

mini manuel

C++

Jean-Michel Réveillac

→ DUT
→ L1/L2/L3

COURS
+ EXOS
corrigés

DUNOD

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.

Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements



d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée. Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2010
ISBN 978-2-10-055571-0

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2^e et 3^e a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

*À Vanna, mon épouse, merci pour sa patience.
À Léa, ma seconde fille, merci pour sa relecture.*

*À Océane, ma première fille,
Jean-Jacques, un talentueux collègue,
mes amis Anne et Jean Philippe,
merci pour leur soutien.*

Table des matières

Avant-propos	1
Contenu de cet ouvrage	1
Conseils de lecture et d'écriture	2
Introduction	5
Historique	5
En route vers C++	10
1 Expressions arithmétiques en C++	11
1.1 Introduction	11
1.2 Les entiers	14
1.3 Les réels	17
1.4 Règles de conversion implicites	18
1.5 Règles de conversion explicites	20
1.6 Les constantes	21
1.7 Les opérateurs et la priorité des opérations	22
1.8 Cas particulier des opérateurs ++ et --	25
1.9 Récapitulatif	26
<i>Exercices</i>	26
<i>Solutions</i>	27
2 Chaînes et types énumération	29
2.1 Introduction	29
2.2 Représentations et manipulations suivant C	30
2.3 Opérations sur les chaînes	31
2.4 Les chaînes en C++ avec la classe string	32
2.5 Les nouvelles fonctions de la classe string	33

2.6	Les fonctions membres principales de la classe string	36
2.7	Les types énumération	36
2.8	Les types personnalisés	38
2.9	Récapitulatif	38
	<i>Exercices</i>	38
	<i>Solutions</i>	39
3	Tableaux et vecteurs	41
3.1	Première approche	41
3.2	Traitement des tableaux et déclarations	42
3.3	Construction suivant C	43
3.4	Tableaux construits avec la classe vector	46
3.5	Tableaux construits avec la classe valarray	49
3.6	Récapitulatif	50
	<i>Exercices</i>	51
	<i>Solutions</i>	52
4	Structures de contrôle	55
4.1	Introduction	55
4.2	Les instructions d'entrées-sorties	56
4.3	La gestion des cas avec switch	58
4.4	Les boucles	59
4.5	Les contrôles d'itérations	64
4.6	Récapitulatif	67
	<i>Exercices</i>	68
	<i>Solutions</i>	68
5	Pointeurs	71
5.1	Première approche	71
5.2	Déclaration et utilisation	72
5.3	Tableau et pointeur	73
5.4	Deux nouveaux opérateurs, new et delete	75
5.5	Pointeur de pointeur	77
5.6	Déclarations de pointeurs	78
5.7	Récapitulatif	79

<i>Exercices</i>	79
<i>Solutions</i>	80
6 Fonctions	83
6.1 Introduction	83
6.2 Écriture, syntaxe et utilisation	84
6.3 Premier programme utilisant une fonction	86
6.4 Récursivité	89
6.5 Fonction void	90
6.6 Passage par valeur et référence	91
6.7 Surcharge d'une fonction	94
6.8 Tableau et fonction	96
6.9 Particularités et remarques	98
6.10 Fonction exit()	100
6.11 Fonction inline	101
6.12 Récapitulatif	102
<i>Exercices</i>	102
<i>Solutions</i>	103
7 Classes	107
7.1 Introduction	107
7.2 Structure C en C++	108
7.3 Les classes	114
7.4 Constructeurs	118
7.5 Destructeurs	126
7.6 Récapitulatif	127
<i>Exercices</i>	127
<i>Solutions</i>	128
8 La surcharge des opérateurs	133
8.1 Surcharge des opérateurs	133
8.2 Surcharge des opérateurs arithmétiques	134
8.3 La fonction friend	136
8.4 Surcharge des opérateurs relationnels	138
8.5 Le pointeur this	140

8.6	Surcharge de l'opérateur d'affectation	141
8.7	Surcharge des opérateurs d'entrée-sortie	143
8.8	Surcharge des autres opérateurs	144
8.9	Récapitulatif	145
	<i>Exercices</i>	146
	<i>Solutions</i>	146
9	Agrégation, héritage, polymorphisme et patrons	151
9.1	L'agrégation	151
9.2	L'héritage	152
9.3	L'héritage multiple	155
9.4	Le polymorphisme	157
9.5	Les patrons de fonctions	159
9.6	Les patrons de classes	161
9.7	Récapitulatif	162
	<i>Exercices</i>	163
	<i>Solutions</i>	163
	Annexes	167
	1 Compiler en mode console	167
	2 Les principales séquences d'échappement	188
	3 Code ASCII	189
	4 Liste des opérateurs, priorité et arité	194
	5 Mots-clés ou mots réservés	198
	6 La bibliothèque C++ standard	201
	7 Les fonctions externes prédéfinies	203
	Références bibliographiques	209
	Index	211

Avant-propos

CONTENU DE CET OUVRAGE

Cet ouvrage suit une progression logique qui vous fera découvrir le langage C++ pas à pas.

Les termes spécifiques du langage comme les mots-clés, les opérateurs, la terminologie objet... sont mentionnés en gras lors de leur première rencontre dans le texte. Les mots-clés sont ensuite écrits en police courrier et les autres termes en *italique*.

Les exemples et les solutions des exercices sont disponibles en téléchargement sur la page dédiée à l'ouvrage sur le site de Dunod : www.dunod.com

Ce livre est une introduction à C++, il essaie de présenter, de façon claire et précise, sur un peu plus de deux cents pages, les principales fonctionnalités et les concepts fondamentaux du langage.

Écrire un livre aussi concis sur un langage comme C++ est une tâche ardue et difficile, j'ai donc été contraint de faire des choix dans l'approche de certaines fonctionnalités parmi la multitude de possibilités offertes.

C++ est complexe mais l'étude d'exemples simples facilite son apprentissage pour passer ensuite à des programmes plus conséquents. J'ai essayé de rassembler ici les fondements de C++ en considérant que le lecteur possède déjà une expérience de la programmation.

À la suite de votre lecture, si vous voulez approfondir vos connaissances, vous trouverez en fin d'ouvrage une liste de liens et une bibliographie qui vous fourniront les moyens d'aller plus loin.

Les annexes 1 à 7 viennent compléter les notions présentes au sein de chaque chapitre.

CONSEILS DE LECTURE ET D'ÉCRITURE

Chaque chapitre contient de nombreux exemples et plusieurs exercices.

Dans les chapitres 1, 2 et 3, les programmes utilisent des mots-clés, des opérateurs et des traitements qui sont expliqués à partir du chapitre 4, notamment les tests (`if...else`), les boucles (`for`), les opérateurs d'incrémentation et de décrémentation (`++` et `--`), les opérateurs d'insertion et d'extraction (`<<` et `>>`), les appels au flux d'entrée et de sortie (`cin` et `cout`)...

Je vous conseille donc d'essayer les exemples de ces premiers chapitres, mais d'attendre d'avoir lu le chapitre 4 afin de pouvoir réaliser correctement les exercices.

Vous noterez que chaque code source de cet ouvrage est abondamment commenté. Prenez le temps d'analyser et d'étudier en profondeur la structure de chacun des programmes, vous y trouverez des éléments clés.

Il me semble impératif de signaler au lecteur que l'écriture du langage C++, comme beaucoup d'autres, demande de la rigueur.

Essayez d'écrire un code clair et lisible, ne négligez pas la mise en place de commentaires, gardez toujours à l'esprit qu'un programme nécessite souvent une maintenance ou de nouvelles mises à jour.

Au cours de ce livre j'ai essayé de respecter un style de programmation et de présentation du code toujours identique.

Les exemples ont tous été testés sous Microsoft Windows et sous Mac OS-X.

L'édition du code et la compilation ont été réalisées avec les environnements de programmation Code::blocks sous Windows et X-Code sous Mac OS-X (voir annexe 1).



Lors de l'exécution des programmes, il peut apparaître des distorsions de l'affichage du texte dans la fenêtre console, en particulier sur les caractères accentués. Modifiez le code source en supprimant les accents et tout devrait rentrer dans l'ordre.

Dans les exemples et les solutions des exercices, vous pourrez remarquer que le code commence toujours par une ou plusieurs lignes du type « `#include...` », suivies de « `using namespace std;` », ce sont les appels à la bibliothèque standard C++ et à l'espace de nom. Ils sont obligatoires, vous trouverez plus de détails sur leurs utilisations au chapitre 6 et dans l'annexe 6 de cet ouvrage.

Un programme C++ contient toujours une fonction principale « `main` ». Elle est obligatoire. Le code de chaque fonction est contenu dans un bloc d'instructions délimitées par une paire d'accolades `{ }`.

Le bloc d'instructions qui constitue la fonction principale « `main` » se termine par la ligne « `return 0` », afin de préciser que le code s'est terminé sans erreur et que le contrôle est redonné au système.

Les exemples et les solutions des exercices sont commentés. L'écriture d'un commentaire en C++ se fait de deux façons, la ligne peut commencer par `//` (double-slash), dans ce cas il n'est pris en compte que jusqu'à la fin de la ligne ou bien l'ensemble du texte peut être contenu entre `/*` et `*/` (slash-astérisque et astérisque-slash), il peut alors s'étendre sur plusieurs lignes.

Il est recommandé de ne pas inclure de commentaires du second type au sein d'un bloc d'instructions.

Dans l'ensemble des codes sources de ce livre vous ne trouverez que des commentaires de type `//` que je trouve beaucoup plus lisibles.

Introduction

Pour commencer cet ouvrage dédié au langage C++, vous trouverez un historique détaillé.

Il me semble important, lorsque l'on programme avec un langage de connaître son histoire, cela montre les différentes évolutions qui ont permis d'arriver à un outil aussi complexe et puissant que C++.

HISTORIQUE

Le premier langage de programmation de haut niveau qui peut être considéré comme tel est sûrement PLANKALKUL, imaginé par Konrad Suze¹ entre 1942 et 1946. Celui-ci restera sous forme de description textuelle jusqu'en 1975 où il fut décrit et implémenté pour la thèse de Joachim Hohmann. Le premier compilateur utilisant ce langage, a été conçu entre 1998 et 2000. Ce langage était très novateur et construit suivant une syntaxe proche de la notation algébrique.

Avant les années 1950, l'écriture des programmes est difficile, tout est fait en langage binaire, le premier langage de bas niveau, dit « Langage de la première génération ».

Au début des années 1950, l'assembleur apparaît, c'est le second langage de bas niveau dit « Langage de la seconde génération ». Les codes binaires sont remplacés par des mnémoniques plus à faciles à retenir. C'est le précurseur d'une longue lignée de langages impératifs qui décrivent les opérations à exécuter par l'ordinateur comme une séquence d'instructions qui viennent modifier l'état du programme.

Le A-0 system (Arithmatic Language version 0), écrit par Grace Hopper en 1951, et implanté en 1952 sur l'UNIVAC est le premier **compilateur** (ou plutôt dans ce cas : **éditeur de lien – linker**) écrit pour un ordinateur. Il donnera naissance à ARITH-MATIC, MATH-MATIC et FLOW-MATIC en 1955.

¹ Konrad Suze, 22 juin 1910 – 18 décembre 1995, ingénieur allemand pionnier de l'informatique, père de l'ordinateur Z3.

En 1954, John Backus, publie l'article « *Preliminary Report, Specifications for the IBM Mathematical FORMula TRANslating System* » qui jette les bases d'un nouveau langage, FORTRAN, le premier « Langage de la troisième génération », qui est encore utilisé aujourd'hui. FORTRAN était avant tout un langage conçu pour effectuer des calculs scientifiques.

Deux ans après, le premier compilateur pour FORTRAN est opérationnel. Il a été conçu par l'équipe que J. Backus dirige chez IBM pour l'ordinateur IBM 704.

FORTRAN est à la base de nombreux autres langages comme ALGOL (*ALGorithmic Oriented Language*) créé en 1958, ou BASIC¹ (*Beginner's All-purpose Symbolic Instruction Code*) créé en 1963.

Quelques années plus tard, en 1958, le langage LISP, inventé par John McCarthy du MIT (Massachusetts Institute of Technology), est le premier langage fonctionnel.

Il décrit les opérations à réaliser par le programme en une séquence d'instructions exécutées par l'ordinateur, comme un langage impératif, mais chaque opération est une évaluation de fonctions mathématiques qui rejette le changement d'état et la mutation des données. LISP est toujours utilisé aujourd'hui notamment dans les programmes dédiés à l'IA (intelligence artificielle).

Les langages comme le A-0 system, le FLOW-MATIC et le COMTRAN d'IBM créés par Bob Berner² ont inspiré COBOL (*COmmon Business Oriented Language*), langage créé en 1959, par un sous-comité du Short Range Committee (Comité constitué des constructeurs d'ordinateurs, de l'institut national des standards, et des trois agences du gouvernement). Ce langage a été très largement utilisé jusqu'à aujourd'hui, notamment dans les institutions administratives et financières.

C'est mai 1962 qui voit naître, le langage SIMULA (*SIMULATION Language*), développé par Ole-Johan Dahl et Kristen Nygaard du Norwegian Computing Center à Oslo. Il apporte la notion d'objet, de classes, de sous-classes, de méthodes virtuelles... C'est le premier langage orienté objet, il est basé sur ALGOL.

¹ Le BASIC a été crée par John George Kemeny et Thomas Eugene Kurz au Darmouth College pour faire utiliser des ordinateurs par des étudiants présents dans des filières non-scientifiques.

² Robert Berner, 8 février 1920 – 22 juin 2004, informaticien américain, directeur de la programmation chez UNIVAC, co-inventeur du code ASCII en 1961.

En 1968, Seymour Papert et Wally Feurzeig, dans un laboratoire privé de Cambridge, développent le langage LOGO, inspiré du LISP, mais apportant une meilleure lisibilité au niveau de la syntaxe.

L'année 1971, marque l'apparition du langage PASCAL, un des premiers à amener le paradigme de programmation structurée, sous-ensemble de la programmation impérative qui recommande une organisation hiérarchique simple du code suivant des procédures (aussi appelées fonctions ou modules dans d'autres langages) et le bannissement de l'instruction « `goto` », seulement réservée à ce qu'aujourd'hui nous appellerions des exceptions. PASCAL est un langage fortement typé, toutes les variables doivent avoir un type défini.

Le langage C, de Dennis Ritchie et Ken Thompson, apparaît en 1972. Développé, en même temps qu'UNIX dans les laboratoires Bell, il jette les bases de la programmation moderne et devient l'un des langages les plus utilisés.

L'année 1972 marque aussi l'arrivée de PROLOG (PROgrammation LOGique), créé par Alain Colmerauer¹ et Philippe Roussel². Il utilise l'expressivité de la logique pour définir une succession d'instructions que doit exécuter l'ordinateur.

En 1980, Bjarrne Stroustrup, travaille à un langage basé sur SIMULA et C, nommé « C WITH CLASSES » qui donnera naissance, en 1983, à C++. C'est aussi en 1980 que le langage SMALLTALK apparaît. Son développement a débuté en 1972, c'est un langage de programmation orienté objet, réflexif et dynamiquement typé, qui dispose d'un des premiers EDI (Environnement de Développement Intégré) ; il est inspiré de LISP et de SIMULA. Ces concepteurs sont Alan Kay³, Ted Kaehler, Adele Goldberg qui officient au Palo Alto Research Center de Xerox.

L'année 1983 marque, en parallèle de C++, la naissance d'ADA (nom inspiré d'Augusta Ada King, comtesse Lovelace, appelé aussi Ada Lovelace⁴), un langage commandité par le DoD (*Department of Defense*

¹ Alain Marie Albert Colmerauer, 24 janvier 1941, informaticien français, professeur à l'université d'Aix-Marseille, spécialiste du traitement informatique des langues.

² Philippe Roussel, 13 mai 1945, informaticien, consultant en analyse, conception et implémentation de systèmes reposant sur la programmation objet.

³ Alan Kay, 17 mai 1940, informaticien américain, pionnier de la programmation orientée objet (POO) et des interfaces utilisateurs graphiques.

⁴ Ada Lovelace, 10 décembre 1815 – 27 novembre 1852, célèbre mathématicienne, connue pour avoir décrit la machine analytique de Charles Babbage, une machine mécanique qui préfigurait l'ordinateur.

– Département de la défense américaine), et développé par l'équipe de Jean Ichbiah¹ chez CII-Honeywell Bull. ADA est un langage de programmation orienté objet qui offre un haut niveau de fiabilité et de sécurité, souvent destiné aux systèmes embarqués et temps réel.

Toujours en 1983, il faut signaler l'apparition de TURBO PASCAL, EDI pour le langage PASCAL compatible avec les micro-ordinateurs PC et écrit à l'origine par Anders Hejlsberg², pour l'ordinateur Nascom, en 1981.

En 1985, QUICKBASIC de Microsoft, le langage interprété et son EDI sont disponibles sous MS-DOS (Système d'exploitation de Microsoft). CAML (*Categorical Abstract Machine Language*) de l'INRIA (Institut National de Recherche en Informatique et Automatique) fait aussi son apparition.

1986, c'est le langage EIFFEL, orienté objet, de Bertrand Meyer³ qui offre la programmation par contrat (paradigme de programmation dans lequel le déroulement des traitements est régi par des règles appelées « assertions ») et l'héritage de type.

Dans l'histoire des langages, il me semble nécessaire de ne pas omettre les langages de script comme HYPERTALK, développé par Bill Atkinson⁴ et inclus dans l'environnement de programmation HYPERCARD sur Apple Macintosh qui fera son apparition en 1987 et disparaîtra en 2004. La même année Sun va présenter SELF, un autre langage de script qui inspirera le célèbre JAVASCRIPT.

1988 voit naître, TCL, un autre langage de script conçu par John Outerhout et inspiré de C, et LISP. Il sera couplé un peu plus tard au TK pour la gestion des interfaces graphiques. Cette même année PERL est développé par Larry Wall⁵.

Entre 1980 et 1990, Tim Berners Lee⁶ travaille au *World Wide Web* et le langage HTML (*HyperText Markup Language*) est développé sur les

¹ Jean Ichbiah, 25 mars 1940 – 26 janvier 2007, informaticien français spécialiste des langages et des systèmes chez CII Honeywell Bull, membre de l'académie des sciences.

² Anders Hejlsberg, programmeur danois, membre fondateur de la société Bordland, qu'il quitte en 1996 pour aller chez Microsoft travaillé sur J++.

³ Bertrand Meyer, informaticien français, fondateur de la société de services ISE (*Interactive Software Engineering*), théoricien, souvent cité, spécialiste de la programmation objet.

⁴ Bill Atkinson, informaticien américain, développeur de programme sous Mac OS, dont les célèbres MacPaint et Hypercard, ainsi que la célèbre bibliothèque graphique QuickDraw.

⁵ Larry Hall, né le 27 septembre 1964, informaticien américain, linguiste à l'origine du langage Perl qu'il perfectionne depuis plus de vingt ans.

⁶ Timothy John Berners Lee, né le 8 juin 1955, informaticien anglais, co-inventeur avec Robert Cailliau du *World Wide Web*, président du W3C (*World Wide Web Consortium*).

bases de SGML (*Standard Generalized Markup Language*, créé par Charles Goldfarb¹, Edward Mosher et Raymond Lorie, en 1969 chez IBM et standardisé² en 1986). Ce langage est dédié aux navigateurs internet depuis Mosaïc³ jusqu'à Google Chrome aujourd'hui. Depuis sa création, il n'a cessé d'évoluer en fonction du *World Wide Web* pour arriver à la version 4.0. Depuis début 2008, une spécification HTML 5.0 est à l'étude.

En 1991, le langage PYTHON de Guido van Rossum⁴ voit le jour, il est multi-paradigmes favorisant la programmation impérative, structurée et orienté objet. Il est construit sur des concepts liés au C et à MODULA. Apple lance APPLESCRIPT en 1993, issu du projet HYPERCARD⁵. Cette année donne aussi naissance à RUBY, langage lui aussi multi-paradigmes, interprété et libre, créé par Yukihiro Matsumoto⁶.

1995 est une année phare et riche avec PHP, de Rasmus Lerdorf⁷, de DELPHI chez Borland, de JAVA de chez Sun et de JAVASCRIPT.

En 1999, GAMBAS (*Gambas Almost Means BASic*) est conçu par Benoît Minisini⁸. C'est un BASIC enrichi de fonctions graphiques pour Linux.

Depuis notre entrée dans le XXI^e siècle, peu de langages nouveaux sont apparus, les langages comme le C, le C++ et le JAVA règnent en maître, cependant Walter Bright⁹ avec sa société Digital Mars développe le langage D, qui fait suite aux langages C et C++, tout en apportant une syntaxe très épurée et un ramasse-miettes¹⁰ (système de gestion automatique de la mémoire). D est un langage impératif orienté objet qui utilise des éléments de la programmation par contrat déjà citée plus haut.

¹ Charles Goldfarb, informaticien, chef de projet chez IBM ou il a créé le langage descriptif GML (*Generalized Markup Language*). Il quittera rapidement IBM pour développer SGML.

² ISO 8879 :1986

³ Premier navigateur web, multi-plateformes, développé au NCSA (National Center for Supercomputing Applications).

⁴ Guido van Rossum, créateur principal du langage Python et du navigateur web Grail. Travaille aujourd'hui pour Google.

⁵ Environnement de programmation de la société Apple, très graphique et très flexible, intégrant le langage de script hypertext. Il existera jusqu'en mars 2004 sur les systèmes Mac OS.

⁶ Yukihiro Matsumoto, né le 4 avril 1965, informaticien japonais, surnommé « Matz ». Fervent adepte des logiciels libres.

⁷ Rasmus Lerdorf, né le 22 novembre 1968, programmeur danois et canadien, créateur de PHP avec Andi Gutmans et Zeev Suraski, aujourd'hui ingénieur chez Yahoo.

⁸ Benoît Minisini, né en 1973, développeur français.

⁹ Walter Bright, informaticien américain qui a participé au développement de Zortech C et créé des compilateurs pour ECMAScript et Java.

¹⁰ On dit aussi : récupérateur de mémoire, glaneur de cellules ou *garbage collector* en anglais.

J'arrêterais là cet historique, je suis conscient d'avoir oublié ou passé sous silence de nombreux langages, cependant j'ai essayé de présenter les plus représentatifs et les principaux.

EN ROUTE VERS C++

Maintenant que vous en savez un peu plus sur l'histoire et la naissance de C++, nous allons pouvoir entrer réellement dans le vif du sujet.

N'hésitez pas à modifier les exemples et à faire vos propres essais et expériences.

Le meilleur moyen pour maîtriser un langage reste avant tout la pratique, et passe par l'écriture de nombreuses lignes de code.

Chaque erreur rencontrée et corrigée vous apportera un peu plus d'expérience.

Si vous n'avez encore pas choisi d'outils pour rédiger et essayer vos futurs codes sources C++, vous trouverez en annexe 1, une petite introduction à l'utilisation de plusieurs environnements de développement intégré (EDI).



Expressions arithmétiques en C++

PLAN

- 1.1 Introduction
- 1.2 Les entiers
- 1.3 Les réels
- 1.4 Les règles de conversion implicites
- 1.5 Les règles de conversion explicites
- 1.6 Les constantes
- 1.7 Les opérateurs et la priorité des opérations
- 1.8 Récapitulatif

OBJECTIFS

- Connaître les différents types pris en compte par C++
- Utiliser les opérateurs
- Appliquer les bonnes règles de conversion

1.1 INTRODUCTION

Tout langage de programmation manipule des **variables**.

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

Pour associer une valeur à une *variable* on utilise le mécanisme d'**affection**. En C++, l'*opérateur d'affection* est le signe =.

Une *variable* possède obligatoirement un *type* qui définit l'encombrement mémoire que prendra la *variable*.

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**, **signed**, **unsigned** peuvent enrichir les *types* de données.

En C++, le compilateur doit être informé des *types* de *variables* qui seront utilisés dans le programme, pour ce faire, le développeur va donc faire une **déclaration**.

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

L'*identificateur* d'une *variable* est composé d'un ensemble de chiffres ou de lettres dans un ordre quelconque qui suit les règles suivantes :

- Le premier caractère doit être obligatoirement une lettre.
- Les minuscules ou les majuscules (la **casse**) sont autorisées et considérées comme différentes.
- Le caractère _ (underscore ou blanc souligné) est autorisé.
- Il faut veiller à ne pas utiliser des **mots-clés** ou **mots réservés** du langage C++ (voir annexe 5) ou des **séquences d'échappement** (voir annexe 2)

Voici quelques exemples d'*identificateurs* admis :

X	x	X1a
somme_totale		X1a
taux	N5	_montant
Total	N5	PRODUIT

La *déclaration* d'une *variable* peut être assimilée à la création en mémoire d'un contenant dont le *type* serait la taille (ou dimension) et le contenu, la valeur.

Quand aucune valeur n'a encore été affectée à une *variable*, seule sa place est réservée. Son contenu n'étant pas encore défini, il viendra ultérieurement.

Au cours de la rédaction d'un programme, une *variable* peut être déclarée à tout moment, toutefois, les développeurs ont pour habitude de regrouper les déclarations, ce qui apporte une meilleure lisibilité et une compréhension accrue lors de la lecture.

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

L'endroit où est déclarée une *variable* définit son **accessibilité** ou sa **visibilité**.

Quand une *variable* est déclarée dans le code même, c'est-à-dire à l'extérieur d'un bloc d'instructions ou d'une fonction, elle est considérée comme une **variable globale** et reste accessible depuis n'importe quel endroit du code.

Une *variable* déclarée dans un bloc d'instruction voit sa portée limitée à l'intérieur de ce bloc, on parle de **variable locale**. En C++, un bloc d'instructions est délimité par des accolades.

Il existe un opérateur de résolution de **portée** qui offre l'accès à des *variables globales* plutôt que *locales*, il se note ::.



PORTEE DES VARIABLES

```
#include <iostream>
using namespace std;
//déclarations et initialisations des variables
globales e et i
double e=2.718 ;
int i=1;
//programme principal
int main(){
    //déclarations et initialisations des variables
    locales j et e
    int j=2;
    double e=1.282 ;
    //somme e:globale + e:locale
    e=::e+e;
    //affichage des résultats
    cout<<"La somme e = "<<e<<endl;
    cout<<"La variable locale j = "<<j<<endl;
    cout<<"La variable globale i = "<<i<<endl;
    cout<<"la variable globale e = "<<e<<endl;
    cout<<"La somme i + e (avec i:globale et e:locale)
est égale à : "<<i+e<<endl;
}
```

Résultat après exécution :

```
La somme e = 4
La variable locale j = 2
La variable globale i = 1
```

```
la variable globale e = 2.718
La somme i + e (avec i:globale et e:locale) est égale à
: 5
```

1.2 LES ENTIERS

Ils s'expriment sous trois *types* différents : **short int** (entier court), **int** (entier simple) et **long int** (entier long).

Chaque *type* peut être **signed** (signé) ou **unsigned** (non signé).

Il faut rajouter au *type* entier le *type char* (caractère) qui est aussi un *type* entier. Il manipule des caractères comme des entiers à l'intérieur d'un programme, chacun est pris en compte sous sa valeur **ASCII** (voir annexe 3).

Tableau 1-1 ENSEMBLE DES COMBINAISONS POSSIBLES EN C++ POUR LES ENTIERS.

LES PLAGES PRÉCISÉES PEUVENT VARIER EN FONCTION DE LA MACHINE ET DU COMPILATEUR.

Type	Plage couverte		Taille en octets
	Limite inférieure	Limite supérieure	
char	-128	127	1
unsigned char	0	255	1
short int	-32 768	32767	2
unsigned short int	0	65535	2
int	-2 147 483 648	2 147 483 647	4
unsigned int	0	4 294 967 295	4
long int	-2 147 483 648	2 147 483 647	4
unsigned long int	0	4 294 967 295	4

Le fichier d'en-tête¹ `<climits>` défini l'étendue des nombres entiers.


MANIPULATION DES ENTIERS

```
#include <iostream>
using namespace std;
int main ()
{
    //initialisation des variables
    char char_std=65;
```

¹ Voir annexe 6.

```
unsigned char char_uns=65;
short int int_short=1;
int int_std=1;
long int int_long=1;
unsigned short int int_short_uns=1;
unsigned int int_std_uns=1;
unsigned long int int_long_uns=1;
//affichage des variables après initialisation
cout<<char_std<<" "<<char_uns<<" "<<endl;
cout<<int_short<<" "<<int_std<<" "<<int_long
<<" ";
cout<<int_short_uns<<" "<<int_std_uns<<" "
<<int_long_uns << endl;
//affichage de la taille en octets prise par chaque
variable
cout<<sizeof(int_short)<<" "<<sizeof(int_std)<<" "
<<sizeof(int_long)<<" ";
cout<<sizeof(int_short_uns)<<" "
<<sizeof(int_std_uns)<<" "
<<sizeof(int_long_uns)<<endl;
cout << endl;
//boucle de 1 à 32 affichant la valeur de chaque
variable suivant une suite croissante des puissances de
2
for (int i=0; i<=32; ++i)
{
    int_short=int_short*2;
    int_std=int_std*2;
    int_long=int_long*2;
    int_short_uns=int_short_uns*2;
    int_std_uns=int_std_uns*2;
    int_long_uns=int_long_uns*2;
    //affichage de l'indice de la boucle
    cout<<"ind : "<<i<<endl;
    //affichage des valeurs calculées pour
    chaque variable
    cout<<"short int : "<<int_short<<endl;
    cout<<"int : "<<int_std<<endl;
    cout<<"long int : "<<int_long<<endl;
    cout<<"unsigned short int : "
    <<int_short_uns<<endl;
    cout<<"unsigned int : "<<int_std_uns<<endl;
    cout<<"unsigned long int : "
    <<int_long_uns<<endl;
    cout<<endl;
}
return(0);
}
```

Résultat après exécution :

```
A A
1 1 1 1 1 1
2 4 4 2 4 4

ind : 1
short int : 2
int : 2
long int : 2
unsigned short int : 2
unsigned int : 2
unsigned long int : 2

ind : 2
short int : 4
int : 4
long int : 4
unsigned short int : 4
unsigned int : 4
unsigned long int : 4

.
.
.

ind : 31
short int : 0
int : -2147483648
long int : -2147483648
unsigned short int : 0
unsigned int : 2147483648
unsigned long int : 2147483648

ind : 32
short int : 0
int : 0
long int : 0
unsigned short int : 0
unsigned int : 0
unsigned long int : 0
```

Remarques. La manipulation des *variables* de type `char` comme des entiers au sein de C++ peut sembler bizarre, cependant de nombreuses applications manipuleront ainsi très facilement des données au format 8 bits ce qui est intéressant pour traiter des caractères de façon simple.

Il faut aussi noter que le type `char` peut être aussi `unsigned` (non signé), comme un `int`.



1.3 LES RÉELS

Ils s'expriment sous trois types différents : **float** (flottant), **double** et **long double**.

Tableau 1-2 ENSEMBLE DES COMBINAISONS POSSIBLES EN C++ POUR LES RÉELS.
LES PLAGES PRÉCISÉES PEUVENT VARIER EN FONCTION DE LA MACHINE ET DU COMPILATEUR.

Type	Plage couverte		Taille en octets
	Limite inférieure	Limite supérieure	
float	-3.4×10^{-38}	3.4×10^{38}	4
double	-1.7×10^{-308}	1.7×10^{308}	8
long double	-3.4×10^{-4932}	3.4×10^{4932}	10

Le fichier d'en-tête `<cfloat>` définit l'étendue des nombres réels.



MANIPULATION DES RÉELS

```
#include <iostream>
using namespace std;
int main() {
    //initialisation de la boucle et des variables
    for(int i=8; i<=65536; i=i+256)
    {
        //calculs
        float flt_std=1.0/i;
        double flt_double=1.0/i;
        long double flt_long=1.0/i;
        //affichage des valeurs calculées
        cout<<"float : "<<flt_std<<endl;
        cout<<"double : "<<flt_double<<endl;
        cout<<"long double : "<<flt_long<<endl;
        cout<<"----- "<<endl;
    }
    return 0;
}
```

Résultat après exécution :

```
float : 0.125
double : 0.125
long double : 0.125
-----
```

```
float : 0.00378788
double : 0.00378788
long double : 0.00378788
-----
float : 0.00192308
double : 0.00192308
long double : 0.00192308
-----
float : 0.00128866
double : 0.00128866
long double : 0.00128866
.
.
.
float : 1.54378e-05
double : 1.54378e-05
long double : 1.54378e-05
-----
float : 1.5377e-05
double : 1.5377e-05
long double : 1.5377e-05
-----
float : 1.53168e-05
double : 1.53168e-05
long double : 1.53168e-05
-----
```

1.4 RÈGLES DE CONVERSION IMPLICITES

Dans une expression mathématique, où les *opérandes* sont de *types* différents, C++ applique quelques règles de *conversion* dites **implicites** car elles sont déterminées par le compilateur :

- Si un des opérandes est de type `long double`, les autres seront aussi convertis en `long double`.
- Si un des opérandes est de type `double`, les autres seront convertis en `double`.
- Si l'un des opérandes est de type `float`, les autres seront convertis en `float`.
- Si l'un des opérandes est de type `unsigned long` les autres seront convertis en `unsigned long`.
- Lorsqu'un des opérandes est de type `long int` et l'autre de type `unsigned int` alors si le `long int` peut représenter toutes les valeurs du

unsigned int, l'unsigned int sera converti en long int, dans l'autre cas, les deux seront convertis en unsigned long int.

- Si un des opérandes est de type long, les autres seront convertis en long.
- Si un des opérandes est unsigned, les autres seront convertis en unsigned.



CONVERSION IMPLICITES

```
#include <iostream>
using namespace std;
int main () {
    //déclarations des variables
    int x=10.125;
    float y=10.125;
    int z=2;
    //affichage de la variable x qui est convertie
    implicitement en nombre entier
    cout<<"x = "<<x<<endl;
    x=x+0.5;
    //attention, ce n'est pas un arrondi, seule la
    partie entière est conservée
    cout<<"x + 0.5 = "<<x<<endl;
    //affichage de la variable y, nombre réel
    cout<<"y = "<<y<<endl;
    //affichage de la variable z, nombre entier
    cout<<"z = "<<z<<endl;
    //affichage du calcul y + z, le résultat est un
    nombre réel par conversion implicite
    cout<<"y + z = "<<y+z<<endl;
    return 0;
}
```

Résultat après exécution :

```
x = 10
x + 0.5 = 10
y = 10.125
z = 2
y + z = 12.125
```

1.5 RÈGLES DE CONVERSION EXPLICITES

Il existe des règles de conversion **explicites**, on les appelle souvent opérations de **cast**. Elles consistent en une modification du *type* de donnée forcée par le programmeur. On utilise pour cela un opérateur de *cast*, qui reprend tout simplement le *type* de la donnée. Il est là pour spécifier la conversion à réaliser.

```
x = (int)2.718;
```

C++ gère aussi une **notation fonctionnelle**, avec la syntaxe suivante :

```
x = int(2.718);
```



CONVERSION EXPLICITES

```
#include <iostream>
using namespace std;
int main () {
    //déclaration des variables
    int x;
    char y;
    float z;
    //conversion explicite
    x=(int)('A');
    y='A';
    //affichage de x et y
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
    //conversion explicite et affichage
    z=(int) 2.718;
    cout<<"z = "<<z<<endl;
    //conversion explicite et affichage
    z=int(2.718);
    cout<<"z = "<<z<<endl;
    return 0;
}
```

Résultat après exécution :

```
x = 65
y = A
z = 2
z = 2
```



Remarque. Dans certains cas, les calculs avec les *réels* présentent des erreurs d'arrondi, il faut donc rester prudent.

1.6 LES CONSTANTES

Lors de l'écriture d'un programme, nous sommes souvent amenés à utiliser des valeurs fixes. Ces dernières peuvent être affectées définitivement à une *variable* via l'utilisation de **constantes**.

C++ est capable de manipuler plusieurs *types* de *constantes* : entière, réelle, caractère, chaîne et énumération.

Une *constante* est signée ou non. Lorsqu'elle est composée d'un caractère unique, celui-ci est précisé entre apostrophes (simple-quote). Dans le cas d'une chaîne de caractères, c'est un nombre quelconque de caractères encadrés par des guillemets (apostrophes doubles ou double-quotes).

Le mot-clé pour définir une *constante* est le *spécificateur const*.

Si une tentative de modification de *constante* a lieu, elle provoque un message d'erreur du compilateur.

Une *constante* ne peut pas être placée en tant que membre gauche d'une affectation, ni être passée en paramètre de référence non constant d'une fonction. Un opérateur de *cast explicite* peut modifier une *constante*, mais c'est souvent une source d'incohérence.

Les *constantes* sont par défaut internes, mais on peut leur donner un lien externe en les déclarant *explicitement* externe¹.

Nous verrons un peu loin que C++ étend *const* aux *fonctions membres* et aux *classes*.

MANIPULATION DES CONSTANTES

```
#include <iostream>
using namespace std;
int main () {
    //déclarations des constantes
    const float e=2.718;
    const double PI=3.14159265;
    const char beep='\b';
    const float e2=e*2;
    //affichage des constantes
    cout<<"e = "<<e<<endl;
```

¹ Voir le modificateur : *extern*.

```

cout<<"PI = "<<PI<<endl;
cout<<"Beep!"<<beep<<endl;
cout<<"2 * e = "<<e2<<endl;
//opération de cast explicite sur la constante e
cout<<"int(e) = "<<int(e)<<endl;
return 0;
}

```

Résultat après exécution :

```

e = 2.718
PI = 3.14159
Beep!
2 * e = 5.436
int(e) = 2

```

1.7 LES OPÉRATEURS ET LA PRIORITÉ DES OPÉRATIONS

Le langage C++ a une véritable prédilection pour les calculs mathématiques et les *opérateurs* qu'il peut manipuler sont nombreux. Il respecte la **priorité** ou la **préséance** des opérateurs et les règles d'**associativité**. La **priorité** ou la **préséance** fait référence à l'ordre d'application des *opérateurs*, par exemple dans l'expression $(a - b * c)$, l'opérateur $*$ est le premier évalué (priorité 13) et l'opérateur $-$ est le second (priorité 12). L'**associativité** peut être exprimée de droite à gauche ou de gauche à droite. L'emploi de parenthèses permet d'outrepasser les règles de priorité, en forçant le calcul de l'expression qu'elles contiennent. Par exemple l'expression $(a - b - c)$ est évaluée $((a - b) - c)$ car l'opérateur $-$ est associatif à gauche.

Si des opérateurs ont une même priorité, c'est l'**associativité** de l'opérateur qui détermine le sens d'évaluation de l'expression. L'annexe 4 rassemble les éléments nécessaires à la compréhension et la mise en œuvre des *opérateurs*, de leurs *priorités* et de leurs *associativités*.

La diversité des *opérateurs* du langage C++ fait sa richesse et sa puissance. On peut dénombrer les familles distinctes suivantes :

- Opérateur **arithmétique**
- Opérateur **binaire** ou **bit à bit**
- Opérateur de **décalage**
- Opérateur d'**expression conditionnelle**
- Opérateur d'**affectation** ou d'**assignation**
- Opérateur **logique** ou **booléen**

- Opérateur **unaire**
- Opérateur d'**insertion**
- Opérateur d'**extraction**
- Opérateur **relationnel**

Parmi l'ensemble des *opérateurs arithmétiques* précédents il peut vous sembler bizarre qu'il n'y est pas l'exponentiation, c'est tout à fait normal, en C++ c'est une fonction externe¹ qui remplit ce rôle.

Le programme qui suit rassemble différents cas d'utilisation des *opérateurs* de C++. Il est abondamment commenté pour vous permettre de mieux comprendre les traitements effectués.



UTILISATION DES OPÉRATEURS

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation des variables
    int a=100, b=7, c=100, DIV=-69, div=10,
        x=1;

    //quelques opérations arithmétiques

    //100+7=1007
    cout<<"a+b="<<a+b<<endl;
    //100-7=93
    cout<<"a-b="<<a-b<<endl;
    //100x7=700
    cout<<"a*b="<<a*b<<endl;
    // -7
    cout<<"-b="<<-b<<endl;
    //division entière
    //-69/10=-6
    cout<<"DIV/div="<<DIV/div<<endl;
    //Reste de 69/10=-9
    cout<<"modulo(DIV/div) ="<<DIV%div<<endl;
    //décrémentation, incrémentation pré ou post
    //a pré-incrémenté : il est incrémenté
    cout<<"++a="<<++a<<endl;
    //puis placé dans le flux de sortie
    cout<<"a="<<a<<endl;
    //b est post-incrémenté : il reste avec sa valeur
    //initiale
    cout<<"b++="<<b++<<endl;
```

¹ Fichier d'en-tête `<cmath>`, voir annexe 7, fonction `pow(d1, d2)`.

```

//puis est incrémenté et placé dans le flux de
sortie
cout<<"b="<<b<<endl;
//maintenant a=101 et b=8
//opérateur conditionnel
//en fonction de la condition, on a 1 ou 0
//avec a=101 et b=8
//a est n'est pas inférieur à b alors 0
cout<<"a<b -> "<<((a<b)?1:0)<<endl;
//a est supérieur à b alors 1
cout<<"a>b -> "<<((a>b)?1:0)<<endl;
//a n'est pas égal à b alors 0
cout<<"a==b -> "<<((a==b)?1:0)<<endl;
//a est différent de b alors 1
cout<<"a!=b -> "<<((a!=b)?1:0)<<endl;
//opérateurs logiques
//avec a=101 et b=8
//si a < b et b < c alors vrai sinon faux
cout<<((a<b)&&(b<c)?»vrai»:"faux")<<endl;
//si a < b et a > c alors vrai sinon faux
cout<<((a<b)|| (a>c)?»vrai»:"faux")<<endl;
//si a < b et b >= 100 alors x=x-a-1 sinon x=x-c/-b
soit x=1-(101/-8) => x=1-(-12) => x=13
x-=(a<b)&&(b>=100)?-a:c/-b;
cout<<"x="<<x<<endl;
return 0;
}

```

Résultat après exécution :

```

a+b=107
a-b=93
a*b=700
-b=-7
DIV/div=-6
modulo(DIV/div)=-9
++a=101
a=101
b++=7
b=8
a<b -> 0
a>b -> 1
a==b -> 0
a!=b -> 1
faux
vrai
x=13

```



Remarques. Lors de l'utilisation de l'opérateur de division entière, il peut survenir des problèmes d'arrondis notamment dans la manipulation de nombres négatifs, toutefois C++ s'en tient à la règle mathématique qui dit que le quotient multiplié par le diviseur plus le reste donne le dividende.

Attention à prendre en compte ce que j'ai expliqué plus haut, c'est dire la *conversion implicite de type* pour que le système puisse évaluer correctement le résultat et fournir la plus grande précision possible.

Les *opérateurs* vus précédemment feront l'objet de précisions et d'explications plus détaillées dans les chapitres suivants.

1.8 CAS PARTICULIER DES OPÉRATEURS ++ ET --

Ces deux *opérateurs unaires* qui respectivement augmentent ou diminuent de 1 la valeur de leur *opérande* peuvent être écrits sous deux formes.

Ils suivent ou ils précèdent leur *opérande*, on parle alors de **post** ou de **pré-incrémantion ou décrémantion**.

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*. La différence peut être significative, il faut donc utiliser ces *opérateurs* en connaissance de cause.



PRÉ ET POST-INCRÉMENTATION

```
#include <iostream>
using namespace std;
int main(){
    //déclaration et initialisation des variables x et
    Y
    int x=1, y=1;
    //affichage de x, pré-incrémenté
    //x est incrémenté, puis placé dans le flux de
    sortie
    cout << "++x = " << ++x << endl;
    cout << "x    = " << x << endl;
    cout << endl;
    //affichage de y, post-incrémenté
    //y est placé dans le flux de sortie, puis
    incrémenté
    cout << "y++ = " << y++ << endl;
    cout << "y    = " << y << endl;
    return 0;
}
```

Résultat après exécution :

```
++x = 2
x   = 2

y++ = 1
y   = 2
```

1.9 RÉCAPITULATIF

- Il existe 4 *types* de données de base : int, char, float et double.
- Les *entiers* ont 3 types différents : short int, int et long int.
- Les *réels* ont 3 types différents : float, double et long.
- Lorsque, dans une expression mathématique, les *opérandes* sont différents, C++ applique des règles de *conversion implicite*.
- Il existe des règles de *conversion explicite* ou opération de *cast* qui permettent au développeur de forcer le *type* d'une donnée.
- Dans un programme, on peut définir des valeurs fixes sous forme de *constantes* via le *spécificateur const*.
- Comme dans tout calcul mathématique, le langage C++ respecte une *priorité des opérateurs*.
- C++ possède une dizaine de familles d'*opérateurs*.
- La *post* et la *pré-incrémantion* traitent les *opérandes* différemment.

EXERCICES

1-1 Code ASCII

Écrivez un programme qui affiche : le code *ASCII* suivi de ses caractères signés et non signés équivalents sous la forme :

```
codeascii = caract.. caract_non_signé
```

1-2 Erreurs d'arrondi

Écrivez un programme qui montre, via le calcul d'un quotient, les erreurs d'arrondi.

SOLUTIONS

1-1 Code ASCII

```
#include <iostream>
using namespace std;
int main () {
    //déclarations et affectations
    char c=0;
    unsigned char c_uns=0;
    //boucle d'affichage
    for (int i=33; i<127; ++i)
    {
        cout<<i<<" = ";
        c=i;
        c_uns=i;
        //affichage du caractère signé et non-signé
        cout<<c<<" "<<c_uns<<endl;
        cout<<"----- "<<endl;
    }
    return 0;
}
```

Résultat après exécution :

```
33 = !
-----
34 = << <<
-----
.
.
.
-----
122 = z z
-----
123 = { {
-----
124 = | |
-----
125 = } }
-----
126 = ~ ~
-----
```

1-2 Erreurs d'arrondi

```
#include <iostream>
using namespace std;
int main () {
```

```
//déclarations des variables
double x=10000;
double y=x/9-1111;
//affichage des résultats
cout<<"y = "<<y<<endl;
cout<<"y * 9 = "<<y*9<<endl;
return 0;
}
```

Résultat après exécution :

```
y = 0.111111
y * 9 = 1
```



Chaînes et types énumération

PLAN

- 2.1 Introduction
- 2.2 Représentation et manipulation suivant C
- 2.3 Opérations sur les chaînes
- 2.4 Les chaînes en C++ avec la classe `string`
- 2.5 Nouvelles fonctions
- 2.6 Les fonctions membres principales
- 2.7 Les types énumération
- 2.8 Les types personnalisés
- 2.9 Récapitulatif

OBJECTIFS

- Manipuler les chaînes de caractères et savoir comment les représenter
- Maîtriser les opérations de traitement portant sur les chaînes
- Savoir utiliser la classe `string`
- Manipuler et comprendre les types énumération

2.1 INTRODUCTION

Un caractère est un élément d'un jeu de caractères prédéfinis.

Le langage C++ utilise le jeu de caractères **ASCII** (*American Standard Code for Information Interchange*) disponible en annexe 3.

Une chaîne de caractères littérale est une suite de caractères entourés d'apostrophes doubles (" " - guillemets américains).

En C++, les *chaînes de caractères* peuvent supporter deux représentations. La première est basée sur le langage C, la seconde traite les *chaînes* comme des *objets* via la classe `string`.

2.2 REPRÉSENTATIONS ET MANIPULATIONS SUIVANT C

Par l'intermédiaire du programme qui suit, vous allez pouvoir vous faire une idée du traitement basique d'une *chaîne de caractère*.



VALEUR DÉCIMALE, OCTALE ET HEXADÉCIMALE

```
#include <iostream>
using namespace std;
int main () {
    //déclaration de la chaîne majuscule
    char majminchif []=>ABCDEFIGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzijklmno
    pqrstuvwxyz0123456789»;
    //affectation à taillemaj de la taille de la chaîne
    majuscule
    int taillemaj=sizeof(majminchif);
    //affichage de la taille de la chaîne
    cout<<"Taille de la chaîne : "<<taillemaj<<endl;
    //boucle d'affichage des différentes valeurs :
    décimale, octal, et hexadécimal de chacun des
    caractères de la chaîne majminchif
    for (int i=0; i<taillemaj; ++i)
    {
        cout<<"-----" <<endl;
        cout<<majminchif[i]<<" (en décimal) -> "
        <<int(majminchif[i])<<endl;
        cout<<majminchif[i]<<" (en octal) -> "
        <<oct<<int(majminchif[i])<<endl;
        cout<<majminchif[i]<<" (en hexadécimal) -> "
        <<hex<<int(majminchif[i])<<endl;
    }
    return 0;
}
```

Résultat après exécution :

```
Taille de la chaîne : 63
-----
A (en décimal) -> 65
A (en octal) -> 101
A (en hexadécimal) -> 41
-----
B (en décimal) -> 42
B (en octal) -> 102
B (en hexadécimal) -> 42
```

```
-----
C (en décimal) -> 43
C (en octal) -> 103
C (en hexadécimal) -> 43
.
.
.
-----
9 (en décimal) -> 39
9 (en octal) -> 71
9 (en hexadécimal) -> 39
-----
(en décimal) -> 0
(en octal) -> 0
(en hexadécimal) -> 0
```

Remarques. Le programme précédent se prête à plusieurs questions :

Pourquoi la déclaration de `majminchif` est suivie de `[]` ? La taille du tableau est calculée par le compilateur.

Pourquoi la taille de la chaîne calculée est 63, alors qu'elle ne contient que 62 caractères (26 lettres majuscules + 26 lettres minuscules + 10 chiffres) ? Le langage C++ ajoute toujours, en fin de tableau, un caractère NUL (`\0`).

Quel est le rôle du mot-clé `sizeof` ? `sizeof` renvoie la taille en octets du tableau de caractères `majminchif`.

A quoi servent les mots-clés `oct` et `hex` ? Ils affichent respectivement les valeurs converties en base 8 (octal) ou en base 16 (hexadécimal).

2.3 OPÉRATIONS SUR LES CHAÎNES

Le langage C possédait les *fonctions* de traitement des *chaînes* suivantes :

- `strcpy(chaine1, chaine2)` : pour copier `chaine2` dans `chaine1`
- `strcmp(chaine1, chaine2)` : pour comparer `chaine1` et `chaine2`
- `strlen(chaine1)` : pour connaître la longueur de `chaine1`
- `strcat(chaine1, chaine2)` : pour concaténer `chaine1` et `chaine2`
- `strchr(chaine1, car1)` : pour localiser `car1` dans `chaine1`

Pour les utiliser, il faut obligatoirement inclure le fichier d'en-tête `<string>`.



OPÉRATIONS SUR CHAÎNES

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

using namespace std;
int main () {
    //déclaration des variables texte1, texte2 et
    texte3
    char texte1 []="Il était un";
    char texte2 []=" petit navire";
    char texte3 [25];
    //affichage des résultats et utilisation des
    fonctions
    cout<<"texte1 + texte2 : "<<strcat(texte1,texte2)<<endl;
    cout<<"texte3 : "<<strcpy(texte3,texte1)<<endl;
    cout<<"Taille de la chaîne texte1 : "
    <<strlen(texte1)<<endl;
    cout<<"Quel texte commence à partir de la lettre
    'v' : "<<strchr(texte1,'v')<<endl;
    return 0;
}
```

Résultat après exécution :

```
texte1 + texte2 : Il était un petit navire
texte3 : Il était un petit navire
Taille de la chaîne texte1 : 25
Quel texte commence à partir de la lettre 'v' : vire
```

2.4 LES CHAÎNES EN C++ AVEC LA CLASSE STRING

C++ offre plus de fonctionnalités pour traiter les *chaînes de caractères* et les *opérateurs* conventionnels peuvent être utilisés.

Reprendons le début de l'exemple précédent avec la nouvelle syntaxe.



CONCATÉNATION

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

```

int main () {
    //déclaration des variables texte1, et texte3
    string texte1, texte3;
    //affectation
    texte1="Il était un";
    //affectation et concaténation avec l'opérateur
    //conventionnel
    texte1+=" petit navire";
    texte3=texte1;
    //affichage des résultats
    cout<<"texte1 : "<<texte1<<endl;
    cout<<"texte3 : "<<texte3<<endl;
    return 0;
}

```

Résultat après exécution :

```

| texte1 : Il était un petit navire
| texte3 : Il était un petit navire

```

2.5 LES NOUVELLES FONCTIONS DE LA CLASSE STRING

Nous venons de voir la *concaténation* et la copie *via* la *classe string* de C++, mais il existe beaucoup d'autres fonctionnalités comme la recherche d'une *chaîne* ou d'un caractère, la présence ou l'absence d'un caractère dans une suite, l'insertion, la suppression et le remplacement.



FONCTIONS DE LA CLASSE STRING

```

#include <iostream>
#include <string>
using namespace std;
int main () {
    //déclarations des variables
    string texte1="Il était un tout petit petit
    navire";
    string mot1("petit ");
    string mot2("enfant");
    int i;
    //recherche la position du mot 'était' dans texte1
    cout<<"Le mot 'était' est placé en "
    <<texte1.find("était")<<"ième position dans la
    chaîne"<<endl;
}

```

```
//recherche la position de la variable mot1 dans
texte1
    cout<<"Le mot 'petit ' est placé en "
<<texte1.find(mot1)<<"ième position dans la
chaîne"<<endl;
    //recherche le dernier mot 'petit' dans texte1
    cout<<"Le dernier mot 'petit' est placé en "
<<texte1.rfind(mot1)<<"ième position dans la
chaîne"<<endl;
    //recherche la première occurrence d'un caractère
d'une chaîne 'bcade'
    cout<<"La première occurrence d'un caractère de la
suite 'bcade' est placée en "
<<texte1.find_first_of("bcade")<<"ième position dans
la chaîne"<<endl;
    //recherche la dernière occurrence d'un caractère
d'une chaîne 'bcade'
    cout<<"La dernière occurrence d'un caractère de la
suite 'bcade' est placée en "
<<texte1.find_last_of("bcade")<<"ième position dans
la chaîne"<<endl;
    //recherche la première occurrence d'un caractère qui
n'appartient pas à la chaîne 'bcade'
    cout<<"La première occurrence d'un caractère qui
n'appartient pas à la suite 'bcade' est placée en "
<<texte1.find_first_not_of("bcade")<<"ième position
dans la chaîne"<<endl;
    //recherche la dernière occurrence d'un caractère
qui n'appartient pas à la chaîne 'bcade'
    cout<<"La dernière occurrence d'un caractère qui
n'appartient pas à la suite 'bcade' est placée
en "<<texte1.find_last_not_of("bcade")<<"ième position
dans la chaîne"<<endl;
    //affectation du résultat de la recherche à la
variable i
    i=texte1.find("était");
    //affichage de i
    cout<<"Le mot 'était' est placé en position N°: "
<<i<<endl;
    //insertion de mot1 dans texte1
    cout<<"La nouvelle chaîne texte1 : "
<<texte1.insert(18, mot1)<<endl;
    //remplacement du mot 'navire' par le mot 'enfant'
    cout<<"Après remplacement la chaîne texte1 devient : "
<<texte1.replace(36, 6, mot2)<<endl;
    //suppression du mot 'petit' dans texte1
    cout<<"Après suppression la chaîne texte1 devient : "
```

```
<<textel.erase(23, 6)<<endl;
    //affiche la longueur de la chaîne textel
    cout<<"La longueur de textel est : "
<<textel.length()<<endl;
    return 0;
}
```

Résultat après exécution :

```
Le mot 'était' est placé en 3ième position dans la
chaîne
Le mot 'petit ' est placé en 18ième position dans la
chaîne
Le dernier mot 'petit' est placé en 24ième position
dans la chaîne
La première occurrence d'un caractère de la suite
'bcade' est placée en 6ième position dans la chaîne
La dernière occurrence d'un caractère de la suite
'bcade' est placée en 35ième position dans la chaîne
La première occurrence d'un caractère qui n'appartient
pas à la suite 'bcade' est placée en 0ième position dans
la chaîne
La dernière occurrence d'un caractère qui n'appartient
pas à la suite 'bcade' est placée en 34ième position dans
la chaîne
Le mot 'était' est placé en position N° : 3
La nouvelle chaîne textel : Il était un tout petit
petit petit navire
Après remplacement la chaîne textel devient : Il était
un tout petit petit enfant
Après suppression la chaîne textel devient : Il était
un tout petit enfant
La longueur de textel est : 36
```

2.6 LES FONCTIONS MEMBRES PRINCIPALES DE LA CLASSE STRING

=	Affection
begin()	Itérateur renvoyant le début
end()	Itérateur renvoyant la fin
length()	Renvoie la longueur de la chaîne
+=	Ajout à une chaîne
insert(position de départ, chaîne)	Insertion dans une chaîne
erase(position de départ, nb caractères à effacer)	Efface les caractères d'une chaîne
replace(position de départ, longueur, chaîne)	Remplace les caractères d'une chaîne
find(chaîne)	Cherche les caractères dans une chaîne
find_first_of(chaîne)	Cherche la première occurrence d'un caractère d'une chaîne dans une chaîne
find_last_of(chaîne)	Cherche la dernière occurrence d'un caractère d'une chaîne dans une chaîne
find_first_not_of(chaîne)	Cherche la première absence d'un caractère d'une chaîne dans une chaîne
find_last_not_of(chaîne)	Cherche la dernière absence d'un caractère d'une chaîne dans une chaîne

2.7 LES TYPES ÉNUMÉRATION

C++ offre au développeur la possibilité de définir ses propres *types* en plus des *types prédefinis*. Il existe plusieurs façons de le faire, en passant par l'utilisation des *classes*, les *structures* ou en utilisant les *types énumération*.

Un *type énumération* est constitué d'un ensemble fini de valeurs appelées **énumérateurs**. Une fois le *type* créé, il peut être utilisé comme tous les autres *types*, il est simplement précédé du mot-clé **enum**.



MANIPULATION DE TYPES ÉNUMÉRATION

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

```
//création des types énumérations
enum jour{lundi, mardi, mercredi, jeudi, vendredi,
samedi, dimanche};
enum couleur{brun, cyan, rouge, vert, bleu, magenta,
jaune, noir};
enum logique{vrai=1, faux=0};
int main () {
    //déclarations
    jour j1, j2;
    j1=mardi;
    j2=mardi;
    couleur tomate=rouge;
    couleur ciel=bleu;
    logique ok=vrai;
    //test comparatif entre les variables j1 et j2 et
affiche le résultat
    if (j1==j2) cout<<"Jours identiques"<<endl;
    //test si la variable tomate a pour valeur
d'énumérateur 2 et affiche le résultat
    if(tomate==2) cout<<"Une tomate est rouge"<<endl;
    //test si la variable ciel a pour valeur
d'énumérateur 5 et affiche le résultat
    if(ciel==5) cout<<"Un canari est jaune";
        else cout<<"Pas toujours"<<endl;
    //affiche la valeur de la variable ok
    cout<<"Valeur de ok : "<<ok<<endl;
    return 0;
}
```

Résultat après exécution :

```
Jours identiques
Une tomate est rouge
Pas toujours
Valeur de ok : 1
```

Les *types énumérations* participent à une meilleure lisibilité du code, toutefois il ne faut pas en abuser. Chaque *énumérateur* d'une liste définit un nouvel *identificateur de variable* si bien que certaines d'entre elles peuvent devenir indisponibles.

Si vous définissez le *type* voyelles comme enum voyelles{a, e, i, o, u, y}, il faut prendre conscience que ces lettres ne pourront plus être utilisées pour autre chose dans toute la *portée* de leur définition.

2.8 LES TYPES PERSONNALISÉS

En parallèle des *types énumération* précédemment cités, il existe un moyen pour créer ses propres *types* de données personnels (*alias*) équivalant à des *types* existants.

La directive **typedef** se charge de cette opération avec la syntaxe suivante :

```
typedef type typealias
```

Il faut remarquer que *typealias* n'est en fait qu'un nouveau nom d'un *type standard*.

L'utilité de **typedef** sera abordée plus précisément lors de la manipulation des *structures*, au chapitre 7 §7.2 de cet ouvrage.

2.9 RÉCAPITULATIF

- C++ traite une *chaîne de caractères* comme une succession de caractères ou comme un *objet*.
- Il existe un ensemble de *fonctions* dédiées au traitement des *chaînes de caractères* (`strcpy`, `strcmp`, `strlen`, `strcat` et `strchr`).
- La classe `string` offre une plus grande souplesse pour traiter les *chaînes de caractères*.
- La classe `string` possède de puissantes *fonctions* de traitement des *chaînes*.
- En C++, le développeur peut définir ses propres *types* : les *types énumération* et les *types personnalisés*.

EXERCICES

2-1 Pierre, papier, ciseaux

Écrivez un programme qui joue à « pierre, papier, ciseaux ». Deux joueurs doivent dire ou représenter l'un après l'autre, par une attitude de la main, les objets « pierre », « papier » ou « ciseaux ».

Le papier est supérieur à la pierre qui l'enveloppe, la pierre est supérieure aux ciseaux qu'elle casse et les ciseaux sont supérieurs au papier qu'ils coupent.

Le joueur gagnant est celui dont l'objet est supérieur à celui de l'autre.

SOLUTIONS

2-1 Pierre, papier, ciseaux

```
#include <iostream>
using namespace std;
//déclaration des types énumérations
enum choix{pierre, papier, ciseaux};
enum score{joueur1, joueur2, egalite};
int main () {
    //déclaration d'une variable n
    int n;
    //déclarations de 2 variables c1 et c2 de type
    choix
    choix c1, c2;
    //déclaration d'une variable s de type score
    score s;
    //affichage des consignes et saisie des scores
    cout<<"Votre choix (pierre:0 - papier:1 -
ciseaux:3) : "<<endl;
    cout<<"Joueur N°1 : ";
    cin>>n;
    //affectation du choix à c1
    c1=choix(n);
    cout<<"Joueur N°2 : ";
    cin>>n;
    //affectation du choix à c2
    c2=choix(n);
    //tests pour déterminer quel est le joueur gagnant
    if(c1==c2) s=egalite;
    else if(c1==pierre)
        if(c2==papier) s=joueur2;
        else s=joueur1;
    else if (c1==papier)
        if (c2==pierre) s=joueur1;
        else s=joueur2;
    else if(c1==ciseaux)
        if (c2==pierre) s=joueur2;
        else s=joueur1;
    //test la variable s pour afficher le résultat
    if (s==egalite) cout<<"Egalité"<<endl;
        else if (s==joueur1) cout<<"Le joueur
N°1 a gagné"<<endl;
            else cout<<"Le joueur N°2 a
gagné"<<endl;
        return 0;
}
```

Résultat après exécution :

1 - Le joueur 1 joue « pierre » et le 2 « ciseaux »

```
Votre choix (pierre:0 - papier:1 - ciseaux:3) :  
Joueur N°1 : 1  
Joueur N°2 : 3  
Le joueur N°2 a gagné
```

2 - Les joueurs 1 et 2 jouent « papier »

```
Votre choix (pierre:0 - papier:1 - ciseaux:3) :  
Joueur N°1 : 1  
Joueur N°2 : 1  
Egalité
```

CHAPITRE 3

Tableaux et vecteurs

PLAN

- 3.1 Première approche
- 3.2 Traitement des tableaux et déclarations
- 3.3 Construction de tableaux suivant C
- 3.4 Tableaux construits avec la classe `vector`
- 3.5 Tableaux construits avec la classe `valarray`
- 3.6 Récapitulatif

OBJECTIFS

- Créer et manipuler des tableaux de tous types et de toutes dimensions
- Utiliser et exploiter la classe `vector`
- Utiliser et exploiter la classe `valarray`

3.1 PREMIÈRE APPROCHE

Un *tableau* est une suite de données de même *type*.

Chacune de ces données est un élément du **tableau**. Elles peuvent être rangées sous la forme d'un **vecteur** (*tableau* à une dimension – une seule entrée) ou d'une **matrice** (*tableau* multi-dimensionnel – plusieurs entrées, lignes et colonnes).

En C++, chaque élément du *tableau* va être repéré via un **indice**. Il y aura autant de séries d'*indices* qu'il y aura de dimensions dans le *tableau*.

Par l'intermédiaire des *indices* qui déterminent la position d'une cellule dans le *tableau* nous aurons ainsi un **accès direct** à la donnée contenue.

Il faut noter que lors de notre utilisation des *classes* `vector` et `valarray`, vous verrez apparaître dans le code une notation de type `tableau.fonction()`. Celle-ci correspond à l'appel de la *fonction membre* d'une *classe*. Nous dirons que l'objet `tableau` est propriétaire de l'appel de `fonction()`.

Nous retrouverons ces notions avec plus de détails dans le chapitre 7 consacré aux *classes*.

3.3 CONSTRUCTION SUIVANT C

Un premier exemple avec un *vecteur*.

La déclaration est identique à celle d'une *variable* ordinaire que l'on fait suivre de son **indice** entre crochets.



CALCUL DE NOTES

```
#include <iostream>
using namespace std;
int main () {
    //déclarations et initialisation du vecteur et des
variables
    float note[6];
    float somme=0;
    int i;
    //boucle de saisie des notes
    for (i=0; i<6; ++i)
    {
        cout<<"Entrez la note n°"<<i+1<<" : ";
        cin>>note[i];
    }
    //boucle de lecture du vecteur et de calcul du
total
    for (i=0; i<6; ++i)
        somme=somme+note[i];
    //affichage et calcul de la moyenne
    cout<<"Moyenne = "<<somme/6<<endl;
    return 0;
}
```

Résultat après exécution :

```
Entrez la note n°1 : 9.5
Entrez la note n°2 : 10
Entrez la note n°3 : 8
```

```
Entrez la note n°4 : 14
Entrez la note n°5 : 15
Entrez la note n°6 : 7.5
Moyenne = 10.6667
```

Il faut faire attention à dimensionner le *tableau* correctement car le compilateur ne vérifie pas le dépassement de l'*indice* maximal et des résultats pour le moins fantaisistes peuvent alors apparaître (la donnée manquante est prise en mémoire et possède une valeur totalement imprévisible).

Passons maintenant à un exemple avec une *matrice* (*tableau* multi-dimensionnel)



TAILLE ET POIDS

```
#include <iostream>
using namespace std;
int main () {
    //déclaration de la matrice
    float pt[3] [2];
    //déclaration et initialisation des variables
    float poids=0, taille=0;
    int i, j;
    //boucles de saisie et d'affichage des poids et
taille
    for (i=0; i<3; i++)
    {
        for(j=0; j<2; j++)
        {
            cout<<"Indices :
["<<i<<"] ["<<j<<"] "<<endl;
            if (j==0) {
                cout<<"Poids : ";
                cin>>pt [i] [j];
            }
            else
            {
                cout<<"Taille : ";
                cin>>pt [i] [j];
            }
        }
    }
    //boucles de calcul des sommes des poids et des
tailles
    for (i=0; i<3; i++)
```

```

        for(j=0; j<2; j++)
    {
        if (j==0) poids+=pt[i][j];
        else taille+=pt[i][j];
    }
//affichage des moyennes des poids et tailles
cout<<"Moyenne poids : "<<poids/3<<endl;
cout<<"Moyenne âge : "<<taille/3<<endl;
return 0;
}

```

Résultat après exécution :

```

Indices : [0] [0]
Poids : 85
Indices : [0] [1]
Taille : 185
Indices : [1] [0]
Poids : 65
Indices : [1] [1]
Taille : 161
Indices : [2] [0]
Poids : 92
Indices : [2] [1]
Taille : 178
Moyenne poids : 80.6667
Moyenne âge : 174.667

```

En C++, on peut initialiser un *tableau* via une liste d'initialisation. Les valeurs de la liste sont attribuées à chacun des éléments du *tableau* en respectant leur ordre d'apparition.

L'exemple suivant montre la syntaxe à utiliser. Le nombre d'éléments de la liste détermine la taille du *tableau* si celle-ci n'a pas été précisée.



TABLEAU A 2 ENTRÉES

```

#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation du tableau carre
    int carre[2][3]={ {1,2,3}, {1,4,9} };
    //boucles d'affichage du contenu du tableau
    for (int ligne=0; ligne<2; ligne++)
    {
        for(int colonne=0; colonne<3; colonne++)

```

```

        cout<<carre[ligne][colonne]<<endl;
    }
return 0;
}

```

Résultat après exécution :

```

1
2
3
1
4
9

```

Pour terminer, il faut préciser que les définitions de *types personnalisés*, via la directive *typedef* (voir chapitre 2 §2.8), sont utilisables dans la déclaration de *tableaux*, suivant la syntaxe ci-dessous :

```
typedef float angle
angle degre[25], radian[25], grade[25];
```

ou de façon équivalente :

```
typedef float angle[25]
angle degre, radian, grade;
```

3.4 TABLEAUX CONSTRUITS AVEC LA CLASSE VECTOR

La *classe vector* pallie aux nombreuses déficiences des *tableaux classiques* construits selon le langage C. Elle fait maintenant partie de la **STL** (*Standard Template Library*) pour la plupart des compilateurs.

Pour mieux en juger construisons un premier exemple avec un *tableau* uni-dimensionnel.



CALCUL DE NOTES

```

#include <iostream>
#include <vector>
using namespace std;
int main () {
    //déclarations et initialisation des variables
    float somme=0;
    int i;
    //déclaration d'un tableau note qui pourra stocker
    6 données de type float
}

```

```
vector <float> note(6);
//boucle de saisie des notes
//la fonction size récupère la taille du tableau
for(i=0; i<note.size();++i)
{
    cout<<"Entrez la note N°"<<i+1<<" : ";
    cin>>note[i];
}
//boucle d'affichage des notes
for(i=0;i<note.size();++i)
    cout<<"note n° "<<i<<" = "<<note[i]<<endl;
//boucle de calcul du total
for(i=0;i<note.size();++i)
    somme+=note[i];
//affichage et calcul de la moyenne
cout<<"La moyenne est : "<<somme/6<<endl;
return 0;
}
```

Résultat après exécution :

```
Entrez la note N°1 : 10.5
Entrez la note N°2 : 11.5
Entrez la note N°3 : 12
Entrez la note N°4 : 14
Entrez la note N°5 : 9
Entrez la note N°6 : 7.5
note n° 0 = 10.5
note n° 1 = 11.5
note n° 2 = 12
note n° 3 = 14
note n° 4 = 9
note n° 5 = 7.5
La moyenne est : 10.75
```

Une possibilité intéressante, liée à l'utilisation de la *classe vector*, se situe dans la facilité déconcertante avec laquelle un *tableau* peut être transmis à un autre.



TRANSFERT ENTRE TABLEAUX

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

```
int main () {
    //déclarations de i et des deux tableaux de type flottant
    valeur1 et valeur2
    int i;
    vector<float> valeur1(10), valeur2(10);
    cout<<"Saisissez 10 nombres" << endl;
    //boucle de saisie de 10 nombres rangés dans valeur1
    for(i=0; i<valeur1.size(); ++i)
        cin>>valeur1[i];
    cout<<"Contenu du tableau valeur1 :" << endl;
    //boucle qui affiche le contenu de valeur1
    for(i=0; i<valeur1.size(); ++i)
        cout<<"A l'indice "<<i<<" on a : "<<valeur1[i]<< endl;
    //affectation de valeur1 à valeur2
    valeur2=valeur1;
    //boucle d'affichage du contenu de valeur2
    cout<<"Contenu du tableau valeur2 :" << endl;
    for(i=0; i<valeur2.size(); ++i)
        cout<<"A l'indice "<<i<<" on a : "<<valeur2[i]<< endl;
    return 0;
}
```

Résultat après exécution :

```
Contenu du tableau valeur1 :
A l'indice 0 on a : 13
A l'indice 1 on a : 134.5
A l'indice 2 on a : 0.05
A l'indice 3 on a : 10000
A l'indice 4 on a : 77
A l'indice 5 on a : 137
A l'indice 6 on a : 4
A l'indice 7 on a : 0.25
A l'indice 8 on a : 256.125
A l'indice 9 on a : 77
Contenu du tableau valeur2 :
A l'indice 0 on a : 13
A l'indice 1 on a : 134.5
A l'indice 2 on a : 0.05
A l'indice 3 on a : 10000
A l'indice 4 on a : 77
A l'indice 5 on a : 137
A l'indice 6 on a : 4
A l'indice 7 on a : 0.25
A l'indice 8 on a : 256.125
A l'indice 9 on a : 77
```

3.5 TABLEAUX CONSTRUITS AVEC LA CLASSE VALARRAY

De nombreux scientifiques utilisent le langage C++. Pour éviter les problèmes redondants de gestion des *tableaux* de nombres, signalés par la communauté scientifique, tout en apportant un niveau de performance élevée digne d'un langage évolué, la classe `valarray` comme la *classe vector* a aussi été ajoutée à la bibliothèque *STL* (*Standard Template Library*).

Attention, il faut vérifier que votre compilateur implémente `valarray`, ce qui n'est pas toujours le cas, contrairement à `vector` qui semble plus répandu. Une situation qui devrait progresser avec la mise à jour des derniers *environnements de développement intégré* (EDI).

Le code qui suit reprend, en utilisant `valarray`, le premier exemple (§ 3.4), exprimé avec la *classe vector*.



CALCUL DE NOTES

```
#include <iostream>
#include <valarray>
using namespace std;
int main () {
    //déclarations et initialisation des variables
    float somme=0;
    int i;
    //déclaration d'un tableau note qui pourra stocker
    6 données de type float
    valarray <float> note(6);
    //boucle de saisie des notes
    //la fonction size récupère la taille du tableau
    for(i=0; i<note.size();++i)
    {
        cout<<"Entrez la note N°"<<i+1<<" : ";
        cin>>note[i];
    }
    //boucle d'affichage des notes
    for(i=0;i<note.size();++i)
        cout<<"note n° "<<i<<" = "<<note[i]<<endl;
    //boucle de calcul du total
    for(i=0;i<note.size();++i)
        somme+=note[i];
    //affichage et calcul de la moyenne
    cout<<"La moyenne est : "<<somme/6<<endl;
    return 0;
}
```

Résultat après exécution :

```
Entrez la note N°1 : 10.5
Entrez la note N°2 : 12.5
Entrez la note N°3 : 7.5
Entrez la note N°4 : 8.5
Entrez la note N°5 : 14
Entrez la note N°6 : 2.5
note n° 0 = 10.5
note n° 1 = 12.5
note n° 2 = 7.5
note n° 3 = 8.5
note n° 4 = 14
note n° 5 = 2.5
La moyenne est : 9.25
```

En conclusion, nous pouvons dire que la *classe vector* doit être utilisée en lieu et place de la gestion classique des *tableaux* selon C. Elle apporte vraiment plus de souplesse.

La *classe valarray* est plutôt réservée à des applications demandant une optimisation ou une vitesse de traitement élevée, là où la notion de performances en termes de calculs est prioritaire.

Il faut noter que les *classes vector* et *valarray* possèdent de nombreuses autres méthodes.

3.6 RÉCAPITULATIF

- Une suite de données de mêmes *types* est un *tableau*.
- Il y a deux sortes de *tableaux*, les *vecteurs* et les *matrices*.
- Les éléments d'un *tableau* sont repérés grâce à leurs *indices*.
- Il existe trois moyens pour traiter des *tableaux* : l'approche suivant le langage C, l'utilisation de la *classe vector* ou l'utilisation de la *classe valarray*.
- Les *classes vector* et *valarray* font appel à la bibliothèque *STL* (*Standard Template Library*)

EXERCICES

3-1 Recherche de valeurs nulles

Soit le tableau de *triplets* (*x*, *y*, *z*) suivant :

x	y	z
12	0	10,5
13	7,5	0
14	6	8
17	0	10
15	13	12,5
0	19	6

Écrivez un programme qui formalise ce *tableau* dans une *matrice* de type *tab [x] [y] [z]*, puis consultez chacune des cellules afin de déterminer les nombres de valeurs nulles.

3-2 Inversion de *tableau*

Écrivez un programme qui inverse le *tableau* suivant. Le premier élément devient le dernier et ainsi de suite (A prend la place de S, Z la place de G...).

A	Z	C	N	P	F	G	S
---	---	---	---	---	---	---	---

3-3 Décalage et rotation

Soit un *vecteur* (*tableau* à 1 dimension) contenant les chiffres de 0 à 9. Écrivez un programme, utilisant la *classe valarray*, qui affiche le contenu de ce *vecteur* sous trois formes :

- La liste des chiffres, soit : 0 1 2 3 4 5 6 7 8 9
- La liste des chiffres ayant subis un décalage à gauche de deux positions, soit : 2 3 4 5 6 7 8 9 0 0
- La liste des chiffres ayant subis une rotation à droite de deux positions, soit : 8 9 0 1 2 3 4 5 6 7

SOLUTIONS

3-1 Recherche de valeurs nulles

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation des variables
    //l est le n° de la ligne
    //c est le n° de la colonne
    //nbzero contiendra le nombre de 0
    int l=6, c=3, nbzero=0;
    //déclaration et initialisation du tableau bi-
dimensionnel tab
    float
tab[6][3]={ {12,0,10.5}, {13,7.5,0}, {14,6,8}, {17,0,10}, {1
5,13,12.5}, {0,19,6} };
    //boucles de lecture des valeurs de tab
    for (int i=0; i<l; ++i)
        for(int j=0; j<c; ++j)
            //test si la valeur lue est égale à 0
            et dans ce cas incrémenter nbzero
            if (tab[i][j]==0) ++nbzero;
    //affichage du résultat
    cout<<"Le nombre de valeur égale à 0 est : "
<<nbzero<<endl;
    return 0;
}
```

Résultat après exécution :

```
Le nombre de valeur égale à 0 est : 4
```

3.2 Inversion de tableau

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation du tableau de
    caractère tab et déclaration d'une variable caractère
    temp
    char tab[8]={ 'A', 'Z', 'C', 'N', 'P', 'F', 'G',
'S' }, temp;
    //boucle de lecture et d'inversion via la variable
    temporaire temp
    for (int i=0; i<4; ++i)
    {
```

```
    temp=tab[i];
    tab[i]=tab[7-i];
    tab[7-i]=temp;
}
//boucle d'affichage des valeurs séparées par une
tabulation
for (int j=0; j<8; ++j) cout<<tab[j]<<>\t>;
return 0;
}
```

Résultat après exécution :

S G F P N C Z A

3.3 Décalage et rotation

```
#include <iostream>
#include <valarray>
using namespace std;
int main(void)
{
    //déclaration et initialisation d'un tableau tab1
    int tab1[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    //déclaration d'un valarray tab2
    valarray <int> tab2(tab1, 10);
    //déclaration d'un valarray tab3
    valarray <int> tab3(10);
    //affiche les entiers contenus dans tab2 séparés
    par une tabulation
    for (int i=0; i<tab2.size(); ++i) cout << tab2[i]
    << "\t";
    //passe à la ligne suivante
    cout<<endl;
    //décale tab2 à gauche de deux positions
    tab3 = tab2.shift(2);
    //affiche les entiers contenus dans tab3 séparés
    par une tabulation
    for (int i=0; i<tab2.size(); ++i) cout << tab3[i]
    << "\t";
    //passe à la ligne suivante
    cout<<endl;
    // effectue une rotation de tab2 de 2 positions
    vers la droite
    tab3 = tab2.cshift(-2);
    //affiche les entiers contenus dans tab3 séparés
    par une tabulation
    for (int i=0; i<tab2.size(); ++i) cout << tab3[i]
    << "\t";
    return 0;
}
```

Résultat après exécution :

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

CHAPITRE 4

Structures de contrôle

PLAN

- 4.1 Introduction
- 4.2 Les instructions d'entrées-sorties cin et cout
- 4.3 La gestion des cas avec switch
- 4.4 Les boucles
- 4.5 Les contrôles d'itérations
- 4.6 Récapitulatif

OBJECTIFS

- Créer et manipuler les instructions d'entrées-sorties cin et cout pour la saisie et l'affichage
- Utiliser et exploiter la gestion de cas avec l'instruction switch
- Créer et manipuler les différents types de boucles
- Gérer et contrôler les boucles via les contrôles d'itérations

4.1 INTRODUCTION

Une **structure de contrôle** est un élément clé de la programmation impérative, c'est une commande qui contrôle l'ordre d'exécution des différentes instructions d'un programme ou d'un algorithme.

Ils existent deux grandes familles de *structure de contrôle*, les structures **séquentielles** et les structures **itératives**.

Les premières sont attachées aux instructions d'entrées sorties comme **cin** et **cout** de C++. Elles arrêtent le flot d'exécution des instructions jusqu'à ce que le périphérique ait traité les données.

Les secondes sont dites *conditionnelles*, elles testent si une condition est vraie ou fausse, comme un test du type : **si condition alors faire instruction**. On les trouve aussi au sein des *boucles* qui se répètent tant qu'une *condition* est vraie ou jusqu'à ce qu'elle le soit.

4.2 LES INSTRUCTIONS D'ENTRÉES-SORTIES

Les deux instructions, `cin` et `cout` (prononcez « see in » et « see out ») sont dédiées à l'écriture et la lecture sur l'entrée et la sortie standard. Nous les avons déjà rencontrées au cours des exemples et des exercices dans les chapitres précédents. Elles sont au cœur de la programmation C++ en mode console¹.

Pour que ces instructions fonctionnent, il faut obligatoirement inclure le fichier d'en-tête `<iostream>`. Il contient l'ensemble des déclarations utiles à la manipulation et l'utilisation de `cin` et `cout`.

L'instruction `cout` demande au système de diriger une *chaîne de caractère*, un entier signé ou non, un flottant, un caractère ou un **pointeur**² vers le flux de sortie standard qui représente le plus souvent l'écran. Le nom de `cout` vient de **Console OUTput** (sortie console).

Je ne reviendrais pas sur la syntaxe que vous avez déjà rencontrée de nombreuses fois dans cet ouvrage. J'ajouterais simplement un petit commentaire sur ce qui a déjà été écrit. La présence de l'opérateur `<<` est obligatoire, elle symbolise l'envoi du flot. Cet opérateur a plusieurs rôles, il peut se contenter de transmettre une *chaîne de caractère* ou bien convertir une valeur *binaire* en une suite de caractères. Cela est lié au principe de **surdéfinition**³ d'un *opérateur* présent au sein de C++.

L'utilisation de `endl` effectue un retour à la ligne et purge le *buffer de sortie*. Son utilisation rend plus lisible le code, contrairement à `\n` qui fait la même chose mais peut prêter à confusion avec un nom de *variable*.



L'utilisation de l'instruction `endl` peut dégrader les performances d'un programme car l'opération de purge du tampon, que ne fait pas `\n`, prend du temps machine. En effet, dans C++, `endl` fait partie des manipulateurs non-paramétriques et il appelle un autre manipulateur, `flush()` de la classe `ostream`⁴.

On peut associer l'opérateur `<<` à des séparateurs comme à la tabulation horizontale (`\t`), verticale (`\v`), la fin de ligne (`\n`), le retour chariot (`\r`) ou le changement de page (`\f`).

¹ Le mode console est un mode où les programmes se déroulent dans une fenêtre en mode texte, contrairement aux fenêtres que nous côtoyons habituellement qui sont en mode graphique. Dans ce mode les commandes sont exécutées plus rapidement. Beaucoup de logiciels que nous utilisons sont en fait des interfaces graphiques de logiciels écrits en mode texte...

² Un *pointeur* est une *variable* contenant une adresse mémoire (voir chapitre 5).

³ Voir chapitre 8 §8.1.

⁴ `ostream` est une *classe* dérivée de la *classe* de base `ios` comme la *classe* `istream`. La *classe* `iostream` est elle-même dérivée des *classes* `ostream` et `istream`.

L'instruction `cin`, au contraire de `cout`, fait une lecture sur l'entrée standard : **Console INput** (entrée console), généralement le clavier de l'utilisateur. Lorsque l'instruction `cin` s'exécute, le système s'arrête et attend une entrée. Comme pour l'instruction `cout`, vous avez déjà pu utiliser `cin` au travers des exemples proposés précédemment.

La présence de l'opérateur `>>` est obligatoire, il symbolise la réception du flot. Il accepte les *types* de base quelconques signés ou non signés ainsi que des *chaînes de caractères*.

Quand un caractère invalide est envoyé, l'exploration du flot s'arrête.

Dans l'exemple ci-dessous, vous pourrez découvrir le moyen de réaliser une entrée multiple avec `cin` et un affichage tabulé via `cout`.



ENTRÉE-SORTIE

```
#include <iostream>
using namespace std;
int main () {
    //déclaration des variables
    char prenom, nom;
    int entier1, entier2, entier3;
    //affichage de la chaîne "Entrez vos 2 initiales : "
    cout<<"Entrez vos 2 initiales : "<<endl;
    //attente de l'entrée des 2 initiales
    cin>>prenom>>nom;
    //affichage de la chaîne "Initiales : " suivie
    des 2 initiales espacées de 2 tabulations et suivi d'une
    fin de ligne
    cout<<"Initiales : "<<nom<<"\t\t"<<prenom<<"\n";
    //affichage de la chaîne "Entrez 3 nombres entiers
    : "
    cout<<"Entrez 3 nombres entiers : "<<endl;
    //attente de l'entrée de 3 nombres entiers
    cin>>entier1>>entier2>>entier3;
    //affichage de la chaîne "Les nombres étaient : "
    suivie des 3 nombres saisis séparés par une tabulation et
    suivis d'une fin de ligne
    cout<<"Les nombres étaient : "
    <<entier1<<"\t"<<entier2<<"\t"<<entier3<<"\n";
    return 0;
}
```

Résultat après exécution :

```
Entrez vos 2 initiales :
P
M
```

```
Initiales : M          P
Entrez 3 nombres entiers :
10
5
25
Les nombres étaient : 10      5      25
```

4.3 LA GESTION DES CAS AVEC SWITCH

L'instruction **switch** contrôle le choix d'un groupe d'instructions à exécuter parmi d'autres. La sélection se fait par l'évaluation d'une expression.

S'il y a égalité entre l'expression (caractère ou entier généralement saisi par l'utilisateur dans le flux d'entrée) et les choix disponibles dans **switch**, un bloc d'instructions associé est exécuté jusqu'au **break** qui termine le traitement du cas.

S'il n'y a pas de correspondance, c'est un bloc d'instructions nommé **default** qui est pris en compte. Ce bloc est optionnel, s'il n'est pas précisément indiqué aucun traitement n'est réalisé.

L'instruction **break** est obligatoire et marque la fin d'une séquence d'instructions.



CIRCONFÉRENCE, SURFACE ET VOLUME

```
#include <iostream>
using namespace std;
int main () {
    //déclaration des variables
    float rayon;
    char calcul;
    //saisie du Rayon
    cout<<"Rayon : ";
    cin>>rayon;
    //saisie du calcul : C, S ou V
    cout<<"Calcul C, S ou V : ";
    cin>>calcul;
    //sélection d'un choix en fct de calcul
    switch (calcul){
        case 'C':
            //affichage et calcul de la
            circonference
            cout<<"Circonference du cercle = "
```

```
<<2*3.14*rayon<<endl;
    //arrêt du traitement
    break;
case 'S':
    //affichage et calcul de la surface
    cout<<"Surface du cercle = "
<<3.14*rayon*rayon<<endl;
    //arrêt du traitement
    break;
case 'V':
    //affichage et calcul du volume
    cout<<"Volume de la sphère : "
<<4/3.0*3.14*rayon*rayon<<endl;
    //arrêt du traitement
    break;
default:
    //affichage message alerte en cas de
    choix inexistant
    cout<<"Calcul inexistant, seuls C, S
et V sont autorisés"<<endl;
    //arrêt du traitement
    break;
    //fin de la sélection
}
return 0;
}
```

Résultat après exécution :

1 – Calcul de la surface d'un cercle de 2.5 de rayon

Rayon : 2.5

Calcul C, S ou V : S

Surface du cercle = 19.625

2 – Calcul suivant P qui n'existe pas

Rayon : 2.5

Calcul C, S ou V : P

Calcul inexistant, seul C, S et V sont autorisés

4.4 LES BOUCLES

Les **boucles** forment une succession d'*itérations* répétitives constituées d'un ensemble de plusieurs instructions.

Le langage C++ dispose de 3 instructions *itératives* : **while**, **do...while** et **for**.

Boucle avec **while**

Elle est construite suivant un bloc de la forme :

```
while (condition) expression
```

L'expression qui contient la ou les instructions est répétée tant que la valeur de la *condition* n'est pas fausse (égale à 0).



CARRÉ D'UN NOMBRE ENTIER

```
#include <iostream>
using namespace std;
int main () {
    //déclaration de la variable n
    int n;
    //saisie d'un premier nombre entier
    cout<<"Saisissez un nombre entier positif : ";
    cin>>n;
    //boucle tant que n est différent de 0
    while(n!=0)
    {
        //affichage du carré de n
        cout<<"Le carré de "<<n<<" est égal à : "
<<n*n<<endl;
        //saisie du nombre suivant
        cout<<"Nombre suivant (0 pour terminer) : ";
        cin>>n;
    }
    return 0;
}
```

Résultat après exécution :

```
Saisissez un nombre entier positif : 4
Le carré de 4 est égal à : 16
Nombre suivant (0 pour terminer) :5
Le carré de 5 est égal à : 25
Nombre suivant (0 pour terminer) :125
Le carré de 125 est égal à : 15625
Nombre suivant (0 pour terminer) :0
```

Boucle avec **do...while**

Il existe une différence importante entre **do...while** et **while**, elle repose sur la position du test de sortie de la *boucle*. Lorsqu'on utilise **do...while**, le test contrairement à ce que fait **while**, s'effectue en fin d'exécution de l'expression.

La ou les instructions sont exécutées de manière répétitive tant que la condition n'est pas fausse (égale à 0).

Il faut remarquer que l'*expression* est toujours évaluée au moins une fois puisque la *condition* est placée en fin de bloc.

La syntaxe de `do...while` est la suivante :

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



FACTORIELLE AVEC DO...WHILE

```
#include <iostream>
using namespace std;
int main () {
    //déclaration des variables
    int i, j, f=1;
    //saisie de i, nombre dont on va calculer la
    factorielle
    cout<<"Entrez le nombre entier dont vous voulez la
    factorielle : ";
    cin>>i;
    //affectation de i à j pour l'affichage du résultat
    j=i;
    //boucle do...while
    do
    {
        //calcul de la factorielle
        f*=i;
        //décrémentation de i
        -i;
    }
    //test si i=0 pour savoir si le calcul doit se
    terminer
    while (i!=0);
    //affichage du résultat suivant j
    cout<<"Factorielle de "<<j<< " = "<<f<<endl;
    return 0;
}
```

Résultat après exécution :

```
Entrez le nombre entier dont vous voulez la factorielle
: 10
Factorielle de 10 = 3628800
```

L'instruction **for**

Voici sa syntaxe générale :

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

Cette instruction est l'une des plus utilisée, le bloc qui la contient est composé d'une *expression* qui gère la valeur initiale d'un *index d'initialisation* (*expression1*), d'une *expression conditionnelle* qui teste l'*index* (*expression2*), d'une *expression* qui modifie l'*index* afin d'amener l'*itération* suivante (*expression3*), et enfin des *instructions* qui constituent la séquence qui sera répétée (*expression4*).

L'*index* doit, comme toute *variable*, être déclaré, cependant *expression1* peut comporter la déclaration et l'initialisation.

L'exemple suivant montre les deux cas de figure.



FACTORIELLE AVEC FOR

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et intialisation des variables
    int i, facto=1, puiss=2;
    //boucle for de 2 à 12 suivant un pas i égal à 1
    for (i=2; i<=12; ++i)
    {
        //calcul de la factorielle
        facto=facto*i;
        //affichage des factorielles
        cout<<"Factorielle de "<<i<< " = "
<<facto<<endl;
    }
    //boucle de 2 à 16 suivant un pas j égal à 1 et
    déclaré entier
    for (int j=2; j<=16; ++j)
    {
        //calcul de la somme
        puiss=puiss*2;
        //affichage des puissances de 2
        cout<<"2 à la puissance "<<j<< " = "
<<puiss<<endl;
    }
    return 0;
}
```

Résultat après exécution :

```

Factorielle de 2 = 2
Factorielle de 3 = 6
.
.
.
Factorielle de 8 = 40320
Factorielle de 9 = 362880
Factorielle de 10 = 3628800
Factorielle de 11 = 39916800
Factorielle de 12 = 479001600
2 à la puissance 2 = 4
2 à la puissance 3 = 8
.
.
.
2 à la puissance 15 = 32768
2 à la puissance 16 = 65536

```

Une boucle `for` peut utiliser plusieurs *index*, notifiés et initialisés de façon individuelle.

Le *pas* (*incrément de l'itération*) d'une *boucle for* peut varier et être négatif ou positif.



BOUCLES FOR

```

#include <iostream>
using namespace std;
int main () {
    //déclaration des variables
    int i, j, k;
    //2 boucles i avec un pas de 1 et j avec un pas de
    5 jusqu'à ce que i soit égal à 10
    for(i=0, j=0; i<=10; ++i, j+=5)
        //affichage de i et j
        cout<<"i = "<<i<<" et j = "<<j<<endl;
    //boucle qui se décrémente suivant un pas égal à -
    50
    for(k=100; k>=-50; k-=25)
        //affichage de k
        cout<<"k = "<<k<<endl;
    return 0;
}

```

Résultat après exécution :

```
i = 0 et j = 0
i = 1 et j = 5
i = 2 et j = 10
i = 3 et j = 15
i = 4 et j = 20
i = 5 et j = 25
i = 6 et j = 30
i = 7 et j = 35
i = 8 et j = 40
i = 9 et j = 45
i = 10 et j = 50
k = 100
k = 75
k = 50
k = 25
k = 0
k = -25
k = -50
```

Les *boucles* peuvent être imbriquées les unes dans les autres avec des *structures de contrôles* différentes. Les *index* doivent être différents et les chevauchements de *boucles* sont interdits.

4.5 LES CONTRÔLES D'ITÉRATIONS

C++ dispose de trois instructions de gestion et de contrôle qui sont : **break**, **continue** et **goto**.

L'instruction **break**

Nous avons déjà rencontré, plus haut dans ce chapitre, l'instruction **break** au sein de l'instruction **switch**.

Dans des cas bien spécifiques, il peut être utile de 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** ou **for**.

Quand le programme rencontre cette instruction la *boucle* se bloque et l'*itération* en cours se termine précisément à cet endroit.



PIUSSANCE D'UN NOMBRE ENTIER

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation des variables
    int n, exp, puiss=1;
    //boucle while tant que vrai
    while(1)
    {
        //saisie d'un nombre entier n
        cout<<"Saisissez un nombre entier ou 0 pour
terminer : ";
        cin>>n;
        //test du nombre entier saisi, s'il est égal
        à 0 alors arrêt
        if (n==0) break;
        //saisie de l'exposant exp
        cout<<"Saisissez l'exposant : ";
        cin>>exp;
        //boucle pour mettre le nombre entier n
        saisi à la puissance exp
        for (int i=1;i<=exp;++i)
            //calcul de la variable puiss
            puiss*=n;
        //affichage du résultat
        cout<<n<<" exposant "<<exp<< " = "
<<puiss<<endl;
        //réinitialisation de la variable puiss pour
        le calcul suivant
        puiss=1;
    }
    return 0;
}
```

Résultat après exécution :

```
Saisissez un nombre entier ou 0 pour terminer : 2
Saisissez l'exposant : 32
2 exposant 32 = 0
Saisissez un nombre entier ou 0 pour terminer : 3
Saisissez l'exposant : 3
3 exposant 3 = 27
Saisissez un nombre entier ou 0 pour terminer : 0
```

L'instruction **continue**

L'instruction **continue** arrête, comme **break**, l'exécution du bloc d'instructions en cours mais ne met pas fin à la *boucle*. La reprise de cette dernière peut continuer à l'*itération* suivante.

L'instruction **goto**

Cette instruction possède la particularité de générer un saut vers un emplacement repéré par une *étiquette*, autorisant ainsi un bouleversement dans la logique d'exécution du programme.

L'appel de l'*étiquette*, qui est un *identificateur*, suit **goto** ; il renvoie vers un endroit précis du code où il se retrouve suivi du caractère : (deux points).

La syntaxe utilisée est celle qui suit :

```
goto étiquette;  
.  
.  
.  
.  
étiquette:instruction
```

Dans le programme, cette *étiquette* doit être unique ou bien au sein d'une fonction (voir chapitre 6).



APRÈS 7, JE M'ARRÊTE

```
#include <iostream>  
using namespace std;  
int main () {  
    //déclaration et initialisation des variables  
    int i, j[]={0,1,2,3,4,5,6,7,8,9};  
    //création d'une boucle de 0 à 10  
    for (i=0;i<10;++i)  
    {  
        //si j[i] est supérieur à 7 alors aller à  
        //l'étiquette end  
        if (j[i]>7) goto end;  
        //affichage de i et j  
        cout<<"Quand i vaut "<<i<<" j est égal à "  
        <<j[i]<<endl;  
        //continue si j[i] est inférieur à 7 avec  
        //reprise à l'itération suivante sans exécuter la suite  
        //du bloc d'instructions
```

```
    continue;
    //étiquette end, affichage de l'arrêt de la
    boucle, j[i] est supérieur à 7
    end:cout<<"j[i] = "<<j[i]<<", c'est
    supérieur à 7, donc la boucle s'arrête"<<endl;
    //arrêt de la boucle
    break;
}
return 0;
}
```

Résultat après exécution :

```
Quand i vaut 0 j est égal à 0
Quand i vaut 1 j est égal à 1
Quand i vaut 2 j est égal à 2
Quand i vaut 3 j est égal à 3
Quand i vaut 4 j est égal à 4
Quand i vaut 5 j est égal à 5
Quand i vaut 6 j est égal à 6
Quand i vaut 7 j est égal à 7
j[i] = 8, c'est supérieur à 7, donc la boucle s'arrête
```



L'utilisation de l'instruction `goto` est sujette à controverses dans la communauté des développeurs C++. Elle peut mettre en cause l'intégrité de la logique d'un programme. Je vous conseille donc de l'utiliser qu'en cas de nécessité absolue¹.

4.6 RÉCAPITULATIF

- Le langage C++ possède deux instructions d'entrées-sorties utilisables en *mode console*, ce sont `cin` et `cout`.
- Lors d'une sélection, l'instruction `switch` permet d'orienter le programme vers une séquence de code choisie.

¹ Il est rarissime qu'au cours du développement d'un programme on ne puisse pas pallier au remplacement d'un saut de type `goto`.

En 1966, Böhm et Jacopini ont démontré que tout programme comportant des `goto` pouvait être transformé en programme n'utilisant que des branchements conditionnels de type *if...then....else* ou des boucles de type *while* ou *do...while*. Il est souvent nécessaire de dupliquer des portions de codes, d'utiliser des *variables* intermédiaires voire même les deux, mais c'est toujours possible. Ces deux mêmes auteurs montreront aussi que les *branchements conditionnels* peuvent eux aussi être remplacés par des *boucles*.

- Il existe plusieurs instructions permettant de gérer des boucles : `do`, `do...while` et `for`.
- Pour mieux contrôler les *structures itératives*, comme les *boucles*, on utilise les instructions `break`, `continue` et `goto`.

EXERCICES

4-1 Surface de rectangles

Écrivez un programme qui calcule la surface de tous les rectangles dont les dimensions : longueur et largeur, varient entre 1 et 5 m (suivant un pas de 1 m).

Le code utilisera deux *boucles* imbriquées et affichera l'ensemble des résultats.

4-2 Inversion de chiffres

Écrivez un programme qui inverse tous les chiffres d'un nombre entier positif qui a été saisi au préalable. Par exemple si vous entrez le nombre : 54321, le programme doit afficher le nombre : 12345.

Si le nombre saisi se termine par un 0, lors de l'inversion celui-ci disparaît (il n'est plus significatif).

SOLUTIONS

4-1 Surface de rectangles

```
#include <iostream>
using namespace std;
int main () {
    //initialisation des variables
    int L, l;
    //première boucle
    for (L=1; L<=5; ++L)
    {
        //deuxième boucle imbriquée dans la première
        for (l=1; l<=5; ++l)
            //affichage du résultat
            cout<<"Surface d'un rectangle de
"<<L<<" m par "<<l<<" m = "<<L*l<<" m2"<<endl;
    }
    return 0;
}
```

Résultat après exécution :

```
Surface d'un rectangle de 1 m par 1 m = 1 m2
Surface d'un rectangle de 1 m par 2 m = 2 m2
.
.
.
Surface d'un rectangle de 5 m par 3 m = 15 m2
Surface d'un rectangle de 5 m par 4 m = 20 m2
Surface d'un rectangle de 5 m par 5 m = 25 m2
```

4-2 Inversion de chiffres

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation des variables
    long nb, r, n=0;
    //saisie d'un nombre entier positif
    cout<<"Saisissez un nombre entier positif : ";
    cin>>nb;
    //boucle d'inversion
    while (nb>0)
    {
        r=nb%10;
        nb/=10;
        n=10*n+r;
    }
    //affichage du nombre inversé
    cout<<"Après inversion de ses chiffres, le nombre
devient : "<<n<<endl;
    return 0;
}
```

Résultat après exécution :

1 – Le nombre saisi ne se termine pas par un 0

```
Saisissez un nombre entier positif : 765034
Après inversion de ses chiffres, le nombre devient :
430567
```

2 - Le nombre saisi se termine par un 0

```
Saisissez un nombre entier positif : 34670
Après inversion de ses chiffres, le nombre devient :
7643
```




Pointeurs

PLAN

- 5.1 Première approche
- 5.2 Déclaration et utilisation
- 5.3 Tableau et pointeur
- 5.4 Deux nouveaux opérateurs, new et delete
- 5.5 Pointeur de pointeur
- 5.6 Déclaration de pointeurs
- 5.7 Récapitulatif

OBJECTIFS

- Comprendre le fonctionnement et l'utilité des pointeurs
- Différencier adresses, pointeurs et variables
- Utiliser les pointeurs pour manipuler des variables ou des tableaux
- Savoir gérer et allouer correctement et dynamiquement une zone mémoire

5.1 PREMIÈRE APPROCHE

La compréhension et la manipulation des *pointeurs* sont fondamentales dans l'utilisation du langage C++.

Les *pointeurs* accèdent à la couche basse de la machine, en mettant en œuvre un accès direct à la mémoire. Par leurs intermédiaires, on se déplace de cases mémoire en cases mémoire. Cette technique optimise le code généré par le compilateur et améliore les temps de traitement pour obtenir une meilleure performance du programme.

Dans un langage évolué, nous manipulons constamment des *variables* qui occupent un plus ou moins grand nombre d'octets en mémoire suivant leur type. Chacune d'entre elles est rangée dans un emplacement mémoire particulier.

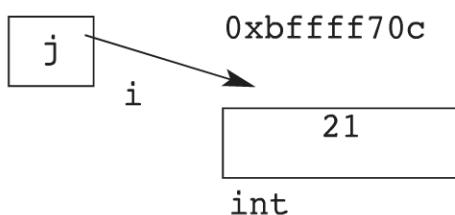
L'adresse de cet emplacement s'obtient en faisant précéder la *variable* du symbole & (et commercial), appelé **opérateur d'adresse**. Ce dernier possède aussi d'autres utilisations comme nous le verrons plus loin¹, dans le chapitre 6 §6.6.

Pour atteindre le contenu, on fait précéder la *variable* du symbole *, nommé **opérateur unaire d'indirection**.

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

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

Vous trouverez ci-dessous un résumé l'ensemble de ces notions :



- *i* est de type **int**
- *i* est rangé à l'emplacement **0xbfffff70c**
- *i* est égal à 21
- pour afficher *i*, on peut utiliser : `cout<<i`
- pour afficher l'adresse de *i*, on utilise : `cout<<&i`
- pour afficher *i*, on peut aussi utiliser le déréférencement avec `cout<<*j`, *j* étant un *pointeur* entier (**int**) qui pointe sur l'emplacement de *i*.

5.2 DÉCLARATION ET UTILISATION

🔍

ADRESSE ET CONTENU

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation de i et déclaration
    du pointeur *adr_i
    int i=21, *adr_i;
```

¹ L'opérateur & peut être utilisé pour effectuer un *passage par référence*.

```
//affichage de i
cout<<"i = "<<i<< " (la valeur de i)"<<endl;
//affichage de l'adresse de i
cout<<"&i = "<<&i<< " (l'adresse de i)"<<endl;
//affectation de l'adresse de i au pointeur adr_i
adr_i=&i;
//affichage du contenu situé à l'adresse i
cout<<"*adr_i = "<<*adr_i<< " (le contenu de
l'adresse i)"<<endl;
//affectation d'une nouvelle valeur au pointeur
adr_i
    *adr_i=7;
//affichage de la nouvelle valeur de i
cout<<"i = "<<i<< " (la nouvelle valeur de
i)"<<endl;
//affichage de l'adresse de i
cout<<"&i = "<<&i<< " (l'adresse de i est tjs la
même)"<<endl;
    return 0;
}
```

Résultat après exécution :

```
i = 21 (la valeur de i)
&i = 0xbffff708 (l'adresse de i)
*adr_i = 21 (le contenu de l'adresse i)
i = 7 (la nouvelle valeur de i)
&i = 0xbffff708 (l'adresse de i est tjs la même)
```

5.3 TABLEAU ET POINTEUR

En C++, le nom d'un *tableau* est considéré comme un *pointeur* vers le premier élément d'un *tableau*.

Nous pouvons donc extrapoler en disant qu'une expression de la forme `&nomtableau[0]` est équivalente à `nomtableau` en termes d'adresse, ce qui nous permet de dire que `nomtableau[i]` correspond à `*[nomtableau+i]` en termes de contenus.

Vérifions ces éléments à l'aide du programme qui suit.



REmplissage et lecture d'un tableau

```
#include <iostream>
using namespace std;
int main(){
```

```
//déclaration et initialisation des variables
int i, j, maxi, mini, tableau[6];
//boucle de saisie et de remplissage (6 nombres
dans tableau)
for (i=0; i<=5;++i)
{
    cout<<"Saisissez un nombre entier : ";
    cin>>tableau[i];
}
//boucle de lecture des valeurs maxi et mini
stockées dans le tableau
//la lecture se fait via le pointeur *(tableau+j)
for(j=0;j<=5;++j)
{
    if (*(tableau+j)>maxi) maxi=*(tableau+j);
    if (*(tableau+j)<mini) mini=*(tableau+j);
}
//affichage des valeurs maxi et mini
cout<<"Valeur maximum : "<<maxi<<endl;
cout<<"Valeur minimum : "<<mini<<endl;
return 0;
}
```

Résultat après exécution :

```
Saisissez un nombre entier : 5
Saisissez un nombre entier : 155
Saisissez un nombre entier : -145
Saisissez un nombre entier : 450
Saisissez un nombre entier : 0
Saisissez un nombre entier : 12
Valeur maximum : 450
Valeur minimum : -145
```

Le parcours d'un *tableau* à l'aide d'un *pointeur* est tout à fait possible. Bien qu'une *adresse* ne soit pas de *type* entier, nous pouvons tout de même l'*incrémenter*. En fait, le compilateur va dimensionner correctement l'*incrément* en fonction du *type* de la *variable* stockée. L'incrément est vu par le compilateur comme un *décalage (offset)* depuis l'*adresse* du premier élément du *tableau* considéré.

Des opérations arithmétiques (comme la somme, l'*incrémantion...*) avec des *pointeurs* sont possibles.



TABLEAU, POINTEUR ET CALCUL

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et initialisation d'un tableau
    //d'entiers
    int tableau[7]={5, 525, 32, 125, -45, 12, 0};
    //déclaration et initialisation de la variable i,
    index du tableau, et de la variable somme
    int i=0, somme=0;
    //boucle de lecture des contenus et de leurs
    adresses
    for(int *adr=tableau; adr<tableau+7; ++adr)
    {
        cout<<"tableau["<<i<<"] = "<<*adr<<" à
        l'adresse : "<<adr<<endl;
        //incrément de i
        ++i;
        //calcul de la somme intermédiaire
        somme+=*adr;
    }
    //affichage de la somme totale des éléments du
    tableau
    cout<<"La somme des éléments du tableau est égale à
    : "<<somme<<endl;
    return 0;
}
```

Résultat après exécution :

```
tableau[0] = 5 à l'adresse : 0xbffff6c8
tableau[1] = 525 à l'adresse : 0xbffff6cc
tableau[2] = 32 à l'adresse : 0xbffff6d0
tableau[3] = 125 à l'adresse : 0xbffff6d4
tableau[4] = -45 à l'adresse : 0xbffff6d8
tableau[5] = 12 à l'adresse : 0xbffff6dc
tableau[6] = 0 à l'adresse : 0xbffff6e0
La somme des éléments du tableau est égale à : 654
```

5.4 DEUX NOUVEAUX OPÉRATEURS, NEW ET DELETE

Lors de la déclaration d'un *pointeur*, on ne peut pas être sûre que l'*adresse* qu'il va utiliser ne soit pas déjà allouée à une autre *variable*.

Dans ce cas, une erreur est générée, le pointeur n'est pas initialisé et aucune zone mémoire n'est allouée.

Pour anticiper et résoudre ce problème nous pouvons utiliser les *opérateurs unaires new et delete*.

Ces mécanismes de *gestion dynamique de la mémoire* peuvent aussi se faire à l'aide des fonctions standards **malloc** et **free** qui existaient en langage C, toutefois on leur préférera **new** et **delete**, plus adaptés à la programmation objet.

L'opérateur **delete** effectue la fonction inverse de **new** puisqu'il libère la mémoire allouée. Un *pointeur* désalloué n'est plus initialisé et ne pointe plus sur rien.

Voici un petit programme de tri utilisant un *tableau* où chacune des valeurs réelles manipulées est déclarée explicitement.



TRI DE RÉELS

```
#include <iostream>
using namespace std;
int main () {
    //déclaration des variables entière
    int i, j, nb;
    //déclarations des pointeurs
    float *tab[25], *temp;
    //déclaration des réels
    float valeur;
    //saisie du nombre de valeurs à trier
    cout<<"Nombre de valeurs à trier (maxi : 25) : ";
    cin>>nb;
    //boucle de saisie des nombres
    for(i=0;i<nb;++i)
    {
        cout<<"Saisissez un nombre : ";
        cin>>valeur;
        //allocation mémoire explicite du nombre dans
        le tableau
        tab[i]=new float(valeur);
    }
    //boucles de tri
    for(i=0;i<nb;++i)
        for (j=i+1;j<nb;++j)
            //test d'inversion des valeurs via une
            variable temporaire
            if(*tab[j]<*tab[i])
            {
```

```
        temp=tab[j];
        tab[j]=tab[i];
        tab[i]=temp;
    }
    //boucle de lecture des valeurs séparées par 2
    tabulations
    for(i=0;i<nb;++i)
        cout<<*tab[i]<<"\t\t";
    return 0;
}
```

Résultat après exécution :

```
Nombre de valeurs à trier (maxi : 25) : 8
Saisissez un nombre : 12
Saisissez un nombre : 27.555
Saisissez un nombre : -123
Saisissez un nombre : -0.555
Saisissez un nombre : 5.55
Saisissez un nombre : 123
Saisissez un nombre : 10345
Saisissez un nombre : -23
-123   -23   -0.555   5.55   12   27.555   123   10345
```

5.5 POINTEUR DE POINTEUR

Au cours du développement d'un programme on peut être amené à définir un *pointeur de pointeur*.

Cette opération est possible et facilement réalisable.



POINTEURS EN CHAÎNE, ADRESSES ET CONTENUS

```
#include <iostream>
using namespace std;
int main () {
    //déclaration et affectation de la valeur 99 à la
    variable x
    int x=99;
    //affichage de x
    cout<<"x = "<<x<<endl;
    //affectation de l'adresse de x au pointeur
    int *px=&x;
    //affichage de la valeur contenue dans le pointeur
```

```

cout<<"Pointeur sur x = "<<*px<<endl;
//affichage de l'adresse de x
cout<<"Adresse de x = "<<&x<<endl;
//affectation de l'adresse du pointeur au pointeur
de pointeur
int **ppx=&px;
//affichage de la valeur contenue dans le pointeur
de pointeur
cout<<"Pointeur de pointeur sur x = "<<**ppx<<endl;
//affichage de l'adresse du pointeur
cout<<"Adresse du pointeur sur x = "<<&px<<endl;
//affectation de la valeur 100 au pointeur de
pointeur
**ppx=100;
//affichage de x
cout<<"x = "<<x<<endl;
return 0;
}

```

Résultat après exécution :

```

x = 99
Pointeur sur x = 99
Adresse de x = 0xbffff668
Pointeur de pointeur sur x = 99
Adresse du pointeur sur x = 0xbffff664
x = 100

```

Dans le programme ci-dessus la dernière *affectation* change la valeur entière contenue dans la *variable* **x**, elle devient égale à 100. En fait, ****ppx=100** envoie 100 au contenu de l'*adresse* **px**, pointée par l'*adresse* **ppx**.

5.6 DÉCLARATIONS DE POINTEURS

Tableau 5-1 QUELQUES EXEMPLES DE DÉCLARATIONS D'UN POINTEUR.

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 retourne un entier.

<code>int (*pt)(char *x)</code>	<code>pt</code> est un pointeur de fonction qui a pour argument un pointeur de type <code>char</code> . Elle retourne un entier.
<code>int *pt(char *x)</code>	<code>pt</code> est une fonction qui a pour argument un pointeur de type <code>char</code> . Elle retourne un pointeur sur un entier.
<code>int *(*pt)(char *a[])</code>	<code>pt</code> est un pointeur vers une fonction qui a pour argument un tableau de pointeurs de type <code>char</code> . Elle retourne un pointeur sur un entier.
<code>int *(*pt)(char (*x)[])</code>	<code>pt</code> est un pointeur vers une fonction qui a pour argument un pointeur vers un tableau de type <code>char</code> . Elle retourne un entier.

5.7 RÉCAPITULATIF

- Les *pointeurs* sont très utilisés en langage C++, il simplifie la manipulation des données, notamment quand celles-ci sont intégrées dans des *tableaux*.
- Pour manipuler correctement les *pointeurs*, il faut souvent gérer l'*allocation dynamique de la mémoire* via les opérateurs `new` et `delete`.
- Un *pointeur* peut pointer lui-même sur un autre *pointeur*.
- Il existe de nombreuses façons de déclarer des *pointeurs*, en fonction du traitement des données que l'on veut obtenir.

EXERCICES

5-1 Destructeur d'occurrences

Écrivez un programme, en utilisant des *pointeurs*, qui supprime une valeur entière `x` au sein d'un tableau `tabint` constitué d'un nombre `n` quelconque d'entiers.

Les valeurs `x`, `n` et le contenu du tableau `tabint` seront saisis par l'utilisateur.

Les éléments restant du tableau seront « compressés » afin d'éliminer les « trous ».

Exemple :

```
x=99  n=6 tabint={10, 15, 25, 99, 44, 99}
résultat à obtenir : tabint={10, 15, 25, 44}
```

5-2 Palindrome

Écrivez un programme qui teste si un mot est un palindrome (Un palindrome est un mot dont la succession des lettres est la même quand on le parcourt de gauche à droite ou de droite à gauche – exemple : « LAVAL »).

SOLUTIONS

5.1 Destructeur d'occurrences

```
#include <iostream>
using namespace std;
int main()
{
    //déclaration des variables et pointeurs
    int tabint[20], n, x;
    int *p1, *p2;

    //saisie de n, x et du contenu de tabint
    cout<<"Saisissez la taille du tableau (maxi 20) : ";
    cin>>n;
    for (p1=tabint; p1<tabint+n; ++p1)
    {
        cout<<"Elément n° "<<p1-tabint<<" : ";
        cin>>*p1;
    }
    cout<<"Saisissez la valeur à supprimer : ";
    cin>>x;
    //affichage du tableau de départ
    cout<<"Le tableau que vous avez saisi : "<<endl;
    for (p1=tabint; p1<tabint+n; ++p1) cout<<*p1<<endl;
    // supprimer x et compresser le tableau
    for (p1=p2=tabint; p1<tabint+n; ++p1)
    {
        *p2 = *p1;
        if (*p2!=x) ++p2;
    }
    //affectation à n de la nouvelle dimension de
    tabint
```

```
n = p2-tabint;
//affichage du tableau compressé
cout<<"Le tableau résultat : "<<endl;
for (p1=tabint; p1<tabint+n; ++p1) cout<<*p1<<endl;
return 0;
}
```

Résultat après exécution :

```
Saisissez la taille du tableau : 6
Elément n°0 : 10
Elément n°1 : 15
Elément n°2 : 25
Elément n°3 : 99
Elément n°4 : 44
Elément n°5 : 99
Saisissez la valeur à supprimer : 99
Le tableau que vous avez saisi :
10
15
25
99
44
99
Le tableau résultat :
10
15
25
44
```

5-2 Palindrome

```
#include <iostream>
using namespace std;
int main()
{
    //déclarations des variables et pointeurs
    char palin[50], *p1, *p2;
    int flag;
    //saisie des données
    cout<<"Saisissez un mot (maxi 50 caractères) : ";
    gets(palin);
    //positionner p2 en fin de chaîne
    for (p2=palin; *p2; ++p2);
    -p2;
    //passe flag1 à 1 ce qui impliquera que palin est
    un palindrome
    flag=1;
    for (p1=palin ; flag && p1<p2 ; ++p1, -p2)
```

```
    if (*p1!=*p2) flag=0;
    //test de flag1 et affichage du résultat
    if (flag) cout<<"Le mot "<<palin<<" est un
palindrome"<<endl;
    else
        cout<<"Le mot "<<palin<<" n'est pas un
palindrome "<<endl;
    return 0;
}
```

Résultat après exécution :

1 - Essai avec le mot : « malayalam »

```
Saisissez un mot (maxi 50 caractères) : malayalam
Le mot malayalam est un palindrome
```

2 - Essai avec le mot : « cocorico »

```
Saisissez un mot (maxi 50 caractères) : cocorico
Le mot cocorico n'est pas un palindrome
```



Fonctions

PLAN

- 6.1 Introduction
- 6.2 Écriture, syntaxe et utilisation
- 6.3 Premier programme utilisant une fonction
- 6.4 Récursivité
- 6.5 Fonction void
- 6.6 Passage par valeur et référence
- 6.7 Surcharge d'une fonction
- 6.8 Tableau et fonction
- 6.9 Particularités et remarques
- 6.10 Fonction exit()
- 6.11 Fonction inline
- 6.12 Récapitulatif

OBJECTIFS

- Créer et manipuler des fonctions simples
- Utiliser et exploiter la récursivité
- Comprendre les techniques du passage par valeur ou par référence
- Manipuler et utiliser la surcharge de fonctions
- Sortir d'un programme depuis une fonction
- Optimiser les temps de traitement et d'exécution d'un programme

6.1 INTRODUCTION

La présence et l'utilisation de *fonctions* au sein du langage C++ apportent une souplesse et une aisance dans la programmation. Elles offrent aux développeurs la possibilité de décomposer un ensemble de code

important en une série de sous-ensembles plus petits qui amènent une modularité et une simplification de la mise au point.

Les *fonctions* sont alors testées et compilées indépendamment. Le *débogage* est plus efficace et la création de programmes complexes plus facile.

En termes de clarté, le code devient plus lisible entraînant ainsi une meilleure *portabilité* ; les fonctionnalités spécifiques du programme étant isolées les unes des autres.

6.2 ÉCRITURE, SYNTAXE ET UTILISATION

Une *fonction* repose sur un ensemble d'instructions destinées à former un sous-programme qui assure une tâche précise.

Au sein d'un programme, on peut trouver de nombreuses *fonctions*, et tout programme C++ en possède au moins une, la fonction **main** (fonction ou programme principal) que nous manipulons depuis le début de cet ouvrage.

Vous trouverez ci-dessous quelques éléments nécessaires à la manipulation, l'écriture et la compréhension des fonctions :

- Toute fonction possède deux parties : l'**en-tête** et le **corps**.
- On appelle **déclaration** de la *fonction* l'en-tête suivi de son point-virgule.
- On appelle **définition** de *fonction*, la *fonction* complète : en-tête et corps.
- Dans un programme, si celui-ci contient plusieurs *fonctions*, l'ordre d'écriture de celles-ci est indifférent pour le compilateur.
- La *déclaration* de la *fonction* doit apparaître avant la première utilisation de la *fonction*, si bien que couramment, on ne donne avant le programme principal, **main**, que l'en-tête de la *fonction*, c'est le **prototypage** (*déclaration du prototype*).
- Si le *prototypage* n'est pas utilisé, la définition de la *fonction* complète doit être présente avant le programme principal qui va l'appeler.
- Le *prototype* est formé du nom de la *fonction*, du nombre et du *type* de ses *arguments*, et de son *type de retour*.
- Le *prototypage* n'est pas obligatoire, mais apporte une plus grande clarté dans la lecture et l'organisation générale du programme.
- Des *fonctions* ne sont jamais imbriquées les unes dans les autres, ni ne se chevauchent.
- L'appel d'une *fonction* au cours d'un programme peut être multiple.

- Une *fonction* traite les *variables* qui lui sont transmises via la liste des paramètres appelés *arguments* ou **paramètres formels** et génère en retour un ou plusieurs résultats.
- Dans le cas standard, une *fonction* ne peut retourner qu'un seul résultat. Si elle est de **type void**, elle n'en retourne aucun et si l'on utilise **le passage par référence**, elle peut en retourner plusieurs¹.
- Les *arguments formels* peuvent être des *constantes*, des *variables* ou des expressions, on mentionne toujours leurs *types*.
- Les paramètres sont des *variables* dites *locales* qui n'existent que pendant l'exécution de la *fonction* et seulement à l'intérieur de celle-ci.
- Une déclaration de *fonction* est comme une déclaration de *variable*, elle donne au compilateur les informations nécessaires à la compilation (nom, type des paramètres formels, et type de retour), le compilateur n'ayant pas besoin de savoir comment la *fonction* travaille (corps de la *fonction*).
- L'en-tête de la *fonction* contient le nom de la *variable* résultat qui sera retournée, son *type* et ses *arguments*.
- Le corps est le code même de la *fonction*. Il s'exécute jusqu'à la rencontre de l'instruction **return** qui le renvoie à l'emplacement de son appel et rend le contrôle à la *fonction main*.
- Dans le programme principal, lors de l'appel de la *fonction*, les paramètres passés sont dénommés **paramètres** ou **arguments effectifs** ou encore **réels**.

La syntaxe générale est la suivante :

- sans *prototypage* :

```
type nomfonction(argumentformel_1, argumentformel_2,...  
argumentformel_n)  
{  
    ...  
    return expression ;  
}  
  
main()  
{  
    ...  
    nomfonction(argumenteffectif_1, argumenteffectif_2,...  
argumenteffectif_n) ;  
    ...  
}
```

¹ Le type **void** et le **passage par référence** cités dans ce paragraphe seront approfondis et développées plus loin dans ce chapitre (§§ 6.5 et 6.6).

► avec prototypage :

```
type nomfonction(type argumentformel_1, type
argumentformel_2,... type argumentformel_n);

main()
{
    ...
nomfonction(argumenteffectif_1, argumenteffectif_2,...
argumenteffectif_n);
    ...
}

type nomfonction(argumentformel_1, argumentformel_2, ...
argumentformel_n)
{
    ...
return expression ;
}
```

6.3 PREMIER PROGRAMME UTILISANT UNE FONCTION

Dans ce premier programme utilisant une *fonction*, l'écriture repose sur le *prototypage*, mode de rédaction très utilisé dans la communauté des développeurs C++.



NOMBRE PREMIER

```
#include <iostream>
//appel de la bibliothèque cmath pour pouvoir utiliser
la fonction sqrt (racine carrée)
#include <cmath>
using namespace std;
//prototype de la fonction, nécessaire au compilateur
int nbpremier(int);
//programme principal
int main () {
    //déclaration de x, le nombre entier à saisir
    int x;
    //saisie de x
    cout<<"Saisissez un nombre entier : ";
    cin>>x;
    //test et affichage du résultat via l'appel de la
    fonction nbpremier auquelle on passe le nombre saisi
    if (nbpremier(x)==0) cout<<"Le nombre "<<"x"<<"
```

```
n'est pas premier" << endl;
    else cout << "Le nombre x est premier" << endl;
    return 0;
}
//fonction nbpremier pour tester la primarité d'un
//nombre, nest le nombre saisi
int nbpremier(int n)
{
    //déclaration de d, diviseur du nombre saisi
    int d;
    //déclaration et affectation de la racine carrée du
    //nombre n à la variable racine_p
    float racine_n=sqrt(n);
    //test si n<2, dans ce cas n n'est pas premier on
    //renvoie 0
    if(n<2) return 0;
    //test si n est égal à 2, dans ce cas n est premier
    //on renvoie 1
    if(n==2) return 1;
    //test si le reste de n diviser par 2 est null,
    //dans ce cas n n'est pas premier on renvoie 0
    if(n%2==0) return 0;
    //boucle suivant les nombres impairs de 3 à la
    //racine carrée de n
    for (d=3; d<=racine_n; d+=2)
        //test si le reste de n/d est null, dans ce
        //cas n n'est pas premier on renvoie 0
        if(n%d==0) return 0;
    return 1;
}
```

Résultat après exécution :

1 – Le nombre premier saisi n'est pas premier

```
Saisissez un nombre entier : 87
Le nombre x n'est pas premier
```

2 – Le nombre saisi est premier

```
Saisissez un nombre entier : 15773
Le nombre x est premier
```

L'expression qui suit l'instruction `return` fait généralement référence à la donnée renvoyée au programme appelant.

L'instruction `return` ne peut renvoyer qu'une seule valeur, toutefois la définition d'une *fonction* peut contenir plusieurs `return` avec des

expressions différentes correspondant aux alternatives générées par le code de la *fonction*.

Lorsque l'instruction `return` est seule, il n'est pas obligatoire de la préciser, mais cela reste tout de même conseillé afin d'assurer une bonne *tracabilité* du code.

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

Le programme qui suit n'utilise pas le *prototypage*.



PIUSSANCE D'UN NOMBRE

```
#include <iostream>
using namespace std;
//déclaration de la fonction puissance sans argument
float puissance ()
{
    //déclaration des variables
    float nb, resultat=1.0;
    int exp;
    //saisie du nombre
    cout<<"Saisissez un nombre : ";
    cin>>nb;
    //saisie de l'exposant
    cout<<"Saisissez un exposant : ";
    cin>>exp;
    //boucle de calcul du résultat
    for (int i=1; i<=exp; ++i) resultat*=nb;
    //renvoi du résultat
    return resultat;
}

//programme principal
int main () {
    //affichage du résultat via un appel de la fonction
    cout<<"Résultat : "<<puissance()<<endl;
    return 0;
}
```

Résultat après exécution :

```
Saisissez un nombre entier : 7.25
Saisissez un exposant : 4
Résultat : 2762.82
```

6.4 RÉCURSIVITÉ

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

L'écriture de son code apporte souvent concision, mais aussi complexité, au prix d'un *débogage* qui reste délicat.

Ce type de programmation offre un palliatif intéressant à des problèmes dont les solutions *itératives* seraient longues et fastidieuses à mettre en place.

Le programme qui suit reprend le calcul de la factorielle d'un nombre déjà évoqué dans le chapitre 4, § 4.4.



FONCTION RÉCURSIVE

```
#include <iostream>
using namespace std;
//prototype de la fonction fact
int fact(int);
int main () {
    //déclaration de nb
    int nb;
    //saisie de nb
    cout<<"Saisissez un nombre entier : ";
    cin>>nb;
    //affichage de la factorielle de nb
    cout<<nb<<"! = "<<fact(nb)<<endl;
    return 0;
}

//fonction fact
int fact(int n)
{
    //test si n est égal à 1 d, dans ce cas renvoie n
    if (n==1) return n;
    else
        //sinon calcule la factorielle via un appel
        //récuratif de la fonction fact
        n=(n*fact(n-1));
    //renvoie n
    return n;
}
```

Résultat après exécution :

```
Saisissez un nombre entier : 10
10! = 3628800
```

6.5 FONCTION VOID

On peut utiliser dans des cas précis des *fonctions* qui ne retournent aucune valeur, pour cela on fait appel à la *fonction void*.

Le mot-clé *void* est mis en lieu et place de la spécification du *type* de la *fonction*.



MOIS AU FORMAT LITTÉRAL

```
#include <iostream>
using namespace std;
//prototype de la fonction
void date(int, int, int);
int main () {
    //déclaration des variables
    int jour, mois, annee;
    //saisie du jour, du mois et de l'année
    cout<<"Saisissez en chiffres le jour : ";
    cin>>jour;
    cout<<"Saisissez en chiffres le mois : ";
    cin>>mois;
    cout<<"Saisissez en chiffres l'année : ";
    cin>>annee;
    //appel de la fonction date
    date(jour, mois, annee);
}

//fonction date
void date(int j, int m, int a)
{
    //déclaration et initialisation d'un tableau des
    mois
    string
    tabmois[12]={"janvier", "février", "mars", "avril", "mai", "
    juin", "juillet", "août", "septembre", "octobre", "novembre"
    , "décembre"};
    //déclaration de ms qui contiendra le mois
    string ms;
    //boucle pour déterminer le mois
    for (int i=0; i<12; ++i)
```

```
//teste le N° du mois  
if (m==i+1) ms=tabmois[i];  
//affichage de la date avec le mois en lettres  
cout<<j<<" "<<ms<<" "<<a<<endl;  
}
```

Résultat après exécution :

```
Saisissez en chiffres le jour : 12  
Saisissez en chiffres le mois : 2  
Saisissez en chiffres l'année : 1965  
12 février 1965
```

6.6 PASSAGE PAR VALEUR ET RÉFÉRENCE

Jusqu'à présent, lors de la transmission des *arguments* vers la *fonction*, nous avons utilisé un processus appelé **passage par valeur**.

Quand la *fonction* est appelée, le ou les *arguments effectifs* sont évalués, puis affectés à ou aux *arguments formels* présents dans l'*en-tête* de la *fonction* avant que son exécution puisse commencer.

Par exemple, lorsqu'on appelle la *fonction test(x)*, si la *variable x* est égale à 100, on passe la valeur 100 à la *variable locale y* de la *fonction* avant que celle-ci ne commence à exécuter le code qui la compose. Dans cette démarche, la valeur 100 est utilisée localement dans la *fonction*, donc *x* n'est pas touché par la *fonction*. La *variable x* est donc simplement un *argument effectif* en **lecture seule**.

En utilisant cette technique, une expression peut être passée comme *argument*. Par exemple, la *fonction test* pourrait avoir été appelée par *test(4/3*3.14*x*x*x)*. Quoi qu'il en soit, ce fonctionnement est très courant et la plupart du temps suffisant. Il permet d'avoir des *fonctions* très autonomes sans risque de *débordements*, toutefois il existe un autre moyen de passation d'un *argument*, c'est le **passage par référence**, utile dans certains cas.

Pour passer un *argument* par *référence*, le *type* spécifié de l'*argument* est suivi de & (et commercial). La *variable locale* devient alors une *référence* à l'*argument effectif* qui lui ait passé. Cet *argument effectif* est maintenant en **lecture-écriture** et non plus en *lecture seule*.



PASSAGE PAR RÉFÉRENCE

```
#include <iostream>
using namespace std;
//fonction test
void test(int x, int& y){
    //affectation de 100 à x et 1000 à y
    x=100;
    y=1000;
}

int main () {
    //déclaration et affectation de i et j
    int i=99, j=999;
    //affichage de i et j
    //x est local et on lui affecte i qui vaut 100,
    //y est une réplique de j dont la valeur est 1000
    cout<<"i = "<<i<<" et j = "<<j<<endl;
    //la fonction test passe i par valeur à x et j par
    //référence à y
    test(i,j);
    //affichage de i et j après appel de test
    //test affecte 100 à x sans manipuler i,
    //test affecte 100 à y et donc le réplique dans j
    //i est en lecture seule, j est en lecture-écriture
    cout<<"i = "<<i<<" et j = "<<j<<endl;
    return 0;
}
```

Résultat après exécution :

```
| i = 99 et j = 999
| i = 99 et j = 1000
```

Nous avons évoqué au début de ce chapitre, dans le § 6.2 « *Écriture, syntaxe et utilisation* », qu'une instruction `return` ne pouvait retourner qu'une seule valeur. Avec le passage des *arguments par référence*, on peut en retourner plusieurs.



RENOVIS MULTIPLES

```
#include <iostream>
using namespace std;
//prototype de la fonction sphere
//volume, surface et circonférence seront passés par
référence, rayon par valeur
void sphere(double&, double&, double&, double);
int main () {
    //déclaration des variables
    double r, v, s, c;
    //saisie du rayon
    cout<<"Saisissez le rayon : ";
    cin>>r;
    //appel de la fonction
    sphere(v, s, c, r);
    //affichage des résultats : volume, surface et
    circonférence
    cout<<"Volume : "<<v<<endl;
    cout<<"Surface : "<<s<<endl;
    cout<<"Circonférence : "<<c<<endl;
    return 0;
}
//fonction sphere
void sphere(double&, double&, double&, double)
{
    //déclaration et initialisation de la constante PI
    const double PI=3.14159265;
    //calcul du volume, de la surface et de la
    circonférence qui seront récupérées par référence
    //le rayon, passé par valeur, est seulement utile
    au calculs
    volume=4.0/3.0*PI*rayon*rayon*rayon;
    surface=4*PI*rayon*rayon;
    circonference=2*PI*rayon;
}
```

Résultat après exécution :

```
Saisissez le rayon : 25
Volume : 65449.8
Surface : 7853.98
Circonférence : 157.08
```

6.7 SURCHARGE D'UNE FONCTION

Elle permet l'utilisation de *fonctions* de mêmes noms qui portent des *arguments* différents. C'est par ce biais que le compilateur les différencie. Les *fonctions surchargées* sont très utilisées dans le langage C++ comme nous le verrons lors de notre étude des *classes* au chapitre 7.



SURCHARGE TRIPLE

```
#include <iostream>
using namespace std;
//3 prototypes différents par leurs arguments de la
fonction volume
double volume(double);
double volume(double, double);
double volume(double, double, double);
int main () {
    //déclaration des variables
    int choix;
    double cote, rayon, hauteur, longueur, largeur,
haut;
    //saisie du choix de calcul
    cout<<"Choisissez le volume à calculer (1-Cube, 2-
Cône, 3-Parallélépipède) : ";
    cin>>choix;
    //sélection en fonction du choix
    switch (choix){
        case 1:
            //saisie de la valeur du côté
            cout<<"Côté du cube : ";
            cin>>cote;
            //affichage et appel de la fonction
volume
            cout<<"Le volume du cube est égal à : "
<<volume(cote)<<endl;
            break;
        case 2:
            //saisie du rayon et de la hauteur du
cône
            cout<<"Rayon du cône : ";
            cin>>rayon;
            cout<<"Hauteur du cone : ";
            cin>>hauteur;
            //affichage et appel de la fonction
volume
```

```
        cout<<"Le volume du cône est égal à : "
<<volume(rayon, hauteur)<<endl;
        break;
    case 3:
        //saisie de la longueur, largeur et
        hauteur du parallélépipède
        cout<<"Longueur du parallélépipède :
";
        cin>>longueur;
        cout<<"Largeur du parallélépipède :
";
        cin>>largeur;
        cout<<"Hauteur du parallélépipède :
";
        cin>>haut;
        //affichage et appel de la fonction
        volume
        cout<<"Le volume du parallélépipède
est égal à : "<<volume(longueur, largeur, haut)<<endl;
        break;
    default:
        //affichage d'une erreur en cas de mauvais
        choix
        cout<<"Erreur !!!"<<endl;
        break;
    }
    return 0;
}
//fonction volume pour le calcul du volume du cube (1
argument)
double volume(double c)
{
    double v;
    v=c*c*c;
    return v;
}
//fonction volume pour le calcul du volume du cône (2
arguments)
double volume(double r, double h1)
{
    double v;
    //déclaration de la constante PI
    const double PI=3.141592653589793;
    v=PI/3*r*r*h1;
    return v;
}
```

```
//fonction volume pour le calcul du volume du
//parallélépipède (3 arguments)
double volume(double L, double l, double h2)
{
    double v;
    v=L*l*h2;
    return v;
}
```

Résultat après exécution :

1 – Calcul du volume d'un cube

Choisissez le volume à calculer (1-Cube, 2-Cône, 3-

Parallélépipède) : 1

Côté du cube : 5

Le volume du cube est égal à : 125

2 – Calcul du volume d'un cône

Choisissez le volume à calculer (1-Cube, 2-Cône, 3-

Parallélépipède) : 2

Rayon du cône : 3

Hauteur du cone : 2.5

Le volume du cône est égal à : 23.5619

3 – Calcul du volume d'un parallélépipède

Choisissez le volume à calculer (1-Cube, 2-Cône, 3-

Parallélépipède) : 3

Longueur du parallélépipède : 2.5

Largeur du parallélépipède : 4.55

Hauteur du parallélépipède : 2

Le volume du parallélépipède est égal à : 22.75

4 – Erreur de choix

Choisissez le volume à calculer (1-Cube, 2-Cône, 3-

Parallélépipède) : 6

Erreur !!!

6.8 TABLEAU ET FONCTION

Dans l'exemple qui suit, nous allons voir la manipulation de *tableaux* au travers de *fonctions*. C'est une opération souvent réalisée par les développeurs.



TRI D'ENTIERS

```
#include <iostream>
using namespace std;
//déclaration d'une constante pour la taille maximale
du tableau
const int nbval=25;
//prototypes des fonctions
void tri(int, float tableau[]);
void affiche(int, float tableau[]);
int main () {
    //déclaration des variables entières
    int nb;
    float tab[nbval];
    //saisie du nombre de valeurs à trier
    cout<<"Nombre de valeurs à trier (maxi : 25) : ";
    cin>>nb;
    //boucle de saisie des nombres
    for(int i=0;i<nb;++i)
    {
        cout<<"Saisissez un nombre : ";
        cin>>tab[i];
    }
    //appel de la fonction tri
    tri(nb, tab);
    //appel de la fonction affiche
    affiche(nb, tab);
    return 0;
}
//définition de la fonction tri
void tri(int nbr, float tableau[])
{
    float temp;
    //boucles de tri
    for(int i=0;i<=nbr;++i)
        for (int j=i+1;j<nbr;++j)
            //test d'inversion des valeurs via
une variable temporaire
            if(tableau[j]<tableau[i])
            {
                temp=tableau[j];
                tableau[j]=tableau[i];
                tableau[i]=temp;
            }
    return;
}
```

```
//fonction affiche
void affiche(int n, float tableau[])
{
    //boucle de lecture des valeurs séparées par 2
    tabulations
    for(int i=0;i<n;++i)
        cout<<tableau[i]<<"\t\t";
    return;
}
```

Résultat après exécution :

```
Nombre de valeurs à trier (maxi : 25) : 6
Saisissez un nombre : 12
Saisissez un nombre : 125
Saisissez un nombre : 5.5
Saisissez un nombre : -45.225
Saisissez un nombre : 0
Saisissez un nombre : 77
-45.225    0      5.5     12     77     125
```

6.9 PARTICULARITÉS ET REMARQUES

Que se passe-t-il quand une déclaration de *fonction* est présente à l'intérieur d'une autre *fonction* ?

Nous sommes amenés à préciser comme *argument* d'une *fonction* le nom d'une autre *fonction*.



FONCTIONS ENCHAÎNÉES

```
#include <iostream>
using namespace std;
//prototypes des fonctions
double add(double (*)(double), int);
double carre(double);
double cube(double);
int main () {
    int n;
    //saisie du nombre de valeurs à traiter
    cout<<"Nombre de valeurs à traiter : ";
    cin>>n;
```

```
//affichage des deux sommes via l'appel des
fonctions carre et cube suivant un nombre de valeurs à
traiter n
cout<<"Somme des carrés : "<<add(carre,n)<<endl;
cout<<"Somme des cubes : "<<add(cube,n)<<endl;
return 0;
}
//définition de la fonction add
//elle évalue la fonction sur laquelle pointe p
double add(double (*p)(double x), int y)
{
    double somme=0, nb;
    //boucle de calcul suivant un nombre d'itération
    égal à y
    for(int i=1;i<=y;++i)
    {
        //saisie d'un nombre
        cout<<"Saisissez un nombre : ";
        cin>>nb;
        //calcul de la somme cumulée
        somme+=(p)(nb);
    }
    return somme;
}
//définition de la fonction carre
double carre(double n1)
{
    double car;
    //calcul du carré
    car=n1*n1;
    //affichage du carré calculé
    cout<<"Carré de "<<n1<<" = "<<car<<endl;
    //retourne la valeur car
    return car;
}
//définition de la fonction cube
double cube(double n2)
{
    double cub;
    //calcul du cube
    cub=n2*n2*n2;
    //affichage du cube calculé
    cout<<"Cube de "<<n2<<" = "<<cub<<endl;
    //retourne la valeur cub
    return cub;
}
```

Résultat après exécution :

```
Nombre de valeurs à traiter : 3
Saisissez un nombre : 3.5
Carré de 3.5 = 12.25
Saisissez un nombre : 12
Carré de 12 = 144
Saisissez un nombre : -4
Carré de -4 = 16
Somme des carrés : 172.25
Saisissez un nombre : 2
Cube de 2 = 8
Saisissez un nombre : 8.2
Cube de 8.2 = 551.368
Saisissez un nombre : -3
Cube de -3 = -27
Somme des cubes : 532.368
```

6.10 FONCTION EXIT()

Dans des cas exceptionnels, il peut être intéressant de terminer un programme à l'intérieur d'une *fonction* différente de `main`, c'est le rôle de la fonction `exit()`.



SORTIE DE BLOC AVEC EXIT

```
#include <iostream>
using namespace std;
//prototype de la fonction inverse
double inverse(double);
int main () {
    double nb;
    //saisie d'un nombre
    cout<<"Saisissez un nombre : ";
    cin>>nb;
    //affichage de l'inverse via l'appel de la fonction
    inverse
    cout<<"L'inverse de "<<nb<<" est égal à "
    <<inverse(nb)<<endl;
    return 0;
}
//fonction inverse
double inverse(double x)
{
```

```
//test si x est égal à 0
if (x==0)
{
    //dans ce cas affiche un message d'erreur et
    termine le programme
    cout<<"Erreur - 0 n'a pas d'inverse !"<<endl;
    exit(0);
}
//retourne le résultat du calcul
return 1.0/x;
}
```

Résultat après exécution :

1 – Le nombre saisi est différent de 0

Saisissez un nombre : 2.5

L'inverse de 2.5 est égal à 0.4

2 – Le nombre saisi est égal à 0

Saisissez un nombre : 0

Erreur - 0 n'a pas d'inverse !

6.11 FONCTION INLINE

L'appel d'une *fonction* au sein d'un programme C++ entraîne des surcoûts de temps de traitement important (appel, passation des paramètres, mémorisation des variables locales, stockage de variables en cours...).

Pour éviter ce problème on peut faire appel à la fonction **inline**.

Dans ce cas, lors de chaque appel de la *fonction* au sein du programme, le compilateur la remplace explicitement par sa séquence de code correspondant, on dit qu'il traite en **extension** la *fonction*. On va donc faire une économie substantielle de temps au détriment d'une plus grande consommation de mémoire (le code du programme devient plus important).

La syntaxe à utiliser est la suivante :

inline type nomfonction(...)

Il faut remarquer que certains compilateurs C++ traitent en *extension* les *fonctions* même si elles ne sont pas déclarées *inline*.

6.12 RÉCAPITULATIF

Les *fonctions* apportent une grande souplesse dans la programmation C++ en découplant un programme complexe en sous-ensembles qui peuvent être testés séparément.

Une *fonction* est toujours constituée d'un en-tête et d'un corps.

Elle traite des *variables* qui lui sont transmises via des *paramètres* ou *arguments formels* qui génèrent en retour des résultats.

Son appel se fait généralement depuis une *fonction* ou le programme principal `main`.

Une *fonction* ne retourne pas obligatoirement une valeur, dans ce cas c'est une *fonction void*.

Elle peut s'appeler elle-même dans le cas de la *récursivité*.

Par l'intermédiaire du *passage par référence*, une *fonction* peut retourner plusieurs résultats.

La *surcharge* offre la possibilité d'utiliser une même *fonction* avec des *arguments* différents.

La *fonction inline*, par son traitement en *extension*, réduit les temps d'exécution.

EXERCICES

6-1 Suite de Fibonacci

Écrivez un programme utilisant une *fonction* récursive qui calcule le nombre de « Fibonacci » d'un nombre entier compris entre 0 et 30.

Un nombre de la suite de « Fibonacci » s'obtient en ajoutant les deux nombres précédents de la suite.

Si on note F_n , le $n^{\text{ième}}$ nombre de Fibonacci, $F_n = F_{n-1} + F_{n-2}$

Voilà les premiers nombres de la suite :

$F_1 = 1$; $F_2 = 1$; $F_3 = 2$; $F_4 = 3$; ...etc.

Indice n	1	2	3	4	5	6	7	8	9	10	11	...
F_n	1	1	2	3	5	8	13	21	34	55	89	...

6-2 Coordonnées cartésiennes vers polaires

Écrivez un programme qui convertit des coordonnées polaires en coordonnées cartésiennes dans l'intervalle $[0; 2\pi[$.

Dans un système où le couple de coordonnées cartésiennes est (x, y) ; le rayon r et l'angle a , des coordonnées polaires sont égales respectivement à :

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \text{et} \\ a &= \text{atan}(x/y) \text{ si } x > 0 \text{ et } y > 0 \\ a &= \text{atan}(y/x) + 2\pi \text{ si } x > 0 \text{ et } y < 0 \\ a &= \text{atan}(y/x) + \pi \text{ si } x < 0 \\ a &= \pi/2 \text{ si } x = 0 \text{ et } y > 0 \\ a &= 3\pi/2 \text{ si } x = 0 \text{ et } y < 0 \end{aligned}$$

SOLUTIONS

6-1 Suite de « Fibonacci »

```
#include <iostream>
using namespace std;
//prototype de la fonction fibonacci
int fibonacci(int);
int main()
{
    //déclaration de n
    int n;
    //saisie de n
    cout<<"Saisissez un nombre entier inférieur à 30 : ";
    cin>>n;
    //test si n est inférieur ou égal à 30 sinon
    affiche un avertissement
    if (n<=30)
        //affichage du résultat via l'appel de la
        fonction fibonacci
        cout << "F("<< n <<") = " << fibonacci(int(n)) << endl;
    else
        cout<<"Le nombre saisi doit être inférieur
ou égal à 30"<<endl;
    return 0;
}
//définition de la fonction fibonacci
int fibonacci(int n)
{
    //test si n est égal à 0, dans ce cas renvoie 0
```

```

    if (n == 0)
        return 0;
    //sinon test si n est égal à 1, dans ce cas renvoie
1
    else if (n == 1)
        return 1;
    //sinon renvoie le résultat via un appel récursif
de la fonction fibonacci sur n-1 + n-2
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

```

Résultat après exécution :

1 – Le nombre saisi est inférieur ou égal à 30

Saisissez un nombre entier inférieur à 30 : 15
F(15) = 610

2 – Le nombre saisi est supérieur à 30

Saisissez un nombre entier inférieur à 30 : 32
Le nombre saisi doit être inférieur ou égal à 30

6-2 Coordonnées cartésiennes vers polaires

```

#include <iostream>
//appel de la bibliothèque <cmath> pour sqrt et atan
#include<cmath>
using namespace std;
//déclaration de la constante PI
const double PI=3.1415926535;
//déclaration du prototype de la fonction cartpol
void cartpol(double&, double&, double, double);
int main () {
    //déclaration des variables
    double x, y, r, a;
    //saisie de x et y
    cout<<"Saisissez x : ";
    cin>>x;
    cout<<"Saisissez y : ";
    cin>>y;
    //appel de la fonction cartpol
    cartpol(r, a, x, y);
    //affichage du résultat
    cout<<"Pour le couple de coordonnées cartésiennes
(x,y) = ("<<x<<", "<<y<<")"<<endl;
    cout<<"le couple de coordonnées polaire (r, a) =
("<<r<<", "<<a<<")"<<endl;
}

```

```
    return 0;
}
//fonction cartpol
//r et a sont passés par référence car la fonction
cartpol doit renvoyer 2 valeurs en sortie
void cartpol(double& r, double& a, double x, double y)
{
    //calcul du rayon r
    r=sqrt(x*x+y*y);
    //calcul de l'angle a
    if (x>0)
        if (y>=0) a=atan(y/x);
        else a=atan(y/x)+2*PI;
    else
        if (x==0)
            if (y>0) a=PI/2;
            else
                if (y<0) a=3*PI/2;
        a=atan(y/x)+PI;
}
```

Résultat après exécution :

```
Saisissez x : -3
Saisissez y : 2
Pour le couple de coordonnées cartésiennes (x,y) = (-3,
2)
le couple de coordonnées polaire (r, a) = (3.60555,
2.55359)
```


PLAN

- 7.1 Introduction
- 7.2 Structures C en C++
- 7.3 Les classes
- 7.4 Constructeurs
- 7.5 Destructeurs
- 7.5 Récapitulatif

OBJECTIFS

- Appréhender les principales notions et le vocabulaire dédiés à la programmation orientée objet
- Comprendre et utiliser la notion de structure dans ses différents contextes
- Définir et manipuler les classes C++
- Instancier des objets
- Comprendre et implémenter les fonctions constructeurs et destructeurs d'une classe C++

7.1 INTRODUCTION

Dans le vieux langage C, on se limitait simplement à la notion de *structure* qui n'était ni plus ni moins qu'un simple *type* défini par l'utilisateur auquel on ne pouvait rien associer et qui, de ce fait, restait très statique. La programmation orientée objet (**POO**) qui est un des fondements du langage C++ franchie allégrement cette barrière en apportant de nouveaux concepts de programmation déjà présents au sein d'autres langages comme *modula*¹.

¹ Langage créé en 1977 par Niklaus Wirth à l'École polytechnique de Zurich. Voir aussi la partie historique dans le chapitre d'introduction, au début de cet ouvrage.

Elle apporte fiabilité, facilité de mise au point, et surtout *réutilisabilité* du code.

Les principaux principes qu'elle met en œuvre sont les *classes*, l'**encapsulation**, l'**héritage** et le **polymorphisme**.

Une *classe* peut être assimilée à un ensemble de *variables* de *types* distincts qui posséderait des *fonctions* manipulant des *opérateurs*.

À partir d'une *classe* on va pouvoir créer un **objet** par **instanciation**.

Définition : Un *objet* (**instance** d'une *classe*) est un composant logiciel avec lequel on va manipuler des données et des fonctions afin d'offrir de nouvelles fonctionnalités au programme.

Un *objet* possède toutes les caractéristiques d'une *variable* ordinaire, on pourrait dire qu'un *objet* n'est en fait qu'une *variable* dont le *type* est une *classe*.

L'*encapsulation* va permettre l'utilisation d'un *objet* tout en masquant son fonctionnement interne.

L'*héritage*, aussi appelé *dérivation*, est un procédé qui autorise la création de **sous-classes** à partir d'une *classe*, dite **super-classe** ou **sur-classe**.

Le *polymorphisme* offre à tout *objet* instancié d'une *classe* ancêtre, la possibilité d'être remplacée par un *objet* d'une *classe* descendante de la *classe* ancêtre.

Si nous possédons, par exemple, deux *objets* instanciés des *classes* : rectangle et triangle, nous allons pouvoir définir un *objet* d'une *classe* polygone qui pourra traiter des rectangles et des triangles. C'est ce que nous appelons le *polymorphisme d'objets*.

7.2 STRUCTURE C EN C++

Nous allons commencer par présenter la manipulation des *structures* comme celles utilisées dans le langage C.

Définition : Une *structure* est un conteneur qui regroupe sous un même nom des éléments de *types* différents. Chacun de ceux-ci est appelé *champs* ou *membres* de la *structure*.

Les *membres* peuvent être des *variables* ordinaires, des *pointeurs*, des *tableaux* ou d'autres *structures*.

Ils doivent tous avoir des noms distincts.

Pour déclarer une *structure*, nous allons définir le *type* et le nom des *variables* qu'elle va devoir manipuler.

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

La syntaxe adoptée est la suivante :

```
struct nomstructure{
    typemembre1 nommembre1;
    typemembre2 nommembre2;
    ...
    typemembren nommembren;
};
```

Pour accéder à un *champ* d'une *structure*, on utilise l'opérateur point (.). Ce dernier possède la *priorité* la plus élevée, il prévaut sur tous les *opérateurs unaires* et les opérateurs arithmétiques, logiques, d'*affectation* ou *relationnels*.



STOCK

```
#include <iostream>
using namespace std;
//déclaration de la structure article
//veste1 et veste2 sont les variables "structurées"
struct article{
    int ref, qte;
    float taille;
    float prix;
}veste1, veste2;

int main () {
    //affectations
    veste1.ref=111;
    veste1.qte=3;
    veste1.taille=50;
    veste1.prix=98.5;
    veste2.ref=122;
    veste2.qte=2;
    veste2.taille=52;
    veste2.prix=98.5;
    //affichage
    cout<<"Ref : "<<veste1.ref<< " / Qté : "
    <<veste1.qte<< " / Taille : "<<veste1.taille<< " / Prix
    : "<<veste1.prix<< " euros"<<endl;
    cout<<"Ref : "<<veste2.ref<< " / Qté : "
    <<veste2.qte<< " / Taille : "<<veste2.taille<< " / Prix
    : "<<veste2.prix<< " euros"<<endl;
    return 0;
}
```

Résultat après exécution :

```
| Ref : 111 / Qté : 3 / Taille : 50 / Prix : 98.5 euros
| Ref : 122 / Qté : 2 / Taille : 52 / Prix : 98.5 euros
```

Un *tableau* peut contenir des éléments issus d'une *structure*.



SALARIÉS

```
#include <iostream>
using namespace std;
//prototypes fonctions
void saisie(int);
void affichage(int);
//déclaration de la structure
struct personnel{
    char nom[25];
    char prenom[25];
    char sexe[1];
    int age;
}salarie[100];

int main () {
    int i, n;
    //saisie du nombre n de salariés à traiter
    cout<<"Nb de salariés à saisir : ";
    cin>>n;
    //appel de la fonction saisie n fois
    for(i=0; i<n; ++i) saisie(i);
    //appel de la fonction affichage n fois
    for(i=0; i<n; ++i) affichage(i);
    return 0;
}
//fonction saisie
void saisie(int i){
    cout<<"Nom : ";
    cin>>salarie[i].nom;
    cout<<"Prénom : ";
    cin>>salarie[i].prenom;
    cout<<"Sexe (M/F) : ";
    cin>>salarie[i].sexe;
    cout<<"Age : ";
    cin>>salarie[i].age;
}
//fonction affichage
void affichage(int i){}
```

```

        cout<<salarie[i].nom<< " - "<<salarie[i].prenom<< " - "
"<<salarie[i].sexe<< " - "<<salarie[i].age<< "
ans"<<endl;
}

```

Résultat après exécution :

```

Nb de salariés à saisir : 3
Nom : DUPONT
Prénom : Pierre
Sexe (M/F) : M
Age : 34
Nom : DURAND
Prénom : Anabelle
Sexe (M/F) : F
Age : 28
Nom : MARTIN
Prénom : Dominique
Sexe (M/F) : F
Age : 31
DUPONT - Pierre - M - 34 ans
DURAND - Anabelle - F - 28 ans
MARTIN - Dominique - F - 31 ans

```

Les *structures* peuvent être imbriquées, une *structure* étant membre d'une autre *structure*.



CLUB CINÉ

```

#include <iostream>
using namespace std;
//prototypes des fonctions
void saisie(int);
void lecture(int);

//déclaration de la structure film
struct film{
    char titre[40];
    int annee;
};

//déclaration de la structure cinefil
struct cinefil{
    char nom[30];
    char prenom[30];
    struct film filmpref;
}membreclub[100];

```

```
int main()
{
    //Saisie du nombre de membres à traiter
    int i, nbm;
    cout<<"Nombre de membres du club à traiter : ";
    cin>>nbm;
    //boucle de saisie
    for (i=0; i<nbm; ++i)
        saisie(i);
    //boucle de lecture et d'affichage
    for (i=0; i<nbm; ++i)
        lecture(i);
    return 0;
}
//fonction saisie
void saisie(int i)
{
    cout<<"Nom : ";
    cin>>membreclub[i].nom;
    cout<<"Prenom : ";
    cin>>membreclub[i].prenom;
    cout<<"Film préfere : ";
    cin>>membreclub[i].filmpref.titre;
    cout<<"Année : ";
    cin>>membreclub[i].filmpref.annee;
}
//fonction lecture
void lecture(int i)
{
    cout << membreclub[i].nom << " " <<
    membreclub[i].prenom << " - film préféré : " <<
    membreclub[i].filmpref.titre << " - "<<
    membreclub[i].filmpref.annee << endl;
}
```

Résultat après exécution :

```
Nombre de membres du club à traiter : 2
Nom : MARTIN
Prenom : Jeanne
Film prefere : Duel
Année : 1971
Nom : DUPONT
Prenom : Pierre
Film prefere : Solaris
Année : 2002
MARTIN Jeanne - film préféré : Duel - 1971
DUPONT Pierre - film préféré : Solaris - 2002
```

Il est possible de créer des *types* de données basés sur des *structures* via la directive `typedef`, que nous avons déjà rencontrée au chapitre 2, § 2.8 de cet ouvrage.

```
typedef struct{
    Int ref ;
    char designation[50] ;
    char couleur[12] ;
    float longueur ;
    float largeur;
    float hauteur ;
    float pxht ;
}meuble

meuble sofa, chaise, table, desserte, commode, console;
```

Dans l'exemple ci-dessus : `sofa`, `chaise`, `table`, `commode`, `desserte` et `console` sont définis comme des variables structurées de type `meuble`.

Les *structures* et les *pointeurs* cohabitent naturellement. L'adresse de rangement des *variables* d'une *structure* s'obtient classiquement via l'*opérateur &*.

Nous pouvons déclarer un *pointeur* sur cette *variable* de façon identique à toute autre.

L'accès à un des *membres* de la *structure* par l'intermédiaire de son *pointeur* répond à la syntaxe suivante :

```
| ptvarstruct->membrestruct
| ou de façon équivalente
| (*ptvarstruct).membrestruct
```

Avec `ptvarstruct`, une *variable pointeur* associée à la *structure* et `->` un *opérateur* d'un *type* équivalent à `. (point)` que nous avons déjà évoqué.



FILM ET ANNÉE

```
#include <iostream>
using namespace std;
//déclaration de la structure film
struct film{
    char titre[40];
    int annee;
}cine1, *cine2;
```

```

int main()
{
    //déclaration d'un tableau buffer de 40 caractères
    int an;
    //affectation de l'adresse de cine1 au pointeur
    cine2
        cine2=&cine1;
    //message de saisie du titre
    cout<<"Titre du film : ";
    //lit le titre saisi et le range dans la variable
    titre de la structure film
    cin.getline(cine2->titre,40);
    //message de saisie de l'année
    cout<<"Annee : ";
    //lit l'année et la range dans la variable buffer
    cin>>an;
    //accède et affecte le contenu de la variable an à
    la variable annee de la structure film via le pointeur
    cine2
        cine2->annee=an;
    //affiche le titre et l'année
    cout<<cine2->titre<<" - Annee : "<<cine2->annee
    <<endl;
    return 0;
}

```

Résultat après exécution :

```

Titre du film : Blade runner
Annee : 1982
Blade runner - Annee : 1982

```

L'exemple ci-dessus mérite quelques commentaires :

- La fonction `getline` facilite la lecture d'une suite de caractères quelconques. Elle lit les caractères sur le flot qui l'a appelé. Un caractère nul est toujours ajouté en fin de *chaîne* derrière les caractères lus.
- La ligne `cin.getline(cine2->titre,40);` pourrait être remplacée de façon équivalente par `cin.getline((*cine2).titre,40);`

7.3 LES CLASSES

Une *classe* peut être vue comme une *structure* dont certains *membres* sont des *fonctions* dites **fonctions membres** aussi appelées **méthodes**. Les autres *membres* étant nommés **données membres** ou **attributs**.

Les *fonctions membres* assurent des manipulations ou des traitements sur les *données membres* qui ne sont ni plus ni moins que des *variables* rangées à l'intérieur de la *classe*.

Elles possèdent un *type*, un nom et sont regroupées sous une *étiquette* qui va déterminer leurs accès.

Il existe trois *étiquettes* possibles appelées aussi **qualificatifs d'accès**. Elles déterminent la **portée**.

- **Private** : un accès aux *membres* est possible seulement depuis l'intérieur de la *classe*
- **Public** : un accès aux *membres* est autorisé depuis l'extérieur de la *classe*
- **Protected** : un accès aux *membres* est possible seulement aux *classes dérivées* ou *amies* (voir chapitre 8 §8.3).

Pour que le compilateur puisse déterminer de quelle *classe* une *fonction* est *membre*, il existe un *opérateur*, nommé **opérateur de résolution de portée**, dont la syntaxe est :

nomclasse::nomfonctionmembre



SURFACE ET VOLUME D'UN CÔNE

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI=3.14159;
//déclaration de la classe cone
class cone {
private:
    //déclaration des données membres ou attributs
    privés
    double rayon, hauteur, vol, surf;
public:
    //déclaration des fonctions membres ou méthodes
    publiques
    void dimensions(double, double);
    double volume(double, double);
    double surface(double, double);
};
//définition de la fonction dimensions
void cone::dimensions(double r, double h) {
    rayon=r*2.54;
    hauteur=h*2.54;
}
```

```

//définition de la fonction volume
double cone::volume(double r, double h) {
    //affectation du volume calculé à la variable vol
    vol=(PI*pow(rayon,2)*hauteur)/3;
    //renvoie de vol
    return vol;
}
//définitions de la fonction surface
double cone::surface(double r, double h) {
    double a;
    a=sqrt(pow(rayon,2)+pow(hauteur,2));
    //affectation de la surface calculée à la variable
    surf
    surf=PI*rayon*(rayon+a);
    //renvoie de surf
    return surf;
}
int main(){
    //déclarations
    double ray, ht;
    //saisie du rayon et de la hauteur
    cout<<"Rayon (en pouces) : ";
    cin>>ray;
    cout<<"Hauteur (en pouces) : ";
    cin>>ht;
    //on déclare C comme une instance de la classe cone
    cone C;
    //appel de la fonction membre dimensions
    C.dimensions(ray, ht);
    //affichage du volume et de la surface via l'appel
    de leurs fonctions membres
    cout<<"Volume : "<<C.volume(ray, ht)<<" cm3" << endl;
    cout<<"Surface : "<<C.surface(ray, ht)<<" cm2"
    << endl;
    return 0;
}

```

Résultat après exécution :

```

Rayon (en pouces) : 2
Hauteur (en pouces) : 4
Volume : 274.568 cm3
Surface : 262.358 cm2

```

Dans l'exemple ci-dessus, nous voyons la *déclaration* de la *classe cone* qui est composée de ses *membres privés* rayon, hauteur, vol et surf et ses membres publics, les fonctions ou méthodes dimensions, volume et surface.

Les variables ou attributs `rayon`, `hauteur`, `vol` et `surf` sont privés et accessibles depuis l'intérieur de la *classe*, suivant le principe de l'*encapsulation* déjà exposé.

L'objet `C` est une *instance* issue de la *classe* `cone`.

L'appel des *fonctions membres* se fait via l'*opérateur .* (point).

L'*opérateur de résolution de portée ::* est là pour spécifier que les fonctions `dimensions`, `volume` et `surface` sont attachés à la *classe* `cone`. Généralement, les *fonctions membres* sont définies hors de la déclaration de la classe afin d'apporter plus de clarté et de lisibilité dans le code et pour respecter le principe même de l'*encapsulation*, mais ce n'est pas une règle stricte, elles peuvent aussi être placées à l'intérieur.

La privatisation des variables : `rayon`, `hauteur`, `vol` et `surf` est motivée par le fait que le reste du programme n'a pas besoin de leur accéder. Par l'intermédiaire d'une même *classe*, le développeur peut instancier plusieurs objets. Cette notion est fondamentale et représente un des avantages principaux de l'utilisation des *classes*.



VOLUME DE 3 CÔNES

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI=3.14159;
//déclaration de la classe cone
class cone {
private:
    //déclaration des données membres ou attributs
    //privés
    double vol;
public:
    //déclaration des fonctions membres ou méthodes
    //publiques
    double volume(double, double);
};
//définition de la fonction volume
double cone::volume(double r, double h){
    //affectation du volume calculé à la variable vol
    vol=(PI*pow(r,2)*h)/3;
    //renvoie de vol
    return vol;
}
int main(){
    //déclarations (instanciations) de 3 instances de
    //la classe cone
```

```

    cone C1, C2, C3;
    //affichage des résultats via l'appel de la méthode
volume
    cout<<"Volume cone N°1 : "<<C1.volume(2, 4)<<endl;
    cout<<"Volume cone N°2 : "<<C2.volume(2.5,
4.25)<<endl;
    cout<<"Volume cone N°3 : "<<C3.volume(3,
4.5)<<endl;
    return 0;
}

```

Résultat après exécution :

```

Volume cone N°1 : 16.7551
Volume cone N°2 : 27.8162
Volume cone N°3 : 42.4115

```

7.4 CONSTRUCTEURS

Définition : Un *constructeur* est une *fonction membre* qui sert à initialiser les *données membres*.

Les caractéristiques principales d'un *constructeur* sont listées ci-dessous et doivent obligatoirement être prises en compte lors de son utilisation :

- Il porte le même nom que la *classe* dans laquelle il est placé.
- C'est la première *fonction membre* à être exécutée.
- Il peut posséder des *arguments*.
- Il n'a pas de *type* de retour.
- Sa définition n'est obligatoire que si elle est nécessaire au bon fonctionnement de la *classe*.

Si le développeur juge que la présence d'un *constructeur* n'est pas utile, un **constructeur par défaut** est défini automatiquement par le compilateur si la *classe* n'en possède pas.



VOLUME ET SURFACE D'UNE PYRAMIDE À BASE CARRÉE

```

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

```

```

//définition de la classe pyramide
class pyramide{
private:
    double c, h;
public:
    //constructeur de la classe pyramide
    pyramide(double x, double y){
        c=2.54*x; h=2.54*y;
    }
    //fonction membre pour le calcul de la surface
    double surface (void){
        double apotheme;
        apotheme=sqrt (pow(h,2)+pow(c/2,2));
        return (c*h*apotheme)/2+pow(c,2);
    }
    //fonction membre pour le calcul du volume
    double volume (void){
        return (pow(c,2)*h/3);
    }
};
int main(){
    //déclaration des variables cote et hauteur
    double cote, hauteur;
    //saisie
    cout<<"Côté (en pouces) : ";
    cin>>cote;
    cout<<"Hauteur (en pouces) : ";
    cin>>hauteur;
    //instanciation de la classe pyramide via la
    création d'un objet P
    pyramide P(cote, hauteur);
    //affichage du volume et de la surface via l'appel
    des méthodes de l'objet P
    cout<<"Volume (en cm3) : "<<P.volume()<<endl;
    cout<<"Surface (en cm2) : "<<P.surface()<<endl;
    return 0;
}

```

Résultat après exécution :

```

Côté (en pouces) : 2
Hauteur (en pouces) : 3
Volume (en cm3) : 65.5483
Surface (en cm2) : 181.268

```

Commentaires :

Dans l'exemple précédent, la fonction *constructeur* initialise l'*objet P* en affectant les valeurs spécifiées à ses *données membres*.

Quand la *déclaration de la classe P* a lieu, le *constructeur* est appelé automatiquement et la conversion de la hauteur et du côté de la pyramide, de pouces en centimètres, est effectuée.

Le langage C++ offre le moyen d'initialiser les *données membres* d'un *objet* par l'intermédiaire de **listes d'initialisation de constructeurs**. Des valeurs par défaut sont choisies, si aucun appel n'est passé lors de l'appel de la *fonction membre*.



SURFACE D'UN TRAPÈZE

```
#include <iostream>
using namespace std;
//déclaration de la classe trapeze
class trapeze {
private:
    double b, B, h;
public:
    //constructeur avec ses paramètres par défaut
    trapeze(double x=1, double y=2, double z=1) :b(x),B(y),h(z) {}
    double surface (void) {
        return ((b+B)*h/2);
    }
};

int main(){
    //création de 3 objets T1, T2 et T3 issus de la
    //classe trapeze
    //T1 utilise les valeurs par défaut spécifiée dans
    //le constructeur
    trapeze T1, T2(15, 20, 3.5), T3(5.5, 15, 4);
    //affichage des résultats
    cout<<"Surface T1 : "<<T1.surface()<<endl;
    cout<<"Surface T2 : "<<T2.surface()<<endl;
    cout<<"Surface T3 : "<<T3.surface()<<endl;
    return 0;
}
```

Résultat après exécution :

```
Surface T1 : 1.5
Surface T2 : 61.25
Surface T3 : 41
```

Nous avons déjà vu que lorsqu'une *classe* est créée, le compilateur génère automatiquement un *constructeur* par défaut. Il faut savoir qu'il existe aussi, un **constructeur par copie** qui est appelé, lui aussi, automatiquement quand un *objet* est dupliqué.

Le *constructeur par copie* possède un paramètre obligatoire qui est l'*objet* qui doit être copié passé par *référence constante*.

En créant notre propre *constructeur par copie* nous pourrons maîtriser plus facilement le programme, comme dans l'exemple qui suit, où la copie est accompagnée d'une conversion.



SURFACE D'UN TRAPÈZE EN CM² ET EN POUCES²

```
#include <iostream>
using namespace std;
//déclaration de la classe trapeze
class trapeze {
private:
    double b, B, h;
public:
    //constructeur avec ses paramètres par défaut
    trapeze(double x=1, double y=2, double
z=1) :b(x),B(y),h(z) {}
    //constructeur par copie de trapeze
    //le passage se fait par référence constante
    //la copie en profite pour effectuer une conversion
    en pouces des variables
    trapeze(const trapeze& T) : b(T.b/2.54),
B(T.B/2.54), h(T.h/2.54){}
    double surface (void){
        return ((b+B)*h/2);
    }
};
int main(){
    //création d'un objet T1 instancié depuis la classe
trapeze
    //petite base b : 15cm, grande base B : 20cm,
hauteur h: 6,5cm
    trapeze T1(15, 20, 6.5);
    //creation d'un objet T2 copie du constructeur T1
    trapeze T2(T1);
    //affichage de la surface calculée en cm2
    cout<<"Surface T1 : "<<T1.surface()<<endl;
```

```
//affichage de la surface calculée en pouces2 via
le constructeur par copie
cout<<"Surface T2 : "<<T2.surface()<<endl;
return 0;
}
```

Résultat après exécution :

```
Surface T1 : 113.75
Surface T2 : 17.6313
```

Dans l'exemple à venir, nous manipulerons un *pointeur* sur une *classe*. Pour accéder aux *membres* d'un *objet* géré par un *pointeur*, nous utiliserons l'*opérateur ->* déjà rencontré plus haut lors de l'étude des *structures*.

Nous remarquerons que comme pour les *structures* la notation via le *.* (point) est aussi possible, bien que les développeurs préfèrent souvent *->* qui semble plus explicite.



SURFACE D'UN TRIANGLE

```
#include <iostream>
using namespace std;
//Définition de la classe triangle
class triangle {
private:
    double b, h;
public:
    void dimensions(double, double);
    //fonction membre surface interne à la classe
    double surface (double b, double h){
        return (b*h/2);
    }
};
int main()
{
    //instanciation de 3 objets de la classe triangle,
    //dont 2 pointeurs
    triangle T1, *T2, *T3;
    cout<<"Surface T1 : "<<T1.surface(10, 20)<<endl;
    //utilisation de l'opérateur -> pour accéder à la
    //méthode surface
    cout<<"Surface *T2 : "<<T2->surface(15, 20)<<endl;
```

```
//utilisation de l'opérateur . équivalent à ->
cout<<"Surface *T3 : "<<(*T3).surface(20,
20)<<endl;
    return 0;
}
```

Résultat après exécution :

```
Surface T1 : 100
Surface *T2 : 150
Surface *T3 : 200
```

Dans le cas où une *donnée membre* est utilisée par toutes les *fonctions membres* de la classe, il existe la possibilité de la déclarer comme *donnée membre statique* en la faisant précéder du mot-clé **static** comme précisé ci-dessous.

```
static typemembre donneemembre
```

Une *variable statique* est automatiquement initialisée à 0. Dans le cas où une valeur non nulle est nécessaire nous pouvons effectuer une *initialisation explicite*.

Elle se comporte comme une *variable globale* ordinaire et il n'existe qu'une seule copie de la *variable* quel que soit le nombre d'*instances* de la *classe*.



SURFACE ET CIRCONFÉRENCE

```
#include <iostream>
using namespace std;
const double PI=3.14159;
//définition de la classe trapèze
class cercle {
private:
    double r;
    //donnée membre static
    static int agrand;
public:
    //constructeur
    cercle(double x):r(x){}
    //fonction membre surface
    double surface (void){
        return (PI*r*r*agrand);
    }
}
```

```

//fonction membre circonférence
double circonf(void) {
    return (2*PI*r*agrand);
}
};

//initialisation explicite de la donnée membre statique
agrand
int cercle::agrand=2;
int main(){
    //instanciation
    cercle C1(12.5);
    cercle C2(25);
    //affichage
    cout<<"Surface C1 : "<<C1.surface()<<endl;
    cout<<"Circonférence C1 : "<<C1.circonf()<<endl;
    cout<<"Surface C2 : "<<C2.surface()<<endl;
    cout<<"Circonférence C2 : "<<C2.circonf()<<endl;
    return 0;
}

```

Résultat après exécution :

```

Surface C1 : 981.747
Circonférence C1 : 157.079
Surface C2 : 3926.99
Circonférence C2 : 314.159

```

Tout comme une *donnée membre statique*, une *fonction membre* peut être elle aussi déclarée *statique*.

Elle sera alors utilisée comme une *fonction* indépendante des *instances* de la *classe* et invoquée comme *membre* d'une *classe*, de façon standard, via l'*opérateur de résolution de portée* `::`.

La syntaxe de sa *déclaration* au sein de la *classe* est la suivante :

```

static type nomfonctionmembre( ... )
{
    ...
    ... return expression ;
}

```

Dans ce cas, elle peut être appelée avant que les *objets* de la *classe* soient instanciés.

Une *fonction membre statique* n'accède qu'à des données *statiques* de sa propre *classe*.



COMPTER LES OBJETS

```
#include <iostream>
using namespace std;
//définition de la classe test
class test{
public:
    //constructeur
    test(){++compteur;}
    //fonction membre statique
    static int nombre(){return compteur;}
private:
    //donnée membre statique
    static int compteur;
};

//invocation de la fonction test via l'opérateur de
portée :: et initialisation
int test::compteur=0;
int main(){
    //affichage
    cout<<"Le nombre d'objets de la classe test est
égal à : "<<test::nombre()<<endl;
    //instanciation de l'objet x et affichage
    test x;
    cout<<"Le nombre d'objets de la classe test est
égal à : "<<test::nombre()<<endl;
    //instanciations des objets y et z et affichage
    test y;
    test z;
    cout<<"Le nombre d'objets de la classe test est
égal à : "<<test::nombre()<<endl;
    //instanciations des objets i, j, k et affichage
    test i, j, k;
    cout<<"Le nombre d'objets de la classe test est
égal à : "<<test::nombre()<<endl;
    return 0;
}
```

Résultat après exécution :

```
Le nombre d'objets de la classe test est égal à : 0
Le nombre d'objets de la classe test est égal à : 1
Le nombre d'objets de la classe test est égal à : 3
Le nombre d'objets de la classe test est égal à : 6
```

Une *donnée membre statique* ou une *fonction membre statique* peut être déclarée *privée* (`private`) ou *publique* (`public`).

7.5 DESTRUCTEURS

Quand un *objet* n'est plus utilisé nous allons pouvoir le supprimer de manière analogue à sa construction. Pour réaliser cette tâche nous utiliserons l'appel d'une *fonction* nommée **destructeur**.

Définition : Un *destructeur* est une *fonction membre* qui porte le même nom que la *classe*. À la différence d'un *constructeur* il est précédé du symbole `~` (tilde). Son rôle est de détruire un *objet* avant que la mémoire soit libérée.



SURFACE DE 2 TRAPÈZES

```
#include <iostream>
using namespace std;
//définition de la classe trapeze
class trapeze {
private:
    double b, B, h;
public:
    //constructeur
    trapeze(double x, double y, double z) :b(x), B(y), h(z)
    {}
    //destructeur
    ~trapeze () {}
    //fonction membre surface
    double surface (void){
        return ((b+B)*h/2);
    }
};
int main(){
    //instanciation de 2 objets T1 et T2 de la classe
    trapèze
    trapeze T1(10, 20, 2.5);
    trapeze T2(50, 100, 12.5);
    //affichage et appel de la fonction surface
    cout<<"Surface de T1 : "<<T1.surface()<<endl;
    cout<<"Surface de T2 : "<<T2.surface()<<endl;
    return 0;
}
```

Résultat après exécution :

```
| Surface de T1 : 37.5  
| Surface de T2 : 937.5
```

Une *classe* ne peut comporter qu'un seul et unique *destructeur*. Si celui-ci n'est pas déclaré explicitement, il est automatiquement créé par le compilateur et existe par défaut.

Un *destructeur* ne retourne aucune valeur.

7.6 RÉCAPITULATIF

La *programmation orientée objet (POO)* est basée principalement sur l'utilisation des *classes*.

À partir d'une *classe*, nous allons pouvoir instancier des *objets* qui peuvent être vus comme des *variables* dont le *type* serait une *classe*.

Au sein du langage C, il n'existait pas de concept *objet* mais on pouvait déjà manipuler des *structures* qui regroupent des éléments de types différents (*variables, tableaux, pointeurs...*).

Une *classe* est l'équivalent d'une super structure ou les *membres* peuvent être des *variables* (*données membres* ou *attributs*) ou des *fonctions* (*fonctions membres* ou *méthodes*).

Parmi les *fonctions* d'une *classe*, certaines d'entre elles ont des particularités comme le *constructeur* pour initialiser les données ou le *destructeur* pour supprimer les données.

Il existe plusieurs types de *constructeurs*, existant automatiquement lors de l'exploitation d'une *classe* par le programme, mais que nous pouvons aussi définir explicitement comme le *constructeur par défaut* ou le *constructeur par copie*.

Dans une *classe*, une donnée utilisée par plusieurs *fonctions* de la *classe* peut être déclarée comme *membre statique*.

EXERCICES

7-1 Déterminants d'un vecteur

Soient v_1 et v_2 , deux vecteurs définis par leurs coordonnées (x, y) et (x', y') extrémités, dans le plan euclidien orienté usuel.

Écrivez un programme, construit autour d'une *classe vecteur*, dont les *fonctions membres (méthodes)* afficheront les 2 vecteurs v_1 et v_2 , puis calculeront les déterminants $\det(v_1, v_2)$ et $\det(v_2, v_1)$.

Les coordonnées extrémités des 2 vecteurs v_1 et v_2 seront imposées.

Rappel mathématique : $\det(v_1, v_2) = xy' - yx'$

7-2 Matrice 2×2

Soit la matrice m1 ci-dessous :

$$\begin{pmatrix} 1 & 5 \\ 3 & 7 \end{pmatrix}$$

Écrivez un programme qui affiche la matrice m1, son déterminant, la matrice inverse $m1^{-1}$ et la copie m2 de la matrice m1.

Le code devra contenir une *classe* matrice qui intégrera un *constructeur par défaut*, un *constructeur par copie*, une *fonction* d'inversion de la matrice, une fonction de calcul du déterminant et une fonction d'affichage.

7-3 Dodécaèdre

Écrivez un programme qui calcule l'aire, le volume, le diamètre de la sphère inscrite et circonscrite d'un dodécaèdre.

Vous utiliserez une *classe*, qui contiendra les *fonctions membres* : aire, volume, diamètre sphère inscrite, diamètre sphère circonscrite et le coefficient c comme une donnée *membre statique*.

Vous prévoirez un *constructeur* et *destructeur* pour la *classe*.

Rappels sur le dodécaèdre :

$$\text{Aire : } A = 3\sqrt{25 + 10ca^2}$$

$$\text{Volume : } V = \frac{1}{4}(15 + 7c)a^3$$

$$\text{Diamètre sphère inscrite : } d = \sqrt{2,5 + 1,1ca}$$

$$\text{Diamètre sphère circonscrite : } D = \frac{1+c}{2}\sqrt{3a}$$

avec a, la longueur de l'arête et c = $\sqrt{5}$

SOLUTIONS

7-1 Déterminants d'un vecteur

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

```
//classe vecteur
class vecteur{
    float x,y;
public:
    // déclaration du constructeur
    vecteur(float,floor);
    //fonction membre affiche et det
    void affiche(int);
    float det(vecteur);
};

//définition du constructeur
vecteur::vecteur(float i=0,floor j=0){
    x=i;
    y=j;
}

//définition de la fonction affiche
void vecteur::affiche(int i){
    cout<<"v"<<i<<" (x, y) = ("<<x<<, "<<y<<") "<<"\n\n";
}

//définition de la fonction det
float vecteur::det(vecteur w){
    float determ;
    determ=x*w.y-y*w.x;
    return determ;
}

int main(){
    //appel du constructeur vecteur, initialisation et
    instantiation de v1 et v2
    vecteur v1(2, 3), v2(2.5, 7);
    //appel de la fonction affiche
    v1.affiche(1);
    v2.affiche(2);
    //affichage des résultats et appel de la fonction
    det
    cout<<"det (v1, v2) = xy' - yx'"<<endl;
    cout <<"det (v1, v2) = "<<v1.det (v2)<<"\n\n";
    cout<<"det (v2, v1) = x'y - y'x"<<endl;
    cout <<"det (v2, v1) = "<<v2.det (v1)<<endl;
    return 0;
}
```

Résultat après exécution :

```
v1 (x, y) = (2, 3)

v2 (x, y) = (2.5, 7)

det(v1, v2) = xy' - yx'
```

```

det(v1, v2) = 6.5

det(v2, v1) = x'y - y'x
det(v2, v1) = -6.5

```

7-2 Matrice 2×2

```

#include <iostream>
using namespace std;
//définition de la classe matrice
class matrice{
public:
    double a, b, c, d;
    //constructeur
    matrice(double w, double x, double y, double
z):a(w), b(x), c(y), d(z) {}
    //constructeur par copie
    matrice(const matrice& n):a(n.a), b(n.b), c(n.c),
d(n.d) {}
    //déclaration de la fonction inverse
    matrice inverse();
    //fonction determinant
    double determinant() {
        return a*d-b*c;
    }
    //déclaration de la fonction affiche
    void affiche();
private:
    double w, x, y, z;
};
//fonction inverse
matrice matrice::inverse(){
    cout<<"Déterminant de m1 : "<<determinant()<<endl;
    double i=1/determinant();
    matrice inv(i*d, -i*b, -i*c, i*a);
    cout<<"La matrice m1 inversée :"<<endl;
    inv.affiche();
    return inv;
}
//fonction affiche
void matrice::affiche(){
    cout<<a<<" " <<b<<endl;
    cout<<c<<" " <<d<<endl;
}
int main() {
    //instanciation de la matrice
    matrice m1(1, 5, 3, 7);
    //affichage et appel des fonctions
}

```

```
cout<<"La matrice m1 : "<<endl;
m1.affiche();
m1.inverse();
//copie de la matrice m1
matrice m2(m1);
//affichage de m2
cout<<"La copie m2 de la matrice m1 : "<<endl;
m2.affiche();
return 0;
}
```

Résultat après exécution :

```
La matrice m1 :
1 5
3 7
Déterminant de m1 : -8
La matrice m1 inversée :
-0.875 0.625
0.375 -0.125
La copie m2 de la matrice m1 :
1 5
3 7
```

7-3 Dodécaèdre

```
#include <iostream>
#include<cmath>
using namespace std;
//class dodeca
class dodeca {
private:
    double arete;
    //déclaration membre statique coef
    static double coef;
public:
    //constructeur
    dodeca(double x):arete(x) {}
    //destructeur
    ~dodeca () {}
    //fonction aire
    double aire (void){
        return (3*sqrt(25+10*coef))*arete*arete;
    }
    //fonction volume
    double volume(void){
        return (0.25*(15+7*coef))*pow(arete,3);
    }
    //fonction diametre sphere inscrite
```

```
double sphinsc(void) {
    return sqrt(2.5+1.1*coef)*arete;
}
//fonction sphcirc
double sphcirc(void) {
    return ((1+coef)/2)*sqrt(3)*arete;
}
};

//initialisation de coef
double dodeca::coef=sqrt(5);
int main() {
    double a;
    //saisie de la longueur de l'arête
    cout<<"Longueur de l'arête du dodécaèdre : ";
    cin>>a;
    //instanciation de l'objet D de la classe dodeca
    dodeca D(a);
    //calculs et affichage
    cout<<"Aire : "<<D.aire()<<endl;
    cout<<"Volume : "<<D.volume()<<endl;
    cout<<"Diamètre sphère inscrite : "<<D.sphinsc()<<endl;
    cout<<"Diamètre sphère circonscrite : "<<D.sphcirc()<<endl;
    return 0;
}
```

Résultat après exécution :

```
Longueur de l'arête du dodécaèdre : 5
Aire : 516.143
Volume : 957.89
Diamètre sphère inscrite : 11.1352
Diamètre sphère circonscrite : 14.0126
```

CHAPITRE 8

La surcharge des opérateurs

PLAN

- 8.1 Surcharge des opérateurs
- 8.2 Surcharge des opérateurs arithmétiques
- 8.3 La fonction friend
- 8.4 Surcharge des opérateurs relationnels
- 8.5 Le pointeur this
- 8.6 Surcharge de l'opérateur d'affectation
- 8.7 Surcharge des opérateurs d'entrée-sortie
- 8.8 Surcharge des autres opérateurs
- 8.9 Récapitulatif

OBJECTIFS

- Comprendre la notion de surcharge des opérateurs.
- Comprendre et maîtriser la fonction friend (amie) lors d'une opération de surcharge.
- Comprendre et manipuler le pointeur this.
- Connaître et savoir écrire les séquences de code nécessaires à la surcharge d'un opérateur.

8.1 SURCHARGE DES OPÉRATEURS

La **surcharge** ou **surdéfinition** des *opérateurs* est aussi appelée par les puristes : **polymorphisme ad hoc**. C'est une technique qui réalise, au travers des *classes*, via la création d'un nouveau *type*, des *fonctions* simulant des calculs spécifiques pour chacun des *opérateurs*. Lors des appels, nous pourrons traiter sous une autre forme les calculs que nous aurons à réaliser.

La *surcharge des opérateurs* est avant tout un moyen d'écrire plus lisiblement et plus clairement du code C++.

Il faut noter que cette technique a été et est encore souvent critiquée car elle permet au développeur de détourner un *opérateur* de ses fonctionnalités premières. Toutefois si l'écriture du programme est réalisée avec soin et bien commentée, cette critique reste banale et bien d'autres fonctionnalités du langage C++ nuisent aussi à la sémantique de compréhension (redéfinition d'une *fonction membre* par exemple).

Comme principe de la *surcharge* nous pouvons retenir ce qui suit.

Si nous instancions deux *objets* « rectangle1 » et « rectangle2 » d'une *classe* « rectangle » et que nous voulions en faire la somme, ou la différence, afin de calculer une surface, il nous faudra définir deux fonctions « additionner » et « soustraire » reprenant les différentes opérations puis affecter le résultat *via* des lignes de code du type :

```
somme = additionner(rectangle1, rectangle2);
difference = soustraire(rectangle1, rectangle2);
```

La surcharge de l'*opérateur +* va nous autoriser à écrire :

```
somme = rectangle1 + rectangle2
difference = rectangle1 - rectangle2
```

Le langage C++ comporte plus de 40 *opérateurs* répartis en plusieurs familles (voir annexe 4). Nous allons définir le mécanisme de *surcharge* pour chacune d'entre elles.

8.2 SURCHARGE DES OPÉRATEURS ARITHMÉTIQUES

Ce type de *surcharge* est très souvent utilisé et reste très pratique.

L'écriture de la *surcharge* d'un *opérateur* passe par une *fonction* dont le nom est **operator**, suivi du symbole d'opération. La liste des *arguments* est identique à celle du *constructeur par copie* de la classe ; ils sont passés par *référence* et sont du type de la *classe*.



RECTANGLES

```
#include <iostream>
using namespace std;
//définition de la classe rectangle
class rectangle{
public:
    float longueur, largeur;
```

```
//constructeur
rectangle(double L=0, double l=0){
    longueur=L;
    largeur=l;
}
//constructeur par copie
//il est précisé ici, mais n'est pas obligatoire
//puisque créé de façon automatique
//par le compilateur
rectangle(const rectangle&);
//fonction membre affiche
void affiche(){
    cout<<"Longueur rectangle rect3 : "<<longueur<<endl;
    cout<<"Largeur rectangle rect3 : "<<largeur<<endl;
}
//surcharge de l'opérateur +
rectangle operator+(rectangle& r1, rectangle& r2){
    rectangle R(r1.longueur+r2.longueur, r1.largeur+r2.largeur);
    return R;
}
int main () {
    //saisie des longueurs (L1, L2)
    //et largeurs (l1, l2) de 2 rectangles
    //rect1 et rect2
    double L1, l1, L2, l2;
    cout<<"Longueur rectangle rect1 : ";
    cin>>L1;
    cout<<"Largeur rectangle rect1 : ";
    cin>>l1;
    cout<<"Longueur rectangle rect2 : ";
    cin>>L2;
    cout<<"Largeur rectangle rect2 : ";
    cin>>l2;
    //instanciation de rect1 et rect2
    rectangle rect1(L1, l1);
    rectangle rect2(L2, l2);
    //addition des objets rect1 et rect2
    //pour créer l'objet rect3
    rectangle rect3=rect1+rect2;
    //affichage des dimensions de rect3
    rect3.affiche();
    return 0;
}
```

Résultat après exécution :

```
Longueur rectangle rect1 : 3
Largeur rectangle rect1 : 4
Longueur rectangle rect2 : 30
Largeur rectangle rect2 : 40
Longueur rectangle rect3 : 33
Largeur rectangle rect3 : 44
```

8.3 LA FONCTION FRIEND

Dans l'exemple précédent, les *données membres*, longueur et largeur de la classe rectangle sont de type public (*public*), si elles avaient été de *type privée* (*private*) l'opérateur surchargé + n'aurait pas eu la possibilité d'y avoir accès. Pour solutionner ce problème, le langage C++ nous offre la possibilité de déroger à cette règle lors d'une *surcharge* en déclarant cette *fonction* comme *amie* de la *classe* par l'intermédiaire du mot-clé **friend**.

Définition : Une fonction *friend* est une *fonction non membre* qui détient un droit d'accès vers l'ensemble de tous les *membres* de la *classe* pour laquelle elle se trouve déclarée.

Ses priviléges deviennent alors égaux à ceux d'une *fonction membre*.

Reprendons notre exemple en déclarant les *données membres* longueur et largeur comme étant privées et utilisons une *fonction amie*.



RECTANGLES ET FONCTION AMIE

```
#include <iostream>
using namespace std;
//définition de la classe rectangle
class rectangle{
    //données membres privées
private:
    float longueur, largeur;
public:
    //constructeur
    rectangle(double L=0, double l=0) {
        longueur=L;
        largeur=l;
    }
}
```

```
//constructeur par copie
//il est précisé ici, mais n'est pas obligatoire
//puisque créé de façon automatique
//par le compilateur
rectangle(const rectangle&);
//déclaration de la fonction amie
friend rectangle operator+(rectangle&, rectangle&);
//fonction membre affiche
void affiche(){
    cout<<"Longueur rectangle rect3 : "
<<longueur<<endl;
    cout<<"Largeur rectangle rect3 : "
<<largeur<<endl;
}
//surcharge de l'opérateur +
rectangle operator+(rectangle& r1, rectangle& r2){
    rectangle R(r1.longueur+r2.longueur,
r1.largeur+r2.largeur);
    return R;
}
int main () {
    //saisie des longueurs (L1, L2)
    //et largeurs (l1, l2) de 2 rectangles
    //rect1 et rect2
    double L1, l1, L2, l2;
    cout<<"Longueur rectangle rect1 : ";
    cin>>L1;
    cout<<"Largeur rectangle rect1 : ";
    cin>>l1;
    cout<<"Longueur rectangle rect2 : ";
    cin>>L2;
    cout<<"Largeur rectangle rect2 : ";
    cin>>l2;
    //instanciation de rect1 et rect2
    rectangle rect1(L1, l1);
    rectangle rect2(L2, l2);
    //addition des objets rect1 et rect2
    //pour créer l'objet rect3
    rectangle rect3=rect1+rect2;
    //affichage des dimensions de rect3
    rect3.affiche();
    return 0;
}
```

Résultat après exécution :

```
Longueur rectangle rect1 : 5
Largeur rectangle rect1 : 6
Longueur rectangle rect2 : 50
Largeur rectangle rect2 : 60
Longueur rectangle rect3 : 55
Largeur rectangle rect3 : 66
```

8.4 SURCHARGE DES OPERATEURS RELATIONNELS

Les six *opérateurs relationnels* `==`, `!=`, `<`, `<=`, `>`, `>=` peuvent aussi être *surchargés*. Ils retournent le plus souvent un type entier (*int*) qui représente une valeur vraie (1) ou fausse (0), comme dans l'exemple suivant.



RECTANGLES ÉGAUX

```
#include <iostream>
using namespace std;
//déclaration de constantes
int const TRUE=1, FALSE=0;
//définition de la classe rectangle
class rectangle {
private:
    double longueur, largeur;
public:
    rectangle(double L, double l) {
        longueur=L;
        largeur=l;
    }
    //surcharge de l'opérateur !=
    double operator !=(rectangle R) {
        if (R.longueur*R.largeur==longueur*largeur)
            return TRUE;
        else return FALSE;
    }
};

int main()
{
    //déclarations
    double L1, l1, L2, l2, L3, l3;
    //saisies
    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;
//instanciation
rectangle R1(l1, l1);
rectangle R2(L2, l2);
rectangle R3(L3, l3);
//tests utilisant l'opérateur surchargé
//et affichage des résultats
if (R1!=R2) cout<<"Rectangles 1 et 2
identiques"<<endl;
else cout<<"Rectangles 1 et 2
différents"<<endl;
if (R2!=R3) cout<<"Rectangles 2 et 3
identiques"<<endl;
else
{
    cout<<"Rectangles 2 et 3 différents"<<endl;
    if (R1!=R3) cout<<"Rectangles 1 et 3
identiques"<<endl;
}
return 0;
}
```

Résultat après exécution :

```
Longueur rectangle 1 : 10
Largeur rectangle 1 : 25
Longueur rectangle 2 : 10
Largeur rectangle 2 : 24.5
Longueur rectangle 3 : 10
Largeur rectangle 3 : 25
Rectangles 1 et 2 différents
Rectangles 2 et 3 différents
Rectangles 1 et 3 identiques
```

8.5 LE POINTEUR THIS

Toutes les *classes* ont un *pointeur* caché qui a pour nom : **this**. Ce pointeur sur l'*objet* est accessible à l'intérieur de la *fonction membre*. De ce fait, ***this** représente l'*objet* lui-même.

Dans une *fonction membre* d'un *objet* qui aurait pour donnée **i**, les deux instructions suivantes sont équivalentes :

```
i = 2;
this->i = 2;
```

Le *pointeur this* est constant, il ne peut pas être modifié, il est donc impossible de l'intégrer comme terme d'une opération arithmétique.

Dans l'exemple qui suit le *constructeur* utilise le *pointeur this*.

Nous retrouverons ce *pointeur* lors de l'opération de *surcharge* de l'opération d'*affectation* **=**, au paragraphe 8.6.



VITESSE MOYENNE EN KM/H

```
#include <iostream>
using namespace std;
//définition de la classe vitesse
class vitesse {
private:
    float t, d;
public:
    //fonction vitesse
    vitesse(float, float);
    float vit (void){
        return (d/t*3.6);
    }
};
//constructeur utilisant this
vitesse::vitesse(float x, float y) {
    this->t=x;
    this->d=y;
}

int main(){
    //déclarations
    float temps, distance;
    //saisies
    cout<<"Temps en secondes : ";
    cin>>temps;
    cout<<"Distance en mètres : ";
```

```
    cin>>distance;
    //instanciation
    vitesse V1(temps, distance);
    //appel de la fonction vit pour l'objet V1
    //et affichage du résultat
    cout<<"Vitesse V1 = "<<V1.vit()<<"km/h"<<endl;
    return 0;
}
```

Résultat après exécution :

```
Temps en secondes : 25
Distance en mètres : 1000
Vitesse V1 = 144km/h
```

8.6 SURCHARGE DE L'OPERATEUR D'AFFECTATION

Parmi l'ensemble des *opérateurs*, l'*opérateur d'affectation* = est le plus utilisé lors du développement d'un programme. Son rôle principal est de transférer le contenu d'une *variable* ou d'un *objet* dans un autre.

Sa *surcharge* est plus délicate que pour les autres *opérateurs* car il faut tester, lors de l'affectation, que le transfert s'effectue sur des *objets* différents. Dans le cas contraire, cela conduirait à une situation anachronique de l'*affectation* d'un *objet* à lui-même.

Il faut retenir qu'une *initialisation* fait appel à un *constructeur* alors qu'une *affectation* fait appel à l'*opérateur d'affectation* comme nous pouvons le voir dans l'exemple suivant.



SURFACES ET AFFECTATIONS

```
#include <iostream>
using namespace std;
//définition de la classe rectangle
class rectangle {
private:
    double L, l;
public:
    //constructeur
    rectangle(double x=2, double y=2.5):L(x),l(y){};
    //constructeur par copie
    rectangle(const rectangle&);
    //surcharge de l'opérateur d'affectation
    //le résultat doit être propriétaire de l'appel
```

```

rectangle& operator =(const rectangle& R) {
    //test qui vérifie
    //que les objets ne sont pas identiques
    //afin d'effectuer une affectation correcte
    if (&R != this){
        //affectations
        this->L=R.L;
        this->l=R.l;
    }
    return *this;
}
//fonction surface
double surface (void){
    return L*l;
};

int main()
{
    //instanciation et initialisations
    rectangle R1, R2(2, 5), R3(2.5, 10);
    //affichage avant affectations
    cout<<"Avant affectations : "<<endl;
    cout<<"R1 = "<<R1.surface()<<" // R2 = "
<<R2.surface()<<" // R3 = "<<R3.surface()<<endl;
    //affectations
    R1=R2;
    R2=R3;
    R3=R1;
    //affichage après affectations
    cout<<"Après affectations : "<<endl;
    cout<<"R1 = "<<R1.surface()<<" // R2 = "
<<R2.surface()<<" // R3 = "<<R3.surface()<<endl;
    //affectations en cascade
    R1=R2=R3;
    //affichage après affectations
    cout<<"Après affectations en cascade : "<<endl;
    cout<<"R1 = "<<R1.surface()<<" // R2 = "<<R2.surface()<<
// R3 = "<<R3.surface()<<endl;
    return 0;
}

```

Résultat après exécution :

```

Avant affectations :
R1 = 5 // R2 = 10 // R3 = 25

```

```
Après affectations :  
R1 = 10 // R2 = 25 // R3 = 10  
Après affectations en cascade :  
R1 = 10 // R2 = 10 // R3 = 10
```

8.7 SURCHARGE DES OPERATEURS D'ENTRÉE-SORTIE

Les *opérateurs d'insertion* `>>` et d'extraction `<<` dans le flux, aussi appelés **opérateurs d'entrée-sortie**, sont très souvent surchargés par les développeurs qui leur confèrent alors une personnalisation adaptée aux traitements en cours.

Cette opération de *surcharge* va devoir utiliser des *classes* existantes au sein du fichier d'en-tête *iostream*, qui sont *ostream* pour l'extraction de flux et *istream* pour l'insertion.

Les paramètres d'entrée comme la valeur de retour seront passés par *référence*. Ces *opérateurs* surchargés seront des *fonctions amies*.

Le code qui suit met en application ces principes.



VOLUME PARALLÉLÉPIPÈDE

```
#include <iostream>  
using namespace std;  
//définition de la classe parallelepiped  
class parallelepiped{  
private:  
    double l, h, p;  
public:  
    //constructeur  
    parallelepiped(double x=0, double y=0, double  
z=0){  
        l=x;  
        h=y;  
        p=z;  
    }  
    //fonctions amies  
    friend ostream& operator <<(ostream&, const  
parallelepiped&);  
    friend istream& operator >>(istream&,  
parallelepiped&);  
};  
//surcharge de l'opérateur <<  
//l'opérateur << retournera le volume,  
//suivi de l'affichage de l'unité
```

```

ostream& operator <<(ostream& ostr, const
parallelepiped& pll){
    //Appel de l'opérateur << surchargé
    return ostr<<(pll.l)*(pll.h)*(pll.p)<<" m3";
}
//surcharge de l'opérateur >>
//l'opérateur >> intègre les invites de saisie
//pour l, h et p
istream& operator >>(istream& istr, parallelepiped&
pll){
    cout<<"Longueur : ";
    istr>>pll.l;
    cout<<"Hauteur : ";
    istr>>pll.h;
    cout<<"Profondeur : ";
    istr>>pll.p;
    return istr;
}
int main()
{
    //instanciation
    parallelepiped P;
    //utilisation de l'opérateur surchargé >>
    cin>>P;
    //affichage du résultat via l'opérateur surchargé
<<
    cout<<"Volume : "<<P<<endl;
    return 0;
}

```

Résultat après exécution :

```

Longueur : 5
Hauteur : 3
Profondeur : 2.5
Volume : 37.5 m3

```

8.8 SURCHARGE DES AUTRES OPÉRATEURS

Nous venons de voir la *surcharge* ou *surdéfinition* des *opérateurs arithmétiques*, *relationnels*, d'*affectation* et d'*entrée-sortie*, il existe d'autres *opérateurs* qui supportent aussi cette opération comme les *opérateurs arithmétiques d'affectation* $+=$, $-=$, $*=$, $/=$, $\%=$, les *opérateurs d'incrémentation* et de *décrémantation* $++$, $--$ et l'*opérateur d'indexation* $[]$.

Nous allons définir la syntaxe d'implémentation de l'*opérateur de surcharge* pour chacun d'entre eux, en sachant que leur emploi se fait de façon semblable à ceux déjà étudiés.

► **Opérateurs arithmétiques d'affectation :**

```
nomclasse operator +=(const nomclasse&);  
nomclasse operator -=(const nomclasse&);  
nomclasse operator *=(const nomclasse&);  
nomclasse operator /=(const nomclasse&);  
nomclasse operator %= (const nomclasse&);
```

► **Opérateurs d'incrémentation, décrémentation :**

```
nomclasse operator++(); pour le type préfixé  
nomclasse operator++(int); pour le type postfixé  
nomclasse operator--(); pour le type préfixé  
nomclasse operator--(int); pour le type postfixé
```

► **Opérateurs d'indexation :**

```
type operator [] (type);
```

D'autres *opérateurs* encore, comme new, delete, &, *, (int) transystème,... précisés en annexe 4 de cet ouvrage, supportent aussi la *surcharge*.

8.9 RÉCAPITULATIF

Le mécanisme de *surcharge* des *opérateurs* améliore la lisibilité et la clarté du code.

De nombreux *opérateurs* du langage C++ peuvent être surchargés.

Par la nature même de l'opération de *surcharge*, on doit souvent utiliser une ou plusieurs *fonctions amies* qui sont des *fonctions membres* capables d'accéder à tous les *membres* d'une *classe*.

Les *classes* ont toutes un *pointeur this* qui est souvent utile dans les opérations de surcharge. Le *pointeur this* représente l'*objet* lui-même.

La *surcharge* de l'*opérateur d'affectation* est très utilisée, mais souvent délicate à mettre en place.

La *surcharge* des *opérateurs d'entrée-sortie* est très utile pour redéfinir un affichage ou une saisie de données.

EXERCICES

8-1 Somme de vecteurs

Soient v_1 , v_2 et v_3 , trois vecteurs définis par leurs coordonnées (x , y), écrivez un programme faisant la somme $v_4 = v_1 + v_2 + v_3$ et $v_5 = v_4 + v_2$.

Il utilisera une *classe vecteur* contenant un *constructeur*, une *fonction d'affichage* des valeurs x et y résultats et une *fonction amie de surcharge* de l'*opérateur +* pour pouvoir faire des sommes en cascade.

8-2 Somme et produit de complexes

Soient c_1 à c_8 des nombres complexes sous la forme $c = (x, y)$ avec $c_1 = (4, 2)$, $c_2 = (6, 3)$ et $c_3 = (12, 6)$.

Écrivez un programme qui calcule les sommes $c_4 = c_1 + c_3$, $c_5 = c_1 + c_4$, $c_6 = c_2 + c_4 + c_5$ et les produits $c_7 = c_1 \times c_3$, $c_8 = c_4 \times c_5$.

Vous devrez utiliser une *classe complexe* munie d'un *constructeur*, d'une *fonction d'affichage* et utilisant la *surcharge des opérateurs + et **.

SOLUTIONS

8-1 Somme de vecteurs

```
#include <iostream>
using namespace std;
// Classe vecteur
class vecteur{
private:
    float x,y;
public:
    //déclaration constructeur
    vecteur(float, float);
    void affichage();
    //fonction amie pour la surcharge de l'opérateur +
    friend vecteur operator+(vecteur, vecteur);
};
//définition constructeur
vecteur::vecteur(float X=0, float Y=0) {
    x=X;
```

```

        y=Y;
    }
//fonction d'affichage
void vecteur::affichage(){
    cout<<"x = "<<x<<" et y = "<<y<<endl;
}
//définition de la surcharge de l'opérateur +
vecteur operator+(vecteur i, vecteur j){
    vecteur s;
    s.x = i.x+j.x;
    s.y = i.y+j.y;
    return s;
}
int main(){
    //instanciation
    vecteur v1(5, 7.5), v2(3, -2), v3(5, 1), v4, v5;
    //somme
    v4 = v1+v2+v3;
    //affichage
    cout<<"v4=v1+v2+v3 avec ";
    v4.affichage();
    //somme
    v5 = v4+v2;
    //affichage
    cout<<"v5=v4+v2 avec ";
    v5.affichage();
    return 0;
}

```

Résultat après exécution :

```

v4=v1+v2+v3 avec x = 13 et y = 6.5
v5=v4+v2 avec x = 16 et y = 4.5

```

8-2 Somme et produit de complexes

```

#include <iostream>
using namespace std;
//classe complexe
class complexe{
private:
    double reel,imagin;
public:
    //constructeur
    complexe(double reel=0, double imagin =0);
    //surcharge de l'opérateur +
    complexe operator+(const complexe&) const;
    //surcharge de l'opérateur *

```

```
complexe operator*(const complexe&) const;
//fonction affichage
void affichage();
}
//définition du constructeur
complexe::complexe(double r, double i){
    reel=r;
    imagin=i;
}
//définition de la surcharge de l'opérateur +
complexe complexe::operator+ (const complexe& c) const
{
    complexe somme;
    somme.reel=(this->reel + c.reel);
    somme.imagin=(this->imagin + c.imagin);
    return somme;
}
//définition de la surcharge de l'opérateur *
complexe complexe::operator* (const complexe& c) const
{
    complexe produit;
    produit.reel=(this->reel * c.reel-this->imagin * c.imagin);
    produit.imagin=(this->reel * c.imagin +this->imagin * c.reel);
    return produit;
}
//fonction affichage
void complexe::affichage(){
    cout<<"partie réelle = "<<reel<<" et partie imaginaire
= "<<imagin<<endl;
}
int main()
{
    //instanciation
    complexe c1(4,2), c2(6,3), c3(12, 6), c4, c5, c6, c7,
c8;
    //calculs et affichages
    cout<<"Sommes : "<<endl;
    c4=c1+c3;
    c4.affichage();
    c5=c1+c4;
    c5.affichage();
    c6=c2+c4+c5;
    c6.affichage();
    cout<<"Produits : "<<endl;
    c7=c1*c3;
```

```
c7.affichage();  
c8=c4*c5;  
c8.affichage();  
}
```

Résultat après exécution :

```
Sommes :  
partie réelle = 16 et partie imaginaire = 8  
partie réelle = 20 et partie imaginaire = 10  
partie réelle = 42 et partie imaginaire = 21  
Produits :  
partie réelle = 36 et partie imaginaire = 48  
partie réelle = 240 et partie imaginaire = 320
```


CHAPITRE 9

Agrégation, héritage, polymorphisme et patrons

PLAN

- 9.1 L'agrégation
- 9.2 L'héritage
- 9.3 L'héritage multiple
- 9.4 Le polymorphisme
- 9.5 Les patrons de fonctions
- 9.6 Les patrons de classe
- 9.7 Récapitulatif

OBJECTIFS

- Connaître et comprendre les différentes techniques liées à la programmation orientée objet (POO).
- Mettre en place des classes dérivées et exploiter les notions d'héritage simple et multiple
- Mettre en œuvre le polymorphisme, utiliser les fonctions virtuelles et comprendre leur utilité.
- Exploiter les patrons au travers des fonctions et des classes en créant des modèles génériques.

9.1 L'AGRÉGATION

C'est une technique qui définit une *classe* comme étant liée à plusieurs *classes* différentes, on l'appelle aussi **composition**.

Dans l'exemple suivant la *classe copain* utilise la *classe string* pour déclarer des données.



RÉPERTOIRE TÉLÉPHONIQUE

```
#include <iostream>
#include <string>
using namespace std;
//classe copain
class copain{
public:
    //constructeur
    copain(char* n="", char* p="", char* t=""):nom(n), pre-
nom(p), tel(t){}
    void affichage(){
        cout<<"Prénom : "<<prenom<<" Nom : "<<nom;
    }
    void affichtel(){
        cout<<" - Téléphone : "<<tel;
    }
private:
    //déclaration de nom, prenom et tel via la classe string
    string nom, prenom, tel;
};
int main() {
    //instanciation de a
    copain a("Marcel", "DUPONT", "0102030405");
    //affichage via les fonctions membres
    a.affichage ();
    a.affichtel();
    return 0;
}
```

Résultat après exécution :

| Prénom : Marcel - Nom : Marcel - Téléphone : 0102030405

9.2 L'HÉRITAGE

À partir d'une *classe primaire* appelée *super-classe*, nous allons créer une *classe dérivée*. C'est le principe de *l'héritage* aussi nommé *dérivation*.

Les *membres publics* (`public`) de la *classe primaire* vont devenir des *membres publics* de la *classe dérivée*.

Pour accéder aux *membres privés* (`private`) d'une *classe primaire* depuis une *classe dérivée*, nous transformerons l'accès de *type privé* en *type protégé* (`protected`).

Le qualificatif d'accès **protected** donne le droit d'accès aux *membres privés* depuis toute *classe dérivée*.



CERCLE, SPHÈRE ET HÉRITAGE

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI=3.14159;
//définition de la classe cercle
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();
//la donnée membre r est en accès protégé
//pour être accessible depuis la classe dérivée sphère
protected:
    double r;
};
//définition de la classe sphere, dérivée de la classe cercle
//elle réutilise r, et les fonctions membres de la classe cercle
class sphere:public cercle
{
public:
    double vol();
    void affsphere();
};
//constructeur de la classe cercle
double cercle::srayon()
{
    cout<<"Rayon : ";
```

```

    cin>>r;
    return r;
}
//définition de la fonction membre affcercle de la classe cercle
void cercle::affcercle()
{
    cout<<"Le cercle a pour : "<<endl;
    cout<<"Diametre : "<<diametre()<<endl;
    cout<<"Circonference : "<<circonf()<<endl;
    cout<<"Surface : "<<surf()<<endl<<endl;
}
//constructeur de la classe sphere
double sphere::vol()
{
    return 4.0/3.0*PI*pow(r,3);
}
//définition de la fonction membre affspher de la classe sphere
void sphere::affspher()
{
    cout<<"La sphere a pour : "<<endl;
    cout<<"Diametre : "<<diametre()<<endl;
    cout<<"Circonference : "<<circonf()<<endl;
    cout<<"Volume : "<<vol()<<endl;
}
int main(){
    //instanciation des objets c1 et c2 de la classe cercle
    cercle c1, c2;
    cout<<"Par defaut, ";
    //appels des fonctions
    c1.affcercle();
    cout<<"CERCLE -> ";
    c2.srayon();
    c2.affcercle();
    //instanciation d'un objet s1 de la classe sphere
    sphere s1;
    //appels des fonctions
    cout<<"SPHERE -> ";
    s1.srayon();
    s1.affspher();
    return 0;
}

```

Résultat après exécution :

```

Par defaut, Le cercle a pour :
Diametre : 10
Circonference : 31.4159
Surface : 78.5397

```

```
CERCLE -> Rayon : 12
Le cercle a pour :
Diametre : 24
Circonference : 75.3982
Surface : 452.389
```

```
SPHERE -> Rayon : 5
La sphere a pour :
Diametre : 10
Circonference : 31.4159
Volume : 523.598
```

9.3 L'HÉRITAGE MULTIPLE

C'est une technique qui offre la possibilité de créer des *classes dérivées* à partir de plusieurs *classes* de base. On peut dire qu'une *classe* hérite 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`.

Pour certains développeurs, *l'héritage multiple* est une possibilité à ne pas utiliser, car son utilisation peut être sujette à des ambiguïtés d'*identificateurs*.

Cet ouvrage ne rentrera pas dans le détail de cette controverse qui dans certains cas me semble justifiée.

Nous nous contenterons de montrer une application simple de cette technique d'*héritage*.

Il faut noter que la librairie standard d'entrées-sorties `iostream` de C++, utilise *l'héritage multiple* pour dériver `iostream` de `istream` et de `ostream`, cela exprime que `iostream` est à la fois un *flot d'entrée* `istream` et un *flot de sortie* `ostream`.



POLYGONE, TRIANGLE ET RECTANGLE

```
#include <iostream>
using namespace std;
//définition de la classe polygone
class polygone{
    //données membres protégées utilisées par les classes dérivées
```

```
protected:  
    double l, h;  
public:  
    void valorise(double a, double b){  
        l=a; h=b;  
    }  
};  
//définition de la classe affiche  
class affiche{  
public:  
    void print(double s){  
        cout<<"Surface : "<<s<<endl;  
    }  
};  
//définition de la classe rectangle qui hérite de la classe  
polygone  
class rectangle:public polygone, public affiche{  
public:  
    double surf(void){  
        return l*h;  
    }  
};  
//définition de la classe triangle qui hérite de la classe  
polygone  
class triangle:public polygone, public affiche{  
public:  
    double surf(void){  
        return l*h/2;  
    }  
};  
int main()  
{  
    //déclarations  
    double largeur, hauteur;  
    //saisie de la largeur ou base  
    cout<<"Largeur/Base : ";  
    cin>>largeur;  
    //saisie de la hauteur  
    cout<<"Hauteur : ";  
    cin>>hauteur;  
    //instanciation d'un objet R de la classe rectangle  
    rectangle R;  
    //instanciation d'un objet T de la classe triangle  
    triangle T;  
    //appel de la fonction membre valorise pour R et T  
    R.valorise(largeur,hauteur);  
    T.valorise(largeur,hauteur);
```

```
//affichage des résultats via l'utilisation de la fonction  
print de la classe affiche  
cout<<"Rectangle - ";  
R.print(R.surf());  
cout<<"Triangle - ";  
T.print(T.surf());  
return 0;  
}
```

Résultat après exécution :

```
Largeur/Base : 25  
Hauteur : 15  
Rectangle - Surface : 375  
Triangle - Surface : 187.5
```

9.4 LE 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 distincts 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 aider à réaliser ce traitement. Elles vont créer 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*.

Le *polymorphisme* décrit ici se nomme plus précisément **polymorphisme d'héritage** (**redéfinition** ou **overriding**). En C++, comme dans d'autres langages typés, le *polymorphisme* peut revêtir de nombreuses formes, comme le *polymorphisme ad-hoc* (déjà cités au chapitre 8, §8.1), le **polymorphisme paramétrique** (**généricité** ou **template**), le **polymorphisme d'inclusion**...et bien d'autres.

Cet ouvrage n'a pas l'ambition de vous emmener dans les méandres du *polymorphisme*, sachez qu'implicitement pour la plupart des développeurs le *polymorphisme* est synonyme de *polymorphisme d'héritage*.



POLYGONE POLYMORPHE

```
#include <iostream>
using namespace std;
//définition de la classe polygone
class polygone{
public :
    void valorise(double a, double b){
        l=a; h=b;
    }
    //définition de la fonction virtuelle surf dans la classe
    //polygone
    virtual double surf(void){
        cout<<"Appel de la fonction surf de la classe de
        base qui renvoie : ";
        return 0;
    }
protected:
    double l, h;
};
//définition de la classe rectangle
class rectangle:public polygone
{
public:
    double surf(void){
        cout<<"Appel de la fonction surf de la classe de
        base qui renvoie : ";
        return l*h;    }
};
//définition de la classe triangle
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;
    }
};
int main(){
    //instanciations des objets R, T, P des classes rectangle,
    triangle et polygone
    rectangle R;
    triangle T;
    polygone P;
    //affectations pointeur-adresse (liens dynamiques)
    polygone *ptP1=&R;
```

```

polygone *ptP2=&T;
polygone *ptP3=&P;
//appel de la fonction membre valorise
ptP1->valorise(5, 3);
ptP2->valorise(2.5, 1.5);
ptP3->valorise(10, 6);
//appel de rectangle::surf(void) et affichage
cout<<ptP1->surf()<<endl;
//appel de triangle::surfvoid() et affichage
cout<<ptP2->surf()<<endl;
//appel de polygone::surf(void) et affichage
cout<<ptP3->surf()<<endl;
return 0;
}

```

Résultat après exécution :

```

Appel de la fonction surf de la classe de base qui renvoie : 15
Appel de la fonction surf de la classe de base qui renvoie :
1.875
Appel de la fonction surf de la classe de base qui renvoie : 0

```

Dans l'exemple précédent les *pointeurs* sont liés dynamiquement à la *fonction surf*.

Les appels sont *polymorphes* car ils fournissent un résultat différent en fonction des *instances* qu'ils manipulent.

La *classe polygone*, qui est la *classe de base* et qui possède une *fonction membre virtuelle surf* est dite *classe polymorphe*.

9.5 LES PATRONS DE FONCTIONS

Un *patron de fonction* est en quelque sorte un modèle avec lequel le compilateur est capable, en fonction des besoins, de générer plusieurs *fonctions génériques* réelles qui différeront par le *type* de données qu'elles auront à manipuler.

Un *patron* condense le code en offrant la possibilité d'écrire une seule fois la définition d'une *fonction*.

Le mot-clé retenu pour les *patrons* est **template** suivi de **class** qui précise le *type*.

Voici la syntaxe à utiliser :

```
template <class type>
```

Le paramètre `type` est le paramètre de `type` qui va remplacer les `types` classiques présents dans la définition de la `fonction`.



ENTIERS, RÉELS ET CARACTÈRES

```
#include <iostream>
using namespace std;
//définition du patron de fonction
template <class T>
//le paramètre de type T est générique pour *tableau
void affiche(T *tableau, int n) {
    for(int i=0;i<n;i++){
        cout<<"Element N° "<<i<<" : "<<tableau[i]<<endl;
    }
    cout<<"-----"<<endl;
}
int main()
{
    //déclaration et affectations pour un tableau d'entiers, de
    réels et de caractères
    int entier[6] = {5, 10, 20, 40, 80, 160};
    double decimal[3] = {1.1, 2.2, 4.4};
    char *chaine[] = {"Bjarne", "STROUSTRUP"};
    //a chaque appel de la fonction affiche le compilateur
    génère une fonction
    //qui tient compte du type comme précisé dans le patron
    //appel de affiche - la fonction générée par le compilateur
    tient compte du type int
    affiche(entier, 6);
    //appel de affiche - la fonction générée par le compilateur
    tient compte du type double
    affiche(decimal, 3);
    //appel de affiche - la fonction générée par le compilateur
    tient compte du type char
    affiche(chaine, 2);
    return 0;
}
```

Résultat après exécution :

```
Element N°0 : 5
Element N°1 : 10
Element N°2 : 20
Element N°3 : 40
Element N°4 : 80
Element N°5 : 160
-----
```

```
Element N°0 : 1.1
Element N°1 : 2.2
Element N°2 : 4.4
-----
Element N°0 : Bjarne
Element N°1 : STROUSTRUP
-----
```

9.6 LES PATRONS DE CLASSES

Les *patrons* sont aussi applicables sur les *classes*, on parle alors de *patrons de classes*.

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*.

De cette façon, nous obtenons des *classes génériques* qui vont pouvoir, comme les *patrons de fonctions*, traiter des *types* de données différents.



PATRON POUR RECTANGLE

```
#include <iostream>
using namespace std;
//définition du patron de la classe rectangle avec son paramètre
générique T
template <class T>class rectangle{
private:
    T L,l;
public:
    rectangle();
    rectangle(T, T);
    void affiche(void);
};
//définition du patron pour la fonction rectangle
template <class T>rectangle<T>::rectangle(){
    L=0; l=0;
}
//définition du patron pour la fonction rectangle
template <class T>rectangle<T>::rectangle(T Lg,T lg){
    L=Lg; l=lg;
}
//définition du patron pour la fonction affiche
template <class T>void rectangle<T>::affiche(){
    cout<<"Surface : "<<L*l<<endl;
}
int main()
```

```

{
    //déclaration et initialisation des variables suivant 3
types
    float L1=5, l1=3;
    double L2=25.5, l2=7.25;
    int L3=15, l3=7.5;
    //instanciations en fonction des types
    rectangle<float>s1(L1,l1);
    rectangle<double>s2(L2,l2);
    rectangle<int>s3(L3,l3);
    rectangle<int>s4;
    //appels de la fonction affiche
    s1.affiche();
    s2.affiche();
    s3.affiche();
    s4.affiche();
    return 0;
}

```

Résultat après exécution :

```

Surface : 15
Surface : 184.875
Surface : 105
Surface : 0

```

9.7 RÉCAPITULATIF

Par le biais de l'*agrégation* une *classe* peut être liée à une ou plusieurs autres.

Une *classe* peut hériter d'une *super-classe*, elle devient ainsi une *classe dérivée* qui partage avec la première ses *membres publics*. L'accès aux *membres privés* peut se faire en transformant le *type privé* (**private**) en *type protégé* (**protected**).

Par l'intermédiaire de la technique de l'*héritage multiple*, une *classe* peut hériter de plusieurs autres.

Il est possible d'appeler une *fonction membre* d'un *objet* sans se soucier de son *type*, en utilisant le mécanisme du *polymorphisme d'héritage*.

On peut créer des *modèles de fonctions* ou de *classes* qui traiteront des *types* de données différents, ce sont les *patrons de fonctions* et les *patrons de classe*.

EXERCICES

9-1 Point de couleur

Écrivez un programme qui montre l'*héritage multiple* d'une *classe* « PointDeCouleur » avec 2 *classes* « Point » et « Couleur ».

La *classe* « Point » précisera sous forme d'un couple (x, y) les coordonnées d'un point, par exemple : (5, -2).

La *classe* « Couleur » précisera sous la forme d'une *chaîne de caractère*, une couleur, par exemple : magenta.

Vous n'oublierez pas de spécifier pour chacune des *classes* un *constructeur* et un *destructeur*.

Un point i, *instance* de la *classe* « PointDeCouleur » pourra être utilisé par votre programme et une *fonction* « affichage » tracera l'utilisation de chacune des *classes*.

9-2 Tri

Écrivez un programme capable de trier une liste de nombres entiers (45, 25, 50, 15, 55, 30, 35, 20, 40, 10) ou de prénoms (Pierre, Marie, Paul, Aline, Jacques, François, Jean, Alain, Michelle, Vincent).

La technique de tri retenue sera celle du *tri bulles* ou *tri par propagation*.

Voici un descriptif de l'algorithme à appliquer :

Le programme parcourt la liste, et compare chaque couple d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre, ils sont permutés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération.

Vous pourrez utiliser la *fonction swap* du fichier d'en-tête <iostream> pour permuter les données et vous devrez créer des *patrons de fonctions* pour le tri et l'affichage des données.

SOLUTIONS

9-1 Point de couleur

```
#include <iostream>
using namespace std;
//définition de la classe Point
class Point{
```

```
public:  
    int x,y;  
    //constructeur  
    Point(int X, int Y){  
        x=X ;  
        y=Y ;  
        cout<<"Constructeur de Point"<<endl;  
    }  
    //destructeur  
    ~Point() {  
        cout<<"Destructeur de Point"<<endl;  
    }  
    //fonction affichage  
    void affichage() {  
        cout<<"Point ("<<x<<, " <<y<<")";  
    }  
};  
//définition de la classe Couleur  
class Couleur{  
public :  
    char* coul;  
    //constructeur  
    Couleur(char* z) {  
        coul=z;  
        cout<<"Construction de Couleur"<<endl;  
    }  
    //destructeur  
    ~Couleur(){  
        cout<<"Destruction de Couleur"<<endl;  
    }  
    //fonction affichage  
    void affichage() {  
        cout<<" de couleur : "<<coul<<endl ;  
    }  
};  
//définition de la classe PointDeCouleur qui hérite des classes  
Point et Couleur  
class PointDeCouleur:public Point, public Couleur{  
public:  
    //constructeur  
    PointDeCouleur(int a, int b, char* c):Point(a,b),Couleur(c){  
        cout<<"Construction de PointDeCouleur"<<endl;  
    }  
    //destructeur  
    ~PointDeCouleur(){  
        cout<<"Destruction de PointDeCouleur"<<endl;  
    }  
}
```

```

//fonction affichage
void affichage(){
    Point::affichage();
    Couleur::affichage();
}
};

int main(){
    //création d'un objet i de la classe PointDeCouleur
    PointDeCouleur i(5, 2, "rouge");
    //appel de affichage de la classe PointDeCouleur
    i.affichage();
}

```

Résultat après exécution :

```

Constructeur de Point
Construction de Couleur
Construction de PointDeCouleur
Point (5, 2) de couleur : rouge
Destruction de PointDeCouleur
Destruction de Couleur
Destructeur de Point

```

9-2 Tri

```

#include <iostream>
using namespace std;
//patron de fonction
template<class T>
//définition de la fonction tri
//T sera remplacé par int ou string lors des appels
void tri(T* tab, int n){
    //les 2 boucles pour le tri
    for(int i=1; i<n; i++)
        for(int j=0; j<n-i; j++)
            if (tab[j]>tab[j+1])
                //permutation des éléments
                swap(tab[j], tab[j+1]);
}
//patron de fonction
template<class T>
//définition de la fonction affichage
void affichage(T* tab, int n){
    for(int i=0; i<n; i++)
        cout<<" "<<tab[i];
    cout<<endl;
}

```

```
}

int main(){
    //déclaration et affectation dun tableau d'entiers
    int nombre[10]={45, 25, 50, 15, 55, 30, 35, 20, 40, 10};
    cout<<"Liste non triée : ";
    //dans les appels qui suivent, T prend en compte le type int
    //appel de la fonction affichage
    affichage(nombre, 10);
    //appel de la fonction tri
    tri(nombre, 10);
    //appel de la fonction affichage
    cout<<"Liste triée : ";
    affichage(nombre, 10);
    cout<<"\n";
    //déclaration et affectation d'un tableau de chaînes
    string prenom[10]={"Pierre", "Marie", "Paul", "Aline",
"Jacques", "François", "Jean", "Alain", "Michelle", "Vincent"};
    cout<<"Liste non triée : ";
    //dans les appels qui suivent, T prend en compte le type
    string
        //appel de la fonction affichage
    affichage(prenom, 10);
    //appel de la fonction tri
    tri(prenom, 10);
    cout<<"Liste triée : ";
    //appel de la fonction affichage
    affichage(prenom, 10);
    return 0;
}
```

Résultat après exécution :

```
Liste non triée : 45 25 50 15 55 30 35 20 40 10
Liste triée : 10 15 20 25 30 35 40 45 50 55
```

```
Liste non triée : Pierre Marie Paul Aline Jacques François Jean
```

```
Alain Michelle Vincent
```

```
Liste triée : Alain Aline François Jacques Jean Marie Michelle
Paul Pierre Vincent
```

Compiler en mode console

Vous trouverez dans les lignes qui suivent, une petite introduction à l'utilisation des compilateurs **Xcode** (Mac OS-X), **Visual C++ 2008 Express**, **Bloodshed Dev C++** (Microsoft Windows) et **Code::blocks** (Mac OS-X, Microsoft Windows, Linux).

Elle sera suivie d'une petite approche des erreurs les plus courantes rencontrées au sein d'un code source C++.

Remarques et conseils

Cette présentation des outils de compilation C++ au sein des environnements de développement intégrés (EDI) reste très sommaire.

Il faudrait plusieurs centaines de pages pour décrire et appréhender l'ensemble des fonctionnalités très étendues de chacun de ces logiciels.

Je me bornerais seulement à préciser, de manière simple, l'édition et la compilation d'une application en mode console.

Je vous conseille de consulter l'aide hors ligne ou en ligne disponible pour chacun de ces produits.

Vous trouverez dans la bibliographie et la page des liens internet de ce livre des références à plusieurs ouvrages et documents spécialisés.

OÙ TROUVEZ LES OUTILS

Les environnements de développement intégré que j'ai sélectionné sont tous gratuits et téléchargeables sur internet.

Mon choix s'est arrêté sur ceux-ci suite à de nombreux échanges avec mes étudiants et à ma propre expérience.

Désignation	OS	Adresse de téléchargement
Xcode (Apple)	Mac OS-X	Fourni en standard avec chaque Mac ou téléchargeable sur le site d'Apple : http://developer.apple.com/technology/xcode.html
Visual C++ 2008 Express (Microsoft)	Microsoft Windows	Téléchargeable sur le site de Microsoft : http://msdn.microsoft.com/fr-fr/express/aa975050.aspx
Dev C++ (Bloodshed)	Microsoft Windows	Téléchargeable sur le site de Bloodshed : http://www.bloodshed.net/
Code::Blocks	Microsoft Windows, Mac OS-X et Linux	Téléchargeable sur le site de Codeblocks http://www.codeblocks.org/

Il existe de nombreux autres outils, gratuits ou payants, qui sont assurément aussi performants, il n'est malheureusement pas possible de tous les citer ici.

XCODE (MAC OS-X)

Après avoir lancé le logiciel, allez dans le menu FILE, et choisissez NEW PROJECT.

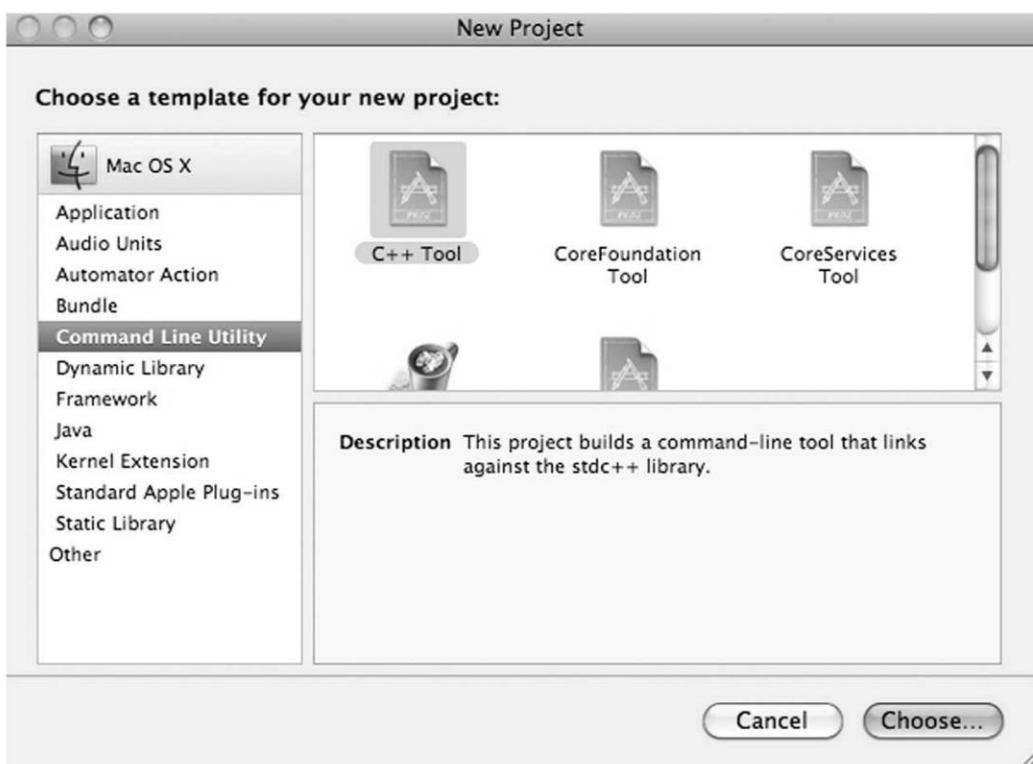


Figure 1-1 Xcode :Nouveau projet.

Dans la fenêtre qui s'ouvre, choisissez « Command Line Utility », puis cliquez sur l'icône « C++ Tool » dans la partie droite.
Cliquez ensuite sur le bouton CHOOSE en bas, à droite.

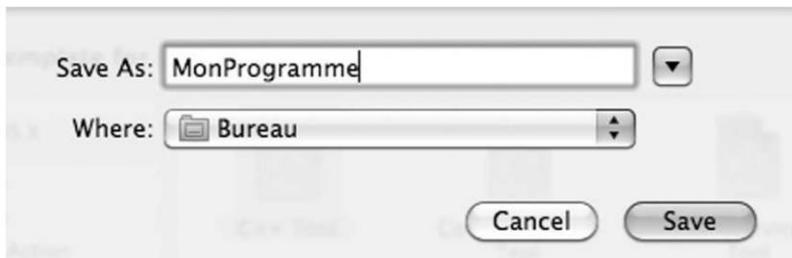


Figure 1-2 Xcode :Saisie du nom du projet.

Entrez le nom de votre futur projet dans le champ « Save As », puis choisissez la destination d'enregistrement de votre travail via le menu déroulant « Where ».

Cliquez sur le bouton SAVE.

Une fenêtre au nom de votre projet s'ouvre.

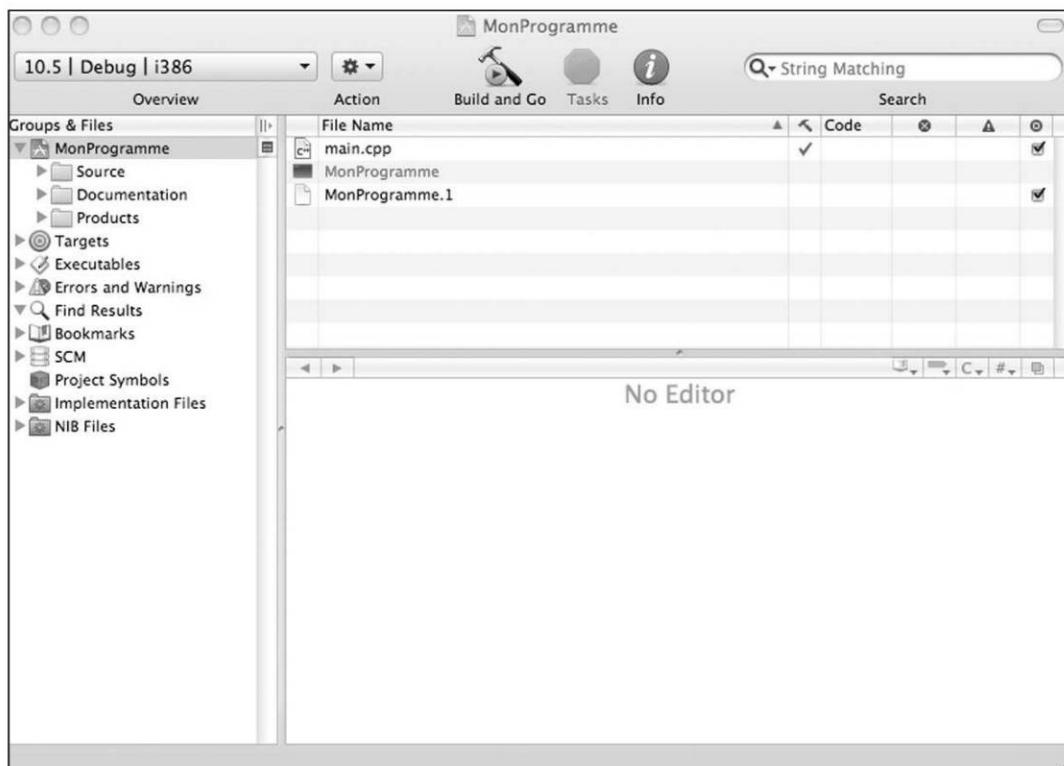


Figure 1-3 Xcode :Environnement de mon projet.

Cliquez sur « main.cpp » dans la fenêtre supérieure droite.

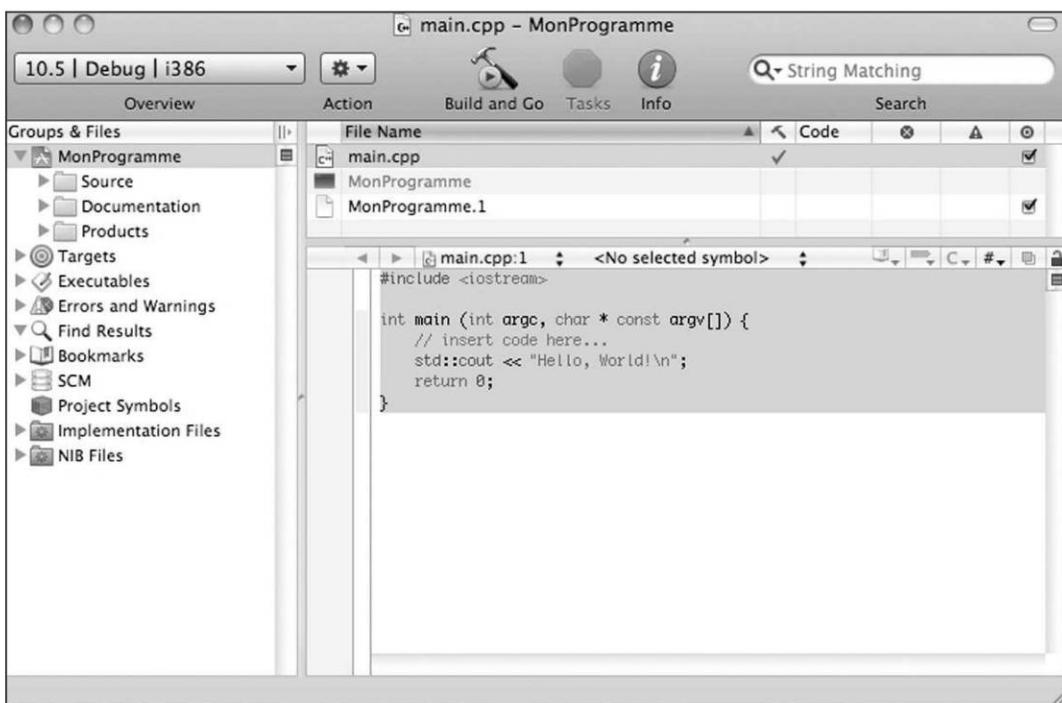


Figure 1-4 Xcode : La zone de saisie du code avec le programme « Hello,World ! » de départ.

Dans la partie inférieure, vous pouvez voir quelques lignes de code C++ (un programme de type « Hello, World ! »).
C'est à la place de celui-ci, dans cette zone, que vous allez saisir vos lignes de code, en remplaçant celles qui existent.

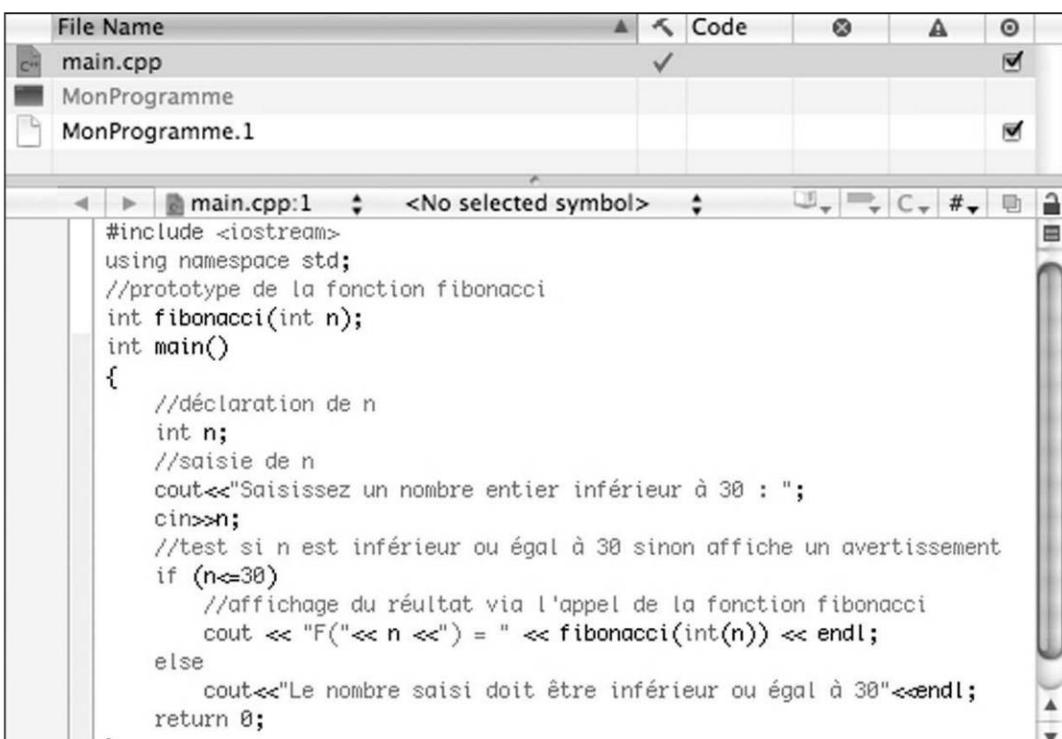


Figure 1-5 Xcode : La zone de saisie du code.

Une fois votre programme saisi, vous allez pouvoir lancer la compilation en cliquant sur l'icône « Build and Go », situé dans la barre supérieure. Le programme vous demande de sauvegarder votre projet, cliquez sur le bouton SAVE ALL.

Deux cas peuvent se présenter :

- Le programme ne contient aucune erreur, la compilation s'exécute et le programme se lance avec succès.
- Le programme contient des erreurs, la compilation est impossible, ne se termine pas correctement et s'arrête. L'éditeur affiche les problèmes rencontrés.

Compilation réussie

Si aucune erreur n'est détectée, le programme se lance.

Vous pouvez vérifier en regardant la barre d'état inférieure de l'éditeur qui doit afficher un message de lancement, à gauche, de type « Xxx launched » et le mot « Succeeded » à droite.

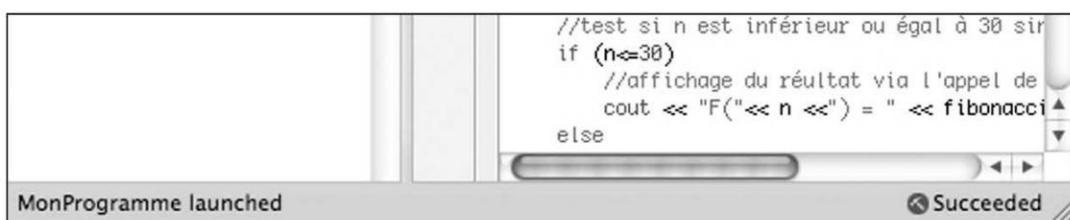


Figure 1-6 Xcode : Programme lancé avec succès.

Dans ce cas, allez dans le menu RUN et choisissez CONSOLE.

Vous devriez trouver le résultat de l'exécution de votre programme ou bien, le curseur clignotant, en attente d'une entrée.



Figure 1-7 Xcode : La fenêtre de la console, avec le curseur en attente d'une entrée.

Compilation impossible

Si un ou plusieurs problèmes sont présents, des messages d'avertissements ou d'erreurs apparaissent, au-dessous de chacune des lignes mises en cause, dans la zone de saisie.

The screenshot shows the Xcode interface with the file `main.cpp` open. The code contains the following:

```

int main()
{
    //déclaration de n
    int n;
    float nb;

    //warning: unused variable 'nb'
    cout<<"Saisissez un nombre entier inférieur à 30 : ";
    cin>>n;
    //test si n est inférieur ou égal à 30 sinon affiche un avertissement
    if (n<=30)
        //affichage du résultat via l'appel de la fonction fibonacci
        cout << "F("<< n <<") = " << fibonacci(int(m)) << endl;
    //error: 'm' was not declared in this scope
    else
        cout<<"Le nombre saisi doit être inférieur ou égal à 30"<<endl;
    return 0
}
//error: expected ';' before '}' token

```

The status bar at the bottom indicates "Build failed (2 errors, 1 warning)".

Figure 1-8 Xcode : La zone de saisie mentionne un message d'avertissement et deux erreurs dans le code C++.

La barre d'état inférieure indique quant à elle, le nombre d'erreurs et d'avertissements trouvés lors de la construction de l'exécutable ainsi qu'un message « Failed » d'arrêt de compilation.



Figure 1-9 Xcode : Indicateurs d'avertissements, d'erreurs et de compilation manqués dans la barre d'état de l'éditeur.

Corrigez les erreurs et relancez la compilation.

MICROSOFT VISUAL C++ 2008 EXPRESS

Lancez le logiciel et allez dans le menu FICHIER, choisissez NOUVEAU, puis PROJET.

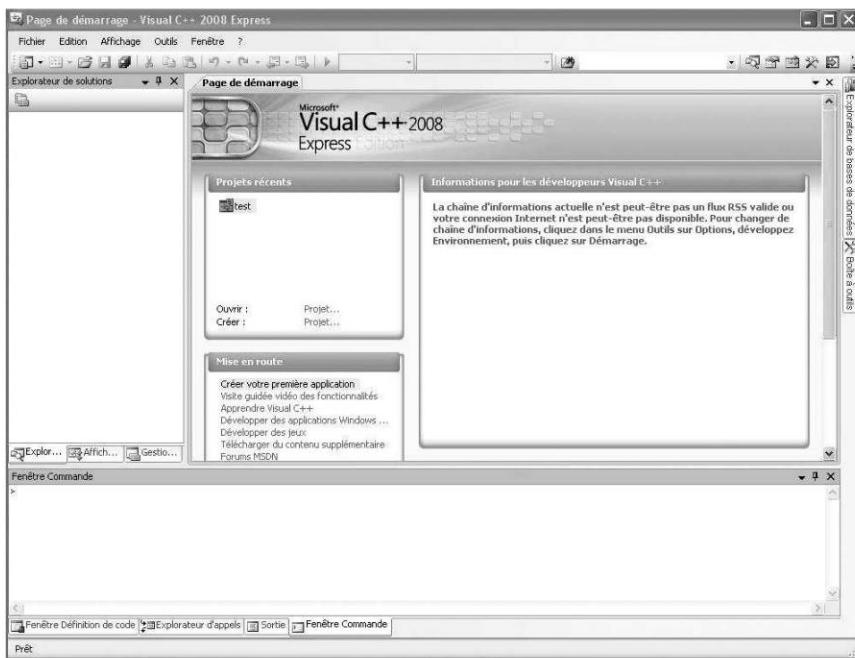


Figure 1-10 Visual C++ :L'environnement de développement intégré Microsoft Visual C++ 2008 Express.

Dans la fenêtre qui vient de s'ouvrir, sélectionnez « Application console Win32 », saisissez un nom pour votre projet dans le champ « Nom ».

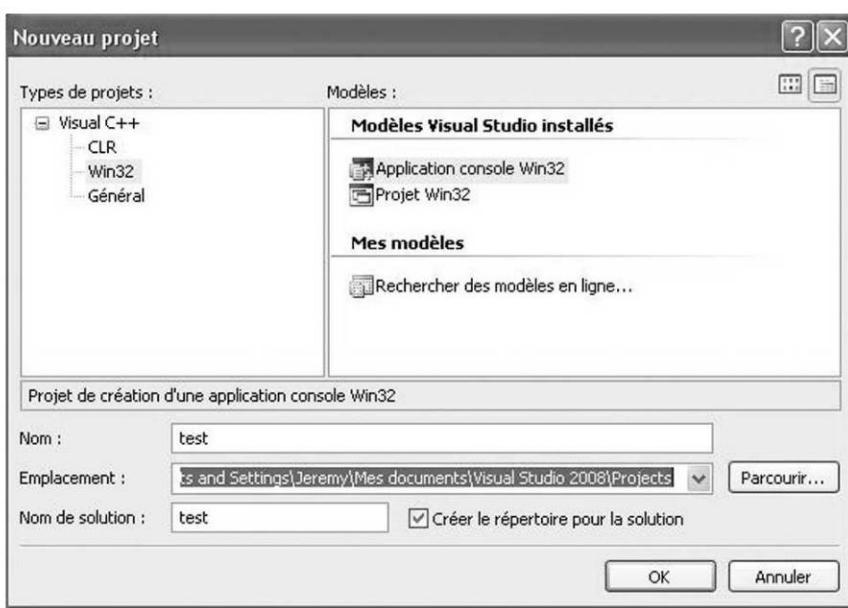


Figure 1-11 Visual C++ :La fenêtre de dialogue pour la création d'un nouveau projet.

En cliquant sur le bouton PARCOURIR vous pourrez choisir une destination pour l'enregistrement de votre travail.

Un dossier, de même nom que votre projet, sera automatiquement créé et contiendra tous les fichiers (si la case « Créer le répertoire pour la solution » est cochée).

Cliquez sur le bouton OK pour valider vos choix.

Une nouvelle fenêtre s'ouvre, c'est l'assistant de création d'application.

Cliquez sur le bouton SUIVANT.



Figure 1-12 Visual C++ :L'assistant de création d'une application Win32.

Laissez les options par défaut et cliquez puis sur le bouton TERMINER.

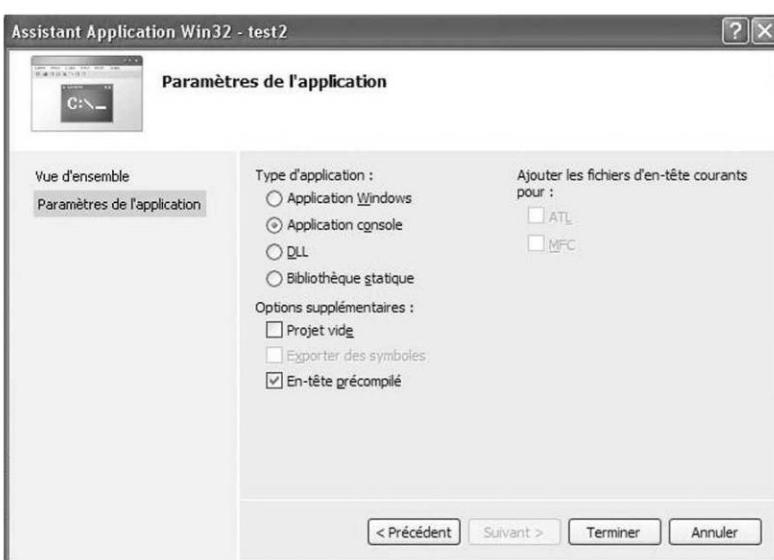


Figure 1-13 Visual C++ :La gestion des paramètres de l'application via l'assistant.

L'éditeur affiche une fenêtre, dont l'onglet porte le nom de votre projet. Elle contient déjà quelques lignes de codes qui forment un programme qui ne fait rien (mais qui peut être compilé – en fait, il ouvre la fenêtre console et la referme aussitôt).

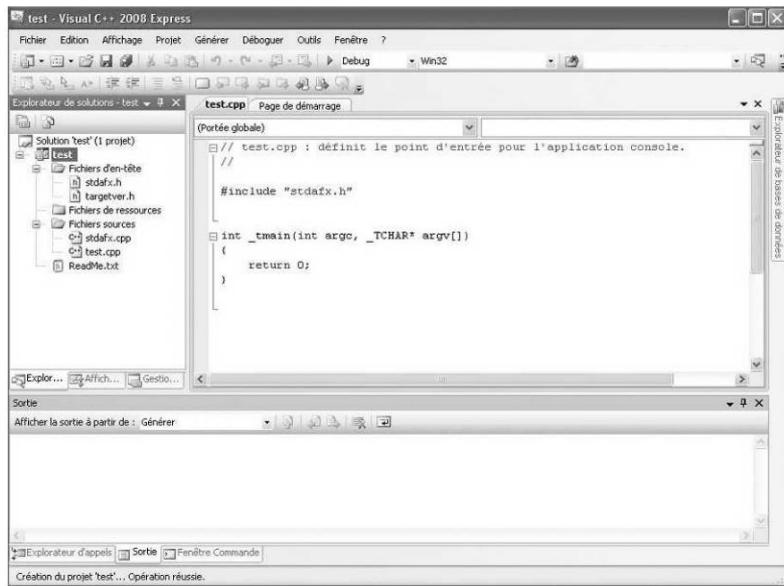


Figure 1-14 Visual C++ :L'éditeur et la fenêtre de saisie du programme.

Vous pouvez supprimer le code présent tout en gardant l'en-tête "#include stdafx.h", obligatoire sous Microsoft Visual C++ Express 2008. Saisissez ensuite vos lignes de programme.

 A screenshot of the Microsoft Visual Studio 2008 Express code editor. The window title is "test.cpp* - Visual C++ 2008 Express". The code editor shows the following C++ code:


```
// test.cpp : définit le point d'entrée pour l'application console.
//
#include "stdafx.h"

#include <iostream>
#include <cmath>

using namespace std;

const double PI=3.1415;

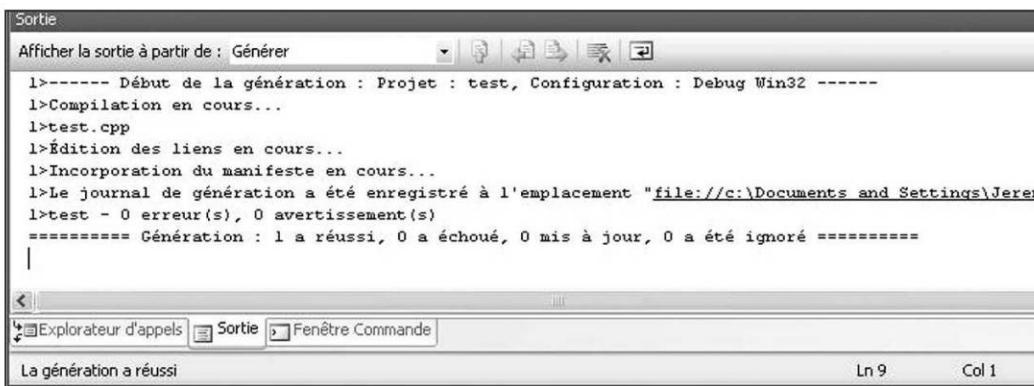
//déclaration de la classe cone
class cone {
private:
    //déclaration des données membres ou attributs privés
    double vol;
public:
    //déclaration des fonctions membres ou méthodes publiques
    double volume(double, double);
```

Figure 1-15 Visual C++ :Les en-têtes et le code du programme dans la fenêtre d'édition.

Pour lancer la compilation, allez dans le menu GENERER, puis choisissez GENERER LA SOLUTION ou appuyer sur la touche de fonction F7.

Compilation réussie

Le résultat s'affiche dans la fenêtre inférieure de l'éditeur qui indique, entre autres, le nombre de compilations réussies ou échouées.



```
Sortie
Afficher la sortie à partir de : Générer
1>----- Début de la génération : Projet : test, Configuration : Debug Win32 -----
1>Compilation en cours...
1>test.cpp
1>Édition des liens en cours...
1>Incorporation du manifeste en cours...
1>Le journal de génération a été enregistré à l'emplacement "file:///c:/Documents and Settings/Jeremy/test/génération.log"
1>test - 0 erreur(s), 0 avertissement(s)
===== Génération : 1 a réussi, 0 a échoué, 0 mis à jour, 0 a été ignoré =====
|
```

Explorateur d'appels Sortie Fenêtre Commande

La génération a réussi

Ln 9 Col 1

Figure 1-16 Visual C++ : La fenêtre indiquant la réussite de la compilation.

Pour lancer votre programme, allez dans le répertoire où vous avez enregistré votre solution et ouvrez le sous-répertoire « debug », votre programme devrait s'y trouver sous la forme d'un fichier exécutable (.exe).

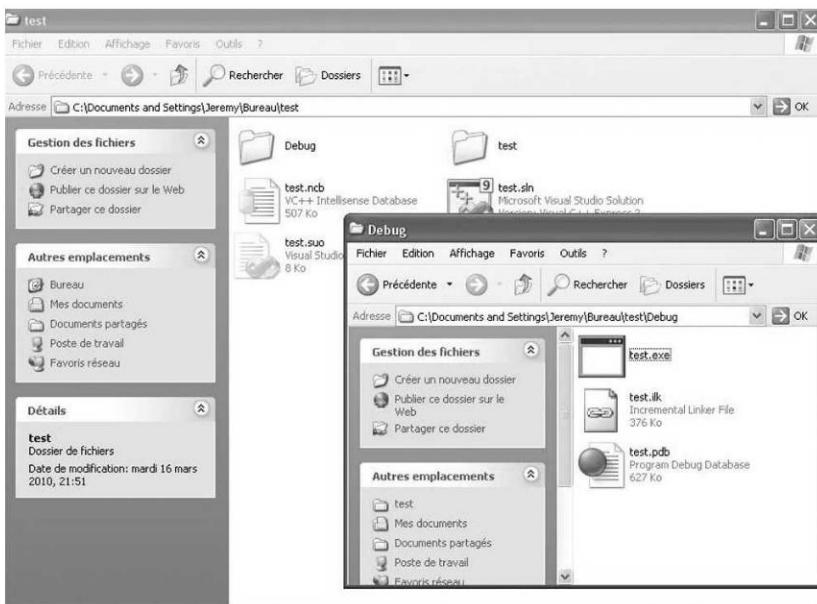


Figure 1-17 Visual C++ : Le répertoire de la solution et son sous-répertoire « Debug » qui contient le programme exécutable.

Compilation impossible

Dans le cas où la compilation a échoué, la fenêtre placée en zone inférieure affiche les erreurs, leurs types et des commentaires associés.

Les codes erreurs sont précédés du n° de ligne entre parenthèses. Vous pouvez retrouver celui-ci dans la barre d'état située en bas de la fenêtre de l'environnement de développement.

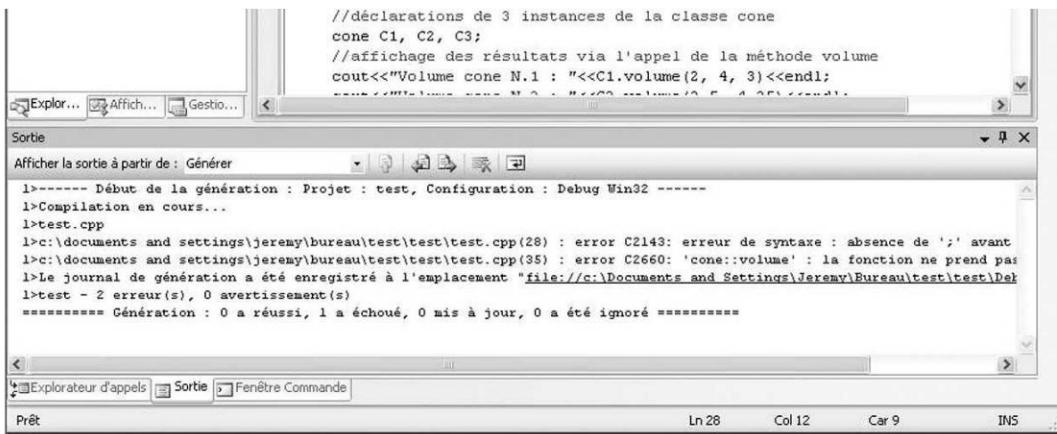


Figure 1-18 Visual C++ : La fenêtre où sont affichés les codes d'erreurs et la barre d'état inférieure qui contient le n° de ligne où est placé le curseur.

Après avoir corrigé les erreurs vous pouvez relancer la compilation.

BLOODSHED DEV C++

Après avoir lancé le logiciel allez dans le file menu FICHIER, choisissez NOUVEAU, puis PROJET.

Cliquez sur l'icône « Console Application », saisissez un nom de projet dans le champ « Nom », vérifiez que l'option « Projet C++ » est bien cochée et validez en cliquant sur le bouton OK.

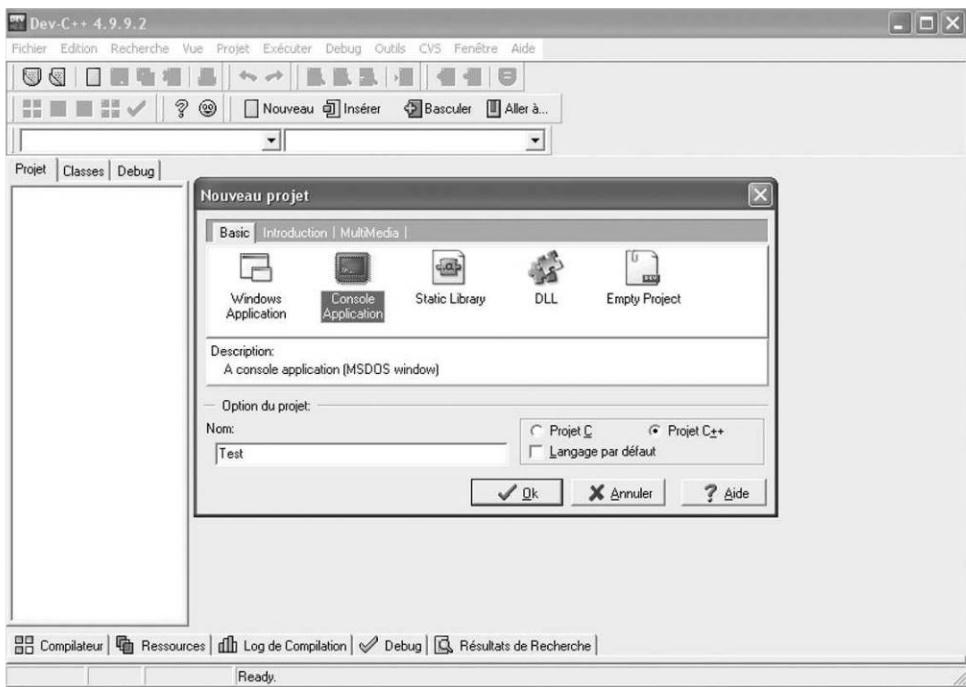


Figure 1-19 Dev C++ :Création d'un nouveau projet.

Une nouvelle fenêtre s'ouvre pour que vous puissiez préciser l'emplacement où votre projet sera enregistré.

Les projets Dev C++ ont comme extension « .dev ».

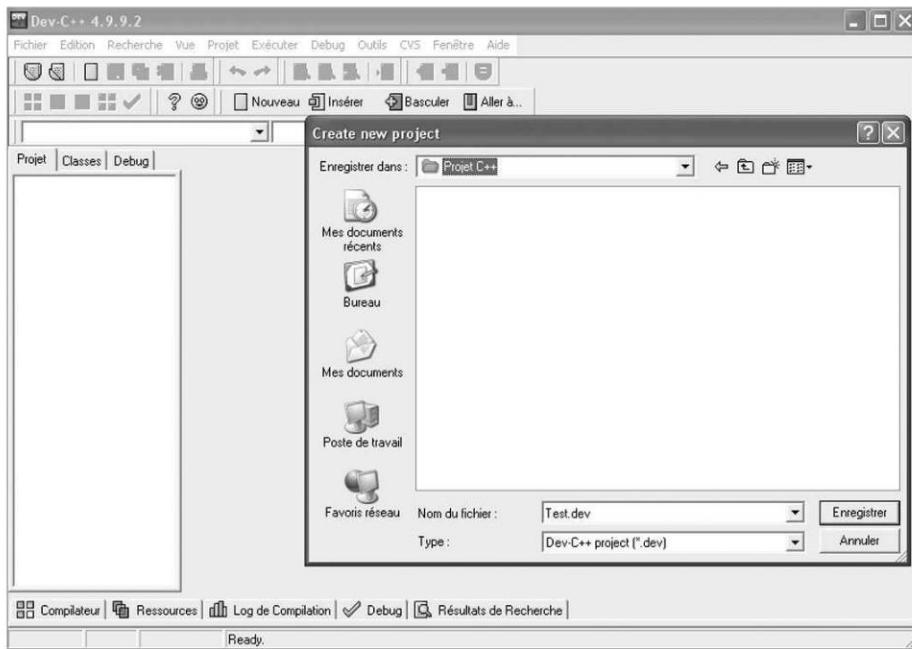


Figure 1-20 Dev C++ :La fenêtre de choix du dossier d'enregistrement du projet.

Une fenêtre se met en place dans l'éditeur, elle porte un onglet nommé « main.cpp ». C'est ici que vous allez devoir saisir le code de votre programme. Par défaut il contient déjà du code sous la forme d'un programme qui ne fait rien (en fait il affiche simplement la fenêtre console en mode pause).

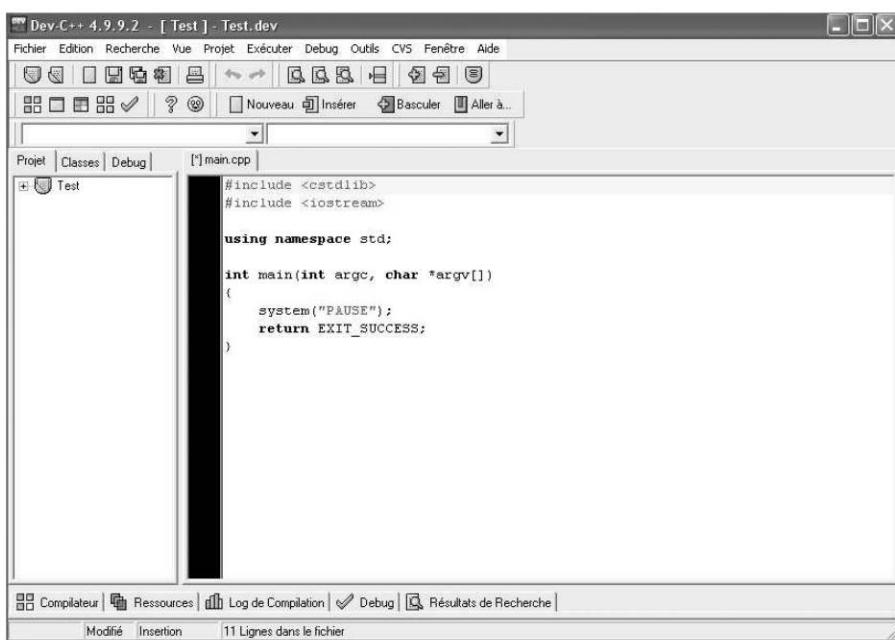
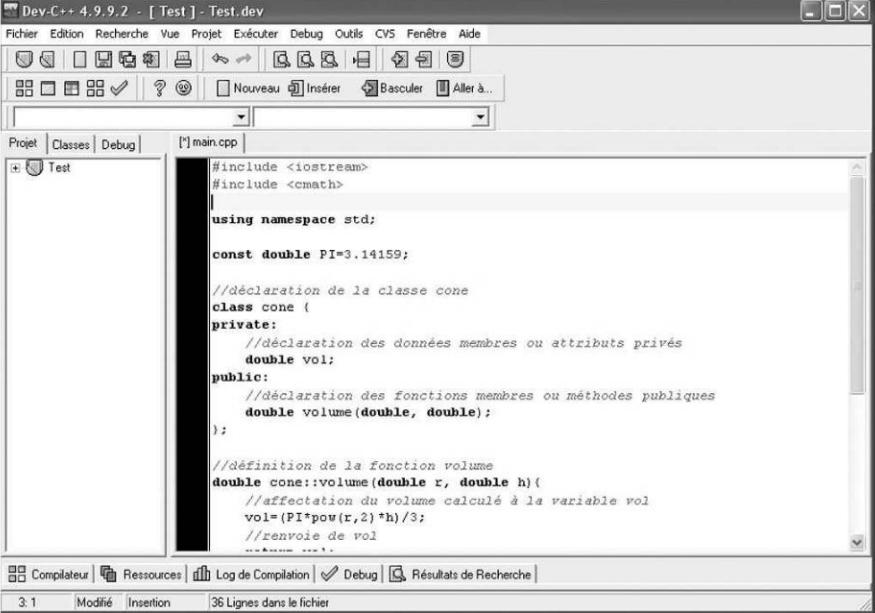


Figure 1-21 Dev C++ :La fenêtre « main.cpp » de l'éditeur avec son code par défaut.

Effacez le code présent, et saisissez votre programme.

Pour compiler et exécuter votre programme, allez dans le menu EXECUTER et choisissez COMPILER & EXECUTER ou appuyez sur la touche de fonction F9 de votre clavier.



```
#include <iostream>
#include <cmath>

using namespace std;

const double PI=3.14159;

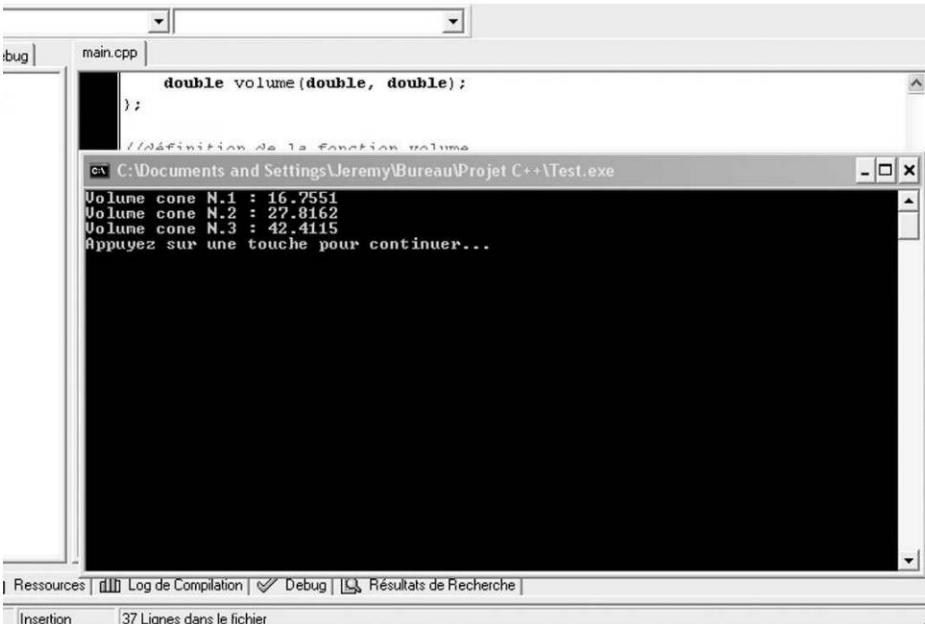
//déclaration de la classe cone
class cone {
private:
    //déclaration des données membres ou attributs privés
    double vol;
public:
    //déclaration des fonctions membres ou méthodes publiques
    double volume(double, double);
};

//définition de la fonction volume
double cone::volume(double r, double h){
    //affectation du volume calculé à la variable vol
    vol=(PI*pow(r,2)*h)/3;
    //renvoie de vol
    return vol;
}
```

Figure 1-22 Dev C++ :La fenêtre « main.cpp » contenant le code de votre programme.

Compilation réussie

L'édition de liens, la compilation et l'exécution devraient s'exécuter et la fenêtre console vous montrer le résultat.



```
double volume(double, double);
};

//Définition de la fonction volume
ex C:\Documents and Settings\Jeremy\Bureau\Projet C++\Test.exe
Volume cone N.1 : 16.7551
Volume cone N.2 : 27.8162
Volume cone N.3 : 42.4115
Appuyez sur une touche pour continuer...
```

Figure 1-23 Dev C++ :La fenêtre console après compilation.

Compilation impossible

Dans le cas où vous auriez commis une erreur, la compilation s'arrête et le debugger est lancé.

Les erreurs sont signalées par un surlignage des lignes dans la fenêtre d'édition. La fenêtre de l'onglet du compilateur s'ouvre en bas de l'éditeur et précise pour chaque ligne les messages d'erreurs.

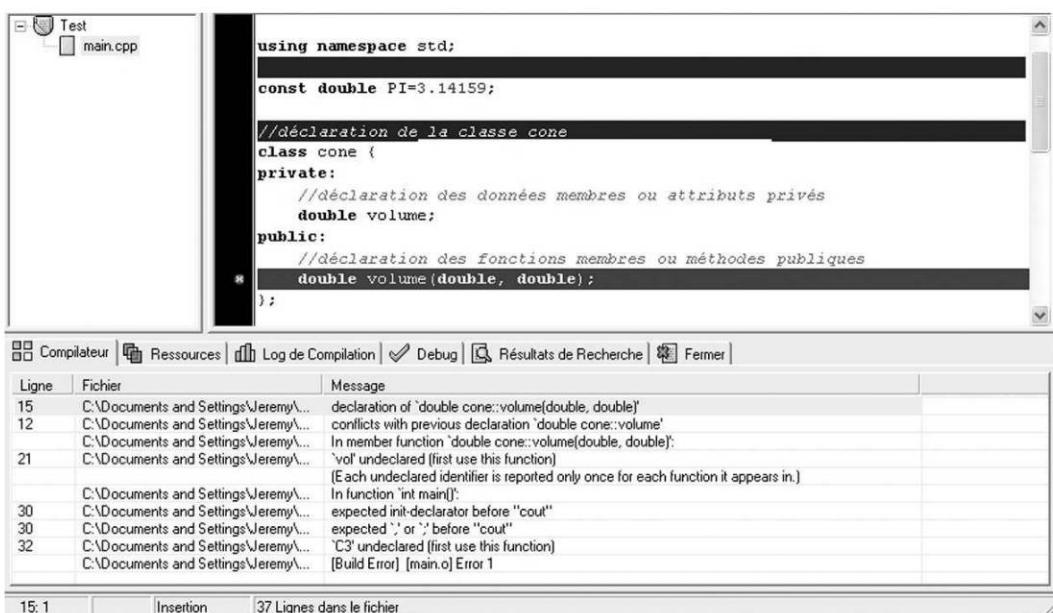


Figure 1-24 Dev C++ :La fenêtre d'édition avec les lignes, contenant des erreurs, surlignées et l'onglet « Compilateur » montrant les messages d'erreurs.

Les numéros de ligne, par défaut invisibles, peuvent être affichés via le menu OUTILS, rubrique OPTIONS DE L'EDITEUR, onglet AFFICHAGE. Il faut cocher, dans la zone « Gouttière », la case « Numéro de ligne ».

Après avoir corrigé les erreurs vous pouvez relancer la compilation.

CODE::BLOCKS

Lancez le logiciel, allez dans le menu FILE et choisissez NEW, puis PROJECT. Vous pouvez aussi cliquer sur l'icône « Create new project » dans la fenêtre principale

Nota : Au moment où j'écris ces lignes, la traduction la version française de CODE::BLOCKS n'est pas encore disponible ou tout au moins n'est pas complète.

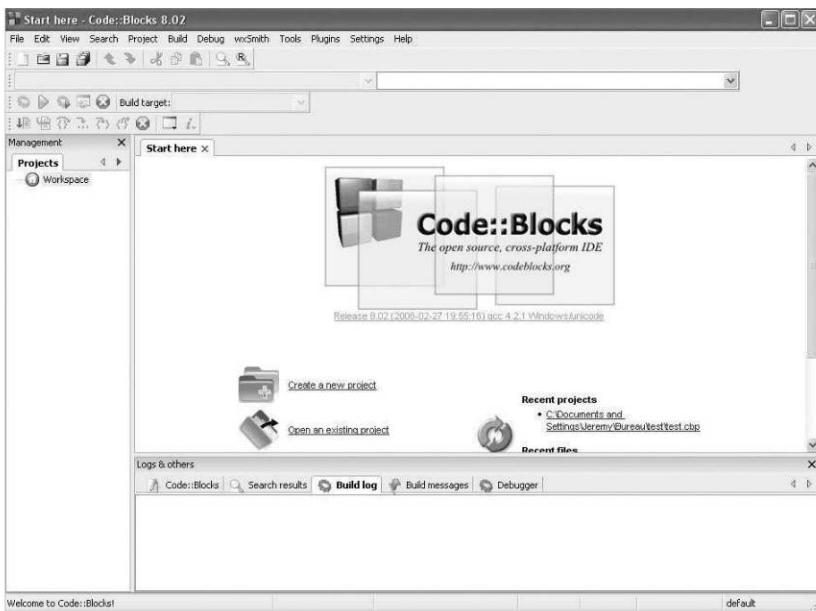


Figure 1-25 Code::Blocks :L'environnement de travail.

Dans la nouvelle fenêtre qui s'ouvre (son contenu peut varier en fonction de l'OS sous lequel vous travaillez : Linux, OS-X ou Microsoft Windows), sélectionnez l'icône CONSOLE APPLICATION, puis cliquez sur le bouton GO.

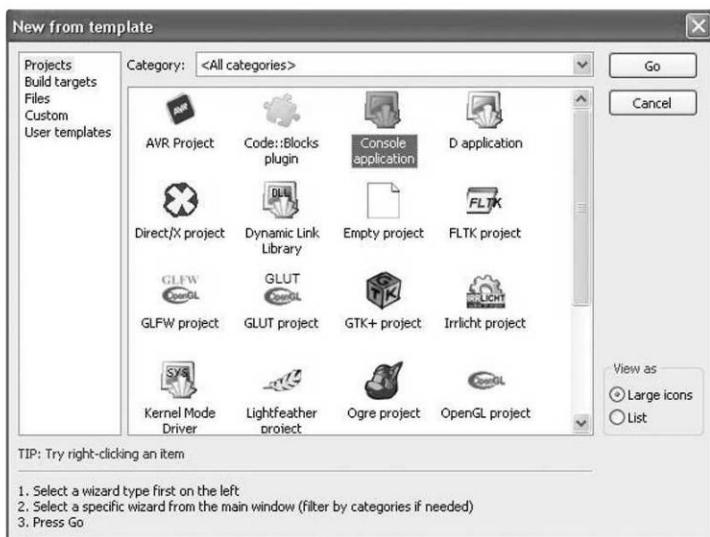


Figure 1-26 Code::Blocks :La fenêtre de choix du type d'application.

Un assistant doit démarrer, cliquez sur le bouton NEXT et choisissez C++, si ce n'est pas déjà sélectionné par défaut.

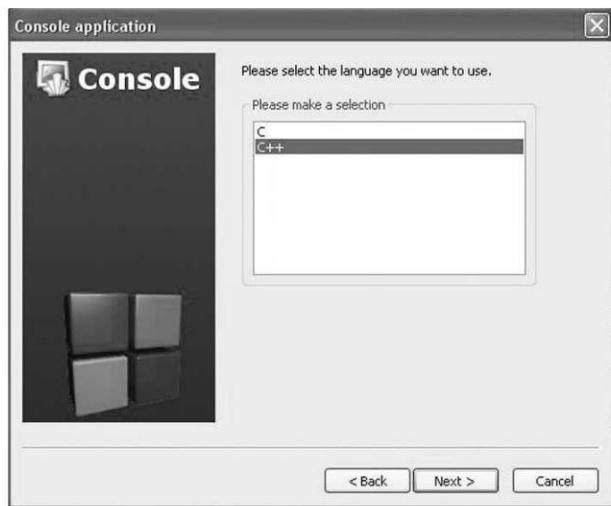


Figure 1-27 Code::Blocks :L'assistant et la fenêtre de dialogue de choix du langage :C ou C++.

Cliquez encore sur le bouton NEXT. Dans la zone « Project title », saisissez le nom de votre projet. Dans la zone « Folder to create project in : », vous pouvez choisir le dossier de destination de votre projet en cliquant sur le bouton « ... ».

Vous pourrez remarquer, dans la zone « Project filename », que votre projet portera le nom que vous avez défini suivi de l'extension « .cbp ».

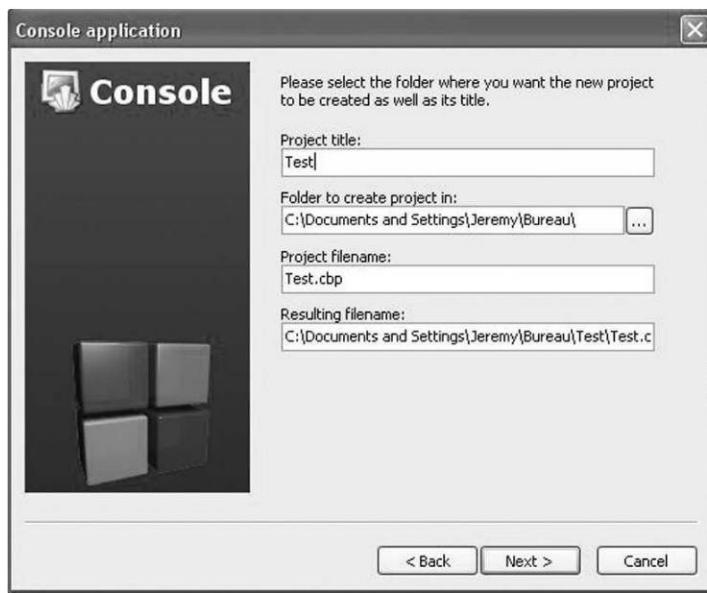


Figure 1-28 Code::Blocks :L'assistant et la fenêtre de dialogue pour la définition du nom et du dossier d'enregistrement du projet.

Cliquez une fois de plus sur le bouton NEXT. Dans la zone de liste déroulante « Compiler », sélectionnez éventuellement le compilateur que vous allez utiliser (dans la plupart des cas c'est GNU CC Compiler ou GCC).

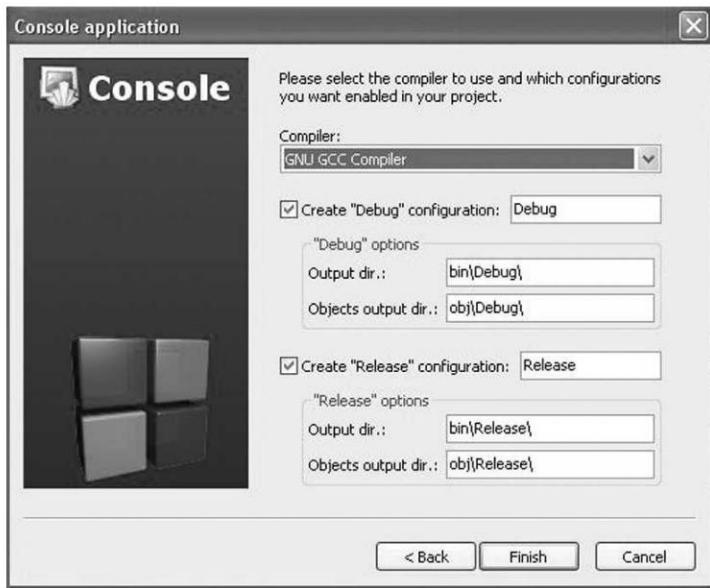


Figure 1-29 Code::Blocks :L'assistant et la fenêtre de dialogue de choix du compilateur.

Enfin, cliquez sur le bouton FINISH. Vous devriez voir dans la partie droite de l'environnement de développement une arborescence de type : « Workspace », « Le nom de votre futur programme », « Sources ».

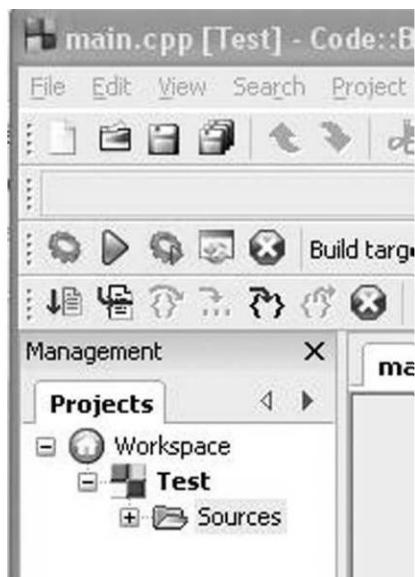


Figure 1-30 Code::Blocks :L'arborescence de programmation.

Cliquez sur le symbole +, placez devant « Sources », vous déplierez le contenu des codes sources et vous devriez voir « main.cpp », le source principal.

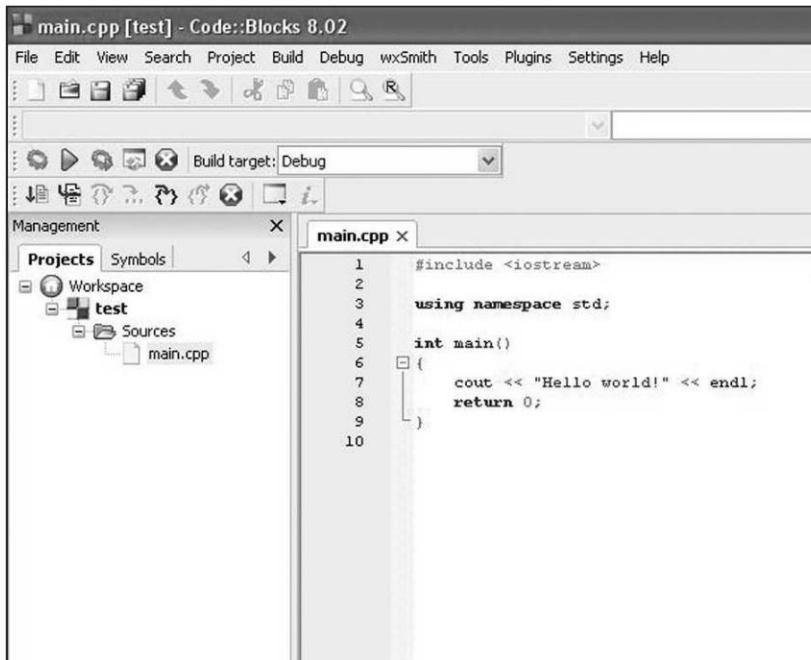


Figure 1-31 Code::Blocks : L’arborescence avec « main.cpp » et la fenêtre d’édition qui contient le programme « Hello world ! » par défaut.

Double-cliquez sur celui-ci, il devrait apparaître dans la grande fenêtre d’édition à droite.

Le code présent est un simple programme qui, lorsqu’il est compilé et exécuté, affiche « Hello, world ! » dans la fenêtre console.

Vous pouvez saisir votre code source à la place.

The screenshot shows the Code::Blocks IDE interface with a more complex code editor window. The title bar reads "*main.cpp [test] - Code::Blocks 8.02". The code in the editor is:

```

10 private:
11     //déclaration des données membres ou attributs privés
12     double vol;
13 public:
14     //déclaration des fonctions membres ou méthodes publiques
15     double volume(double, double);
16 };
17
18 //définition de la fonction volume
19 double cone::volume(double r, double h){
20     //affectation du volume calculé à la variable vol
21     vol=(PI*pow(r,2)*h)/3
22     //renvoie de vol
23     return vol;
24 }
25
26 int main(){
27     //déclarations de 3 instances de la classe cone
28     cone C1, C2, C3;
29     //affichage des résultats via l’appel de la méthode volume
30     cout<<"Volume cone N.1 : "<<C1.volume(2, 4)<<endl;
31     cout<<"Volume cone N.2 : "<<C2.volume(2.5, 4.25)<<endl;
32     cout<<"Volume cone N.3 : "<<C3.volume(3, 4.5)<<endl;
33     system("PAUSE");

```

Figure 1-32 Code::Blocks : La fenêtre d’édition avec votre code source.

Pour compiler et lancer votre programme, allez dans le menu BUILD et choisissez BUILD AND RUN ou cliquez sur la touche de fonction F9 ou encore sur la troisième icône en partant de la gauche, dans la barre d'outils de compilation (Menu VIEW, TOOLBARS, COMPILER).



Figure 1-33 Code::Blocks : La barre d'outils du compilateur (compiler).

Compilation réussie

L'édition de liens, la compilation et l'exécution devraient s'exécuter et la fenêtre console apparaître pour vous montrer le résultat de votre travail.

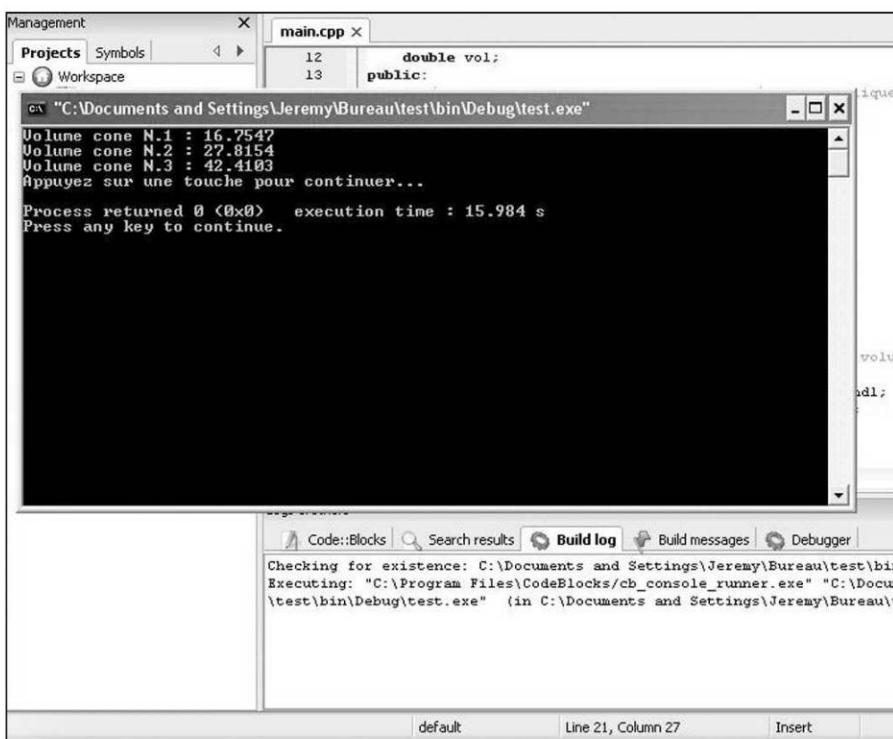


Figure 1-34 Code::Blocks : Compilation réussie et fenêtre console de votre application.

Dans la zone inférieure, sous l'onglet « Build log », la démarche de compilation a dû se dérouler, affichant plusieurs messages.

Compilation impossible

Dans le cas où vous auriez commis une erreur, la compilation s'arrête et le debugger est lancé.

Sous l'onglet inférieur « Build messages », les erreurs ou les alertes s'affichent en précisant un numéro de ligne.

Le numéro de ligne de la première erreur est marqué d'un petit rectangle rouge dans la fenêtre d'édition.

The screenshot shows the Code::Blocks IDE interface. At the top, there is a code editor window containing C++ code. Below it is a 'Logs & others' docked window with tabs for 'Code::Blocks', 'Search results', 'Build log', 'Build messages' (which is selected), and 'Debugger'. The 'Build messages' tab displays a list of compilation errors and warnings. The first error is highlighted with a red rectangle around line 23. The error message is: 'In member function 'double cone::volume(double, double)': error: expected ';' before "return"'. The file path is 'C:\Documents\...'. The bottom of the window shows the status bar with the file name 'u\test\main.cpp', the build configuration 'default', the current line 'Line 23, Column 1', and buttons for 'Insert' and 'Read/Write'.

```

21     vol=(PI*pow(r,2)*h)/3
22     //fif
23     return vol;
24 }
25
26 int main(){
27     //déclarations de 3 instances de la classe cone
28     cones C1, C2, C3;
29     //affichage des résultats via l'appel de la méthode volume
30     cout<<"Volume cone N.1 : "<<C1.volume(2, 4)<<endl;
31     cout<<"Volume cone N.2 : "<<C2.volume(2.5, 4.25)<<endl;

```

File	Line	Message
C:\Documents\...		In member function 'double cone::volume(double, double)':
C:\Documents\...	23	error: expected ';' before "return"
C:\Documents\...		In function 'int main()':
C:\Documents\...	28	error: 'cones' was not declared in this scope
C:\Documents\...	28	error: expected ';' before "C1"
C:\Documents\...	30	error: 'C1' was not declared in this scope
C:\Documents\...	31	error: 'C2' was not declared in this scope
C:\Documents\...	32	error: 'C3' was not declared in this scope
C:\Documents\...	28	warning: unused variable 'cones'
C:\Documents\...	30	warning: unused variable 'C1'
C:\Documents\...	31	warning: unused variable 'C2'
C:\Documents\...	32	warning: unused variable 'C3'
		== Build finished: 6 errors 4 warnings ==

Figure 1-35 Code::Blocks : Compilation impossible, l'onglet « Build messages » et la première erreur, marquée par un petit rectangle rouge (ligne 23) en face du code source, dans la fenêtre d'édition.

Après correction, vous pouvez relancer la compilation.

ERREURS COURANTES EN C++

Voici une liste non-exhaustive d'erreurs souvent rencontrées dans un code source C++.

- Confondre l'égalité **==**, dans un test et l'affectation **=** pour attribuer une valeur à une variable.
- Les points-virgules en trop ou manquant en fin de ligne, comme par exemple dans le cas d'une boucle **for** ou **while**.
- Le mauvais choix de l'opérateur lors de la redirection des flux vers les entrées-sorties via **cout** et **cin** ;<< à la place de **>>**.
- Utilisation d'un type mal approprié lors d'une division, en effet C++ utilise **/** pour les deux divisions, entière et réelle, et détermine le résultat en fonction du type des opérandes.

- Oublie de l'instruction `break` dans une instruction de choix `switch`.
- Valeur de retour mal précisée dans le retour d'une fonction.
- Confusion entre `&&` (et logique) et `&` (et binaire) qui évalue deux opérandes.
- Confusion entre `||` (ou logique) et `|` (ou binaire).

Il existe de nombreuses autres erreurs possibles, prenez le temps de consulter la documentation de chaque EDI, pour en savoir un peu plus.



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* exprimées comme des caractères ordinaires.

Code ASCII

Caractère	Décimal	Hexa	Clavier	Terme anglais	Description
NULL	0	00	Ctrl+@	<i>Null</i>	Nul
SOH	1	01	Ctrl+A	<i>Start of heading</i>	Début d'en-tête
STX	2	02	Ctrl+B	<i>Start of text</i>	Début de texte
ETX	3	03	Ctrl+C	<i>End of text</i>	Fin de texte
EOT	4	04	Ctrl+D	<i>End of transmit</i>	Fin de communication
ENQ	5	05	Ctrl+E	<i>Enquiry</i>	Demande
ACK	6	06	Ctrl+F	<i>Acknowledge</i>	Accusé de réception
BELL	7	07	Ctrl+G	<i>Bell</i>	Sonnerie
BS	8	08	Ctrl+H	<i>Backspace</i>	Retour arrière
HT	9	09	Ctrl+I	<i>Horizontal tab</i>	Tabulation horizontale
LF	10	0A	Ctrl+J	<i>Line feed</i>	Interligne
VT	11	0B	Ctrl+K	<i>Vertical tab</i>	Tabulation verticale
FF	12	0C	Ctrl+L	<i>Form feed</i>	Page suivante
CR	13	0D	Ctrl+M	<i>Carriage return</i>	Retour en début de ligne
SO	14	0E	Ctrl+N	<i>Shift out</i>	Hors code
SI	15	0F	Ctrl+O	<i>Shift in</i>	En code
DLE	16	10	Ctrl+P	<i>Data line escape</i>	Echappement en transmission

DC1	17	11	Ctrl+Q	<i>Device control 1</i>	Commande auxiliaire n° 1
DC2	18	12	Ctrl+R	<i>Device control 2</i>	Commande auxiliaire n° 2
DC3	19	13	Ctrl+S	<i>Device control 3</i>	Commande auxiliaire n° 3
DC4	20	14	Ctrl+T	<i>Device control 4</i>	Commande auxiliaire n° 4
NAK	21	15	Ctrl+U	<i>Negative acknowledge</i>	Accusé de réception négatif
SYN	22	16	Ctrl+V	<i>Synchronous idle</i>	Synchronisation
ETB	23	17	Ctrl+W	<i>End of transmit block</i>	Fin de bloc transmis
CAN	24	18	Ctrl+X	<i>Cancel</i>	Annulation
EM	25	19	Ctrl+Y	<i>End of medium</i>	Fin de support
SUB	26	1A	Ctrl+Z	<i>Substitute</i>	Remplacement
ESC	27	1B	Ctrl+[<i>Escape</i>	Echappement
FS	28	1C	Ctrl+\	<i>File separator</i>	Séparateur de fichier
GS	29	1D	Ctrl+]	<i>Group separator</i>	Séparateur de groupe
RS	30	1E	Ctrl+^	<i>Record separator</i>	Séparateur d'enregistrement
US	31	1F	Ctrl+_	<i>Unit separator</i>	Séparateur d'unité
SP	32	20	-	<i>Space</i>	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
/	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	-	-	Égal
>	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
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
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		<i>Backslash - Antislash</i>	Barre oblique inverse
]	93	5Dh		-	Crochet fermant
^	94	5Eh		-	Accent circonflexe
_	95	5Fh		<i>Underscore</i>	Souligné
`	96	60h		-	Accent grave
a	97	61h		-	Lettre a minuscule

b	98	62h		-	Lettre b minuscule
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
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		<i>Delete</i>	Effacement

Liste des opérateurs, priorité et arité

L'**arité** qui indique si l'opérateur porte sur un, deux ou trois opérandes (*unaire*, *binaire* ou *ternaire*) et la surcharge¹ qui peut être possible ou non.

Opérateur	Nom	Syntaxe	Préséance Priorité	Associa- tivité	Arité	Surcharge
Opérateurs divers (portée, sélection, pointeur,...)						
::	Résolution, portée globale	class_nom :: membre	17	Droite	Unaire	Non
::	Résolution, portée de classe	::nom	17	Gauche	Binaire	Non
.	Sélection de membre	objet.membre	16	Gauche	Binaire	Non
->	Sélection de membre	pointeur ->membre	16	Gauche	Binaire	Oui
[]	Indexation	pointeur[exp]	16	Gauche	Binaire	Oui
()	Appel de fonction	expexp_liste()	16	Gauche		Oui
()	Construction	type(exp_liste)	16	Gauche		Oui

¹ Voir chapitre 8 §8.1, « La surcharge des opérateurs ».

Opérateurs unaires de pré-incrémantation, décrémentation						
++	Post incrémentation	lvalue ++	16	Droite	Unaire	Oui
--	Post décrémentation	lvalue --	16	Droite	Unaire	Oui
Opérateurs unaires de post-incrémantation, décrémentation						
++		++ lvalue	15	Droite	Unaire	Oui
--		-- lvalue	15	Droite	Unaire	Oui
Opérateurs unaires de taille						
sizeof	Taille d'objet		15	Droite	Unaire	Non
sizeof	Taille de type		15	Droite	Unaire	Oui
Opérateurs logiques						
!	NON logique	! exp	15	Droite	Unaire	Oui
+	Plus unaire		15	Droite	Unaire	Oui
-	Moins unaire		15	Droite	Unaire	Oui
*	Déréférence		15	Droite	Unaire	Oui
&	Adresse		15	Droite	Unaire	Oui
new	Allocation		15	Droite	Unaire	Oui
delete	Désallocation		15	Droite	Unaire	Oui
()	Conversion de type		15	Droite	Binaire	Oui
.*	Sélection membre directe		14	Gauche	Binaire	Non
->*	Sélection membre indirecte		14	Gauche	Binaire	Oui
Opérateurs arithmétiques simples						
*	Produit	exp * exp	13	Gauche	Binaire	Oui
/	Quotient	exp / exp	13	Gauche	Binaire	Oui
%	Modulo ou reste	exp % exp	13	Gauche	Binaire	Oui

+	Somme	exp + exp	12	Gauche	Binaire	Oui
-	Différence	exp - exp	12	Gauche	Binaire	Oui
Opérateur d'insertion ou de sortie						
<<	Insertion à gauche	exp << exp	11	Gauche	Binaire	Oui
Opérateur d'extraction ou d'entrée						
>>	extraction à droite	exp >> exp	11	Gauche	Binaire	Oui
Opérateurs relationnels						
<	Inférieur à	exp < exp	10	Gauche	Binaire	Oui
<=	Inférieur ou égal à	exp <= exp	10	Gauche	Binaire	Oui
>	Supérieur à	exp > exp	10	Gauche	Binaire	Oui
>=	Supérieur ou égal à	exp >= exp	10	Gauche	Binaire	Oui
==	Egal à	exp == exp	9	Gauche	Binaire	Oui
!=	Different de	exp != exp	9	Gauche	Binaire	Oui
Opérateurs binaires						
&	ET bit à bit	exp & exp	8	Gauche	Binaire	Oui
^	OU exclusif bit à bit	exp ^ exp	7	Gauche	Binaire	Oui
	OU inclusif bit à bit	exp exp	6	Gauche	Binaire	Oui
~	Complément bit à bit	~ exp	15	Droite	Unaire	Oui
Opérateurs logiques						
&&	ET logique	exp && exp	5	Gauche	Binaire	Oui
	OU logique inclusif	exp exp	4	Gauche	Binaire	Oui
Opérateur d'expression conditionnelle						
?:	Expression conditionnelle	exp ? exp : exp	3	Gauche	Ternaire	Non
Opérateur d'affectation						
=	affectation conventionnelle	lvalue = exp	2	Droite	Binaire	Oui

Opérateurs d'affectations composées						
<code>*=</code>	Produit et affectation	<code>lvalue *= exp</code>	2	Droite	Binaire	Oui
<code>/=</code>	Quotient et affectation	<code>lvalue /= exp</code>	2	Droite	Binaire	Oui
<code>%=</code>	Modulo et affectation	<code>lvalue %= exp</code>	2	Droite	Binaire	Oui
<code>+=</code>	Somme et affectation	<code>lvalue += exp</code>	2	Droite	Binaire	Oui
<code>-=</code>	Différence et affectation	<code>lvalue -= exp</code>	2	Droite	Binaire	Oui
Opérateurs de décalages composés						
<code><<=</code>	Décalage bit gauche et affectation	<code>lvalue <<= exp</code>	2	Droite	Binaire	Oui
<code>>>=</code>	Décalage bit droit et affectation	<code>lvalue >>= exp</code>	2	Droite	Binaire	Oui
Opérateurs binaires d'affectations composées						
<code>&=</code>	ET bit à bit et affectation	<code>lvalue &= exp</code>	2	Droite	Binaire	Oui
<code> =</code>	OU inclusif bit à bit et affectation	<code>lvalue = exp</code>	2	Droite	Binaire	Oui
<code>^=</code>	OU exclusif bit à bit et affectation	<code>lvalue ^= exp</code>	2	Droite	Binaire	Oui
<code>throw</code>	Levée d'exception	<code>throw (exp)</code>	1	Droite	Unaire	Oui
,	virgule	<code>exp, exp</code>	0	Gauche	Binaire	Oui

Dans ce tableau, j'ai rassemblé les *opérateurs* sous leur famille respective tout en respectant leur *priorité*.



Mots-clés ou mots réservés

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 une 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
double	Type nombre réel
dynamic_cast	Opérateur de conversion dynamique
else	Alternative de l'instruction conditionnelle <code>if</code>
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
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
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

La bibliothèque C++ standard

Le tableau ci-dessous présente la *bibliothèque C++ (STL – Standard Template Library)*

La colonne de gauche, contient les 16 fichiers d'en-tête qui appartaient déjà au langage C. La colonne de droite contient les 33 fichiers d'en-tête C++.

La norme ISO/IEC 14882:2003 du langage C++ précise les points suivant pour les dénominations des fichiers d'en-tête :

- Les 16 fichiers hérités du langage C doivent être précédés de la lettre c. Par exemple `stdio.h` devient `cstdio`.
- Les 33 fichiers C++ doivent abandonner l'extension .h. Par exemple `iostream.h` devient `iostream`.

Bibliothèque C	Bibliothèque STL
<code>cassert</code>	<code>algorithm</code>
<code>cctype</code>	<code>bitset</code>
<code>cerrno</code>	<code>complex</code>
<code>cfloat</code>	<code>deque</code>
<code>ciso646</code>	<code>exception</code>
<code>climits</code>	<code>fstream</code>
<code>clocale</code>	<code>functional</code>
<code>cmath</code>	<code>iomanip</code>
<code>csetjmp</code>	<code>ios</code>
<code>csignal</code>	<code>iosfwd</code>
<code>cstdarg</code>	<code>iostream</code>
<code>cstddef</code>	<code>istream</code>
<code>cstdio</code>	<code>iterator</code>
<code>cstdlib</code>	<code>limits</code>
<code>cstring</code>	<code>List</code>
<code>ctime</code>	<code>locale</code>
	<code>map</code>
	<code>memory</code>
	<code>new</code>
	<code>numeric</code>
	<code>ostream</code>
	<code>queue</code>
	<code>set</code>
	<code>sstream</code>
	<code>stack</code>
	<code>stdexcept</code>
	<code>strstream</code>
	<code>streambuf</code>
	<code>string</code>
	<code>typeinfo</code>
	<code>utility</code>
	<code>valarray</code>
	<code>vector</code>

Les compilateurs n'intègrent pas tous les fichiers d'en-tête, certains en possèdent même d'autres, cependant la plupart supportent les suivants :

- **cctype** : Classement des entiers, lettres, majuscules, minuscules etc.
- **climits** : Valeurs maximales et minimales pour les types de base.
- **cmath** : Fonctions mathématiques.
- **cstdio** : Fonctions d'entrées-sorties.
- **cstdlib** : Fonctions standards, fonctions d'allocation mémoire, fonctions de recherche, fonctions de conversion, etc.
- **cstring** : Fonction de traitement des chaînes de caractères, fonctions de manipulation de données en mémoire.
- **ctime** : Fonction de manipulation et de conversion de la date et de l'heure et **iostream**, **istream**, **ostream**, **fstream**, **sstream**, **iuomanip**, **ios**, **fstream**, **streambuf**, **strstream** : fonctions de gestion des flux d'entrée et de sortie.

Comme je l'ai précisé plus haut, beaucoup de compilateurs possèdent des bibliothèques propriétaires spécifiques qui autorisent la gestion d'environnements de développement, notamment graphiques, 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 (voir Avant Propos dans ce livre).

Les fonctions externes prédéfinies

Vous trouverez, dans le tableau qui suit, quelques-unes des principales fonctions prédéfinies de C++.

La colonne de droite précise le fichier d'en-tête auquel appartient la fonction (suivant la norme ISO/IEC 14882).

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 (d1, 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
atol (s)	int	Convertit s en nombre entier long	cstdlib
ceil (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

<code>difftime (u1, u2)</code>	<code>double</code>	Renvoie la différence $u_1 - u_2$, où u_1 et u_2 sont des valeurs de temps écoulé depuis une date de référence (voir la fonction <code>time</code>)	<code>ctime</code>
<code>exit (u)</code>	<code>void</code>	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	<code>cstdlib</code>
<code>exp (d)</code>	<code>double</code>	Élève e à la puissance d ($e = 2.7182818\dots$ est la base des logarithmes népériens)	<code>cmath</code>
<code>fabs (d)</code>	<code>double</code>	Renvoie la valeur absolue de d	<code>cmath</code>
<code>close (f)</code>	<code>int</code>	Ferme le fichier f et renvoie 0 en cas de fermeture normale	<code>cstdio</code>
<code>feof (f)</code>	<code>int</code>	Détermine si une fin de fichier est atteinte. Renvoie dans ce cas une valeur non nulle, et 0 dans le cas contraire	<code>cstdio</code>
<code>fgetc (f)</code>	<code>int</code>	Lit un caractère unique dans le fichier f	<code>cstdio</code>
<code>fgets (s, i, f)</code>	<code>char*</code>	Lit une chaîne s formée de i caractères dans le fichier f	<code>cstdio</code>
<code>floor (d)</code>	<code>double</code>	Renvoie l'arrondi à l'entier immédiatement inférieur de d	<code>cmath</code>
<code>fmod (d1, d2)</code>	<code>double</code>	Renvoie le reste de d_1/d_2 (avec le signe de d_1)	<code>cmath</code>
<code>fopen (s1, s2)</code>	<code>file*</code>	Ouvre le fichier s_1 , de type s_2 . Renvoie un pointeur sur ce fichier	<code>cstdio</code>
<code>fprintf (f, ...)</code>	<code>int</code>	Écrit des données dans le fichier f	<code>cstdio</code>
<code>fputc (c, f)</code>	<code>int</code>	Écrit un caractère simple c dans le fichier f	<code>cstdio</code>
<code>fputs (s, f)</code>	<code>int</code>	Écrit la chaîne s dans le fichier f	<code>cstdio</code>

<code>fread(s,i1, i2,f)</code>	int	Lit <code>i2</code> données de longueur <code>i1</code> (en octets) depuis le fichier <code>f</code> dans la chaîne <code>s</code>	cstdio
<code>free(p)</code>	void	Libère la zone de mémoire débutant à l'adresse indiquée par <code>p</code>	cstdlib
<code>fscanf(f, ...)</code>	int	Lit des données dans le fichier <code>f</code>	cstdlib
<code>fseek(f,i, i)</code>	int	Décale, de 1 octet à partir de l'adresse <code>i</code> , le pointeur sur le fichier <code>f</code> (où <code>i</code> peut représenter le début ou la fin du fichier, ou la position d'un enregistrement donné)	cstdio
<code>ftell(f)</code>	long int	Renvoie la position courante du pointeur dans le fichier <code>f</code>	cstdio
<code>fwrite(s, i1,i2,f)</code>	int	Écrit <code>i2</code> données de longueur <code>i1</code> , depuis la chaîne <code>s</code> dans le fichier <code>f</code>	cstdio
<code>getc(f)</code>	int	Lit un caractère simple dans le fichier <code>f</code>	cstdio
<code>getchar</code>	int	Lit un caractère simple sur l'unité d'entrée standard	cstdio
<code>gets(s)</code>	char*	Lit la chaîne <code>s</code> sur l'unité d'entrée standard	cstdio
<code>isalnum(c)</code>	int	Détermine si <code>c</code> , l'argument donné, est de type alphanumérique. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype
<code>isalpha(c)</code>	int	Détermine si <code>c</code> , l'argument donné, est de type alphabétique. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype
<code>isascii(c)</code>	int	Détermine si <code>c</code> , l'argument donné, correspond à un code ASCII. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype

iscntrl (c)	int	Détermine si c, 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 c, 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 c, l'argument donné, correspond à un caractère graphique imprimable du code ASCII (codes hexa 0 x21 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 c, 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 c, l'argument donné, est un chiffre octal. cctype Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype
isprint (c)	int	Détermine si c, 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
ispunct (c)	int	Détermine si c, l'argument donné, est un caractère de ponctuation. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype
isspace (c)	int	Détermine si c, l'argument donné, est un caractère d'espacement. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype

isupper (c)	int	Détermine si c, 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 c, l'argument donné est un chiffre hexadécimal. Renvoie une valeur non nulle si tel est le cas, nulle sinon	cctype
labs (s)	long int	Renvoie la valeur absolue de s	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 (d1, 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
putc (c, f)	int	Écrit un caractère c, dans le fichier f	cstdio
putchar (c)	int	Écrit un caractère c, 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 cstdio	cstdio
sin (d)	double	Renvoie le sinus de d	cmath

<code>sinh(d)</code>	<code>double</code>	Renvoie le sinus hyperbolique de d	<code>cmath</code>
<code>sqrt(d)</code>	<code>double</code>	Renvoie la racine carrée de d	<code>cmath</code>
<code>srand(u)</code>	<code>void</code>	Initialise le générateur de nombres aléatoires	<code>cstdlib</code>
<code>strempi(s1,s2)</code>	<code>int</code>	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 $s1 > s2$	<code>cstring</code>
<code>strcpy(s1,s2)</code>	<code>char*</code>	Copie de la chaîne $s2$ dans $s1$ Renvoie le nombre de caractères de la chaîne $s1$	<code>cstring</code>
<code>strset(s1,s2)</code>	<code>char*</code>	Comparaison lexicographique de deux chaînes. Renvoie une valeur négative si $s1 < s2$, nulle si $s1$ et $s2$ sont identiques et positive dans le cas où $s1 > s2$	<code>cstring</code>
<code>strcmp(s1,s2)</code>	<code>int</code>	Remplace tous les caractères de la chaîne s par c (à l'exception du caractère de fin de chaîne)	<code>cstring</code>
<code>system(s)</code>	<code>int</code>	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	<code>cstdlib</code>
<code>tan(d)</code>	<code>double</code>	Renvoie la tangente de d	<code>cmath</code>
<code>tanh(d)</code>	<code>double</code>	Renvoie la tangente hyperbolique de d	<code>cmath</code>
<code>time(d)</code>	<code>long int</code>	Renvoie le nombre de secondes écoulées depuis une date d spécifiée	<code>ctime</code>
<code>toascii(c)</code>	<code>int</code>	Convertit la valeur de l'argument c en code ASCII	<code>cctype</code>
<code>tolower(c)</code>	<code>int</code>	Convertit une lettre en minuscule	<code>cctype ou cstdlib</code>
<code>toupper(c)</code>	<code>int</code>	Convertit une lettre en majuscule	<code>cctype ou cstdlib</code>



Références bibliographiques

Bibliographie

- STROUSTRUP B. – *Le langage C++*, Pearson éducation, 2004. ISBN 2744070033 ou 978-2744070037
- STROUSTRUP B. – *Programming : Principles and practice using C++*, Addison Wesley Educationnal, 2009. ISBN 0321543726 ou 978-0321543721
- MEYERS S. – *Le C++ efficace*, Vuibert, 2001. ISBN 271178682X ou 978-2711786824
- DEITEL P.J. – *C++ How to program : international version*, Pearson Education, 2009. ISBN 013246540X ou 978-0132465403
- OUALLINE. – *Practical C++ programming*, O'Reilly, 2003. ISBN 0596004192 ou 978-0596004194
- SOLTER N.A. et KLEPER S.J. – *Professional C++*, Hungry Minds Inc, 2005. ISBN 0764574841 ou 978-0764574849
- COGSWELL J.M., LISCHNER R. et STEPHENS R. – *C++ Cookbook*, O'Reilly media, 2005. ISBN 0596007612 ou 978-0596007614
- DELANNOY C. – *Programmer en langage C++*, Eyrolles, 2000. ISBN 2212088973 ou 978-2212088977
- VASSILIU M. – *Le langage C++*, Pearson education, 2005. ISBN 2744073830 ou 978-2744073830

Liens internet¹

- <http://fr.wikipedia.org/wiki/C%2B%2B> – Le wiki du C++.
- <http://www.cppfrance.com> – Un site de référence pour C++ (*en français*).
- http://fr.wikibooks.org/wiki/Programmation_C%2B%2B_%28d%C3%A9butant%29 – Wikibooks : Programmation C++ débutant (*en anglais*).

¹ Les liens internet ne sont pas pérennes et varient. Dans le cas où certains ne fonctionneraient plus, effectuez une recherche *via* www.google.com.

- <http://www2.research.att.com/~bs/homepage.html> – Le site du maître : Bjarne Stroustrup – Superbe – Une mine... (*en anglais*).
- http://fr.wikibooks.org/wiki/Programmation_C%2B%2B – Un autre Wikibook – Programmation C++ (*en anglais*)
- <http://www.cpp-home.com/> – C++ Home – FAQ, Articles, Tutoriels... (*en anglais*).
- <http://www.developpez.net/forums/f7/c-cpp/> – Forum d’entraide sur C++ et d’autres langages (*en français*).
- <http://www.possibility.com/Cpp/CppCodingStandard.html#stacknames> – Tout sur le codage C++ dans le respect de son standard – Intéressant (*en anglais*).
- <http://www.aidocours.fr/index-cours-597.html> – Le site « aidocours » – Nombreux cours sur des sujets très différents dont un cours sur C++ – Bien fait et pédagogique (*en français*).
- http://artis.imag.fr/~Xavier.Decoret/resources/stl_tutorial/index.html – Un tutoriel sur STL, la bibliothèque standard C++ – Bien écrit et complet (*en français*).
- <http://www.nawouak.net/?doc=course.cpp+lang=fr> – Programmation orientée objet avancée en C++ – Pour aller plus loin en POO (*en français*).
- <http://www.medini.org/stl/> – Une mine d’informations sur STL (*en anglais*).
- http://www-d0.fnal.gov/~dladams/cxx_standard.pdf – La norme ISO 14882 :1998 – La référence ! 1^{ère} édition – Document PDF (*en anglais*).
- <http://openassist.googlecode.com/files/C%2B%2B%20Standard%20-%20ANSI%20ISO%20IEC%2014882%202003.pdf> – La norme ISO 14882 :2003 – La référence ! – Dernière version – Document PDF (*en anglais*).
- <http://casteyde.christian.free.fr/cpp/cours/online/book1.html> – Un cours de C++ très complet (*en anglais*).
- <http://www.cplusplus.com/reference/stl/> – Tout sur C++ – Énorme... (*en anglais*).



Index

<< 143

>> 143

A

accès direct 41

accessibilité 13

affectation 11, 22, 141

agrégation 151, 162

alias 38

amie 133, 136

amies 115

arguments effectifs 85

arguments formels 85

arité 197

arithmétique 22

arrondi 26

ASCII 14, 29

assignation 22

associativité 22

attributs 114

B

bibliothèque C++ 205

binnaire 22, 56, 197

bit à bit 22

Bloodshed Dev C++ 167

booléen 22

boucles 55, 59

break 58, 64

buffer de sortie 56

C

casse 12

cast 20

cctype 206

cfloat 17

champs 108

char 12, 14

cin 55

classe 21, 94, 107, 114

classe dérivée 151, 152, 162

classe polymorphe 159

classes génériques 161

climits 14, 206

cmath 206

Code::blocks 167, 180

composition 151

concaténation 32

concaténer 31

Console OUTput 56

constantes 21

constructeur 107, 118

constructeur par copie 121, 134

constructeur par défaut 118

continue 64, 66

conversion 18

conversion implicite 25

corps 84

cout 55

cstdio 206

cstdlib 206

cstring 206

ctime 206

D

débordements 91

décalage 22, 74

déclaration 12, 84
 décrémentation 25, 144
 default 58
 définition 84
 delete 75
 déréférencement 72
 dérivation 108, 152
 destructeurs 107, 126
 division entière 25
 do...while 59
 données membres 114
 double 12, 17

E

EDI 10
 en-tête 84
 encapsulation 108
 endl 56
 entrée-sortie 143
 enum 36
 énumérateurs 36
 environnements de développement
 intégré 10
 erreurs courantes 186
 étiquette 66, 115
 exit 83
 exit() 100
 explicites 20
 expression conditionnelle 22
 extension 101
 externes prédéfinies 207
 extraction 23

F

float 12, 17
 flot d'entrée 155
 flot de sortie 155
 flush() 56
 fonction 83
 fonctions amies 143
 fonctions génériques 159
 fonctions membre 21, 43, 114
 fonctions virtuelles 151, 157
 for 59
 free 76

friend 133, 136
 fstream 206

G

généricité 157
 getline 114
 goto 64, 66

H

héritage 108, 151, 152
 héritage multiple 155, 162
 hex 31

I

identificateur(s) 11, 155
 implicites 18
 incrément 63
 incrémentation 144
 index d'initialisation 62
 indexation en base 0 42
 indice 41
 inline 83, 101
 insertion 23
 instance 108
 instanciation 108
 instruction 55
 int 12, 14
 ios 206
 iostream 143, 155, 206
 istream 143, 206
 itérations 55
 itératives 55
 iomanip 206

L

lecture seule 91
 lecture-écriture 91
 lien dynamique 157
 lien statique 157
 listes d'initialisation de constructeurs
 120
 locales 85
 logique 22
 long double 17
 long int 14

M

main 83, 84
malloc 76
matrice 41
membres 108
membres privés 152
membres publics 152
méthodes 114
Microsoft Visual C++ 2008 Express
 173

mode console 56, 167
modèles de fonctions 162
mots-clés 12, 201
mots réservés 12, 201

N

\n 56
new 75
notation fonctionnelle 20

O

objet 108
oct 31
offset 74
opérateur d'adresse 72
opérateur de résolution de portée 115
opérateur unaire d'indirection 72
opérateurs 197
opérateurs unaires 25
operator 134
ostream 56, 143, 206
overriding 157

P

paramètres 85
paramètres formels 85
pas 63
passage par référence 85, 91
passage par valeur 91
patrons 151
patrons de classe 161, 162
patrons de fonction 159, 161, 162
pointeur 56, 71
polymorphes 159
polymorphisme 108, 151, 157
polymorphisme ad hoc 133, 157

polymorphisme d'héritage 157, 162
polymorphisme d'inclusion 157
polymorphisme paramétrique 157
POO 107, 151
portée 13, 37, 115
post 25
pré-incrémantation 25
préséance 22
priorité 22, 197
private 115, 126, 136, 152, 162
programmation orientée objet 107
protected 115, 162
prototypage 84, 85
prototype 84
public 115, 126, 152

Q

qualificateurs 12
qualificatifs d'accès 115

R

récursion 89
récurivité 83, 89
redéfinition 157
réels 85
référence 91, 134, 143
relationnel 23

S

séquences d'échappement 12, 189
séquentielles 55
short 12
short int 14
si condition 55
signed 12, 14
sous-classes 108
spécificateurs 12
sstream 206
static 123
STL 46, 205
strcat 31
strchr 31
strcmp 31
strcpy 31
streambuf 206
string 29, 31