémentation	16 - 15	OUI
--	Post et pré-décrémentation	16 - 15	OUI
throw	Levée d'exception	1	OUI
=	Affectation	2	OUI
+=	Affectation addition	2	OUI
-=	Affectation soustraction	2	OUI
*=	Affectation multiplication	2	OUI
/=	Affectation division	2	OUI
%=	Affectation reste	2	OUI
&=	Affectation ET bit à bit	2	OUI
^=	Affection OU exclusif bit à bit	2	OUI
|=	Affectation OU exclusif	2	OUI
<<=	Affection décalage bit à gauche	2	OUI

Opérateur	Nom	Priorité*	Surcharge
>>=	Affectation décalage bit à droite	2	OUI
?:	Condition	3	NON
||	OU logique	4	OUI
&&	ET logique	5	OUI
|	OU bit à bit	6	OUI
^	OU exclusif bit à bit	7	OUI
&	ET bit à bit	8	OUI
!=	Différent	9	OUI
==	Égal	9	OUI
<	Inférieur	10	OUI
<=	Inférieur ou égal	10	OUI
>	Supérieur	10	OUI
>=	Supérieur ou égal	10	OUI
<<	Décalage bit à gauche	11	OUI
>>	Décalage bit à droite	11	OUI
+	Addition	12	OUI
-	Soustraction	12	OUI
*	Multiplication	13	OUI
/	Division	13	OUI
%	Reste	13	OUI
->*	Sélection membre indirecte	14	OUI
.*	Sélection membre directe	14	NON
Sizeof	Taille	15	NON

Opérateur	Nom	Priorité*	Surcharge
`~`	Complément bit à bit	15	OUI
`!`	NON logique	15	OUI
`+`	Plus	15	OUI
`-`	Moins	15	OUI
`*`	Dé-référence	15	OUI
`&`	Adresse	15	OUI
`New`	Allocation	15	OUI
`Delete`	Désallocation	15	OUI
`()`	Conversion de type	15	OUI
`.`	Sélection membre	16	NON
`->`	Sélection membre indirecte	16	OUI
`[]`	Indexation	16	OUI
`()`	Appel fonction	16	OUI
`()`	Construction type	16	OUI
`()`	Construction de type	16	OUI
`::`	Portée globale	17	NON
`::`	Portée de classe	17	NON

* Nombre de 0 à 17 qui caractérise la priorité de l'opérateur.

Par exemple l'opérateur du produit `*` (multiplication) a une priorité de valeur 13 supérieure à l'opérateur de somme `+` (addition) qui possède une priorité de valeur 12.
Lors de l'évaluation de l'expression `x = 4+5*2`, le produit (13) est prioritaire sur la somme (12), ce qui implique que `x = 14`.

Annexe **C**

Les principales séquences d'échappement

Caractère	Séquence	Code ASCII
Caractère nul	\0	000
Sonnerie	\a	007
Retour arrière	\b	008
Tabulation horizontale	\t	009
Retour à la ligne (LF)	\n	010
Tabulation verticale	\v	011
Nouvelle page (FF)	\f	012
Retour chariot	\r	013
Guillemets (")	\"	034
Apostrophe (')	\'	039
Point d'interrogation (?)	\ ?	063
Antislash (backslash)	\\	092

Les nombres hexadécimaux ont la forme `\xhh` avec `h` représentant un chiffre hexadécimal.

```
\x7      \x07      \xB2      \x6B
```

Beaucoup de compilateurs autorisent l'utilisation des caractères apostrophe (`'`) et point d'interrogation (`?`) dans les chaînes de caractères exprimés comme caractère ordinaire.

Annexe **D**

Les types de données C++

Type	Description	Taille*	Intervalle de valeurs*
`int`	Entier	4 octets	–2 147 483 648 à 2 147 483 647
`short`	Entier court	2 octets	–32 768 à 32 767
`long`	Entier long	4 octets	–2 147 483 648 à 2 147 483 647
`unsigned int`	Entier non signé	2 ou 4 octets	0 à 4 294 967 295
`unsigned short`	Entier court non signé	2 octets	0 à 65 535
`unsigned long`	Entier long non signé		0 à 4 294 967 295
`char`	Caractère simple	1 octet	–128 à 127
`unsigned char`	Caractère non signé	1 octet	0 à 255
`float`	Réel	4 octets	$3.4*10^{-38}$ à $3.4*10^{38}$
`double`	Réel double précision	8 octets	$1.7*10^{-308}$ à $1.7*10^{308}$

Type	Description	Taille*	Intervalle de valeurs*
`long double`	Réel double précision long	10 octets	$3.4*10^{-4932}$ à $3.4*10^{4932}$
`bool`	Booléen	1 octet	0 ou 1 *(Toute valeur différente de 0 est considérée comme égale à 1)*
`void`		Type propre à la fonction ne renvoyant aucune valeur	
`class`		Classe	
`struct`		Structure	
`union`		Union	
`enum`		Énumération	

*La taille peut varier en fonction de la machine et du compilateur.

Annexe **E**

Mots réservés ou mots-clés

Mot-clé	Fonction
`asm`	Pour intégrer du code assembleur
`auto`	Classe de stockage d'une variable
`bool`	Type booléen
`break`	Termine une instruction `switch` ou une boucle
`case`	Contrôle uns instruction `switch`
`catch`	Précise les actions en cas d'exception
`char`	Type caractère (entier)
`class`	Déclare une classe
`const`	Définit une constante
`const_cast`	Opérateur de conversion pour les constantes
`continue`	Saut vers l'itération suivante d'une boucle
`default`	Précise le cas par défaut d'une instruction `switch`
`delete`	Libère (désalloue) la mémoire allouée
`do`	Précise le début d'une boucle

Mot-clé	Fonction
`double`	Type nombre réel
`dynamic_cast`	Opérateur de conversion dynamique
`else`	Alternative de l'instruction conditionnelle `if`
`enum`	Déclare un type énuméré
`explicit`	Définit un transtypage (conversion) explicite
`extern`	Classe de stockage d'une variable
`false`	Valeur booléenne fausse
`float`	Type nombre réel
`for`	Début d'une boucle for
`friend`	Définit une fonction amie pour une classe
`goto`	Saut vers une étiquette
`if`	Définit une instruction conditionnelle
`inline`	Demande le texte de la fonction lors de son appel
`int`	Type entier
`long`	Type entier ou réel long
`mutable`	Rend accessible en écriture un champ de structure constant
`namespace`	Définit un espace de nom
`new`	Alloue de la mémoire
`operator`	Déclare un opérateur surchargé (surdéfini)
`private`	Précise les données privées d'une classe
`protected`	Précise les données protégées d'une classe
`public`	Précise les données publiques d'une classe

Mot-clé	Fonction
`register`	Précise une classe d'objets placés dans des registres
`reinterpret_cast`	Opérateur de conversion
`return`	Retourne une valeur dans une fonction
`short`	Type entier court
`signed`	Type entier signé
`sizeof`	Retourne la taille (nombre d'octets)
`static`	Classe de stockage d'une variable
`static_cast`	Opérateur de conversion
`struct`	Définit une structure
`switch`	Définit une suite d'alternatives
`template`	Définit un patron de fonction
`this`	Pointeur sur l'objet courant
`throw`	Gestion des exceptions
`true`	Valeur booléenne vraie
`try`	Définit un bloc d'instructions pour les exceptions
`typedef`	Définit un alias d'un type existant
`typeid`	Précise le type d'un objet
`typename`	Précise qu'un identificateur inconnu est un type
`union`	Définit une structure à membres multiples
`unsigned`	Type entier non signé
`using`	Précise la référence à des identificateurs d'un espace de noms
`virtual`	Déclare une fonction membre d'une sous-classe

Mot-clé	Fonction
`void`	Précise le type pour des fonctions ne retournant aucune valeur
`volatile`	Déclare des objets modifiables hors programme
`while`	Précise la condition d'une boucle

Annexe **F**

Code ASCII

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
NULL	0	00	Ctrl+@	*Null*	Nul
SOH	1	01	Ctrl+A	*Start of heading*	Début d'entête
STX	2	02	Ctrl+B	*Start of text*	Début de texte
ETX	3	03	Ctrl+C	*End of text*	Fin de texte
EOT	4	04	Ctrl+D	*End of transmit*	Fin de communication
ENQ	5	05	Ctrl+E	*Enquiry*	Demande
ACK	6	06	Ctrl+F	*Acknowledge*	Accusé de réception
BELL	7	07	Ctrl+G	*Bell*	Sonnerie
BS	8	08	Ctrl+H	*Backspace*	Retour arrière
HT	9	09	Ctrl+I	*Horizontal tab*	Tabulation horizontale
LF	10	0A	Ctrl+J	*Line feed*	Interligne
VT	11	0B	Ctrl+K	*Vertical tab*	Tabulation verticale

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
FF	12	0C	Ctrl+L	*Form feed*	Page suivante
CR	13	0D	Ctrl+M	*Carriage return*	Retour en début de ligne
SO	14	0E	Ctrl+N	*Shitf out*	Hors code
SI	15	0F	Ctrl+O	*Shift in*	En code
DLE	16	10	Ctrl+P	*Data line escape*	Échappement en transmission
DC1	17	11	Ctrl+Q	*Device control 1*	Commande auxiliaire n° 1
DC2	18	12	Ctrl+R	*Device control 2*	Commande auxiliaire n° 2
DC3	19	13	Ctrl+S	*Device control 3*	Commande auxiliaire n° 3
DC4	20	14	Ctrl+T	*Device control 4*	Commande auxiliaire n° 4
NAK	21	15	Ctrl+U	*Negative acknowledge*	Accusé de réception négatif
SYN	22	16	Ctrl+V	*Synchronous idle*	Synchronisation
ETB	23	17	Ctrl+W	*End of transmit block*	Fin de bloc transmis
CAN	24	18	Ctrl+X	*Cancel*	Annulation
EM	25	19	Ctrl+Y	*End of medium*	Fin de support
SUB	26	1A	Ctrl+Z	*Substitute*	Remplacement
ESC	27	1B	Ctrl+[	*Escape*	Échappement

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
FS	28	1C	Ctrl+\	*File separator*	Séparateur de fichier
GS	29	1D	Ctrl+]	*Group separator*	Séparateur de groupe
RS	30	1E	Ctrl+^	*Record separator*	Séparateur d'enregistrement
US	31	1F	Ctrl+_	*Unit separator*	Séparateur d'unité
SP	32	20	-	*Space*	Espacement
!	33	21h	-	-	Point d'exclamation
"	34	22h	-	-	Guillemets
#	35	23h	-	-	Dièse
$	36	24h	-	-	Dollar
%	37	25h	-	-	Pourcentage
&	38	26h	-	-	Et commercial
'	39	27h	-	-	Apostrophe
(	40	28h	-	-	Parenthèse ouvrante
)	41	29h	-	-	Parenthèse fermante
*****	42	2Ah	-	-	Astérisque
+	43	2Bh	-	-	Plus
,	44	2Ch	-	-	Virgule
-	45	2Dh	-	-	Moins
.	46	2Eh	-	-	Point

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
/	47	2Fh	-	*Slash*	Barre oblique
0	48	30h	-	-	Chiffre 0
1	49	31h	-	-	Chiffre 1
2	50	32h	-	-	Chiffre 2
3	51	33h	-	-	Chiffre 3
4	52	34h	-	-	Chiffre 4
5	53	35h	-	-	Chiffre 5
6	54	36h	-	-	Chiffre 6
7	55	37h	-	-	Chiffre 7
8	56	38h	-	-	Chiffre 8
9	57	39h	-	-	Chiffre 9
:	58	3Ah	-	-	Deux points
;	59	3Bh	-	-	Point virgule
<	60	3Ch	-	-	Inférieur à
=	61	3Dh	-	-	2gal
>	62	3Eh	-	-	Supérieur à
?	63	3Fh	-	-	Point d'interrogation
@	64	40h	-	-	Arobas
A	65	41h	-	-	Lettre A majuscule
B	66	42h	-	-	Lettre B majuscule
C	67	43h	-	-	Lettre C majuscule
D	68	44h	-	-	Lettre D majuscule

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
E	69	45h	-	-	Lettre E majuscule
F	70	46h	-	-	Lettre F majuscule
G	71	47h	-	-	Lettre G majuscule
H	72	48h	-	-	Lettre H majuscule
I	73	49h	-	-	Lettre I majuscule
J	74	4Ah	-	-	Lettre J majuscule
K	75	4Bh	-	-	Lettre K majuscule
L	76	4Ch	-	-	Lettre L majuscule
M	77	4Dh	-	-	Lettre M majuscule
N	78	4Eh	-	-	Lettre N majuscule
O	79	4Fh	-	-	Lettre O majuscule
P	80	50h	-	-	Lettre P majuscule
Q	81	51h	-	-	Lettre Q majuscule
R	82	52h	-	-	Lettre R majuscule
S	83	53h	-	-	Lettre S majuscule

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
T	84	54h	-	-	Lettre T majuscule
U	85	55h	-	-	Lettre U majuscule
V	86	56h	-	-	Lettre V majuscule
W	87	57h	-	-	Lettre W majuscule
X	88	58h	-	-	Lettre X majuscule
Y	89	59h	-	-	Lettre Y majuscule
Z	90	5Ah	-	-	Lettre Z majuscule
[	91	5Bh	-	-	Crochet ouvrant
\	92	5Ch		*Backslash*	Barre oblique inverse (antislash)
]	93	5Dh		-	Crochet fermant
^	94	5Eh		-	Accent circonflexe
_	95	5Fh		*Underscore*	Souligné
`	96	60h		-	Accent grave
a	97	61h		-	Lettre a minuscule
b	98	62h		-	Lettre b minuscule

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
c	99	63h		-	Lettre c minuscule
d	100	64h		-	Lettre d minuscule
e	101	65h		-	Lettre e minuscule
f	102	66h		-	Lettre f minuscule
g	103	67h		-	Lettre g minuscule
h	104	68h		-	Lettre h minuscule
i	105	69h		-	Lettre i minuscule
j	106	6Ah		-	Lettre j minuscule
k	107	6Bh		-	Lettre k minuscule
l	108	6Ch		-	Lettre l minuscule
m	109	6Dh		-	Lettre m minuscule
n	110	6Eh		-	Lettre n minuscule
o	111	6Fh		-	Lettre o minuscule
p	112	70h		-	Lettre p minuscule
q	113	71h		-	Lettre q minuscule

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
r	114	72h		-	Lettre r minuscule
s	115	73h		-	Lettre s minuscule
t	116	74h		-	Lettre t minuscule
u	117	75h		-	Lettre u minuscule
v	118	76h		-	Lettre v minuscule
w	119	77h		-	Lettre w minuscule
x	120	78h		-	Lettre x minuscule
y	121	79h		-	Lettre y minuscule
z	122	7Ah		-	Lettre z minuscule
{	123	7Bh		-	Accolade ouvrante
|	124	7Ch		-	Tube
}	125	7Dh		-	Accolade fermante
~	126	7Eh		-	Tilde
DEL	127	7F		*Delete*	Effacement

Annexe **G**

Fonctions externes prédéfinies

Dans le tableau suivant, la colonne de droite précise le fichier d'en-tête à la norme ISO/IEC 14882-1998 auquel appartient la fonction.

La plupart des compilateurs acceptent les fichiers d'en-tête à l'ancienne norme. Par exemple `stdio.h` en lieu et place de `cstdio` (voir annexe H).

Fonction	Type	Rôle	Fichier d'en-tête
`abs(i)`	int	Renvoie la valeur absolue de i.	cstdlib
`acos(d)`	double	Renvoie l'arc cosinus de d.	cmath
`asin(d)`	double	Renvoie l'arc sinus de d.	cmath
`atan(d)`	double	Renvoie l'arc tangente de d.	cmath
`atan2(dl, d2)`	double	Renvoie l'arc tangente de d1/d2.	cmath
`atof(s)`	double	Convertit s en nombre en double précision.	cstdlib
`atoi(s)`	int	Convertit s en nombre entier.	cstdlib

Fonction	Type	Rôle	Fichier d'en-tête
`atol(s)`	int	Convertit s en nombre entier long.	cstdlib
`cei1(d)`	double	Renvoie la valeur spécifiée, arrondie à l'entier immédiatement supérieur	cstdlib
`cos(d)`	double	Renvoie le cosinus de d.	cmath
`cosh(d)`	double	Renvoie le cosinus hyperbolique de d.	cmath
`difftime(u1, u2)`	double	Renvoie la différence u1-u2, où u1 et u2 sont des valeurs de temps écoulé depuis une date de référence (voir la fonction time)	ctime
`exit(u)`	void	Ferme tous les fichiers et buffers, et termine le programme. La valeur de u est affectée par la fonction et indique le code retour du programme.	cstdlib
`exp(d)`	double	Élève e à la puissance d. (e =2.7182818... est la base des logarithmes népériens).	cmath
`fabs(d)`	double	Renvoie la valeur absolue de d.	cmath
`close(f)`	int	Ferme le fichier f et renvoie 0 en cas de fermeture normale.	cstdio
`feof(f)`	int	Détermine si une fin de fichier est atteinte. Renvoie dans ce cas une valeur non nulle, et 0 dans le cas contraire.	cstdio

Fonction	Type	Rôle	Fichier d'en-tête
`fegtc(f)`	int	Lit un caractère unique dans le fichier f.	cstdio
`fgets(s,i,f)`	char*	Lit une chaîne s formée de i caractères dans le fichier f.	cstdio
`floor(d)`	double	Renvoie l'arrondi à l'entier immédiatement inférieur de d.	cmath
`fmod(dl,d2)`	double	Renvoie le reste de d1/d2 (avec le signe de di).	cmath
`fopen(sl,s2)`	file*	Ouvre le fichier s1, de type s2. Renvoie un pointeur sur ce fichier.	cstdio
`fprintf(f,...)`	int	Écrit des données dans le fichier.	cstdio
`fputc(c, f)`	int	Écrit un caractère simple dans le fichier f.	cstdio
`fputs(s, f)`	int	Écrit la chaîne s dans le fichier f.	cstdio
`fread(s,il,i2,f)`	int	Lit i2 données de longueur i1 (en octets) depuis le fichier f dans la chaîne s.	cstdio
`free(p)`	void	Libère la zone de mémoire débutant à l'adresse indiquée par p.	cstdlib
`fscanf(f,...)`	int	Lit des données dans le fichier f (se reporter à l'annexe G pour le détail des arguments).	cstdlib

Fonction	Type	Rôle	Fichier d'en-tête
`fseek(f, 1, i)`	int	Décale, de 1 octet à partir de l'adresse i, le pointeur sur le fichier f (où i peut représenter le début ou la fin du fichier, ou la position d'un enregistrement donné).	cstdio
`ftell(f)`	long int	Renvoie la position courante du pointeur dans le fichier f.	cstdio
`fwrite (s, il, i2, f)`	int	Écrit i2 données de longueur i1, depuis la chaîne s dans le fichier f.	cstdio
`getc(f)`	int	Lit un caractère simple dans le fichier f.	cstdio
`getchar`	int	Lit un caractère simple sur l'unité d'entrée standard.	cstdio
`gets(s)`	char*	Lit la chaîne s sur l'unité d'entrée standard.	cstdio
`isalnum(c)`	int	Détermine si l'argument donné est de type alphanumérique. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isalpha(c)`	int	Détermine si l'argument donné est de type alphabétique. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isascii(c)`	int	Détermine si l'argument donné correspond à un code ASCII. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype

Fonction	Type	Rôle	Fichier d'en-tête
`iscntrl(c)`	int	Détermine si l'argument donné correspond à un caractère de contrôle du code ASCII. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isdigit(c)`	int	Détermine si l'argument donné est un chiffre décimal. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isgraph(c)`	int	Détermine si l'argument donné correspond à un caractère graphique imprimable du code ASCII (codes hexa 0x21 à 0x7e, ou octal 041 à 176). Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`islower(c)`	int	Détermine si l'argument donné est un caractère minuscule. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isodigit(c)`	int	Détermine si l'argument donné est un chiffre octal. ctype. h Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isprint(c)`	int	Détermine si l'argument donné correspond à un caractère imprimable du code ASCII (codes hexa 0x20 à 0x7e, ou octal 040 à 176). Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype

Fonction	Type	Rôle	Fichier d'en-tête
`ispunct(c)`	int	Détermine si l'argument est un caractère de ponctuation. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isspace(c)`	int	Détermine si l'argument est un caractère d'espacement. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isupper(c)`	int	Détermine si l'argument donné est un caractère majuscule. Renvoie une valeur non nulle si tel est le cas, nulle sinon.	cctype
`isxdigit(c)`	int	Détermine si l'argument donné est un chiffre hexadécimal.	cctype
`labs(1)`	long int	Renvoie une valeur non nulle si tel est le cas, nulle sinon. Renvoie la valeur absolue de 1.	cmath
`log(d)`	double	Renvoie le logarithme népérien de d.	cmath
`log10(d)`	double	Renvoie le logarithme décimal de d.	cmath
`malloc(u)`	void*	Alloue u octets de mémoire. Renvoie un pointeur sur le début de la zone allouée.	cstdlib
`pow(dl, d2)`	double	Renvoie la valeur de d1 élevée à la puissance d2.	cmath
`printf(...)`	int	Écrit des données sur l'unité standard de sortie.	cstdio

Fonction	Type	Rôle	Fichier d'en-tête
`putc(c, f)`	int	Écrit un caractère dans le fichier f.	cstdio
`putchar(c)`	int	Écrit un caractère sur l'unité standard de sortie.	cstdio
`puts(s)`	int	Écrit une chaîne s sur l'unité standard de sortie.	cstdio
`rand(void)`	int	Renvoie un nombre entier positif aléatoire.	cstdlib
`rewind(f)`	void	Positionne le pointeur sur f en début de fichier.	cstdio
`scanf(...)`	int	Lit des données sur l'unité d'entrée standard stdio.h.	cstdio
`sin(d)`	double	Renvoie le sinus de d.	cmath
`sinh(d)`	double	Renvoie le sinus hyperbolique de d.	cmath
`sqrt( d)`	double	Renvoie la racine carrée de d.	cmath
`srand(u)`	void	Intialise le générateur de nombres aléatoires.	cstdlib
`strempi(s1, s2)`	int	Comparaison de deux chaînes sans distinction des majuscules et minuscules. Renvoie une valeur négative si s1<s2, nulle si s1 et s2 sont identiques et positive dans le cas où s1>s2.	cstring
`strcpy(si, s2)`	char*	Copie de lachaîne s2 dans s1.	cstring
`strlen(s)`	int	Renvoie le nombre de caractères de la chaîne s.	cstring

Fonction	Type	Rôle	Fichier d'en-tête
`strset(s,c)`	char*	Remplace tous les caractères de la chaîne s par c (à l'exception du caractère de fin de chaîne \0).	cstring
`strcmp(sl, s2 )`	int	Comparaison lexicographique de deux chaînes. Renvoie une valeur négative si s1<s2, nulle si s1 et s2sont identiques et positive dans le cas où s1>s2.	cstring
`system(s)`	int	Transmet au système d'exploitation la ligne de commande s. Renvoie 0 si la commande est correctement exécutée, et une valeur non nulle (généralement -1) dans le cas contraire.	cstdlib
`tan(d)`	double	Renvoie la tangente de d.	cmath
`tanh(d)`	double	Renvoie la tangente hyperbolique de d.	cmath
`time(p)`	long int	Renvoie le nombre de secondes écoulées depuis une date spécifiée.	ctime
`toascii(c)`	int	Convertit la valeur de l'argument en code ASCII.	cctype
`tolower(c)`	int	Convertit une lettre en minuscule.	cctype ou cstdlib
`toupper(c)`	int	Convertit une lettre en majuscule.	cctype ou cstdlib

Annexe **H**

Les fichiers d'en-tête de la bibliothèque C++ standard

Le tableau suivant présente les *bibliothèques C++ standard* normalisées.

La colonne de gauche, contient les 18 fichiers d'en-tête qui appartenaient déjà au langage C.

La colonne de droite contient les 32 fichiers d'en-tête C++ de la bibliothèque générique standard STL (*Standard Template Library*).

Note : la dernière norme ISO/IEC 14882-1998 du langage C++ précise les points suivant pour les dénominations des fichiers d'en-tête :

– les 18 fichiers hérités du langage C doivent être précédés de la lettre c. Par exemple `stdio.h` devient `cstdio` ;
– les 32 fichiers C++ doivent abandonnés l'extension .h. Par exemple `iostream.h` devient `iostream`.

Bibliothèque C	Bibliothèque STL	
`cassert`	`algorithm`	`map`
`cctype`	`bitset`	`memory`
`cerrno`	`complex`	`new`

Bibliothèque C	Bibliothèque STL	
cfloat	deque	numeric
ciso646	exception	ostream
climits	fstream	queue
clocale	functional	set
cmath	iomanip	sstream
csetjmp	ios	stack
csignal	iosfwd	stdexcept
cstdarg	iostream	streambuf
cstddef	istream	string
cstdio	iterator	typeinfo
cstdlib	limits	utility
cstring	list	valarray
ctime	locale	vector
cwcar		
cwctype		

Tous les compilateurs n'intègrent pas tous les fichiers d'en-tête, cependant la plupart possèdent les suivants :

– `cctype` ou `ctype.h` : classement des entiers, lettres, majuscules, minuscules etc.
– `climits` ou `limits.h` : valeurs maximum et minimum pour les types de base.
– `cmath` ou `math.h` : fonctions mathématiques.
– `cstdio` ou `stdio.h` : fonctions d'entrées-sorties.
– `cstdlib` ou `stdlib.h` : fonctions standards, fonctions d'allocation mémoire, fonctions de recherche, fonctions de conversion, etc.

– `cstring` ou `string.h` : fonction de traitement des chaînes de caractères, fonctions de manipulation de données en mémoire.
– `ctime` ou `time.h` : fonction de manipulation et de conversion de la date et de l'heure.
– `iostream`, `istream`, `ostream`, `fstream`, `sstream`, `iuomanip`, `ios`, `fstream`, `streambuf`, `strstream` (et/ou sans extension .h) : fonction de gestion des flux d'entrée et de sortie.

Beaucoup de compilateurs possèdent des bibliothèques propriétaires spécifiques qui autorisent la gestion d'environnements de programmation liés à un système d'exploitation et/ou une machine.

Lorsque l'on utilise plusieurs fichiers d'en-tête dans un programme, il peut y avoir un problème de « collision de noms », c'est-à-dire qu'un identificateur porte le même nom dans plusieurs bibliothèques. Pour éviter ce problème on utilise la directive :

```
using namespace std ;
```

derrière l'inclusion des fichiers d'en-tête.

Index

C

D

E

K

L

M

N

O

P

Q

R

S

T

U

V

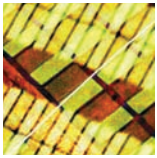
W

047621 - (I) - (2) - OSB 80° - STY-CDD
Dépôt légal Juin 2004
Imprimerie CHIRAT - 42540 Saint-Just-la-Pendue
N° 2353

Imprimé en France

SCIENCES SUP

Série Aide-mémoire



Jean-Michel Réveillac

Aide-mémoire de C++

Cet ouvrage est une initiation à la programmation en C++ ANSI. Il traite de façon didactique, et suivant une progression logique, l'ensemble des fonctionnalités de ce langage. De très nombreux exemples simples ponctuent l'apprentissage de chaque nouvelle notion. Ce livre permet ainsi l'acquisition des mécanismes majeurs et la découverte de tout le potentiel du C++ : les variables, les opérateurs, les fonctions, les structures, les pointeurs, les classes, la programmation objet, l'héritage, les patrons…

Des connaissances basiques des langages de programmation sont conseillées pour aborder cet ouvrage, mais aucun pré-requis lié au langage C ou C++ n'est indispensable.

JEAN-MICHEL RÉVEILLAC

est maître de conférences à l'université de Bourgogne.

MATHÉMATIQUES

PHYSIQUE

CHIMIE

SCIENCES DE L'INGÉNIEUR

INFORMATIQUE

SCIENCES DE LA VIE

SCIENCES DE LA TERRE

1er cycle | 2e cycle | 3e cycle

1 2 3 4 5 6 7 8

LICENCE | MASTER | DOCTORAT

9 782100 076215

ISBN 2 10 007621 3

www.dunod.com

DUNOD