Claude Delannoy

Programmer en C++ moderne

De C++11 à C++20

Éditions EYROLLES

Acquérir une parfaite maîtrise du C++ et de la programmation objet

Les versions C++11, C++14 et C++17 ont apporté au langage C++ plus que de nouvelles fonctionnalités : une nouvelle façon de programmer. Dès lors, une refonte complète du classique *Programmer en langage C++* de Claude Delannoy s'imposait. C'est à cette tâche que s'est attelé l'auteur à l'occasion de cette 10e édition de l'ouvrage, en réécrivant les exemples de code et en préconisant de bonnes pratiques de programmation dans l'esprit de ce C++ moderne.

L'ouvrage ainsi remanié commence par une présentation détaillée de la syntaxe de base du langage, s'appuyant dès que possible sur les structures de données de la bibliothèque standard (types *string* et *vector*) et sur la déclaration automatique (C++11). Puis il expose les techniques de gestion dynamique basées sur les « pointeurs intelligents » introduits par C++11 et C++14.

L'auteur insiste ensuite sur la bonne compréhension des concepts objet et de la programmation générique à l'aide des « patrons ». Un chapitre est consacré à la « sémantique de déplacement » introduite par C++11. Plusieurs chapitres sont dédiés aux conteneurs et aux algorithmes de la STL (*Standard Template LIbrary*). Les nouveautés de C++20 (concepts et contraintes, modules, coroutines…) sont présentées en annexe.

Chaque notion nouvelle et chaque fonction du langage est illustrée de programmes complets écrits en C++ moderne, dont le code source est fourni sur le site www.editions-eyrolles.com. Un équivalent en C++03 est proposé quand nécessaire pour les lecteurs amenés à exploiter d'anciens programmes.

À qui s'adresse ce livre ?

- **Aux étudiants de cursus universitaires (DUT, licence, master), ainsi qu'aux élèves des écoles d'ingénieurs.**
- **À tout programmeur ayant déjà une expérience de la programmation (C, C#, Java, Python, PHP…) et souhaitant s'initier au langage C++.**

Au sommaire

Ingénieur informaticien au CNRS, **Claude Delannoy** possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 500 000 exemplaires vendus.

www.editions-eyrolles.com
Éditions Eyrolles | Diffusion Geodif

Conception de couverture :
Studio Eyrolles © Éditions Eyrolles

Code éditeur : G67895
ISBN : 978-2-212-67895-2

Programmer en C++ moderne

CHEZ LE MÊME ÉDITEUR

Du même auteur

C. Delannoy. – **Exercices en langage C++** - *178 exercices corrigés*.
N°67663, 4e édition, 2018, 396 pages.

C. Delannoy. – **Programmer en Java** (*couvre Java 9*).
N°67536, 10e édition, 2017, 920 pages.

C. Delannoy. – **Exercices en Java** (*couvre Java 8*).
N°67385, 4e édition, 2017, 346 pages.

C. Delannoy. – **Programmer en langage C** *Cours et exercices corrigés*
N°11825, 5e édition, 2016, 268 pages.

C. Delannoy. – **Le guide complet du langage C**.
N°14012, 2014, 844 pages.

C. Delannoy. – **S'initier à la programmation et à l'orienté objet**
Avec des exemples en C, C++, C#, Python, Java et PHP
N°11826, 2e édition, 2016, 360 pages.

Dans la même collection

A. Tasso. – **Le livre de Java premier langage** *Avec 109 exercices corrigés*.
N°67840, 13e édition, 2019, 600 pages.

H. Bersini, P. Alexis, G. Degols. – **Apprendre la programmation web avec Python et Django**.
N°67515, 2018, 396 pages.

H. Bersini, I. Wellesz. – **La programmation orientée objet**.
Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ.
N°67399, 7e édition, 2017, 696 pages.

J. Engels. – **PHP 7**.
N°67360, 2017, 585 pages.

P. Roques. – **UML 2.5 par la pratique**. *Études de cas et exercices corrigés*.
N°67565, 8e édition, 2018, 408 pages.

C. Soutou. – **Programmer avec MySQL**.
N°67379, 5e édition, 2017, 522 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

Claude Delannoy

Programmer en C++ moderne

De C++11 à C++20

Éditions EYROLLES

Éditions Eyrolles
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

À l'occasion de cette 10e édition, l'ouvrage *Programmer en langage C++* de Claude Delannoy change de titre pour s'appeler désormais *Programmer en C++ moderne*.

ISBN : 978-2-212-67895-2

Avant-propos

Depuis sa conception dans les années 1980 par B. Stroustrup, le C++ a tout naturellement évolué à travers diverses normalisations. Mais, avec la version C++11, le langage a été suffisamment modifié pour qu'on se mette à parler de « C++ moderne ». Le présent ouvrage constitue un remaniement profond des éditions précédentes ; non seulement, il intègre les nouvelles possiblités offertes par les versions C++11, C++14, C++17 et C++20 mais, de surcroît, il présente les nouvelles pratiques de progammation induites par ce changement radical. Les exemples de codes ont tous été réécrits dans ce sens.

1 À qui s'adresse l'ouvrage

Destiné à tous ceux qui souhaitent maîtriser l'ensemble des facettes de la programmation en C++, cet ouvrage s'adresse aussi bien aux étudiants qu'aux développeurs ou aux enseignants en informatique.

Il ne requiert aucune connaissance en programmation orientée objet dont les principes seront présentés en détail au fil de l'étude. Il en va de même pour le langage C dans la mesure où les spécificités de C++ issues du C seront exposées avec le même niveau de détail que les autres.

En revanche, il reste préférable que le lecteur possède déjà une petite expérience de la programmation procédurale (ou structurée), c'est-à-dire dire qu'il soit familiarisé avec les notions de variables, de types, d'affectation, de structures de contrôle, de fonctions, etc., communes à la plupart des langages en usage aujourd'hui (C/C++, Python, JavaScript, Java, PHP...).

2 Structure et contenu de l'ouvrage

L'ouvrage commence par un chapitre très général présentant de façon globale les différentes caractéristiques du langage, montrant ainsi comment y sont intégrés les différents concepts intervenant en programmation procédurale ou en programmation orientée objet (P.O.O.) et en quoi C++ se démarque des autres langages.

La programmation procédurale (chapitres 2 à 11)

Après un chapitre présentant de façon informelle les rudiments les plus indispensables à l'écriture de programmes complets, on aborde l'ensemble des possibilités de programmation procédurale : types de variables, opérateurs et expressions, instructions de contrôle, fonctions (y compris les expressions lambdas).

La déclaration automatique (*auto*) est présentée à ce niveau de façon à être exploitée dans la suite de l'ouvrage lorsqu'elle offre un intérêt.

On s'appuie très tôt sur le type *string* dont on présente les principales propriétés, et ceci alors même que la notion de classe n'a pas encore été exposée.

On trouvera ensuite la présentation des pointeurs natifs et celle de la gestion dynamique dans laquelle on introduit les pointeurs intelligents (en se limitant à ce niveau aux types de base).

Cette partie s'achève sur :

- l'étude en parallèle des tableaux natifs et des principales propriétés des vecteurs (alors même que, là encore, les proprités des patrons et des conteneurs ne seront étudiées en détail qu'ultérieurement) ;
- l'étude succinte des chaînes de style C dont la connaissance reste nécessaire dans certaines situations.

Le chapitre 5 propose une première approche des entrées/sorties conversationnelles, de façon à pouvoir les employer dans un code, avant même que n'aient été étudiées leurs fonctionnalités détaillées.

La programmation orientée objet (chapitres 12 à 25)

Les aspects orientés objet sont ensuite abordés de façon progressive, mais sans pour autant nuire à l'exhaustivité de l'ouvrage. Nous y traitons, non seulement les purs concepts de P.O.O. (classe, constructeur, destructeur, héritage, redéfinition, polymorphisme, programmation générique), mais aussi les aspects très spécifiques au langage (surdéfinition d'opérateurs, fonctions amies). Nous pensons ainsi permettre au lecteur de devenir parfaitement opérationnel dans la conception, le développement et la mise au point de ses propres classes. C'est ainsi, par exemple, que nous avons largement insisté sur le rôle du constructeur de recopie, ainsi que sur la redéfinition de l'opérateur d'affectation, éléments qui conduisent à la notion de « classe canonique ». Toujours dans le même esprit, nous avons pris soin de bien développer les notions avancées mais indispensables que sont la ligature dynamique et les classes

abstraites, lesquelles débouchent sur la notion la plus puissante du langage (et de la P.O.O.) qu'est le polymorphisme.

Tout ce qui est lié à la sémantique de déplacement introduite par C++11 (références à des *rvalue*, construction et affectation par déplacement, références universelles, fonction *forward*...) a été regroupé dans un chapitre spécifique (23), de façon à permettre au lecteur d'en aborder l'étude que lorsqu'il le juge nécessaire.

Cette partie s'achève par deux chapitres (24 et 25) plus techniques consacrés aux flots et à la gestion des exceptions.

Les structures de données et les algorithmes (chapitres 26 à 31)

On étudie ensuite en détail les principaux éléments de la S.T.L. (*Standard Template Library*) après avoir pris soin d'exposer préalablement d'une part les notions de classes et de fonctions génériques (patrons), d'autre part celles de conteneur, d'itérateur et d'algorithmes qui conditionnent la bonne utilisation de la plupart de ses composants.

C'est naturellement là qu'on trouvera l'étude détaillée des classes *string* et *vector* présentées succinctement auparavant.

Les chapitres finaux et les annexes

Les derniers chapitres sont consacrés :

- au préprocesseur ;
- aux énumérations et aux structures de bas niveau héritées du C (champs de bits et unions) ;
- à une introduction aux threads.

Enfin quelques annexes fournissent des récapitulatifs ou des compléments. On y trouvera notamment :

- un récapitulatif des règles de recherche d'une fonction surdéfinie ;
- la liste complète des algorithmes standards.

Le C++ moderne : sa place dans le livre

Les nouvelles fonctionnalités du C++ moderne sont tout naturellement intégrées au fil du texte. Par exemple :

- les déclarations automatiques (*auto*) et la nouvelle syntaxe d'initialisation (entre accolades) sont présentées dans le chapitre 3 pour les types de base (leur intérêt étant alors limité) et elles seront complétées au fur et à mesure de la rencontre de nouveaux types ;
- les pointeurs intelligents sont présentés au chapitre 10 et seront ensuite utilisés à chaque fois qu'on devra recourir à l'allocation dynamique ;

- le type *initializer_list* est présenté au chapitre 7 et on le retrouvera ensuite tout au long de l'ouvrage et l'on verra les problèmes qu'il peut poser avec l'initialisation avec accolades ;
- la boucle *for* pour les séquences est introduite au chapitre 8 pour le type *string* et on la retrouvera ensuite pour les autres formes de séquences ;
- la sémantique de déplacement sera exposée dans son intégralité dans le chapitre 23.

D'une manière générale, le C++ moderne offre des avantages considérables, tant sur le plan de l'uniformisation du langage que sur celui de l'efficacité de la bibliothèque standard ou encore de l'optimisation du code. Il n'en reste pas moins que le respect strict de la compatibilité avec les versions antérieures introduit de nouvelles difficultés liées :

- à la multiplicité des rédactions possibles qui s'offre au programmeur ;
- à quelques incohérences internes ; par exemple, dans de rares cas, *auto* n'est pas utilisable ou ne fournit pas le type escompté.

De plus, l'universalité des déclarations souhaitées par les concepteurs des nouvelles normes n'est pas toujours au rendez-vous ; en particulier, elle est souvent perturbée par l'introduction du type *initializer_list*.

Fidèle à nos habitudes, ces difficultés sont clairement identifiées au fil du texte. Ainsi, le lecteur restera en mesure d'effectuer ses propres choix sur ces situations délicates.

3 Guide d'utilisation

L'ouvrage est conçu sous la forme d'un cours progressif. Celui qui aborde le C++ pour la première fois devra tenter de l'étudier de façon séquentielle. Néanmoins certaines parties peuvent ne pas être utiles à la poursuite de l'étude : elles sont alors soit placées dans une rubrique « informations complémentaires » pour les plus brèves, soit indiquées clairement par une mention en italique figurant au début du paragraphe correspondant.

Chaque notion fondamentale est illustrée d'un programme simple mais complet, accompagné d'un exemple d'exécution. Le code correspondant peut être trouvé sur le site de l'éditeur *www.editions-eyrolles.com*.

Outre son caractère didactique, l'ouvrage est conçu d'une manière très structurée, de façon à en permettre la consultation au delà de la phase d'apprentissage. C'est ainsi qu'il est doté d'une table des matières très détaillée et d'un index fourni. Les exemples complets peuvent servir à une remémoration rapide du concept qu'ils illustrent. Des encadrés permettent de retrouver rapidement la syntaxe d'une instruction ou certaines règles fondamentales.

Bien que fondé sur le C++ moderne, l'ouvrage propose différents outils permettant, le cas échéant, de bien distinguer l'apport des versions récentes :

- des pictogrammes appropriés mentionnent les apports de chacune des versions récentes (C++11, C++14, C++17 et C++20) ;

- lorsque tout un paragraphe est concerné par l'une de ces versions, outre le pictogramme évoqué, le titre en mentionnera le numéro correspondant ;
- les codes, bien qu'écrits en C++ moderne, fournissent systématiquement en commentaires une formulation pour C++98 ; en outre lorsqu'une partie de code s'appuie sur C++14 ou C++17, une formulation C++11 équivalente est proposée.

Ces outils devraient permettre au développeur d'exploiter d'anciens codes. L'enseignant qui le souhaiterait pourra se bâtir un cours simplifié fondé sur un C++ réduit plus proche du C++ historique que du C++ moderne.

Par ailleurs, le programmeur C abordant l'étude du C++ pourra tirer profit des remarques titrées « En C », lesquelles viennent signaler les différences les plus importantes existant entre C et C++.

Enfin, compte tenu de la popularité du langage Java, nous avons introduit de nombreuses remarques titrées « En Java ». Elles mettent l'accent sur les différences majeures existant entre Java et C++. Elles seront utiles non seulement au programmeur Java qui apprend ici le C++, mais également au lecteur qui, après la maîtrise du C++, souhaitera aborder l'étude de Java.

4 Ce que vous trouverez sur le site de l'éditeur

Sur le site *www.editions-eyrolles.com*, outre les codes sources des différents exemples, vous trouverez deux chapitres qui figuraient dans les précédentes éditions :

- la description des principales fonctions héritées du langage C ;
- l'étude des principaux « design patterns » et leur programmation en C++, ce chapitre étant été lui aussi remanié pour le C++ moderne.

Table des matières

1

Présentation du langage C++

Nous vous proposons ici d'examiner les caractéristiques essentielles de C++. Pour vous permettre de mieux les appréhender, nous vous fournissons de brefs rappels concernant les concepts de la programmation structurée (ou procédurale) et de la P.O.O.[1] Auparavant, nous commencerons par un bref historique du langage.

1 Historique du langage

Très tôt, les concepts de la programmation orientée objet (en abrégé P.O.O.) ont donné naissance à de nouveaux langages dits « orientés objet » tels que Smalltalk, Simula, Eiffel ou, plus récemment, Java, PHP ou Python. Le langage C++, quant à lui, a été conçu suivant une démarche hybride. En effet, Bjarne Stroustrup, son créateur, a cherché à adjoindre à un langage structuré existant (le C), un certain nombre de spécificités lui permettant d'appliquer les concepts de P.O.O. Dans une certaine mesure, il a permis à des programmeurs C d'effectuer une transition en douceur de la programmation structurée vers la P.O.O. De sa conception jusqu'à sa normalisation, le langage C++ a quelque peu évolué. Initialement, un certain nombre de publications de AT&T ont servi de référence au langage, notamment la version 2.0 en 1989 et les versions 2.1 et 3 en 1991. C'est cette dernière version qui a servi de base au travail du comité ANSI qui, sans la remettre en cause, l'a enrichie de quelques extensions et surtout de composants standards originaux se présentant sous forme de fonctions et de classes génériques qu'on désigne souvent par le sigle S.T.L. (*Standard Template Library*). Une pre-

1. Rappelons que l'ouvrage s'adresse à un public déjà familiarisé avec un langage procédural classique.

mière norme de C++ a été publiée par l'ANSI en juillet 1998. Quelques modifications mineures ont été apportées en 2003, mais elles ne concernent pas l'utilisateur. En revanche, un nouveau standard, dit C++11 (l'ancien standard étant noté indifféremment C++98 ou C++03), publié en 2011 a enrichi notablement le langage, à tel point que, depuis cette date, on s'est mis à parler de « C++ moderne ». De nouvelles versions ont continué d'apparaître à un rythme régulier : 2014 (C++14) et 2017 (C++17) et 2019 (C++20).

On notera cependant que, depuis C++98, les aspects fondamentaux du langage qui font l'objet de ce chapitre ont en fait peu changé.

2 Programmation structurée et programmation orientée objet

2.1 Problématique de la programmation

Jusqu'à maintenant, l'activité de programmation a toujours suscité des réactions diverses allant jusqu'à la contradiction totale. Pour certains, en effet, il ne s'agit que d'un jeu de construction enfantin, dans lequel il suffit d'enchaîner des instructions élémentaires (en nombre restreint) pour parvenir à résoudre n'importe quel problème ou presque. Pour d'autres, au contraire, il s'agit de produire (au sens industriel du terme) des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères, notamment :

- *l'exactitude* : aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux spécifications) ;
- *la robustesse* : aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation ;
- *l'extensibilité* : facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications ;
- *la réutilisabilité* : possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;
- *la portabilité* : facilité avec laquelle on peut exploiter un même logiciel dans différentes implémentations ;
- *l'efficience* : temps d'exécution, taille mémoire...

La contradiction n'est souvent qu'apparente et essentiellement liée à l'importance des projets concernés. Par exemple, il est facile d'écrire un programme exact et robuste lorsqu'il comporte une centaine d'instructions ; il en va tout autrement lorsqu'il s'agit d'un projet de dix hommes-années ! De même, les aspects extensibilité et réutilisabilité n'auront guère d'importance dans le premier cas, alors qu'ils seront probablement cruciaux dans le second, ne serait-ce que pour des raisons économiques.

2.2 La programmation structurée

En programmation structurée (ou procédurale), un programme est formé de la réunion de différentes procédures et de différentes structures de données, généralement indépendantes de ces procédures. D'autre part, les procédures utilisent un certain nombre de structures de contrôle bien définies (on parle parfois de « programmation sans *go to* »).

La programmation structurée a manifestement fait progresser la qualité de la production des logiciels. Notamment, elle a permis de structurer les programmes, et, partant, d'en améliorer l'exactitude et la robustesse. On avait espéré qu'elle permettrait également d'en améliorer l'extensibilité et la réutilisabilité. Or, en pratique, on s'est aperçu que l'adaptation ou la réutilisation d'un logiciel conduisait souvent à « casser » le module intéressant, et ceci parce qu'il était nécessaire de remettre en cause une structure de données. Or, ce type de difficulté apparaît précisément à cause du découplage existant entre les données et les procédures, lequel se trouve résumé par ce que l'on nomme « l'équation de Wirth » :

Programmes = algorithmes + structures de données

2.3 Les apports de la programmation orientée objet

2.3.1 Objet

C'est là qu'intervient la programmation orientée objet (en abrégé P.O.O), fondée justement sur le concept d'**objet**, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données. Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la P.O.O. est :

Méthodes + Données = Objet

2.3.2 Encapsulation

Mais cette association est plus qu'une simple juxtaposition. En effet, dans ce que l'on pourrait qualifier de P.O.O. « pure »[2], on réalise ce que l'on nomme une **encapsulation des données**. Cela signifie qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes, qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un « message » à l'objet.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications[3] de ses méthodes, la manière dont sont réellement implantées les

2. Nous verrons en effet que les concepts de la P.O.O. peuvent être appliqués d'une manière plus ou moins rigoureuse. En particulier, en C++, l'encapsulation ne sera pas obligatoire, ce qui ne veut pas dire qu'elle ne soit pas souhaitable.

données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une « abstraction des données » (ce qui exprime bien que les détails concrets d'implémentation sont cachés). À ce propos, on peut remarquer qu'en programmation structurée, une procédure pouvait également être caractérisée (de l'extérieur) par ses spécifications, mais que, faute d'encapsulation, l'abstraction des données n'était pas réalisée.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel. Elle facilite considérablement la maintenance : une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était bien sûr pas le cas avec la programmation structurée). De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

2.3.3 Classe

En P.O.O. apparaît généralement le concept de classe, qui correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune[4] et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (on dit aussi qu'un objet est une « instance » de sa classe).

2.3.4 Héritage

Un autre concept important en P.O.O. est celui d'héritage. Il permet de définir une nouvelle classe à partir d'une classe existante (qu'on réutilise en bloc !), à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La conception de la nouvelle classe, qui « hérite » des propriétés et des aptitudes de l'ancienne, peut ainsi s'appuyer sur des réalisations antérieures parfaitement au point et les « spécialiser » à volonté. Comme on peut s'en douter, l'héritage facilite largement la réutilisation de produits existants, d'autant plus qu'il peut être réitéré autant de fois que nécessaire (la classe C peut hériter de B, qui elle-même hérite de A).

2.3.5 Polymorphisme

Généralement, en P.O.O, une classe dérivée peut « redéfinir » (c'est-à-dire modifier) certaines des méthodes héritées de sa classe de base. Cette possibilité est la clé de ce que l'on nomme le polymorphisme, c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient tous de classes dérivées de la même classe de base. Plus précisément, on utilise chaque objet comme s'il était de cette classe de base, mais son comportement effectif dépend de sa classe effective (dérivée de cette classe de base), en particulier de la manière dont ses propres méthodes ont été redéfinies. Le polymorphisme améliore l'extensibilité des programmes, en permettant d'ajouter de nouveaux objets dans un

3. Rôles, noms, types des arguments et de la valeur de retour. On parle aussi d'interface.

4. Bien entendu, seule la structure est commune, les données étant propres à chaque objet. En revanche, les méthodes sont effectivement communes à l'ensemble des objets d'une même classe.

scénario préétabli et, éventuellement, écrit avant d'avoir connaissance du type effectif de ces objets.

2.4 P.O.O., langages de programmation et C++

Nous venons d'énoncer les grands principes de la P.O.O. sans nous attacher à un langage particulier.

Or manifestement, certains langages peuvent être conçus (de toutes pièces) pour appliquer à la lettre ces principes et réaliser ce que nous nommons de la P.O.O. « pure ». C'est par exemple le cas de Simula, Smalltalk ou, plus récemment, Eiffel ou Java. Le même phénomène a eu lieu, en son temps, pour la programmation structurée avec Pascal.

À l'opposé, on peut toujours tenter d'appliquer, avec plus ou moins de bonheur, ce que nous aurions tendance à nommer « une philosophie P.O.O. » à un langage classique (Pascal, C...). On retrouve là une idée comparable à celle qui consistait à appliquer les principes de la programmation structurée à des langages comme Fortran ou Basic.

Le langage C++ se situe à mi-chemin entre ces deux points de vue. Il a en effet été obtenu en **ajoutant** à un langage procédural répandu (C) les outils permettant de mettre en œuvre tous les principes de la P.O.O.. Programmer en C++ va donc plus loin qu'adopter une philosophie P.O.O. en C, mais moins loin que de faire de la P.O.O. pure avec Eiffel ! D'ailleurs, son concepteur Stroustrup a dit lui-même que considérer C++ comme un pur langage objet revient à se priver de certaines de ses fonctionnalités.

À l'époque où elle est apparue, la solution adoptée par B. Stroustrup avait le mérite de préserver l'existant, grâce à la quasi-compatibilité avec C++, de programmes déjà écrits en C. Elle permettait également une « transition en douceur » de la programmation structurée vers la P.O.O.. Malheureusement, la contrepartie de cette souplesse est que la qualité des programmes écrits en C++ dépendra étroitement des décisions du développeur. Par exemple, il restera tout à fait possible de faire cohabiter des objets (dignes de ce nom, parce que réalisant une parfaite encapsulation de leurs données) avec des fonctions classiques réalisant des effets de bord sur des variables globales... Quoi qu'il en soit, il ne faudra pas perdre de vue que, de par la nature même du langage, on ne pourra exploiter toute la richesse de C++ qu'en se plaçant dans un contexte hybride mêlant programmation procédurale (notamment des fonctions « usuelles ») et P.O.O.. Ce n'est que par une bonne maîtrise du langage que le programmeur pourra réaliser du code de bonne qualité.

3 C++ et la programmation structurée

Les possibilités de programmation structurée de C++ sont pratiquement celles du langagre C et elles sont également assez proches de celles de nombreux langages actuels (Java, C#, PHP, Python...), à l'exception des pointeurs.

En ce qui concerne les types de base des données, on trouvera :

- les types numériques usuels : entiers avec différentes capacités, flottants avec différentes capacités et précisions ;
- le type caractère ;

Par ailleurs, on trouvera les agrégats de données que sont :

- les chaînes de caractères représentées par le type *string* (il s'agit en fait d'objets) ; toutefois, nous verrons qu'il subsiste encore des traces de la convention de représentation des chaînes héritée du langage C ;
- les tableaux : ensembles d'éléments de même type, de taille fixée à la compilation ; là encore, il s'agit d'un héritage « historique » du langage C et, souvent, ces tableaux (que nous qualifierons de « natifs » seront avantageusement remplacés par le type *vector* introduit dans la bilbliothèque standard dès C++98.
- les structures : ensembles d'éléments de types quelconques ; il s'agit ici aussi d'un héritage du C et nous verrons que ces structures se généralisent en C++ au point de devenir un cas particulier des classes.

Les opérateurs de C++ sont très nombreux. En plus des opérateurs arithmétiques (+, -, *, /) et logiques (et, ou, non), on trouvera notamment des opérateurs d'affectation originaux permettant de simplifier en x += y des affectations de la forme $x = x + y$ (on notera qu'en C++, l'affectation est un opérateur, pas une instruction !).

Les structures de contrôle comprennent :

- la structure de choix : instruction *if* ;
- la structure de choix multiple : instruction *switch* ;
- les structures de boucle de type « tant que » et « jusqu'à » : instructions *do... while* et *while* ;
- une structure très générale permettant de programmer, entre autres, une « boucle avec compteur » : instruction *for* ; depuis C++11, on trouvera également une boucle adaptée aux « séquences ».

Les pointeurs sont assez spécifiques à C++ (et à C). Assez curieusement, on verra qu'ils sont également liés aux tableaux et à la convention de représentation des chaînes du C. Ces aspects sont en fait inhérents à l'historique du langage, dont les germes remontent finalement aux années 1980 : à l'époque, on cherchait plus à simplifier l'écriture des compilateurs du langage qu'à sécuriser les programmes ! Nous verrons que C++11 a introduit les « pointeurs intelligents » qui, dans certains cas, pourront se substituer avantageusement aux pointeurs historiques (que nous nommerons « pointeurs natifs »).

La notion de procédure se retrouvera en C++ dans la notion de fonction. La transmissions des arguments pourra s'y faire, au choix du programmeur : par valeur, par référence (ce qui n'était pas possible en C) ou encore par le biais de manipulation de pointeurs. On notera que ces fonctions sont définies indépendamment de toute classe ; on les nommera souvent des « fonctions ordinaires », par opposition aux méthodes des classes.

4 C++ et la programmation orientée objet

Les possibilités de P.O.O. représentent bien sûr l'essentiel de l'apport de C++ au langage C.

C++ dispose de la notion de classe (généralisation de la notion de type défini par l'utilisateur). Une classe comportera :

- la description d'une structure de données ;
- des méthodes.

Sur le plan du vocabulaire, C++ utilise des termes qui lui sont propres. On parle en effet de :

- « membres données » pour désigner les différents membres de la structure de données associée à une classe ;
- « fonctions membres » pour désigner les méthodes.

À partir d'une classe, on pourra « instancier » des objets (nous dirons aussi créer des objets) de deux façons différentes :

- soit par des déclarations usuelles, les emplacements étant alors gérés automatiquement sous forme de ce que l'on nomme une « pile » ;
- soit par allocation dynamique dans ce que l'on nomme un « tas », les emplacements étant alors gérés par le programmeur lui-même.

C++ permet l'encapsulation des données, mais il ne l'impose pas. On peut le regretter mais il ne faut pas perdre de vue que, par sa conception même (extension de C), le C++ ne peut pas être un langage de P.O.O. pure. Bien entendu, il reste toujours possible au concepteur de faire preuve de rigueur, en s'astreignant à certaines règles telles que l'encapsulation absolue.

Comme la plupart des langages objets, C++ permet de définir ce que l'on nomme des « constructeurs » de classe. Un constructeur est une fonction membre particulière qui est exécutée au moment de la création d'un objet de la classe. Le constructeur peut notamment prendre en charge l'initialisation d'un objet, au sens le plus large du terme, c'est-à-dire sa mise dans un état initial permettant son bon fonctionnement ultérieur ; il peut s'agir de banales initialisations de membres données, mais également d'une préparation plus élaborée correspondant au déroulement d'instructions, voire d'une allocation dynamique d'emplacements nécessaires à l'utilisation de l'objet. L'existence d'un constructeur garantit que l'objet sera toujours initialisé, ce qui constitue manifestement une sécurité.

De manière similaire, une classe peut disposer d'un « destructeur », fonction membre exécutée au moment de la destruction d'un objet. Celle-ci présentera surtout un intérêt dans le cas d'objets effectuant une « allocation de ressources » (allocation dynamique d'emplacements mémoire, ouverture de fichier, établissement d'une connexion...) ; ces ressources pourront alors être libérés par le destructeur.

Une des originalités de C++ par rapport à d'autres langages de P.O.O. réside dans la possibilité de définir des « fonctions amies d'une classe ». Il s'agit, soit de fonctions usuelles, soit de fonctions membres qui sont autorisées (par une classe) à accéder aux données (encapsulées)

de la classe. Certes, le principe d'encapsulation est violé, mais uniquement par des fonctions dûment autorisées à le faire.

La notion de « surdéfinition d'opérateurs » va permettre de doter une classe d'opérations analogues à celles que l'on rencontre pour les types prédéfinis. Par exemple, on pourra définir une classe *complexe* (destinée à représenter des nombres complexes) et la munir des opérations d'addition, de soustraction, de multiplication et de division. Qui plus est, ces opérations pourront utiliser les symboles existants : +, -, *, /. On verra que, dans certains cas, cette surdéfinition nécessitera le recours à la notion de fonction amie.

Le langage C disposait déjà de possibilités de conversions explicites ou implicites. C++ permet de les élargir aux types définis par l'utilisateur que sont les classes. Par exemple, on pourra donner un sens à la conversion *int -> complexe* ou à la conversion *complexe -> float* (*complexe* étant une classe).

Naturellement, C++ dispose de l'héritage et même (ce qui est peu commun) de possibilités dites « d'héritage multiple » permettant à une classe d'hériter simultanément de plusieurs autres. Le polymorphisme est mis en place, sur la demande explicite du programmeur, par le biais de ce que l'on nomme (curieusement) des fonctions virtuelles (en Java, le polymorphisme est « natif » et le programmeur n'a donc pas en s'en préoccuper).

Les entrées-sorties de C++ sont différentes de celles du C, car elle reposent sur la notion de « flots » (classes particulières), ce qui permet notamment de leur donner un sens pour les types définis par l'utilisateur que sont les classes (grâce au mécanisme de surdéfinition d'opérateur).

Avec sa normalisation, le C++ a été doté de la notion de patron (*template* en anglais). Un patron permet de définir des modèles paramétrables par des types, et utilisables pour générer différentes classes ou différentes fonctions qualifiées parfois de génériques, même si cette généricité n'est pas totalement intégrée dans le langage lui-même, comme c'était par exemple le cas avec ADA.

5 C et C++

Précédemment, nous avons dit, d'une façon quelque peu simpliste, que C++ se présentait comme un « sur-ensemble » du langage C[5], offrant des possibilités de P.O.O.

En toute rigueur, certaines des extensions du C++ ne sont pas liées à la P.O.O ; on peut citer : la notion de référence, la surdéfinition des fonctions, les expressions lambdas (C++11), les déclarations automatiques (C++11)... Elles pourraient en fait être ajoutées au langage C, sans qu'il soit pour autant « orienté objet ». Ici, nous étudierons directement le C++, de sorte que ces extensions non P.O.O. seront tout naturellement présentées au fil des prochains chapitres.

5. Pour être précis, il faudrait dire qu'il s'agit d'un surensemble de C99 jusqu'à C++14 et de C11 depuis C++17.

Par ailleurs, certaines possibilités du C deviennent inutiles (ou redondantes) en C++. Par exemple, C++ a introduit de nouvelles possibilités d'entrées-sorties (basées sur la notion de flot) qui rendent superflues les fonctions standards de C telles que *printf* ou *scanf*. Ou encore, C++ dispose de méthodes de gestion dynamique qui remplacent avantageusement les fonctions *malloc*, *calloc* et *free* du C : il s'agit des opérateurs *new* et *delete* du C++98, avantageusement complétés depuis C++11 par les pointeurs intelligents et les fonctions *make_unique* et *make_shared*.

Comme ici, nous étudions directement le langage C++, il va de soi que ces « possibilités inutiles » du C ne seront pas étudiées en détail. Nous nous contenterons de les mentionner à simple titre informatif, dans des remarques titrées « En C ».

Par ailleurs, il existe quelques incompatibilités mineures entre C et C++. Là encore, elles ne poseront aucun problème à qui ne connaît pas le C. À titre d'information, elles seront récapitulées en Annexe E.

6 C++ et les bibliothèques standards

Comme tout langage, C++ dispose (depuis sa normalisation C++98) d'une bibliothèque standard, c'est-à-dire de fonctions et de classes prédéfinies. Elle comporte notamment de nombreux patrons de classes et de fonctions permettant de mettre en œuvre :

- les structures de données les plus importantes (vecteurs dynamiques, listes chaînées, chaînes...) et les algorithmes les plus usuels ;
- les entrées-sorties (flots) ;
- la gestion des exceptions.

Tous ces composants seront naturellement étudiés en détail le moment venu, au même titre que le langage de base.

En outre, C++ dispose de la totalité de la bibliothèque standard du C, y compris de fonctions devenues inutiles ou redondantes. Bien entendu, là encore, les fonctions indispensables seront introduites au fil des différents chapitres. Vous trouverez sur le site de l'éditeur *www.editions-eyrolles.com* un complément (Annexe G) venant récapituler les principales fonctions héritées de C[6].

En fait, la distinction entre langage de base et bibliothèque standard reste assez peu formelle. Simplement, comme nous le verrons, l'emploi de composants d'une bibliothèque nécessitera des instructions d'accès particulières (*#include*).

6. Vous en trouverez une descrition exhaustive dans l'ouvrage *Le guide complet du langage C* du même auteur, chez le même éditeur.

7 A propos du C++ moderne

Les fondamentaux de la programmation structurée et la P.O.O. en C++ n'ont pratiquement pas été impactés par les récentes versions. Il n'en reste pas moins que ces dernières ont assez profondémment modifié la manière de coder.

Ainsi l'arrivée des pointeurs intelligents aide à sécuriser les opérations de gestion dynamique, en prenant automatiquement en charge les allocations et libération de mémoire, supprimant du même coup les risques de double libération ou de fuite de mémoire.

Les déclarations automatiques (auto) permettent de simplifier les déclarations parfois complexes, notamment celles induites par l'utilisation des conteneurs. Comme nous le verrons, il existera cependant quelques situations où leur usage n'est pas possible ou ne fournit pas ce que l'on attend.

La boucle *for* pour les séquences facilite l'écriture d'une boucle portant sur l'ensemble des éléments de ce que l'on nomme une « séquence », sans avoir à recourir à l'emploi d'un compteur ; l'exemple typique est celui de l'ensemble des éléments d'un vecteur.

Les expressions lambda simplifient la rédaction de ce que l'on nomme des « fonctions de rappel », en permettant d'en fournir directement le code dans l'appel ; elles s'avéreront très précieuses lorsqu'il s'agira de fournir à un algorithme une fonction représentant un prédicat.

La sémantique dite de « déplacement » sert à optimiser les opérations de copie et d'affectation d'objets. Elle se fonde sur la référence à une *rvalue* qui désignera en fait une référence à quelque chose de temporaire. Sa prise en compte dans les patrons conduira à la définition de « références universelles ».

De nouvelles formes d'initialisation ont été introduites, dans un souci d'universalité, lequel au demeurant n'est pas entièrement atteint puisqu'aucune forme (ancienne ou nouvelle) n'est utilisable de façon systématique. Notamment, l'existence d'un type *initializer_list* destiné à représenter des séquences de valeurs va créer des ambiguïtés. En définitive cette extension apparaîtra autant comme une complexification (il faut connnaître les différentes formes) que comme une simplification.

Naturellement, toutes ces nouveautés seront étudiées au fur et à mesure des besoins, sans faire l'impasse sur les difficultés qu'elles peuvent induire.

2

Généralités sur le langage C++

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C++, basée sur deux exemples commentés. Vous y découvrirez, de manière encore informelle pour l'instant, comment s'expriment certaines instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux structures de contrôle (boucle avec compteur, choix).

Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont l'édition, la compilation, l'édition de liens et l'exécution.

Notez bien que le principal objectif de ce chapitre est de vous permettre de lire et d'écrire d'emblée des programmes complets, quitte à ce que l'exposé détaillé de certaines notions soit différé. Nous nous sommes donc limités à ce qui s'avère indispensable pour l'étude de la suite de l'ouvrage et, donc, en particulier, à des aspects de programmation procédurale. Autrement dit, aucun aspect P.O.O. ne sera abordé ici et vous ne trouverez donc aucune classe dans nos exemples.

1 Présentation par l'exemple de quelques instructions du langage C++

1.1 Un exemple de programme en langage C++

Voici un exemple de programme en langage C++, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonctionnement.

```
#include <iostream>
#include <cmath>
using namespace std ;
int main()
{  int i ;
   float x ;
   float racx ;
   const int NFOIS = 5 ;
   cout << "Bonjour\n" ;
   cout << "Je vais vous calculer " << NFOIS << " racines carrees\n" ;
   for (i=0 ; i<NFOIS ; i++)
   { cout << "Donnez un nombre : " ;
     cin >> x ;
     if  (x < 0.0)
        cout << "Le nombre " << x << "ne possede pas de racine carree\n " ;
      else
        { racx = sqrt (x) ;
          cout << "Le nombre " << x << " a pour racine carree : " << racx << "\n" ;
        }
   }
   cout << "Travail termine - au revoir " ;
}
```

```
Bonjour
Je vais vous calculer 5 racines carrees
Donnez un nombre : 8
Le nombre 8 a pour racine carree : 2.82843
Donnez un nombre : 4
Le nombre 4 a pour racine carree : 2
Donnez un nombre : 0.25
Le nombre 0.25 a pour racine carree : 0.5
Donnez un nombre : 3.4
Le nombre 3.4 a pour racine carree : 1.84391
Donnez un nombre : 2
Le nombre 2 a pour racine carree : 1.41421
Travail termine - au revoir
```

Premier exemple de programme C++

1.2 Structure d'un programme en langage C++

Nous reviendrons un peu plus loin sur le rôle des trois premières lignes.

La ligne :

```
int main()
```

se nomme un « en-tête ». Elle précise que ce qui sera décrit à sa suite est en fait le *programme principal* (*main*). Lorsque nous aborderons l'écriture des fonctions en C++, nous verrons que celles-ci possèdent également un tel en-tête ; ainsi, en C++, le programme principal apparaîtra en fait comme une fonction dont le nom (*main*) est imposé et nous verrons plus précisément la signification du mot *int* qui le précède.

Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades « *{* » et « *}* ». On dit que les instructions situées entre ces accolades forment un « bloc ». Ainsi peut-on dire que la fonction *main* est constituée d'un en-tête et d'un bloc ; il en ira de même pour toute fonction C++. Notez qu'un bloc peut lui-même contenir d'autres blocs (c'est le cas de notre exemple). En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions.

1.3 Déclarations

Les quatre instructions :

```
int i ;
float x ;
float racx ;
const int NFOIS = 5 ;
```

sont des « déclarations ».

La première précise que la variable nommée *i* est de type *int*, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C++ il existe plusieurs types d'entiers.

Les deux autres déclarations précisent que les variables *x* et *racx* sont de type *float*, c'est-à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C++ il existe plusieurs types flottants.

Enfin, la quatrième déclaration indique que *NFOIS* est une constante de type entier, ayant la valeur 5. Contrairement à une variable, la valeur d'une constante ne peut pas être modifiée. Elle doit donc obligatoirement être initialisée (c'est-à-dire recevoir une valeur) au moment de sa déclaration.

En C++, comme dans beaucoup de langages actuels, les déclarations des types des variables sont obligatoires. Ici, nous les avons regroupées au début du programme (on devrait plutôt dire : au début de la fonction *main*) mais ce n'est pas une obligation : il est seulement nécessaire qu'une variable soit **déclarée avant d'être utilisée**. Les mêmes considérations vaudront pour toutes les variables définies dans une fonction ; on les appelle « variables locales » (en toute rigueur, les variables définies dans notre exemple sont des variables locales de la fonction *main*). Nous verrons également (dans le chapitre consacré aux fonctions) qu'on peut définir des variables en dehors de toute fonction : on parlera alors de variables globales.

1.4 Pour écrire des informations : utiliser le flot *cout*

L'interprétation détaillée de l'instruction :

```
cout << "Bonjour\n" ;
```

nécessiterait des connaissances qui ne seront introduites qu'ultérieurement : nous verrons que *cout* est un « flot de sortie » et que << est un opérateur permettant d'envoyer de l'information sur un flot de sortie. Pour l'instant, admettons que *cout* désigne la fenêtre dans laquelle s'affichent les résultats. Ici, donc, cette instruction peut être interprétée ainsi : *cout* reçoit l'information :

```
"Bonjour\n"
```

Les guillemets servent à délimiter une « chaîne de caractères » (suite de caractères). La notation *n* est conventionnelle : elle représente un caractère de fin de ligne, c'est-à-dire un caractère qui, lorsqu'il est envoyé à l'écran, provoque le passage à la ligne suivante. Nous verrons que, de manière générale, C++ prévoit une notation de ce type (\ suivi d'un caractère) pour un certain nombre de caractères dits « de contrôle », c'est-à-dire ne possédant pas de graphisme particulier.

L'instruction suivante :

```
cout << "Je vais vous calculer " << NFOIS << " racines carrees\n" ;
```

ressemble à la précédente avec cette différence qu'ici on envoie trois informations différentes à l'écran :

- l'information *" Je vais vous calculer"* ;
- l'information *NFOIS*, c'est-à-dire en fait la valeur de cette constante, à savoir 5 ;
- l'information *" racines carrees\n"* .

1.5 Pour faire une répétition : l'instruction *for*

Comme nous le verrons, en C++, il existe plusieurs façons de réaliser une répétition (on dit aussi une « boucle »). Ici, nous avons utilisé l'instruction *for* :

```
for (i=0 ; i<NFOIS ; i++)
```

Son rôle est de répéter le bloc (délimité par des accolades « *{* » et « *}* ») figurant à sa suite, en respectant les consignes suivantes :

- avant de commencer cette répétition, réaliser :

  ```
  i = 0
  ```

- avant chaque nouvelle exécution du bloc (tour de boucle), examiner la condition :

  ```
  i < NFOIS
  ```

  si elle est satisfaite, exécuter le bloc indiqué, sinon passer à l'instruction suivant ce bloc ;

- à la fin de chaque exécution du bloc, réaliser :

  ```
  i++
  ```

Il s'agit là d'une notation propre au C++ qui est équivalente à :

```
i = i + 1
```

En définitive, vous voyez qu'ici notre bloc sera répété cinq fois.

1.6 Pour lire des informations : utiliser le flot *cin*

La première instruction du bloc répété par l'instruction *for* affiche simplement le message *Donnez un nombre :*. Notez qu'ici nous n'avons pas prévu de changement de ligne à la fin. Là encore, l'interprétation détaillée de la seconde instruction du bloc :

```
cin >> x ;
```

nécessiterait des connaissances qui ne seront introduies qu'ultérieurement : nous verrons que *cin* est un "flot d'entrée" associé au clavier et que << est un opérateur permettant d'"extraire" (de lire) de l'information à partir d'un flot d'entrée. Pour l'instant, admettons que cette instruction peut être interprétée ainsi : lire une suite de caractères au clavier et la convertir en une valeur de type *float* que l'on place dans la variable *x*. Ici, nous supposerons que l'utilisateur « valide » son entrée au clavier. Plus tard, nous verrons qu'il peut fournir plusieurs informatins par anticipation. De même, nous supposerons pour l'instant qu'il ne fait pas de « faute de frappe ».

1.7 Pour faire des choix : l'instruction *if*

Les lignes :

```
if  (x < 0.0)
    cout << "Le nombre " << x << "ne possede pas de racine carree\n " ;
 else
    { racx = sqrt (x) ;
      cout << "Le nombre " << x << " a pour racine carree : " << racx << "\n" ;
    }
```

constituent une instruction de choix basée sur la condition $x < 0.0$. Si cette condition est vraie, on exécute l'instruction suivante, c'est-à-dire :

```
cout << "Le nombre " << x << "ne possede pas de racine carree\n " ;
```

Si elle est fausse, on exécute l'instruction suivant le mot *else*, c'est-à-dire, ici, le bloc :

```
{ racx = sqrt (x) ;
  cout << "Le nombre " << x << " a pour racine carree : " << racx << "\n" ;
}
```

Notez qu'il existe un mot *else* mais pas de mot *then*. La syntaxe de l'instruction *if* (notamment grâce à la présence de parenthèses qui encadrent la condition) le rend inutile.

La fonction *sqrt* fournit la valeur de la racine carrée d'une valeur flottante qu'on lui transmet en argument.

Remarques

1 Une instruction telle que :

```
racx = sqrt (x) ;
```

est une instruction classique d'affectation : elle donne à la variable *racx* la valeur de l'expression située à droite du signe égal. Nous verrons plus tard qu'en C++ l'affectation peut prendre des formes plus élaborées.

2 D'une manière générale, C++ dispose de trois sortes d'instructions :

– des instructions simples, terminées obligatoirement par un point-virgule,

– des instructions de structuration telles que *if* ou *for*,

– des blocs (délimités par { et }).

Les deux dernières ont une définition « récursive » puisqu'elles peuvent contenir, à leur tour, n'importe laquelle des trois formes.

Lorsque nous parlerons d'instruction, sans précisions supplémentaires, il pourra s'agir de n'importe laquelle des trois formes ci-dessus.

1.8 Les directives à destination du préprocesseur

Les deux premières lignes de notre programme :

```
#include <iostream>
#include <cmath>
```

sont un peu particulières. Il s'agit de directives qui seront prises en compte avant la traduction (compilation) du programme, par un programme nommé « préprocesseur » " (parfois « précompilateur »). Ces directives, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne. Leur emplacement au sein du programme n'est soumis à aucune contrainte (mais une directive ne s'applique qu'à la partie du programme qui lui succède). D'une manière générale, il est préférable de les placer au début, avant toute fonction, comme nous l'avons fait ici.

Ces deux directives demandent en fait d'introduire (avant compilation) des instructions (en C++) situées dans les fichiers *iostream* et *cmath*. Leur rôle ne sera complètement compréhensible qu'ultérieurement.

Pour l'instant, notez que :

- *iostream* contient des déclarations relatives aux flots donc, en particulier, à *cin* et *cout*, ainsi qu'aux opérateurs << et >> (dont on verra plus tard qu'ils sont en fait considérés comme des fonctions particulières) ;
- *cmath* contient des déclarations relatives aux fonctions mathématiques (héritées de C), donc en particulièr à *sqrt*.

D'une manière générale, dès que vous utilisez une fonction dans une partie d'un programme, il est nécessaire qu'elle ait été préalablement déclarée. Cela vaut également pour les fonctions prédéfinies. Plutôt que de s'interroger sur les déclarations exactes de ces fonctions prédéfinies, il est préférable d'incorporer les fichiers en-têtes correspondants.

Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. Généralement, vous ne les utiliserez pas toutes dans un programme donné ; cela n'est guère gènant, dans la mesure où les déclarations ne produisent pas de code exécutable.

1.9 L'instruction *using*

La norme de C++ a introduit la notion d'« espaces de noms » (*namespace*). Elle permet de restreindre la « portée » des symboles à une certaine partie d'un programme et donc, en particulier, de règler les problèmes qui peuvent se poser quand plusieurs bibliothèques utilisent les mêmes noms. Cette notion d'espace de noms sera étudiée par la suite. Pour l'instant, retenez que les symboles déclarés dans le fichier *iostream* appartiennent, par défaut, à l'espace de noms *std*. L'instruction *using* sert précisément à indiquer que l'on se place "dans cet espace de noms *std*" (attention, si vous placez l'instruction *using* avant l'incorporation des fichiers en-tête, vous obtiendrez une erreur car vous ferez référence à un espace de noms qui n'a pas encore été défini !). En toute rigueur, nous pourrions nous passer de cette instruction, à condition de « préfixer » tous les symboles concernés par *std::*, par exemple *std::cout* ou *std::cin*, ce qui, à notre sens, rend les programmes moins lisibles.

1.10 Exemple de programme utilisant le type caractère

Voici un second exemple de programme, accompagné de deux exemples d'exécution, destiné à vous montrer l'utilisation du type « caractère ». Il demande à l'utilisateur de choisir une opération parmi l'addition ou la multiplication, puis de fournir deux nombres entiers ; il affiche alors le résultat correspondant.

```
#include <iostream>
using namespace std ;
int main()
{  char op ;
   int n1, n2 ;
   cout << "operation souhaitee (+ ou *) ? " ;
   cin >> op ;
   cout << "donnez 2 nombres entiers : " ;
   cin >> n1 >> n2 ;
   if (op == '+') cout << "leur somme  est : " << n1+n2 << "\n" ;
           else  cout << "leur produit est : " << n1*n2 << "\n" ;
}
```

```
operation souhaitee (+ ou *) ? +
donnez 2 nombres entiers : 25 13
leur somme  est : 38
```

```
operation souhaitee (+ ou *) ? *
donnez 2 nombres entiers : 12 5
leur produit est : 60
```

Utilisation du type char

Ici, nous déclarons que la variable *op* est de type caractère (*char*). Une telle variable est destinée à contenir un caractère quelconque (codé, bien sûr, sous forme binaire !).

L'instruction *cin >> op* permet de lire un caractère au clavier et de le ranger dans *op*. L'instruction *if* permet d'afficher la somme ou le produit de deux nombres, suivant le caractère contenu dans *op*. Notez que :

- la relation d'égalité se traduit par le signe == (et non = qui représente l'affectation et qui, ici, comme nous le verrons plus tard, serait admis mais avec une autre signification !).
- la notation '+' représente une constante caractère. Notez bien que C++ n'utilise pas les mêmes délimiteurs pour les constantes chaînes (il s'agit de ") et pour les constantes caractères.

Remarquez que, tel qu'il a été écrit, notre programme calcule le produit, dès lors que le caractère fourni par l'utilisateur n'est pas +.

2 Quelques règles d'écriture

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme en C++. Nous y parlerons précisément de ce que l'on appelle les « identificateurs » et les « mots-clés », du format libre dans lequel on écrit les instructions, ainsi que de l'usage des séparateurs et des commentaires.

2.1 Les identificateurs

Les identificateurs servent à désigner les différentes « choses »[1] manipulées par le programme, telles les variables et les fonctions (nous rencontrerons ultérieurement les autres choses manipulés par le C++ : objets, structures, unions ou énumérations, membres de classe, de structure ou d'union, types, étiquettes d'instruction *goto*, macros). Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les **lettres** ou les **chiffres**, le premier d'entre eux étant nécessairement une lettre.

En ce qui concerne les lettres :

- le caractère souligné (_) est considéré comme une lettre. Il peut donc apparaître au début d'un identificateur. Voici quelques identificateurs corrects :

1. En dehors d'un contexte de P.O.O, nous aurions pu parler des « objets » manipulés par un programme. Il est clair, qu'ici, ce terme devient trop restictif. Nous aurions pu utiliser le terme « entité » à la place de « chose ».

```
lg_lig    valeur_5   _total   _89
```

- les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Ainsi, en C++, les identificateurs *ligne* et *Ligne* désignent deux choses différentes.

Aucune restriction ne pèse sur la longueur des identificateurs (en C, seuls les 31 premiers caratères étaient significatifs).

2.2 Les mots-clés

Certains « mots-clés » sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. Vous en trouverez la liste complète, classée par ordre alphabétique, en Annexe E.

2.3 Les séparateurs

Dans NOTRE langue écrite, les différents mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne.

Il en va quasiment de même en C++ dans lequel les règles vont donc paraître naturelles. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier (tel que : *, = ; * ( ) [ ] { }*) doivent impérativement être séparés soit par un espace, soit par une fin de ligne. En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme).

Ainsi, vous devrez impérativement écrire :

```
int x,y
```

et non :

```
intx,y
```

En revanche, vous pourrez écrire indifféremment :

```
int n,compte,total,p
```

ou plus lisiblement :

```
int n, compte, total, p
```

2.4 Le format libre

Comme tous les langages récents, le C++ autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites ; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment, le risque existe, si l'on n'y prend garde, d'aboutir à des programmes peu lisibles.

À titre d'exemple, voyez comment pourrait être (mal) présenté notre programme précédent :

```
#include <iostream>
#include <cmath>
using namespace std ; int main() {  int i ;  float
      x ;   float racx ;   const
          int NFOIS
      = 5 ;  cout << "Bonjour\n" ;  cout
  << "Je vais vous calculer " << NFOIS << " racines carrees\n" ;  for (i=0 ;
  i<NFOIS ; i++) { cout << "Donnez un nombre : " ; cin >> x
  ; if  (x < 0.0) cout << "Le nombre "
    << x << "ne possede pas de racine carree\n " ; else { racx = sqrt
    (x) ; cout << "Le nombre " << x << " a pour racine carree : " << racx <<
    "\n" ;    }  }  cout << "Travail termine - au revoir " ; }
```

Exemple de programme mal présenté

2.5 Les commentaires

Comme tout langage évolué, C++ autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation. Il existe deux types de commentaires :

- les commentaires « libres », hérités du langage C ;
- les commentaires de fin de ligne (introduits par C++).

2.5.1 Les commentaires libres

Ils sont formés de caractères quelconques placés entre les symboles /* et */. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de tels commentaires :

```
/* programme de calcul de racines carrees */

/* commentaire fantaisiste &ç§{<>} ?%!!!!!! */

/* commentaire s'étendant
   sur plusieurs lignes
   de programme source    */

/* ============================================
   *   commentaire quelque peu esthetique      *
   *   et encadre, pouvant servir,                *
   *   par exemple, d'en-tete de programme   *
   ========================================== */
```

Voici un exemple de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int i ;            /* compteur de boucle */
float x ;          /* nombre dont on veut la racine carree */
float racx ;       /* racine carrée du nombre */
```

Voici enfin un exemple légal mais peu lisible :

```
int  /* compteur de boucle */ i ;  float x ;
 /* nombre dont on veut la racine
carrée */  float racx ;       /* racine carrée du nombre */
```

2.5.2 Les commentaires de fin de ligne

Comme son nom l'indique, il se place à la fin d'une ligne. Il est introduit par les deux caractères : //. Dans ce cas, tout ce qui est situé entre // et la fin de la ligne est un commentaire. Notez que cette nouvelle possibilité n'apporte qu'un surcroît de confort et de sécurité ; en effet, une ligne telle que :

```
cout << "bonjour\n" ; // formule de politesse
```

peut toujours être écrite ainsi :

```
cout << "bonjour\n" ; /* formule de politesse */
```

Vous pouvez mêler (volontairement ou non !) les commentaires libres et les commentaires de fin de ligne. Dans ce cas, notez que, dans :

```
/* partie1 // partie2 */ partie3
```

le commentaire « ouvert » par /* ne se termine qu'au prochain */ ; donc *partie1* et *partie2* sont des commentaires, tandis que *partie3* est considéré comme appartenant aux instructions. De même, dans :

```
partie1 // partie2 /* partie3 */ partie4
```

le commentaire introduit par // s'étend jusqu'à la fin de la ligne. Il concerne donc *partie2, partie3 et partie 4.*

Remarques

1 Le commentaire de fin de ligne constitue l'un des deux cas où la fin de ligne joue un rôle significatif. L'autre cas concerne les directives destinées au préprocesseur (il ne concerne donc pas la compilation proprement dite).

2 Si l'on utilise systématiquement le commentaire de fin de ligne, on peut alors faire appel à /* et */ pour « inhiber » un ensemble d'instructions (contenant éventuellement des commentaires de fin de ligne) en phase de mise au point.

3 Nos exemples de commentaires doivent être considérés comme des exemples didactiques et, en aucun cas, comme des modèles de programmation. Ainsi, généralement, il sera préférable d'éviter les commentaires redondants par rapport au texte du programme lui-même.

3 Création d'un programme en C++

La manière de développer et d'utiliser un programme en C++ dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir : édition du programme, compilation et édition de liens.

3.1 L'édition du programme

L'édition du programme (on dit aussi parfois « saisie ») consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme qu'on nomme « programme source ». En général, ce texte sera conservé dans un fichier que l'on nommera « fichier source ».

Chaque système possède ses propres conventions de dénomination des fichiers. En général, un fichier peut, en plus de son nom, être caractérisé par un groupe de caractères (au moins 3) qu'on appelle une « extension » (ou, parfois un « type ») ; la plupart du temps, en C++, les fichiers source porteront l'extension *cpp*.

3.2 La compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En C++ (comme en C), compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes :

- **traitement par le préprocesseur** : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en C++ pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.
- **compilation** proprement dite, c'est-à-dire traduction en langage machine du texte C++ fourni par le préprocesseur.

Le résultat de la compilation porte le nom de module objet.

3.3 L'édition de liens

En général, un module objet créé ainsi par le compilateur n'est pas directement exécutable. Il lui manquera, en effet, au moins les fonctions de la bibliothèque standard dont il a besoin ; dans notre exemple précédent, il sagirait : de la fonction *sqrt*, des fonctions correspondant au travail des opérateurs << et >>.

C'est effectivement le rôle de l'éditeur de liens que d'aller rechercher dans la bibliothèque standard les modules objet nécessaires. Notez que cette bibliothèque est une collection de modules objets organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers. Nous verrons que, grâce aux possibilités de compilation séparée de C++, il vous sera également possible de rassembler au moment de l'édition de liens différents modules objets, compilés de façon indépendante.

Le résultat de l'édition de liens est ce que l'on nomme un programme exécutable, c'est-à-dire un ensemble autonome d'instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu'il soit nécessaire de faire appel à un quelconque composant de l'environnement de programmation en C++.

3.4 Les fichiers en-tête

Nous avons vu que, grâce à la directive *#include*, vous pouviez demander au préprocesseur d'introduire des instructions (en langage C++) provenant de ce que l'on appelle des fichiers « en-tête ». Ces fichiers comportent, entre autres choses, des déclarations relatives aux fonctions prédéfinies (attention, ne confondez pas ces déclarations des fichiers en-têtes, avec les modules objets qui contiendront le code exécutable de ces différentes fonctions).

3

Les types de base de C++

Les types *char*, *int* et *float* que nous avons déjà rencontrés sont souvent dits « scalaires » ou « simples », car, à un instant donné, une variable d'un tel type contient une seule valeur. Ils s'opposent aux types « structurés » (on dit aussi « agrégés ») qui correspondent à des variables qui, à un instant donné, contiennent plusieurs valeurs (de même type ou non). Ici, nous étudierons en détail les propriétés de ce que l'on appelle les **types de base** du langage C++ ; il s'agit des types scalaires à partir desquels pourront être construits tous les autres, dits « types dérivés », qu'il s'agisse :

- de types structurés comme les chaînes, les vecteurs, les tableaux et surtout les classes ;
- d'autres types simples comme les pointeurs ou les énumérations.

Nous verrons ensuite comment déclarer et initialiser des variables de ces différents types de base. Puis, nous examinerons le cas des constantes (variables déclarées avec le qualificatif *const* ou *constexpr*). Nous terminerons sur les possibilités de déclaration automatique (*auto*, *decltype*) introduites par C++11.

Auparavant, cependant, nous vous proposons de faire un bref rappel concernant la manière dont l'information est représentée dans un ordinateur, ce qui nous permettra de mieux comprendre la notion de type qui en découle.

1 La notion de type

La mémoire centrale est un ensemble de positions binaires nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par ce qu'on nomme son adresse.

L'ordinateur, compte tenu de sa technologie actuelle, ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être **codée** sous cette forme. Dans ces conditions, on voit qu'il ne suffit pas de connaître le contenu d'un emplacement de la mémoire (d'un ou de plusieurs octets) pour être en mesure de lui attribuer une signification. Par exemple, si vous savez qu'un octet contient le « motif binaire » suivant :

```
01001101
```

vous pouvez considérer que cela représente le nombre entier *77* (puisque le motif ci-dessus correspond à la représentation en base 2 de ce nombre). Mais pourquoi cela représenterait-il un nombre ? En effet, toutes les informations (nombres entiers, nombres réels, nombres complexes, caractères, instructions de programme en langage machine, graphiques, images, sons, vidéos...) devront, au bout du compte, être codées en binaire.

Dans ces conditions, les huit bits ci-dessus peuvent peut-être représenter un caractère ; dans ce cas, si nous connaissons la convention employée sur la machine concernée pour représenter les caractères, nous pouvons lui faire correspondre un caractère donné (par exemple M, dans le cas du code ASCII). Ils peuvent également représenter une partie d'une instruction machine ou d'un nombre entier codé sur 2 octets, ou d'un nombre réel codé sur 4 octets, ou...

On comprend donc qu'il n'est pas possible d'attribuer une signification à une information binaire tant que l'on ne connaît pas la manière dont elle a été codée. Qui plus est, en général, il ne sera même pas possible de « traiter » cette information. Par exemple, pour additionner deux informations, il faudra savoir quel codage a été employé afin de pouvoir mettre en œuvre les bonnes instructions (en langage machine). Par exemple, on ne fait pas appel aux mêmes circuits électroniques pour additionner deux nombres codés sous forme « entière » et deux nombres codés sous forme « flottante ».

D'une manière générale, la notion de type, telle qu'elle existe dans les langages évolués, sert à régler (entre autres choses) les problèmes que nous venons d'évoquer.

Les types de base du langage C++ se répartissent en quatre catégories en fonction de la nature des informations qu'ils permettent de représenter :

- nombres entiers (mot-clé **int**) ;
- nombres flottants (mot-clé **float** ou **double**) ;
- caractères (mot-clé **char**) ;
- valeurs booléennes, c'est-à-dire dont la valeur est soit vrai, soit faux (mot-clé **bool**).

2 Les types entiers

2.1 Les différents types usuels d'entiers prévus par C++

C++ prévoit que, sur une machine donnée, on puisse trouver jusqu'à quatre tailles différentes d'entiers, désignées par les mots-clés suivants :

- **short int** (qu'on peut abréger en *short*) ;
- **int** (c'est celui que nous avons rencontré dans le chapitre précédent) ;
- **long int** (qu'on peut abréger en *long* ;
- **long long int** (qu'on peut abréger en *long long*), introduit par C++11.

Chaque taille impose naturellement ses limites. Toutefois, ces dernières dépendent non seulement du mot-clé considéré, mais également de la machine utilisée : tous les *int* n'ont pas la même taille sur toutes les machines ! Fréquemment, deux des trois mots-clés correspondent à une même taille[1]. Le type *long long int* introduit par C++11 doit être représenté sur un minimum de 64 bits.

2.2 Leur représentation en mémoire

Pour fixer les idées, nous raisonnerons ici sur des nombres entiers représentés sur 16 bits , mais il sera facile de généraliser notre propos à une taille quelconque.

Quelle que soit la machine (et donc, a fortiori, le langage !), les entiers sont codés en utilisant un bit pour représenter le signe (0 pour positif et 1 pour négatif).

a) Lorsqu'il s'agit d'un nombre **positif** (ou nul), sa valeur absolue est écrite en base 2, à la suite du bit de signe. Voici quelques exemples de codages de nombres (à gauche, le nombre en décimal, au centre, le codage binaire correspondant, à droite, le même codage exprimé en hexadécimal) :

```
  1          0000000000000001          0001
  2          0000000000000010          0002
  3          0000000000000011          0003
 16          0000000000010000          0010
127          0000000001111111          007F
255          0000000011111111          00FF
```

b) Lorsqu'il s'agit d'un nombre **négatif**, sa valeur absolue est codée généralement suivant ce que l'on nomme la « technique du complément à deux »[2]. Pour ce faire, cette valeur est d'abord exprimée en base 2 puis tous les bits sont inversés (1 devient 0 et 0 devient 1) et, enfin, on ajoute une unité au résultat. Voici quelques exemples (avec la même présentation que précédemment) :

1. Dans une implémentation donnée, on peut connaître les caractéristiques des différents types entiers grâce à des constantes (telles que *INT_MAX*, *INT_MIN*) définies dans le fichier en-tête *climits*.

2. Bien que non imposée totalement par la norme, cette technique tend à devenir universelle. Dans les (anciennes) implémentations qui se contentaient de respecter les contraintes imposées par la norme, les différences restent mineures (deux représentations du zéro : +0 et -0, différence d'une unité sur la plage des valeurs couvertes pour une taille d'entier donnée).

```
  -1              1111111111111111              FFFF
  -2              1111111111111110              FFFE
  -3              1111111111111101              FFFD
  -4              1111111111111100              FFFC
 -16              1111111111110000              FFF0
-256              1111111100000000              FF00
```

Remarques

1 Le nombre 0 est codé d'une seule manière (0000000000000000).

2 Si l'on ajoute 1 au plus grand nombre positif (ici 0111111111111111, soit 7FFF en hexadécimal ou 32767 en décimal) et que l'on ne tient pas compte de la dernière retenue (ou, ce qui revient au même, si l'on ne considère que les 16 derniers bits du résultat), on obtient... le plus petit nombre négatif possible (ici 1000000000000000, soit 8000 en hexadécimal ou -32768 en décimal). Nous verrons qu'en C++, la situation dite « de dépassement de capacité » (correspondant au cas où un résultat d'opération s'avère trop grand pour le type prévu) sera traité ainsi, en ignorant un bit de retenue...

2.3 Les types entiers non signés

De façon quelque peu atypique, C++ vous autorise à définir trois autres types voisins des précédents en utilisant le qualificatif *unsigned*. Dans ce cas, on ne représente plus que des nombres positifs pour lesquels aucun bit de signe n'est nécessaire. Cela permet théoriquement de doubler la taille des nombres représentables ; par exemple, avec 16 bits, on passe de l'intervalle [-32768; 32767] à l'intervalle [0 ; 65535]. Mais cet avantage est bien dérisoire, par rapport aux risques que comporte l'utilisation de ces types (songez qu'une simple expression telle que *n-p* va poser problème dès que la valeur de *p* sera supérieure à celle de *n* !).

En pratique, ces types non signés seront souvent réservés à la manipulation directe d'un « motif binaire » (tel un « mot d'état ») et non pas pour faire des calculs. Nous verrons d'ailleurs qu'il existe des opérateurs spécialisés dits « de manipulation de bits ». Comme nous aurons l'occasion de le rappeler, **il est conseillé d'éviter de méler des entiers signés et des entiers non signés dans une même expression**, même si cela est théoriquement autorisé par la norme.

2.4 Notation des constantes littérales entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire simplement sous forme décimale, avec ou sans signe, comme dans ces exemples :

```
+533    48    -273
```

Vous pouvez également utiliser une notation octale (base 8) ou hexadécimale (base 16). La forme octale se note en faisant précéder le nombre écrit en base 8 du chiffre 0.

Par exemple :

014 correspond à la valeur décimale *12*,

037 correspond à la valeur décimale *31*.

La forme hexadécimale se note en faisant précéder le nombre écrit en hexadécimal (les dix premiers chiffres se notent 0 à 9, A correspond à dix, B à onze... F à quinze) des deux caractères *0x* (ou *0X*). Par exemple :

0x1A correspond à la valeur décimale *26* (*16+10*)

Les deux dernières notations doivent cependant être réservées aux situations dans lesquelles on s'intéresse plus au motif binaire qu'à la valeur numérique de la constante en question. D'ailleurs, ces constantes sont de type non signé (alors que les constantes écrites en notation décimale sont bien signées).

Remarque

Nous parlerons souvent de « constante littérale entière » ou, parfois, tout simplement de « littéral entier » pour une notation telle que +533, afin de la distinguer des constantes dites symboliques qualifiées par *const*.

Informations complémentaires

Par défaut, une constante entière écrite en notation décimale est codée dans l'un des deux types signé *int* ou *long* (on utilise le type le plus petit, suffisant pour la représenter). On peut imposer un type précis à une constante litérale en la « suffixant » (sans espace) par une ou plusieurs lettres (majuscules ou minuscules, dans un ordre quelconque) parmi *u* pour *unsigned, l* pour *long* et *ll* pour *long long* (en C++11). Ainsi, *3u* ou *-35u* seront de type *unsigned int* ; *456l* sera du type *long* ; *45ul* sera du type *unsigned long ; 34ULL* sera du type *unsigned long long* (en C++11).

3 Les types flottants

3.1 Les différents types et leur représentation en mémoire

Les types flottants permettent de représenter, **de manière approchée**, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation scientifique (ou exponentielle) bien connue qui consiste à écrire un nombre sous la forme $1.5\ 10^{22}$ ou $0.472\ 10^{-8}$; dans une telle notation, on nomme « mantisses » les quantités telles que *1.5* ou *0.472* et « exposants » les quantités telles que *22* ou *-8*.

Plus précisément, un nombre réel sera représenté en flottant, en déterminant deux quantités M (mantisse) et E (exposant) telles que la valeur

$$M . B^{E}$$

représente une approximation de ce nombre. La base B est généralement unique pour une machine donnée (il s'agit souvent de 2 ou de 16) et elle ne figure pas explicitement dans la représentation machine du nombre.

C++ prévoit trois types de flottants correspondant à des tailles différentes : *float*, *double* et *long double.*

La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable, sauf lorsque l'on doit faire une analyse fine des erreurs de calcul[3]. En revanche, il est important de noter que de telles représentations sont caractérisées par deux éléments :

- *La précision* : lors du codage d'un nombre décimal quelconque dans un type flottant, il est nécessaire de ne conserver qu'un nombre fini de bits. Or la plupart des nombres s'exprimant avec un nombre limité de décimales ne peuvent pas s'exprimer de façon exacte dans un tel codage. On est donc obligé de se limiter à une représentation approchée en faisant ce que l'on nomme une erreur de troncature. Quelle que soit la machine utilisée, on est assuré que cette erreur (relative) ne dépassera pas 10^{-6} pour le type *float* et 10^{-10} pour le type *long double*.
- *Le domaine couvert*, c'est-à-dire l'ensemble des nombres représentables à l'erreur de troncature près. Là encore, quelle que soit la machine utilisée, on est assuré qu'il s'étendra au moins de 10^{-37} à 10^{+37}.

3.2 Notation des constantes littérales flottantes

Comme dans la plupart des langages, les constantes littérales flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale ;
- exponentielle.

La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais, bien sûr, pas toutes les deux en même temps !). En voici quelques exemples corrects :

```
12.43    -0.38    -.38    4.    .27
```

En revanche, la constante *47* serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance, si ce n'est au niveau du temps d'exécution, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant).

La notation exponentielle utilise la lettre *e* (ou *E*) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès que l'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents) :

3. À titre indicatif, le fichier en-tête *cfloat* contient de nombreuses constantes définissant les propriétés des différents types flottants (limites, précisions, « epsilon machine »...). On trouvera plus d'informations sur ces éléments dans *Le guide complet du langage C*, du même auteur, aux éditions Eyrolles.

```
   4.25E4          4.25e+4          42.5E3
 54.27E-32        542.7E-33        5427e-34
   48e13            48.e13           48.0E13
```

Par défaut, toutes les constantes littérales flottantes sont créées par le compilateur dans le type *double*. Il est toutefois possible d'imposer à une constante flottante :

- d'être du type *float*, en faisant suivre son écriture de la lettre *F* (ou *f*) : cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision (le gain en place et la perte en précision dépendant de la machine concernée).
- d'être du type *long double*, en faisant suivre son écriture de la lettre *L* (ou *l*) : cela permet de gagner éventuellement en précision, en contrepartie d'une perte de place mémoire (le gain en précision et la perte en place dépendant de la machine concernée).

4 Les types caractères

4.1 La notion de caractère en langage C++

Comme la plupart des langages, C++ permet de manipuler des caractères codés en mémoire sur un octet. Bien entendu, le code employé, ainsi que l'ensemble des caractères représentables, dépend de l'environnement de programmation utilisé (c'est-à-dire à la fois de la machine concernée et du compilateur employé). Néanmoins, on est toujours certain de disposer des lettres (majuscules et minuscules), des chiffres, des signes de ponctuation et des différents séparateurs (en fait, tous ceux que l'on emploie pour écrire un programme !). En revanche, les caractères nationaux (caractères accentués ou ç) ou les caractères semi-graphiques ne figurent pas dans tous les environnements.

Par ailleurs, la notion de caractère en C++ dépasse celle de caractère imprimable, c'est-à-dire auquel est obligatoirement associé un graphisme (et qu'on peut donc imprimer ou afficher sur un écran). C'est ainsi qu'il existe certains caractères de changement de ligne, de tabulation, d'activation d'une alarme sonore (cloche)... Nous avons d'ailleurs déjà utilisé le premier (sous la forme \n).

Remarque

Les caractères non imprimables sont souvent nommés « caractères de contrôle ». Dans le code ASCII (restreint ou non), ils ont des codes compris entre 0 et 31.

4.2 Notation des constantes littérales de type caractère

Les constantes littérales de type « caractère », lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quotes) le caractère voulu, comme dans ces exemples :

```
'a'     'Y'     '+'     '$'
```

Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère « \ », nommé « antislash » (en anglais, il se nomme « back-slash », en français, on le nomme aussi « barre inverse » ou « contre-slash »). Dans cette catégorie, on trouve également quelques caractères (\, ', " et ?) qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteur qui les empêche d'être notés de manière classique entre deux apostrophes.

Voici la liste de ces caractères :

Notation en C++	Code ASCI	Abréviation	Signification usuelle
\a	07	BEL	cloche ou bip (alert ou audible bell)
\b	08	BS	Retour arrière (Backspace)
\f	0C	FF	Saut de page (Form Feed)
\n	0A	LF	Saut de ligne (Line Feed)
\r	0D	CR	Retour chariot (Carriage Return)
\t	09	HT	Tabulation horizontale (Horizontal Tab)
\v	0B	VT	Tabulation verticale (Vertical Tab)
\\	5C	\	
\'	2C	'	
\"	22	"	
\?	3F	?	

De plus, il est possible d'utiliser directement le code du caractère en l'exprimant, toujours à la suite du caractère « antislash » :

- soit sous forme **octale** ;
- soit sous forme **hexadécimale** précédée de **x**.

Voici quelques exemples de notations équivalentes, dans le code ASCII :

```
'A'   '\x41'\101'
'\r'  '\x0d'\15'\015'
'\a'  '\x07'\x7'\07'\007'
```

Remarques

1 En fait, il existe plusieurs versions de code ASCII, mais toutes ont en commun la première moitié des codes (correspondant aux caractères qu'on trouve dans toutes les implémentations) ; les exemples cités ici appartiennent bien à cette partie commune.

2 Le caractère \, suivi d'un caractère autre que ceux du tableau ci-dessus ou d'un chiffre de 0 à 7 est simplement ignoré. Ainsi, dans le cas où l'on a affaire au code ASCII, \9 correspond au caractère 9 (de code ASCII 57), tandis que \7 correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».

3 En fait, la norme prévoit trois types de caractères : *char*, *signed char* et *unsigned char*. Pour l'instant, sachez que cet attribut de signe n'agit pas sur la représentation d'un caractère en mémoire. En revanche, nous verrons dans le prochain chapitre que C++ permet de convertir une valeur de type caractère en une valeur entière ; dans ce cas, l'attribut de signe du caractère pourra intervenir.

En Java

Les caractères sont représentés sur 2 octets, en utilisant le codage dit « Unicode ». Il n'existe qu'un seul type *char*.

Informations complémentaires

Il existe d'autre types caractères, issus du langage C :

– *char16_t* : il utilise 16 bits et peut servir à représenter des caractères utilisant le code Unicode UTF16 ; les constantes de ce type sont suffixées par *U* ;

– *char32_t :* il utilise 32 bits et peut servir à représenter des caractères utilisant le code Unicode UTF2 ; les constantes de ce type sont suffixées par *L* ;

– *wchar_t* : les caractères sont codés sur un nombre quelconque d'octets.

5 Le type bool

Ce type est tout naturellement formé de deux valeurs notées *true* (vrai) et *false* (faux). Il peut intervenir dans des constructions telles que :

```
bool ok = false ;
   .....
if (.....) ok = true ;
   .....
if (ok) .....
```

6 Déclaration des variables

Nous venons de décrire les différents types de base, ainsi que la manière d'écrire les constantes littérales de chacun d'entre eux. La **déclaration** d'une variable d'un type de base se présente sous la forme du nom de type, suivi d'une liste comportant un ou plusieurs noms de variable comme dans :

```
int p ;                 // p est de type int
long double x, y, z ;   // x, y et z sont de type long double
```

En ce qui concerne l'emplacement de ces déclarations, rappelons qu'une variable doit simplement être déclarée avant son utilisation. Notez qu'il existe des styles de programmation assez opposés, entre lesquels nous ne chercherons pas à trancher ici, à savoir :

- déclarer systématiquement toutes les variables en début de la fonction concernée ;
- placer la déclaration d'une variable le plus près possible de son utilisation.

7 Initialisation des variables

7.1 Généralités

Les variables déclarées dans la fonction *main* (ou dans d'autres fonctions) ne reçoivent pas de valeur par défaut. Plus précisément, l'emplacement qui leur est attribué est laissé tel quel, ce qui signifie que sa valeur est quasiment imprévisible :

```
int n ;
 .....       // ici, la valeur de n est imprévisible
cout << n ;  // en général, avertissement du compilateur
             // on affiche une valeur quelconque
n = 20 ;
cout << n  ; // ici, on affiche 20
```

Il est généralement conseillé d'initialiser explicitement une variable lors de sa déclaration, comme dans :

```
float x = 2.5, y = 5.25 ;
```

Cependant, cette démarche n'est pas toujours réaliste. C'est par exemple le cas pour une variable dont la valeur est fixée par une lecture :

```
int n ;    // ou, artificiellement :   int n = 0 ;
  ....
cin >> n ;
```

Une initialisation prématurée (artificielle) de *n* (par exemple à 0) peut troubler le lecteur du programme. Qui plus est, certains outils de détection des variables non initialisées ne fonctionneront naturellement plus en cas d'initialisation artificielle.

7.2 Notations de l'initialisation

La notation utilisée précédemment pour initialiser les variables était la seule existant dans les anciennes versions de C++. Elle reste assez naturelle et elle ressemble à celle qui est utilisée dans bon nombre d'autres langages dont Java ou C.

Mais il existe d'autres notations moins usitées dans ce contexte de types simples, à savoir :

- la notation parenthésée ;
- la notation avec accolades introduite par C++11.

7.2.1 La notation parenthésée

Elle a introduite par C++98 pour faciliter l'utilisation des patrons ;

```
int n (5) ;    // entièrement équivalente ici à int n = 5 ;
```

Nous verrons que cette notation permettra d'écrire un patron de classes, sans avoir à se préoccuper de savoir si le type générique utilisé est de type classe ou non. Elle est rarement utilisée en dehors de ce contexte de patrons.

7.2.2 La notation avec accolades (C++11)

Cette notation a été introduite par C++11 dans le but de définir une notation universelle utilisable pour tous les types de variables (simples, agrégats, objets...) :

```
int n { 5 } ;
int n = { 5 } ;
```

Dans notre exemple d'initialisation d'un entier à 5, les quatre formes d'initialisation présentées sont équivalentes. Cependant, les notations avec accolades n'autorisent pas ce que l'on nomme des « conversions dégradantes » (étudiées dans le prochain chapitre), alors que les autres les acceptent :

```
int n  = 2.35 ;     // accepté - n vaut 2
int n (2.35) ;      // accepté - n vaut 2
int n {2.35} ;      // rejeté en compilation
int n = {2.35} ;    // rejeté en compilation
```

De plus, la notation {} représente une valeur dite « nulle » :

```
int n  { } ;       // ou int n = {} ;  équivalent à : int n {0} ou int n = 0
```

Un telle valeur nulle correspond naturellement à la valeur 0 pour des variables de type numérique. Pour le type caractère, il s'agira du caractère de code nul. Plus tard, nous rencontrerons d'autres significations telles que chaîne de longueur nulle ou vecteur vide.

Informations complémentaires

Les formes d'initialisation utilisant le signe égal sont dites **initialisations copies**, tandis que celles n'y recourant pas sont dites **initialisations directes**. Si leur rôle est identique dans le cas des variables simples, il n'en ira plus de même dans le cas des objets. Nous y reviendrons le moment venu.

On pourrait penser que la notation avec accolades est utilisable dans toutes les situations. En fait, nous verrons que la notion de liste d'initialisation (type *initializer_list*), également introduite par C++11, viendra quelque peu perturber la situation.

8 Constantes et expressions constantes

Il est possible de demander que la valeur d'une variable ne soit pas modifiée pendant l'exécution du programme. Pour ce faire, on utilise soit le modificateur *const*, soit le modificateur *constexpr* introduit par C++11.

8.1 Le modificateur *const*

Avec :

```
const int n = 20 ;
```

on déclare *n* de type *int* et de valeur (initiale) *20* mais, de surcroît, les éventuelles instructions modifiant la valeur de *n* seront rejetées par le compilateur. Nous en avions déjà rencontré un exemple dans notre premier programme du paragraphe 1.1 du chapitre 2.

Comme on peut s'y attendre, une variable déclarée *const* doit obligatoirement être initialisée lors de sa déclaration. En revanche, l'expression utilisée pour en fixer la valeur n'a pas besoin d'être constante, comme le montre cet exemple :

```
int p ; cin >> p ;
   .....
const int n = 2 * p + 4  ;
```

Ici, la valeur de *n* n'est pas connue à la compilation et elle sera fixée à un moment donné de l'exécution du programme et elle ne pourra plus varier ensuite.

8.2 Le modificateur *constexpr* (C++11)

Depuis C++11, il existe un mot-clé *constexpr* qui sert à désigner une variable constante dont la **valeur est connue lors de la compilation** :

```
const int n = 6 ;
constexpr int p = 3 * n + 2 ;   // OK : p sera calculé à la compilation
```

Comme on peut s'y attendre *constexpr* implique *const* mais la réciproque est fausse : une variable déclarée *const* n'est pas toujours utilisable dans une expression *constexpr*. Ainsi, cet exemple est correct :

```
const int n = 3 ;         // ici, n est calculable à la compilation
constexpr int p = 2*n ;   // OK car p est calculable à la compilation
```

alors que celui-ci ne l'est pas :

```
int n = 2 ;             // ici n n'est pas calculable à la compilation
const int p = n ;       // p est const mais pas const expr
constexpr int q = 3 * n + 2 ;   // erreur de compilation
```

L'emploi de *constexpr* s'avérera intéressant lorsqu'il sera couplé à des fonctions déclarées elles-mêmes *constexpr*, ce qui signifiera que leur valeur pourra alors être calculée à la compilation, ce qui conduira à un gain de temps à l'exécution. Pour être exhaustifs, disons également que, dans certains environnements, le compilateur peut profiter du fait qu'une variable est déclarée *constexpr* pour optimiser le code, par exemple en la plaçant dans une mémoire en lecture seule.

9 Déclarations automatiques (C++11)

C++11 a jouté des fonctionnalités permettant de définir automatiquement le type d'une variable, soit à partir d'une expression d'initialisation, soit à partir d'une autre expression.

9.1 Le mot-clé *auto*

Le type d'une variable peut être déduit de l'expression utilisée pour son initialisation :

```
auto n = 12 ;    // 12 est de type int, donc n sera de type int
auto x = 2.5 ;   // 2.5 est de type double, donc x sera de type double
auto y = 2.5L ;  // 2.5L est de type long double, donc y sera de type long double
auto p = 2 * n ; // l'expression 2*n est de type int, donc p sera de type int
```

On notera bien que les qualificatifs *const* ou *constexpr* ne sont pas considérés avec la déclaration automatique :

```
const int n = 12 ;    // ou constexpr
auto p = n ;          // p est de type int et non const int ou constexpr int
```

On peut toutefois ajouter le qualificatif dans la déclaration automatique de type :

```
const int n = 12 ;
const auto p = n ;    // (ou auto const) cette fois, p est bien de type const int
```

Certes, l'usage de *auto* semble ici ne guère présenter d'intérêt, par rapport à une déclaration classique. Mais cette fonctionnalité s'avérera fort précieuse lorsque l'on sera amené à utiliser des expressions dont le type n'est pas facile à connaître ou, tout simplement, difficile à écrire. Il existera également des circonstances où l'emploi de *auto* nous évitera d'avoir à écrire deux fois le même nom de type, limitant ainsi les risques d'erreurs. Nous en rencontrerons de nombreux exemples dans la suite de l'ouvrage.

9.2 Le mot-clé *decltype*

On peut aussi déclarer qu'une variable est du même type qu'une autre variable :

```
long n1 = 10 ;
const int p1 = 5 ;
decltype (n1) n2 = 20 ;    // n2 sera du type de n1, donc long
decltype (p1) p2 = 10 ;    // p2 sera du type de p1, donc const int
```

Cette fois, les qualificatifs *const* ou *constexpr* sont bien conservés.

On peut aussi employer *decltype* avec une expression :

```
int n ;
decltype (2*n+1) p ;   // p sera du type de l'expression 2*n+1, donc int
```

Cette facilité s'avérera surtout utile pour définir des types peu connus du programmeur. Nous en rencontrerons des exemples par la suite.

10 Le mot-clé *volatile*

N.B. *Ce paragraphe peut-être ignoré dans un premier temps*

Il existe une déclaration peu usitée employant le mot-clé *volatile* de la même manière que *const*, comme dans ces exemples :

```
volatile int p ;
volatile int q = 50 ;
```

Elle indique au compilateur que la valeur de la variable correspondante (ici *p* ou *q*) peut évoluer, indépendamment des instructions du programme. Son usage est limité à des situations très particulières dans lesquelles l'environnement extérieur au programme peut agir directement sur des emplacements mémoire, comme peuvent le faire certains périphériques d'acquisition. L'emploi de *volatile* dans ce cas peut se révéler précieux puisqu'il peut alors empêcher le compilateur de procéder à des « optimisations » basées sur l'examen des seules instructions du programme. Considérez, par exemple :

```
for (int i=1 ; i<15 ; i++)
{ k = j*j ;
  // ici, on utilise k
}
```

Si la valeur de *j* n'est pas modifiée dans la boucle, le compilateur traduira ces instructions comme si l'on avait écrit :

```
k = j*j ;
for (int i=1 ; i<15 ; i++)
{  // ici, on utilise k
}
```

En revanche, si la variable *k* a été déclarée *volatile*, le compilateur conservera l'affectation en question dans la boucle.

4

Opérateurs et expressions

Ce chapitre vous présente la plupart des opérateurs du C++ ainsi que les règles de priorité et de conversion de type qui interviennent dans les évaluations des expressions. Les (quelques) autres opérateurs concernent essentiellement les pointeurs, les accès aux éléments des tableaux, les accès aux membres des objets et, enfin, ce que l'on nomme « résolution de portée ». Ils seront exposés dans la suite de cet ouvrage (mais ils figureront quand même dans le tableau récapitulatif fourni en fin de chapitre).

1 Originalité des notions d'opérateur et d'expression en C++

Commes les langages C ou Java, C++ est un langage très fourni en opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (*arithmétiques*, *relationnels*, *logiques*) ou moins classiques (*manipulations de bits*). Mais, de surcroît, C++ dispose d'un important éventail d'opérateurs originaux *d'affectation* et *d'incrémentation*.

Ce dernier aspect nécessite une explication. En effet, dans la plupart des langages, on trouve, comme en C++ :

- d'une part, des *expressions* formées (entre autres) à l'aide d'opérateurs ;
- d'autre part, des *instructions* pouvant éventuellement faire intervenir des expressions, comme l'instruction d'affectation :

```
y = a * x + b ;
```

ou encore l'instruction d'affichage :

```
cout << "valeur = " << n + 2 * p ;
```

dans laquelle apparaît l'expression *n + 2 * p* ;

Dans beaucoup de langages, l'expression possède une valeur mais ne réalise aucune action, en particulier aucune attribution d'une valeur à une variable. Au contraire, l'instruction d'affectation y réalise une attribution d'une valeur à une variable mais ne possède pas de valeur. On a affaire à deux notions parfaitement disjointes. En C++, il en va différemment puisque :

- D'une part, les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possédera une valeur), mais également agir sur le contenu de variables. Ainsi, l'expression (car, comme nous le verrons, il s'agit bien d'une expression en C++) :

  ```
  ++i
  ```

  réalisera une action, à savoir : augmenter la valeur de *i* de *1* ; en même temps, elle aura une valeur, à savoir celle de *i* après incrémentation.

- D'autre part, une affectation apparemment classique telle que :

  ```
  i = 5
  ```

  pourra, à son tour, être considérée comme une expression (ici, de valeur *5*). D'ailleurs, en C++, l'affectation (=) est un opérateur. Par exemple, la notation suivante :

  ```
  k = i = 5
  ```

  représente une expression en C++ (ce n'est pas encore une instruction – nous y reviendrons). Elle sera interprétée comme :

  ```
  k = (i = 5)
  ```

  Autrement dit, elle affectera à *i* la valeur *5* puis elle affectera à *k* la valeur de l'expression *i = 5*, c'est-à-dire *5*.

En fait, en C++, les notions d'expression et d'instruction sont étroitement liées puisque **la principale instruction** de ce langage est **une expression terminée par un point-virgule**. On la nomme souvent « instruction expression ». Voici des exemples de telles **instructions** qui reprennent les **expressions** évoquées ci-dessus :

```
++i ;
i = 5 ;
k = i = 5 ;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans ces deux cas, il y a évaluation d'une expression (*++i* ou *i=5*) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression *i=5*, c'est-à-dire *5*, est à son tour affectée à *k* ; en revanche, la valeur finale de l'expression complète est, là encore, inutilisée.

2 Les opérateurs arithmétiques en C++

2.1 Présentation des opérateurs

Comme tous les langages, C++ dispose d'opérateurs classiques « binaires » (c'est-à-dire portant sur deux « opérandes »), à savoir l'addition (+), la soustraction (-), la multiplication (*) et la division (/), ainsi que d'un opérateur « unaire » (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté - (comme dans *-n* ou dans *-x+y*).

Les opérateurs binaires ne sont a priori définis que pour deux opérandes ayant le même type parmi : *int*, *long int*, *float*, *double* et *long double* (ainsi que *unsigned int* et *unsigned long*) et ils fournissent un résultat de même type que leurs opérandes.

Mais nous verrons, au paragraphe 3, que, par le jeu des conversions implicites, le compilateur saura leur donner une signification :

- soit lorsqu'ils porteront sur des opérandes de type *short* qui est un type numérique à part entière (il en ira de même pour *unsigned short*) ;
- soit lorsqu'ils porteront sur des opérandes de type *char* ou même *bool*, bien qu'ils ne s'agisse plus de vrais types numériques (il en ira de même pour *signed char* et *unsigned char*) ;
- soit lorsqu'ils porteront sur des opérandes de types différents[1].

De plus, il existe un opérateur de modulo noté *%* qui ne peut porter que sur des entiers et qui fournit le reste de la division de son premier opérande par son second. Par exemple, *11%4* vaut *3*, *23%6* vaut *5*. La norme ANSI ne définit cet opérateur que pour des valeurs positives de ses deux opérandes. Dans les autres cas, le résultat dépend de l'implémentation.

Notez bien qu'en C++ le quotient de deux entiers fournit un entier. Ainsi, *5/2* vaut *2* ; en revanche, le quotient de deux flottants (noté, lui aussi */*) est bien un flottant (*5.0/2.0* vaut bien approximativement *2.5*).

Remarque

Il n'existe pas d'opérateur d'élévation à la puissance. Il est nécessaire de faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x^3 comme *x*x*x*), soit à la fonction *power* de la bibliothèque C standard, dont l'en-tête figure dans *cmath* (voyez éventuellement l'Annexe G fournie sur le site).

En Java

L'opérateur % est défini pour les entiers (positifs ou non) et pour les flottants.

1. En langage machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.

2.2 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En C++, comme dans les autres langages, les règles sont naturelles et rejoignent celles de l'algèbre traditionnelle (du moins, en ce qui concerne les opérateurs arithmétiques dont nous parlons ici).

Les opérateurs unaires + et - ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs binaires *, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et -.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a affaire à une *associativité de gauche à droite* (nous verrons que quelques opérateurs, autres qu'arithmétiques, utilisent une associativité de droite à gauche).

Enfin, des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez que ces parenthèses peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

```
a + b * c                    a + ( b * c )
a * b + c % d                ( a * b ) + ( c % d )
- c % d                      ( - c ) % d
- a + c % d                  ( - a ) + ( c % d )
- a / - b + c                ( ( - a ) / ( - b ) ) + c
- a / - ( b + c )            ( - a ) / ( - ( b + c ) )
```

Remarque

Les règles de priorité interviennent pour définir la signification exacte d'une expression. Néanmoins, lorsque deux opérateurs sont théoriquement commutatifs, on ne peut être certain de l'ordre dans lequel ils seront finalement exécutés. Par exemple, une expression telle que *a*+*b*+*c* pourra aussi bien être calculée en ajoutant *c* à la somme de *a* et *b*, qu'en ajoutant *a* à la somme de *b* et *c*. Même l'emploi de parenthèses dans ce cas ne suffit pas à « forcer » l'ordre des calculs. Notez bien qu'une telle remarque n'a d'importance que lorsque l'on cherche à maîtriser parfaitement les erreurs de calcul.

2.3 Comportement des opérateurs en cas d'opération impossible

Comme dans tous les langages, il existe trois circonstances où un opérateur ne peut pas fournir un résultat correct :

- *dépassement de capacité* : résultat de calcul trop grand (en valeur absolue) pour la capacité du type (il peut aussi se produire si une opération portant sur des entiers non signés fournit un résultat négatif) ;

- *sous-dépassement de capacité* : résultat de calcul trop petit (en valeur absolue) pour la capacité du type ; cette situation ne se présente que pour les types flottants ;
- tentative de *division par zéro*.

La norme de C++ se contente de dire que, dans ces circonstances, « le comportement du programme est indéterminé »[2]. En théorie, on peut donc aboutir :

- à un résultat faux ;
- à une valeur particulière servant conventionnellement à indiquer qu'un résultat n'est plus un nombre ou qu'il est infini : c'est ce qui se produit pour les flottants dans les implémentations qui utilisent les conventions dites IEEE ;
- à un arrêt du programme accompagné (peut-être) d'un message d'erreur ;
- ...

En pratique, cependant, on constate que :

- le dépassement de capacité des entiers n'est pas détecté et on se contente de conserver les bits les moins significatifs du résultat ;
- le dépassement de capacité des flottants conduit à la valeur *+INF* ou *-INF* dans les implémentations respectant les conventions IEEE, à un arrêt de l'exécution dans les autres ;
- le sous-dépassement de capacité des flottants conduit soit à un résultat nul, soit à un arrêt de l'exécution ;
- la tentative de division par zéro conduit à l'une des valeurs *+INF*, *-INF* ou *NaN (Not a Number*) dans les implémentations respectant les conventions IEEE, à un arrêt de l'exécution dans les autres.

En Java

Le comportement est imposé par le langage : pas de détection des dépassements de capacité en entier, utilisation des conventions IEEE pour les flottants. Seule la tentative de division par zéro déclenche une « exception » qui peut éventuellement être interceptée par le programme (ce qui n'est pas le cas en C++, malgré l'existence d'un mécanisme de gestion des exceptions et l'existence des exceptions standards *overflow_error* et *underflow_error* qui, en fait, ne sont pas utilisées par les opérateurs usuels).

2. Seul le dépassement de capacité de l'addition d'entiers non signés est défini.

3 Les conversions implicites pouvant intervenir dans un calcul d'expression

Comme nous l'avons déjà évoqué, les différents opérateurs arithmétiques ne sont définis que pour des opérandes de même type parmi *int* et *long* (et leurs variantes non signées), ainsi que *float*, *double* et *long double*. Mais, fort heureusement, C++ vous permet :

- de mélanger plusieurs types au sein d'une même expression ;
- d'utiliser les types *short* et *char* (avec leurs variantes non signées), ainsi que le type *bool*.

C'est ce que nous allons examiner ici. Pour faciliter le propos, nous examinerons tout d'abord les situations usuelles ne faisant pas intervenir les types non signés. Un paragraphe ultérieur fera ensuite le point sur ces cas peu usités ; vous pourrez éventuellement l'ignorer dans un premier temps.

3.1 Notion d'expression mixte

Comme nous l'avons dit, les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais C++ vous permet d'écrire ce que l'on nomme des « expressions mixtes » dans lesquelles interviennent des opérandes de types différents. Voici un exemple d'expression autorisée, dans laquelle *n* et *p* sont supposés de type *int*, tandis que *x* est supposé de type *float* :

```
n * x + p
```

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit *n*x*. Pour que cela soit possible, il va mettre en place des instructions de conversion de la valeur de *n* dans le type *float* (car on considère que ce type *float* permet de représenter à peu près convenablement une valeur entière, l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type *float* et elle fournira un résultat de type *float*.

Pour l'addition, on se retrouve à nouveau en présence de deux opérandes de types différents (*float* et *int*). Le même mécanisme sera mis en place, et le résultat final sera de type *float*.

Remarque

Attention, le compilateur ne peut que prévoir les instructions de conversion (qui seront donc exécutées en même temps que les autres instructions du programme) ; il ne peut pas effectuer lui-même la conversion d'une valeur que généralement il ne peut pas connaître.

3.2 Les conversions usuelles d'ajustement de type

Une conversion telle que *int -> float* se nomme une « conversion d'ajustement de type ». Une telle conversion ne peut se faire que suivant une hiérarchie qui permet de ne pas dénatu-

rer la valeur initiale (on dit parfois que de telles conversions respectent l'intégrité des données), à savoir :

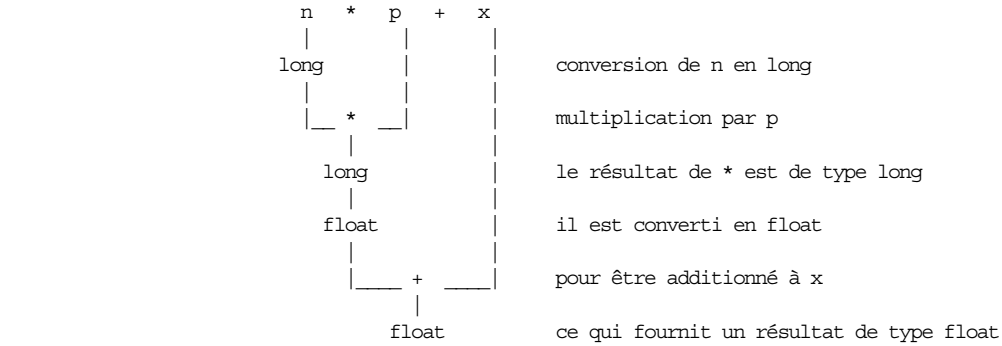
```
int -> long -> long long -> float -> double -> long double
```

On peut bien sûr convertir directement un *int* en *double* ; en revanche, on ne pourra pas convertir un *double* en *float* ou en *int*.

Notez que le choix des conversions à mettre en œuvre est effectué en considérant un à un les opérandes concernés et non pas l'expression de façon globale. Par exemple, si *n* est de type *int*, *p* de type *long* et *x* de type *float*, l'expression :

```
n * p + x
```

sera évaluée suivant ce schéma :



3.3 Les promotions numériques usuelles

3.3.1 Généralités

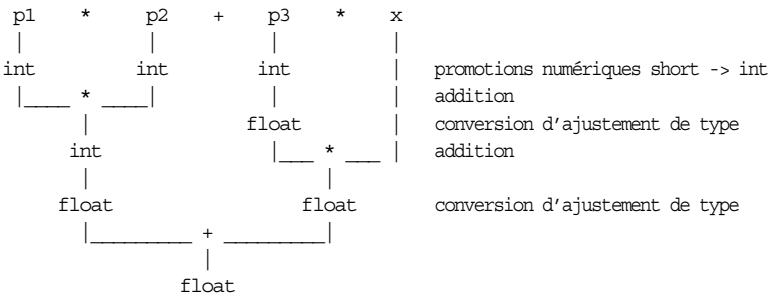
Les conversions d'ajustement de type ne suffisent pas à régler tous les cas. En effet, comme nous l'avons déjà dit, les opérateurs arithmétiques ne sont théoriquement pas définis pour le type *short* (bien qu'il s'agisse d'un vrai type numérique), ni pour les types *char* et *bool* qui peuvent cependant apparaître, eux aussi, dans des expressions arithmétiques.

En fait, C++ prévoit tout simplement que toute valeur de l'un de ces trois types apparaissant dans une expression est d'abord convertie en *int*, et cela sans considérer les types des éventuels autres opérandes. On parle alors, dans ce cas, de « promotions numériques » (ou encore de « conversions systématiques »).

Par exemple, si *p1*, *p2* et *p3* sont de type *short* et *x* de type *float*, l'expression :

```
p1 * p2 + p3 * x
```

est évaluée comme l'indique le schéma de la page suivante.



Notez bien que les valeurs des trois variables de type *short* sont d'abord soumises à la promotion numérique *short -> int* ; après quoi, on applique les mêmes règles que précédemment.

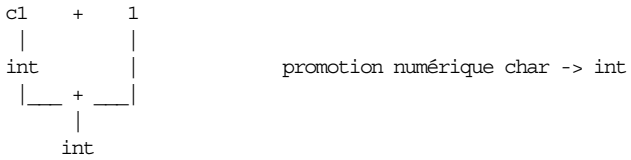
3.3.2 Cas du type *char*

A priori, vous pouvez être surpris de l'existence d'une conversion systématique (promotion numérique) de *char* en *int* et vous interroger sur sa signification. En fait, il ne s'agit que d'une question de point de vue. En effet, une valeur de type caractère peut être considérée de deux façons :

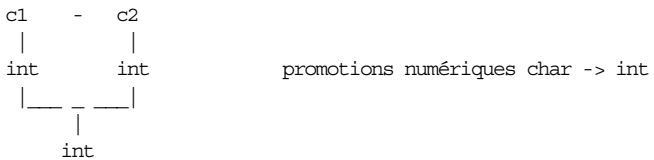
- comme le caractère concerné : *a*, *Z*, fin de ligne ;
- comme le code de ce caractère, c'est-à-dire un motif de 8 bits ; or à ce dernier on peut toujours faire correspondre un nombre entier (le nombre qui, codé en binaire, fournit le motif en question) ; par exemple, dans le code ASCII, le caractère *E* est représenté par le motif binaire *01000101*, auquel on peut faire correspondre le nombre *65*.

Effectivement, on peut dire qu'en quelque sorte C++ confond facilement un caractère avec la valeur (entier) du code qui le représente. Notez bien que, comme toutes les machines n'emploient pas le même code pour les caractères, l'entier associé à un caractère donné ne sera pas toujours le même.

Voici quelques exemples d'évaluation d'expressions, dans lesquels on suppose que *c1* et *c2* sont de type *char*, tandis que *n* est de type *int*.



L'expression *c1+1* fournit donc un résultat de type *int*, correspondant à la valeur du code du caractère contenu dans *c1* augmenté d'une unité.



Ici, bien que les deux opérandes soient de type *char*, il y a quand même conversion préalable de leurs valeurs en *int* (promotions numériques).

3.3.3 Cas du type bool

Le type *bool* a été introduit tardivement dans C++ (il n'existe pas en C). Auparavant, les valeurs correspondantes (*true* et *false*) étaient simplement représentées par un entier (1 ou 0). Pour préserver la compatibilité avec d'anciens codes, la norme a prévu une conversion systématique (promotion numérique) de *bool* en *int*. Voici un exemple :

```
bool ok = true ;
   .....
cout << ok + 2 ;    // affiche 3
   .....
ok = false ;
cout << ok + 2 ;    // affiche 2
```

Certains opérateurs (relationnels, logiques) fournissent un résultat de type *bool*. Dans les anciennes versions de C++, ainsi qu'en langage C, il fournissent une valeur entière 0 ou 1. Là encore, la conversion implicite évoquée assure la compatibilité.

3.4 Les conversions en présence de types non signés

Nous examinons ici les situations peu usuelles où apparaissent dans une expression des opérandes de type non signé. Ce paragraphe peut être ignoré dans un premier temps.

3.4.1 Cas des entiers

Tant qu'une expression ne mélange pas des types entiers signés et des types entiers non signés, les choses restent naturelles. Il suffit simplement de compléter les conversions (promotions numériques et conversions d'ajustement de type) *short -> int -> long* par les conversions *unsigned short -> unsigned int -> unsigned long*, mais aucun problème nouveau ne se pose (on peut toujours obtenir des dépassements de capacité qui ne seront pas détectés[3]).

En revanche, le mélange des types signés et des types non signés, bien qu'il soit fortement déconseillé, est accepté par le langage ; mais il conduit à mettre en place des conversions (généralement de signé vers non signé) n'ayant guère de sens et ne respectant pas, de toute façon, l'intégrité des données (que pourrait d'ailleurs bien valoir -5 converti en non signé ?). Une telle liberté est donc à proscrire. À simple titre indicatif, sachez que les conversions prévues

3. Attention, cette fois, une simple expression telle que *n-p* (où *n* et *p* sont de type non signé) peut conduire à un dépassement de capacité dès que la valeur de *n* est inférieure à celle de *p*.

par la norme, dans ce cas, se contentent de préserver le motif binaire (par exemple, la conversion de *signed int* en *unsigned int* revient à conserver tel quel le motif binaire concerné)[4].

3.4.2 Cas des caractères

Le type *char* peut, lui aussi, recevoir un attribut de signe ; en outre, la norme ne dit pas si *char* (tout court) correspond à *unsigned char* ou à *signed char* (alors que, par défaut, tous les types entiers sont considérés comme signés).

L'attribut de signe d'une variable de type caractère n'a aucune incidence sur la manière dont un caractère donné est représenté (codé) : il n'y a qu'un seul jeu de codes sur 8 bits, soit 256 combinaisons possibles en comptant le *\0*. En revanche, cet attribut va intervenir dès lors qu'on s'intéresse à la valeur numérique associée au caractère et non plus au caractère lui-même. C'est le cas, par exemple, dans des situation telles que :

```
char c ;
cout << c+1 ;
```

Pour fixer les idées, supposons, ici encore, que les entiers de type *int* sont représentés sur 16 bits et voyons comment se déroule précisément la promotion numérique *char -> int*, nécessaire à l'évaluation de l'expression *c+1*.

- Si le caractère n'est pas signé, on ajoute à gauche de son code 8 bits égaux à 0. Par exemple :

  01001110 devient 0000000001001110
  11011001 devient 0000000011011001

- Si le caractère est signé, on ajoute à gauche de son code 8 bits égaux au premier bit du code du caractère. Par exemple :

  01001110 devient 0000000001001110
  11011001 devient 1111111111011001

Ainsi, l'instruction d'affichage précédente utilisera pour la valeur de *c* convertie en *int* (la valeur affichée étant cette dernière, augmentée de *1*) :

- une valeur comprise entre *-128* et *127* si *c* est de type *unsigned char* ;
- une valeur comprise entre *0* et *255* si *c* est de type *signed char*.

On n'oubliera pas que, si *c* est de type *char* (tout court), celui-ci peut correspondre à un *signed char* ou à un *unsigned char* suivant l'implémentation.

Notez qu'aucun problème de ce type n'apparaît dans une instruction telle que :

```
cout << c ;
```

En Java

Il existe également une promotion numérique de *char* en entier ; tout se passe comme si le type *char* était non signé (rappelons qu'il n'existe qu'un seul tye *char*).

4. On trouvera une étude détaillée de ces possibilités dans *Le guide complet du langage C* du même auteur, chez le même éditeur.

4 Les opérateurs relationnels

Comme tout langage, C++ permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple :

```
2 * a > b + 5
```

Le résultat de la comparaison est une valeur booléenne prenant l'une des deux valeurs *true* ou *false*.

Les expressions comparées pourront être d'un type de base quelconque et elles seront soumises aux règles de conversion présentées dans le paragraphe précédent. Cela signifie qu'au bout du compte on ne sera amené à comparer que des expressions de type numérique, même si dans les opérandes figurent des valeurs de type *short*, *char* ou *bool* (*true*>*false* sera vraie).

Voici la liste des opérateurs relationnels existant en C++ :

Opérateur	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

Les opérateurs relationnels

Remarquez bien la notation (==) de l'opérateur d'égalité, le signe = étant réservé aux affectations.

En ce qui concerne leur priorité, il faut savoir que les quatre premiers opérateurs (<, <=, > et >=) sont de même priorité. Les deux derniers (== et *!*=) possèdent également la même priorité, mais celle-ci est inférieure à celle des précédents. Ainsi, l'expression :

```
a < b == c < d
```

est interprétée comme :

```
(a < b) == (c < d)
```

ce qui, en C++, a effectivement une signification, étant donné que les expressions $a<b$ et $c<d$ sont, finalement, des quantités entières. En fait, cette expression prendra la valeur *1* lorsque les relations $a < b$ et $c < d$ auront toutes les deux la même valeur, c'est-à-dire soit lorsqu'elles seront toutes les deux vraies, soit lorsqu'elles seront toutes les deux fausses. Elle prendra la valeur *0* dans le cas contraire.

D'autre part, ces opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques. Cela permet souvent d'éviter certaines parenthèses dans des expressions.

Ainsi :

```
x + y < a + 2
```

est équivalent à :

```
( x + y ) < ( a + 2 )
```

Remarque

Une erreur courante consiste à utiliser l'opérateur = à la place de ==. Elle peut conduire à du code accepté par le compilateur, mais n'aboutissant pas au résultat voulu. Voyez cet exemple :

```
if (a = b) // ici, on a utilisé = au lieu de ==
{ ..... }
```

L'expression *a=b* affecte la valeur de *b* à *a* et sa valeur est celle de *a* après affectation (donc, celle de *b*). Nous verrons que l'instruction *if* convertit alors cette valeur numérique en un booléen suivant la règle : non nul devient vrai, nul devient faux. Ainsi, le bloc suivant *if* est-il exécuté si la valeur de *b* est non nulle ! À noter que certains compilateurs vous fournissent un avertissement en cas d'utilisation douteuse de l'opérateur =.

Cas des comparaisons de caractères

Compte tenu des règles de conversion, **une comparaison peut porter sur deux caractères**. Bien entendu, la comparaison d'égalité ne pose pas de problème particulier ; par exemple (*c1* et *c2* étant de type *char*) :

- *c1* == *c2* sera vraie si *c1* et *c2* ont la même valeur, c'est-à-dire si *c1* et *c2* contiennent des caractères de même code, donc si *c1* et *c2* contiennent le même caractère ;
- *c1* == '*e*' sera vraie si le code de *c1* est égal au code de '*e*', donc si *c1* contient le caractère *e*.

Autrement dit, dans ces circonstances, l'existence d'une conversion *char* --> *int* n'a guère d'influence. En revanche, pour les comparaisons d'inégalité, quelques précisions s'imposent. En effet, par exemple *c1* < *c2* sera vraie si le code du caractère de *c1* a une valeur inférieure au code du caractère de *c2*. Le résultat d'une telle comparaison peut donc varier suivant le codage employé (et, éventuellement, l'attribut signé ou non signé du type *char* employé). Cependant, il faut savoir que, quel que soit ce codage :

- l'ordre alphabétique est respecté pour les minuscules d'une part, pour les majuscules d'autre part ; on a toujours '*a*' < '*c*', '*C*' < '*S*'...
- les chiffres sont classés par ordre naturel ; on a toujours '*2*' < '*5*'...

En revanche, aucune hypothèse ne peut être faite sur les places relatives des chiffres, des majuscules et des minuscules, pas plus que sur la place des caractères accentués (lorsqu'ils existent) par rapport aux autres caractères, laquelle peut varier suivant l'attribut de signe !