Alain Gibaud

Programmer en C++

Des premiers pas à la maîtrise de C++23

2e édition

ellipses

Première partie

Préambule

1 | À propos de ce livre

1.1 À qui s'adresse-t-il ?

Comme dans sa première édition, ce livre a été conçu pour accompagner les étudiants d'université ou d'école d'ingénieurs dès le début de leur cursus. Mais il conviendra aussi à toute personne désirant compléter ou actualiser ses compétences de programmeur en C++. Par le traitement de sujets de difficulté croissante, il se propose de vous guider de manière progressive vers la maîtrise de ce langage très puissant qu'est C++, dans sa plus récente version normalisée : C++23.

1.2 Que contient-il ?

Cet ouvrage est consacré au langage C++. Toutefois, ce dernier est construit sur le langage C, et la relation qui existe entre ces langages est davantage qu'une simple filiation : C++ est un *surensemble* de C. Le choix qui a été fait dans cette édition est de traiter l'héritage du langage C dans une première partie intitulée « *Le socle de C++* ». On y présente uniquement les éléments du langage C repris par C++, en excluant ceux qui sont inutiles, voire nuisibles.

Il est impossible de décrire un langage de programmation d'une manière à la fois détaillée et accessible aux débutants. Il est en effet fréquent qu'une description approfondie fasse référence à des informations ou à des concepts qui n'ont pas encore été présentés. Pour cette raison, la présentation du langage C++ a été scindée en deux parties.

La première, « *Débuter avec le langage C++* », contient une description du langage largement suffisante pour écrire des programmes intéressants. Elle débute par une description de ce qu'est la *programmation par objets*, et des concepts associés à ce type d'approche. L'objectif de cette présentation est de montrer que la programmation par objets ne s'appuie pas uniquement sur l'usage de « boîtes à outils » logicielles : c'est aussi une démarche d'analyse des problèmes permettant de concevoir l'architecture de leurs solutions.

Une introduction progressive des fonctionnalités de base de C++ vient ensuite. Elle s'appuie sur des exemples d'abord rudimentaires, puis plus élaborés au fil des chapitres. « Progressive » signifie ici que les éléments du langage sont présentés dans un ordre logique, en commençant par les plus primitifs. Chaque sujet abordé s'appuie ainsi sur les sujets précédents afin de minimiser les risques d'incompréhension.

La seconde partie intitulée « *Approfondir le langage C++* » traite de sujets un peu plus ardus, et même parfois difficiles si l'on s'intéresse aux aspects les plus avancés. Il faut s'attendre à ce que certaines de ces pages nécessitent plusieurs lectures pour être assimilées. Mais ne vous inquiétez pas, car vous n'êtes pas obligés de tout savoir sur le langage C++ pour écrire des programmes intéressants et très performants. Certains chapitres portent le même nom dans la première et la seconde partie, car traitent du même sujet, mais à des niveaux de détail différents.

1.3 Les exemples

Apprendre un langage de programmation peut être frustrant si chaque point abordé n'est pas illustré par un exemple immédiatement utilisable. Cependant, même simples, les premiers exemples sont difficiles à introduire car il n'est pas possible de présenter simultanément les connaissances requises pour les comprendre. On est donc parfois amené à présenter du code comportant des

instructions ou utilisant des concepts qui n'ont pas encore été introduits. Dans ce cas, ou lorsque le sujet est difficile, le texte comporte de très nombreux renvois (comportant numéros de sections et de pages) permettant d'approfondir les points évoqués.

Les exemples courts ont l'avantage d'illustrer de manière facilement compréhensible les fonctionnalités présentées. Toutefois, ils ne permettent généralement pas d'en démontrer l'intérêt concret. C'est pour cela que ce livre contient beaucoup d'exemples plus ambitieux qui replacent le sujet dans un contexte pratique. Ceux-ci sont plus difficiles à comprendre, car ils tiennent compte de détails qui, bien qu'ils débordent parfois du cadre du sujet traité, rendent le code réaliste.

Il existe dans ce livre trois sortes d'exemples, tous imprimés avec une police à chasse fixe, mais avec une présentation spécifique :

(a) Un *fragment de code* est un court extrait destiné à illustrer un point précis. Les fragments ne sont pas utilisables en l'état, car ils sont incomplets : il leur manque généralement un contexte, par exemple des déclarations ou initialisations de variables. Voici un exemple de fragment de code :

```
/* Schéma d'HORNER */
y = ((a * x) + b ) * x + c  ;
```

Dans tous les exemples, les commentaires avec points de suspension tels que :

```
// ...
/* ... */
// ... suite de l'exemple précédent
```

indiquent l'emplacement d'un code qui n'est pas en relation directe avec le sujet, et qui a été volontairement omis pour simplifier la présentation.

(b) Les *fonctions* (ou les *classes*) constituent des outils presque complets, et sont donc généralement utilisables dans un programme par simple copier-coller. Pour des raisons de simplicité et de brièveté, certains éléments sont cependant omis dans les exemples présentés : c'est généralement le cas des inclusions de fichiers, pourtant nécessaires à la compilation de la fonction. Par exemple, une fonction qui réalise des entrées-sorties en langage C++ devrait être précédée de l'inclusion du fichier d'en-tête `<iostream>`, mais ce n'est pas forcément le cas dans ce livre afin d'éviter de le rendre encore plus volumineux.

Voici un exemple de fonction :

```
1  // ...
2  /*
3    Calcule la valeur d'un polynôme du second degré,
4    de coefficients a,b,c au point x
5  */
6  double poly2(double x, double a, double b, double c)
7  {
8    double y ;
9    // Schéma d'HORNER
10   y = ((a * x) + b ) * x + c  ;
11   return y ;
12 }
13 // ...
```

(c) Les *programmes* sont des exemples autonomes, car ils sont utilisables tels quels, par simple copie. Certains programmes sont parfois présentés en plusieurs parties pour pouvoir introduire des explications intermédiaires. Dans ce cas, un commentaire `// ...` ou `/* ... */` montre comme d'habitude que du code a été omis. Voici un programme très court qui met en œuvre la fonction précédente.

Programme poly2.cpp

```
1
2  #include <iostream>
3  /*
4    Calcule la valeur d'un polynome du second degré,
5    de coefficients a,b,c au point x
6  */
7  double poly2(double x, double a, double b, double c)
8  {
9    double y ;
10   // Schéma d'HORNER
11   y = ((a * x) + b ) * x + c  ;
12   return y ;
13 }
14
15 int main()
16 {
17   double x,a,b,c ;
18   std::cout << "Entrez a b c puis x : " << std::endl  ;
19   std::cin >> a >> b >> c >> x ;
20   std::cout << "Valeur du polynome = " <<  poly2(x,a,b,c) << std::endl ;
21   return 0 ;
22 }
```

(d) Finalement, le texte affiché par un programme sera représenté dans ce livre avec une barre latérale de la façon suivante :

```
Valeur du polynome = 23.56
```

Tous les exemples de ce livre (et d'autres qui n'ont pas pu y figurer faute de place) peuvent être téléchargés sous la forme d'une archive depuis le site de l'éditeur à l'adresse suivante :

`https://www.editions-ellipses.fr/`

Chaque exemple est contenu dans un répertoire qui comporte les codes sources et les ressources nécessaires à leur mise en œuvre :

- Un script `bash` nommé `build.sh` pour construire les applications en ligne de commande sous Linux.
- Un fichier projet (extension `.pro`) utilisable par `creator`, l'excellent IDE venant avec la bibliothèque **Qt**. Cette bibliothèque est gratuite pour les usages non lucratifs. Utiliser `creator` est certainement la meilleure option, car cet IDE fonctionne aussi bien sous Linux et MacOS que sous Windows.

Toutes remarques et suggestions concernant ce livre peuvent être directement adressées à l'auteur par e-mail :

`progcpp23.ag@free.fr`

1.4 Pictogrammes utilisés dans le texte

Le texte de ce livre utilise plusieurs sortes de pictogrammes permettant d'attirer l'attention sur un point particulier.

 Introduit des explications concernant le code qui précède.

 Introduit un résumé qui rappelle les principaux points à assimiler.

 Introduit un renvoi vers la partie avancée du livre. Ceci permet au lecteur d'approfondir immédiatement le sujet en train d'être traité, s'il le désire.

 Introduit des explications techniques concernant l'implantation d'une fonctionnalité particulière. Lire ces explications permet d'avoir une connaissance plus profonde du langage.

 Signale une difficulté particulière, ou un piège à éviter.

 Signale une fonctionnalité du langage C qu'il est recommandé de ne pas utiliser en C++.

2 | À propos du langage C++

Le langage C++ est né des réflexions de Bjarne Stroustrup, qui a eu l'idée d'allier l'efficacité du langage C à la puissance d'expression permise par la programmation par objets. Le langage C a été créé dans les années 70 pour réaliser des logiciels système. Il s'agit d'un excellent langage, dépouillé mais performant, qui a eu et continue d'avoir un grand succès dans son domaine d'application. C'est pour cette raison que C++ a donc été conçu (au milieu des années 80) comme un surensemble du langage C, ce qui a permis de conserver le bénéfice de l'énorme logithèque de ce dernier.

Au fil des années, le langage a été doté de nombreuses fonctionnalités importantes : *l'héritage multiple*, la gestion des *exceptions* et surtout la *généricité*, qui a radicalement changé la manière de l'utiliser, et qui l'a rendu particulièrement extensible.

À cause de sa genèse et des similitudes syntaxiques, C++ est considéré par ceux qui ne le connaissent pas comme un « super C », mais c'est une erreur. Non seulement le C++ initial est très différent de C, mais le C++ moderne (c'est-à-dire à partir de C++11) permet une approche de la programmation très différente de celle que proposait son prédécesseur.

On dit parfois que C++ est un langage *multiparadigmes*, ce qui signifie qu'il supporte plusieurs façons d'aborder les problèmes et de créer les solutions correspondantes : il est à la fois *procédural* [1] (comme le C), *à objets* [2], et *générique* [3]. Ces différentes façons de voir les choses peuvent être combinées selon le type de problème à résoudre : on peut ainsi écrire des programmes dans un style *procédural pur*, ou qui seraient *à objets et génériques*, ou *procéduraux et génériques*, ou enfin combiner les trois paradigmes, ce qui est la possibilité la plus utilisée. Enfin, grâce à son approche de la généricité, C++ permet une certaine forme de *métaprogrammation* [4].

C++ est un langage évoluant en permanence, et son comité de normalisation est très actif. Les formes initiales du langage s'appellent C++98 et C++03. Les versions suivantes constituent le *C++ moderne* : il s'agit de C++11, C++14, C++17, C++20 et C++23 [5]. Cette dernière norme n'était d'ailleurs pas complètement supportée par tous les compilateurs lors de la réalisation de ce livre.

Malgré les différences, il y a un domaine dans lequel la philosophie du C++ est identique à celle du C : chaque fonctionnalité est conçue pour favoriser l'efficacité, et vous ne payez pas pour une fonctionnalité que vous n'utilisez pas. Pour cette raison, C++ est bien adapté aux projets complexes pour lesquels d'excellentes performances sont requises. C'est le cas par exemple des programmes et environnements graphiques, que nous utilisons tous les jours sur nos ordinateurs.

C++ est un langage très puissant dont les fonctionnalités de base sont abordables, mais il est tellement riche et vaste qu'il y a toujours des subtilités à y découvrir, même pour un programmeur expérimenté. C'est pourquoi, si à l'issue de la lecture de ce livre (ou de tout autre ouvrage traitant du même sujet), vous pensez avoir tout compris du C++, il est probable qu'on vous ait mal expliqué.

Bonne lecture !

1. La programmation procédurale est basée sur des assemblages de procédures (aussi appelées sous-programmes ou fonctions) qui implantent des algorithmes.
2. Ce sujet est traité au chapitre 18.
3. Ce sujet est traité au chapitre 27.
4. La métaprogrammation est une manière de programmer permettant dans certains cas d'obtenir des résultats à la compilation, donc sans exécuter de code. Ce sujet est traité au chapitre 56.
5. Le nom de chaque norme correspond à son année de publication. Il faut aussi savoir qu'il peut s'écouler plusieurs années entre la publication d'une norme et son support complet par les compilateurs. La prochaine version du langage s'appellera C++26.

Deuxième partie

Le socle de C++

3 | Composition des programmes C++

3.1 La notion de fonction en C++

Un programme rédigé en langage C++ comporte au moins une *fonction*. Une fonction est un groupe d'instructions qui réalisent un traitement à partir de données qu'on lui fournit. L'effet de ce traitement est généralement la production d'un résultat, comme pour une fonction mathématique. Par exemple, la fonction définie en mathématiques comme

$$f(x, y) \rightarrow x + y$$

reçoit deux nombres, en calcule la somme et retourne cette valeur. Il s'agit là du genre de fonction le plus classique, mais il en existe d'autres :

- Fonction sans paramètre : Il s'agit d'une fonction qui produit un résultat sans qu'on lui fournisse explicitement de données,
$$heure() \rightarrow heureCourante$$
- Fonction sans résultat : Par extension du concept de fonction, le langage C++ permet aussi de définir des fonctions qui ne produisent pas de résultat. On dit dans ce cas que leur résultat est *vide* (ou `void`). Ceci pourrait être représenté comme :
$$g(x) \rightarrow$$
Une fonction à résultat vide peut être très utile en programmation. En effet, ses instructions, (comme d'ailleurs celles de toutes les autres fonctions) peuvent produire un *effet*[1] : elles peuvent par exemple lancer une fusée, éteindre une lampe, afficher un message ou plus simplement modifier l'état de la mémoire. La notion de fonction en programmation (et particulièrement en langage C++) est donc assez différente de ce qu'elle est en mathématiques.
- Fonction membre : Cette sorte de fonction est liée à la programmation par objet. Les fonctions membres (ou méthodes) existent donc en C++, mais pas en C. Leur spécificité est d'être attachées au traitement des objets. On n'en dira pas davantage dans cette partie du livre, car ce sujet sera traité à partir du chapitre 18 (page 189), qui introduit ce type de programmation.

3.1.1 Invocation des fonctions

Bien entendu, pour qu'une fonction soit utile, il faut qu'elle soit *appelée*[2], ce qui déclenche son exécution. C'est l'interaction entre les différentes fonctions qui composent le programme qui constitue l'activité de ce programme.
Le schéma de la figure 3.1 représente un programme très simple composé de 5 fonctions. Dans ce schéma, les flèches représentent des relations d'appel : la fonction `main` peut appeler les fonctions `b` et `d`. On notera que `d` n'appelle aucune autre fonction, mais que `b` peut appeler `a` qui peut appeler `c`, qui elle-même peut appeler `d`. On remarque donc que `d` peut être appelée de façon *directe* ou *indirecte*. Remarquez aussi que `a` est susceptible de s'appeler elle-même : il s'agit d'un *appel récursif*. Ce genre d'appel est traité au chapitre 16, page 159.

3.2 La fonction `main`

Comme la totalité du code constituant un programme se trouve dans des fonctions, il est nécessaire qu'une de ces fonctions soit lancée automatiquement et appelle les autres, directement ou indirectement. Il s'agit d'une fonction conventionnellement nommée `main`, et elle peut être la seule

1. On peut aussi utiliser le terme *effet de bord*.
2. On utilise aussi le terme *invocation* pour désigner un appel de fonction.

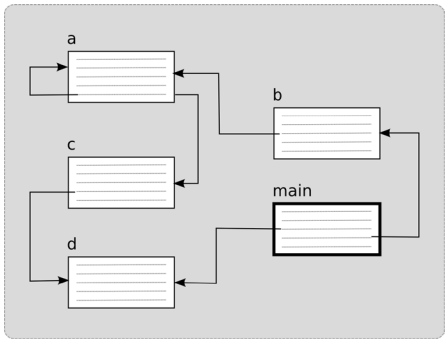

Figure 3.1 – Un programme simple composé de cinq fonctions.

du programme. Voici donc ce que vous devez retenir : si un programme écrit en langage C++ (ou C) ne comporte pas de fonction **main**, l'éditeur de liens [3] refusera de générer le fichier exécutable.

Toutes les fonctions **main** sont construites sur le modèle de celle qui suit. Pour l'instant, je ne donnerai pas d'autre explication sur les fonctions (qui seront traitées plus en détail au chapitre 6), car nous avons beaucoup d'autres choses à voir auparavant. La fonction **main** nous servira donc uniquement à accueillir les différents exemples, au fur et à mesure de leur présentation.

NeFaitRien/ne-fait-rien.cpp

```
1
2 #include <iostream>
3 // --------------- Exemple de fonction main vide (commentaire style C++)
4 int main()
5 {
6   /* Le code de la fonction doit se trouver ici (commentaire style C) */
7   return 0 ;
8 }
```

La fonction **main** débute ici à la ligne 4, et se termine à la ligne 8. Elle ne comporte aucune instruction, mais les lignes 3 et 6 comportent des *commentaires*. Ceux-ci n'ont aucun effet sur l'exécution des programmes, mais ils sont extrêmement importants, car ils permettent d'éclaircir ou de rappeler un point qui rendrait le programme difficile à comprendre. N'hésitez jamais à ajouter un commentaire, même si vous êtes la seule personne à lire votre programme.
Pour l'instant, ne vous préoccupez pas du reste du code : contentez-vous de recopier cet exemple tel quel avec votre éditeur de texte favori. Enregistrez ce code source dans le fichier **ne-fait-rien.cpp**, puis générez le programme exécutable. Par exemple, si vous travaillez en mode texte dans une console Linux, vous devez taper [4] :

```
g++ -o ne-fait-rien ne-fait-rien.cpp
```

Exécutez maintenant le programme en tapant :

```
./ne-fait-rien
```

S'il ne se passe rien, c'est plutôt bon signe. Bravo ! Vous venez d'exécuter votre premier programme en langage C++, celui qui ne fait rien.

Vous trouverez davantage des renseignements sur la fonction **main** au chapitre 10, page 113.

3. Si ce terme ne vous dit rien, lisez l'annexe A.1.2.3, page 799.
4. Les commandes de compilation/édition de liens sont décrites en annexe A.2, page 805.

4 | Les données et leur type

Ce qu'on appelle *donnée* dans le contexte des langages de programmation n'est autre que la « matière » sur laquelle travaillent les programmes. C'est pourquoi, avant de parler des outils mis à notre disposition par un langage de programmation, il faut préciser les concepts en relation avec les données.

4.1 Variables et constantes

Une donnée peut être une *constante* ou une *variable*. Ces deux termes rappellent de façon claire que la valeur d'une constante ne peut pas être modifiée, contrairement à celle d'une variable. Les variables sont désignées par un nom, tel que `hauteur` ou `xflr13`. Dans le vocabulaire informatique, on trouve souvent les termes *identificateur* ou *identifiant*, comme synonymes de nom. Les constantes n'ont pas de nom : elles peuvent être des valeurs intermédiaires résultant d'un calcul, ou des *littéraux* tels que `12.5` ou `"Bonjour"`.

4.2 À quoi sert une déclaration ?

La très grande majorité des langages compilés (tels C ou C++) exigent que les variables soient déclarées avant d'être utilisées. Une déclaration ressemble à une présentation : *« Compilateur, je te présente* `x`*, c'est une variable contenant une donnée de type entier »*. Grâce à cela, le compilateur pourra entreprendre les actions aboutissant à la création de la variable. Il saura aussi comment la manipuler lorsqu'il rencontrera l'identificateur `x`.

4.3 Qu'est-ce qu'un type ?

Les variables et constantes ont un type. Si vous n'êtes pas familier avec ce concept, lisez cette section. Sinon, vous pouvez passer directement aux sections suivantes qui décrivent les différents types disponibles en langage C++.

L'annexe A.1 (page 789) consacrée à la structure et au fonctionnement des ordinateurs montre que toutes les informations stockées en mémoire sont représentées de la même manière [1]. Si l'on s'intéresse à un octet particulier pris au hasard, il est absolument impossible de dire ce qu'il représente, car il ne comporte aucune information pouvant nous renseigner.

Or, selon que cet octet contribue à représenter une adresse, un entier, un réel ou autre chose, le programme devra le manipuler de différentes façons. Par exemple, la façon de multiplier deux réels est différente de celle qui permet de multiplier deux entiers.

L'information décrivant la nature d'une entité informatique s'appelle un *type*. Connaître le type d'une entité peut être utile au moment de l'exécution du programme, mais pour les langages compilés, c'est surtout lors de la compilation que le type est exploité. Il permet au compilateur de :

(a) faire en sorte que la place nécessaire et suffisante pour stocker chaque entité soit réservée. Corollairement, ceci lui permet d'associer à chaque identificateur utilisé (`x`, `y`, `masse`, `vitesse`, `prix`, etc.) une *adresse* en mémoire.

(b) fabriquer le code machine manipulant correctement cette entité. C'est tout de même le point le plus important !

1. Avec les technologies actuelles, il s'agit d'une représentation binaire.

(c) vérifier que le programmeur utilise bien les données de façon cohérente. Par exemple, l'addition de deux nombres réels semble cohérente. Par contre, additionner un nombre réel avec un texte est probablement une erreur.


Le dernier point cité est très important : la notion de type a pour effet *d'introduire du sens* dans vos programmes, ce qui permet au compilateur d'en vérifier (dans une certaine mesure) la cohérence. Dans le jargon informatique, on utilise le terme *sémantique* pour désigner le sens attaché à un type.

La sémantique des types (quels qu'ils soient) peut être définie par :

(a) le langage (c'est le cas des types primitifs disponibles en C ou C++),

(b) la bibliothèque standard du langage (en C++ uniquement),

(c) l'utilisateur (également en C++).


La sémantique des types permet de détecter automatiquement beaucoup d'erreurs de programmation, mais ne vous faites pas d'illusions : il existe tant de façons de se tromper qu'il reste très facile d'écrire des programmes faux. Par conséquent, le fait qu'un programme ait été compilé sans message d'erreur n'est absolument pas une garantie que ce programme soit correct, malgré l'apport du typage.

4.4 Les types primitifs

On dira qu'une entité est d'un type *fondamental* (ou *primitif simple*) si son type ne s'appuie pas sur l'existence d'un autre type. Il existe en C++ trois sortes de *types primitifs simples* : les *entiers*, les *réels* et les *booléens*. Il existe aussi des types *primitifs composés*, tels que les *pointeurs*, les *tableaux* et les *structures*. Le premier sera traité ici, mais les autres seront traités au chapitre 8, page 83.

Qu'elles soient d'un type simple ou composé, les entités primitives ont en commun de pouvoir être manipulées par des opérateurs prédéfinis du langage.


En C++, déclarer une variable d'un type *primitif* ne l'initialise pas, à moins que le programmeur le demande explicitement. Vous ne devez donc jamais supposer qu'une telle variable possède *par défaut* une valeur particulière [a]. En revanche, on verra qu'il est possible de garantir que tous les *objets* (que vous pouvez considérer comme des variables d'un type non primitif) soient initialisés.

a. Ceci est un héritage contestable du langage C.

4.4.1 Entiers

Les nombres entiers utilisés dans les ordinateurs ne représentent qu'approximativement l'ensemble mathématique des entiers, car on ne sait représenter qu'un nombre fini d'entiers. Le langage C++ utilise généralement la représentation de la machine cible. Comme presque tous les ordinateurs utilisent la même représentation, cela limite les surprises. Les entiers peuvent être signés (entiers relatifs) ou non signés (entiers naturels). Les entiers signés sont presque toujours représentés par un procédé nommé *complément à deux*. À partir de C++20 , ce procédé est même obligatoire.

Le langage ne propose pas moins de cinq types entiers qui s'appellent `char`, `short`, `int`, `long` et `long long`. Sans spécification explicite, ils sont tous signés, à l'exception de `char`, qui peut être signé ou non selon le compilateur. Remarquez toutefois que la plupart des compilateurs considèrent que les `char` sont signés par défaut.

Le tableau de la figure 4.1 indique les intervalles de valeurs représentables par des entiers codés sur N bits.

Type d'entier	Plus petite valeur	Plus grande valeur
Entiers non signés	0	$(2^N) - 1$
Entiers signés	$-(2^{N-1})$	$(2^{N-1}) - 1$

Figure 4.1 – Intervalle des valeurs représentables par des entiers codés sur N bits

Pour obtenir la version non signée de ces types, il suffit de préfixer leur nom par le mot-clé **`unsigned`**, comme dans **`unsigned long`**. La taille de l'entier résultant est la même que celle de l'entier signé correspondant. Pour mettre en évidence le fait qu'un entier est signé, vous pouvez aussi utiliser le préfixe **`signed`**, mais à part pour les **`char`**, ceci est redondant. Le langage impose quelques contraintes concernant la quantité de mémoire utilisée pour représenter les entiers : les **`char`** doivent être par définition codés sur un octet, et la taille minimale des **`int`** est de 2 octets. À part cela, les relations suivantes entre les différentes tailles doivent être vérifiées :

$$taille(\texttt{char}) \leq taille(\texttt{short}) \leq taille(\texttt{int}) \leq taille(\texttt{long}) \leq taille(\texttt{long long})$$

Pour fixer les idées, le tableau de la figure 4.2 indique les tailles typiquement utilisées, dans deux environnements classiques. Ces informations sont purement indicatives : il est bon de consulter les spécifications de votre compilateur avant d'utiliser un type plutôt qu'un autre. Cependant, il est probable que ces informations correspondent exactement à votre système.

Type C++ de l'entier	Ordinateur personnel	Microcontrôleur
`char`	1	1
`short` (ou `short int`)	2	2
`int`	4	2
`long` (ou `long int`)	4	4
`long long`	8	4

Figure 4.2 – Tailles typiques (en octets) des entiers supportés par les compilateurs C++, dans deux environnements très différents.

Avant de passer à un exemple, voici quelques précisions importantes concernant les types **`char`** et **`int`**, qui sont très utilisés.

- Le nom **`char`** est l'abréviation de « *character* ». Le choix de ce nom est sans doute une décision de conception malheureuse, car il fait invariablement croire aux débutants que ce type de donnée sert forcément à représenter des caractères. En fait, les **`char`** peuvent effectivement servir à cela, mais aussi à tout autre chose. Le **`char`** est simplement un entier, sur lequel vous pouvez faire exactement les mêmes opérations que sur les autres entiers. Sa seule spécificité est d'être toujours codé sur 8 bits [2], ce qui ne permet de représenter que 256 valeurs.
- Le type **`int`** est très souvent équivalent à **`short`** ou **`long`**, selon la machine. On peut donc se demander à quoi il sert, puisqu'il semble redondant. Ce type a été créé pour supporter de façon efficace les calculs en nombres entiers sur la machine cible. Pour cela, les variables **`int`** ont, en principe, exactement la même taille que les registres de données du processeur, ce qui assure théoriquement un fonctionnement optimal sur celui-ci. Ceci explique sa fréquente utilisation.

2. En théorie, les **`char`** pourraient cependant être codés sur plus de 8 bits, mais les architectures de processeurs exploitant cette possibilité sont extrêmement rares.

4.4.1.1 Déclaration des variables entières

Déclarer une variable de type entier est très simple. Cette déclaration sera de la forme :

type_de_la_variable nom_de_la_variable;

Par exemple :

```
int hauteur ;    /* hauteur est un int */
short vitesse ;  /* vitesse est un short */
```

4.4.1.2 Exemple d'utilisation de nombres entiers

Calculs entre nombres entiers

```
1
2  #include <iostream>
3
4  int main()
5  {
6    int a ;   // a est un int
7    int b,c ; // b et c sont aussi des int
8
9    b = 2 ;   // mettons la valeur 2 dans b
10   c = 10 ;  // et la valeur 10 dans c
11   a = b * c ;  // mettons le produit de b par c dans a
12
13   std::cout << a << std::endl ; // affichons la valeur de a
14
15   return 0 ;
16 }
```

Cet exemple est très court, mais comme c'est le premier vrai programme de ce livre il y a beaucoup à en dire. On va donc le décomposer pour l'expliquer.

```
  int a ;   // a est un int
  int b,c ; // b et c sont aussi des int
```

Ces deux lignes *définissent* les variables `a`,`b` et `c`. Elles demandent au compilateur de faire ce qu'il faut pour qu'un petit morceau de mémoire soit réservé pour chacune de ces variables. Elles précisent aussi le type de chaque variable, ce qui permet au compilateur de générer un code approprié à ce type.

D'un point de vue syntaxique, chaque instruction est terminée par un « ; », et vous avez le droit de mettre plusieurs instructions sur la même ligne, ce qui permet de mettre en évidence le fait qu'elles sont logiquement liées. Hors de cette situation, n'abusez pas des lignes de code longues, car elles rendent les programmes difficiles à lire.

```
  b = 2 ;      // mettons la valeur 2 dans b
  c = 10 ;     // et la valeur 10 dans c
  a = b * c ;  // mettons le produit de b par c dans a
```

Ces instructions réalisent des *affectations*. L'opérateur `=` sert ici à transférer la valeur qui est à sa droite dans la variable qui est à sa gauche. L'opérateur `=` sert donc à changer le contenu de cases en mémoire. L'opérateur `*` sert quant à lui à faire le produit des valeurs des variables qui l'entourent, qu'on appelle des *opérandes*. Finalement, vous l'avez compris, la variable `a` devrait contenir la valeur 20 à l'issue de l'exécution de ce morceau de code.

```
std::cout << a << std::endl ; // affichons la valeur de a
```

Cette instruction (qui est composée d'une seule expression) permet d'afficher sur l'écran la valeur de `a`, suivie d'un saut de ligne. Dans cette expression, l'opérateur `<<` permet d'écrire son opérande de droite sur le flux représenté par son opérande de gauche.
Comme l'*objet*[a] prédéfini `std::cout` représente l'écran, l'expression affiche `a` sur l'écran. La même opération est ensuite effectuée avec `std::endl`, qui représente un saut de ligne.

a. Pour l'instant, vous pouvez considérer les objets comme des variables ayant des propriétés un peu particulières.


Vous trouverez au chapitre 7 (page 79) des explications plus détaillées sur le fonctionnement des entrées-sorties sur flux utilisant les opérateurs `<<` et `>>` .

Exécutons maintenant ce programme grâce aux deux commandes qui suivent. La première permet de générer le fichier exécutable, alors que la seconde permet de lancer le programme. Le nom du fichier source est ici `calculs-entiers.cpp` .

```
g++ -o calculs-entiers calculs-entiers.cpp
./calculs-entiers
```

```
20
```

4.4.2 Entiers techniques

Il existe en C++ quelques types entiers qui ne sont pas dédiés au calcul numérique, mais à la gestion des structures de données. I s'agit de `size_t`, `ssize_t` et `ptrdiff_t`. Faute d'une terminologie officielle, on les appellera ici *entiers techniques*. Leur usage n'est pas absolument indispensable, mais recommandé si l'on veut éviter d'éventuels débordements. En outre, il faut savoir qu'ils sont utilisés par certains opérateurs (l'opérateur `sizeof` retourne par exemple un entier de type `size_t`), ou par des fonctions de la bibliothèque standard.

(a) `size_t` est un type non signé permettant de représenter la taille de n'importe quelle donnée. Il est aussi utilisable pour déclarer les indices permettant d'accéder aux éléments des conteneurs n'acceptant pas les indices négatifs. Le conteneur `vector` (section 32.1, page 397) fournit par exemple pour cet usage un type synonyme de `size_t`.

(b) `ssize_t` est un type signé jouant le même rôle que `size_t` pour les conteneurs manipulables via des indices positifs ou négatifs. Cela peut être le cas lorsqu'on manipule des tableaux au travers d'un pointeur pour obtenir des indices décalés (voir section 14.4, page 146).

(c) `ptrdiff_t` est un type signé permettant de représenter la différence de deux pointeurs. Reportez vous à la section 14.1 (page 141) consacrée à l'arithmétique des pointeurs pour en savoir davantage sur ce sujet.

Depuis C++23 , il est possible d'utiliser les littéraux de types `size_t` ou `ssize_t`. Ceci permet de supporter la déduction automatique de type, lorsqu'une donnée est déclarée avec `auto` (cette fonctionnalité est décrite en section 17.5, page 181). La table de la figure 4.3 indique les suffixes à utiliser.

Type désiré	Suffixe	Exemple
`ptrdiff_t`	*inexistant*	
`size_t`	*uz* ou *UZ*	`0uz`, `4UZ`
`ssize_t`	`z` ou `Z`	`0z`, `0xFFFFFFFFZ`

Figure 4.3 – Les différents suffixes permettant de préciser le type d'une constante entière technique.

4.4.3 Booléens

Le type `bool` permet en C++ de représenter les valeurs logiques. Les variables de ce type ne peuvent prendre que deux valeurs : `true` (*vrai*), ou `false` (*faux*). Tous les opérateurs de comparaison retournent par exemple un résultat de type `bool`.

Ce type n'existait pas dans les premières versions de C++, qui utilisaient la convention du langage C pour la représentation des valeurs logiques. En C, les valeurs logiques étaient représentées par un `int`, avec la convention suivante : `0` signifie *faux*, et toutes les autres valeurs signifient *vrai*. Par exemple, les opérateurs de comparaison retournaient `0` (*faux*) ou `1` (*vrai*). On en parle ici, parce que le type `bool` est compatible avec l'ancienne représentation, qui est toujours utilisable. Par exemple :

```
bool a, b, c, d ;
a = true ;  // a vaut true
b = 1 ;     // b vaut true
c = 34 ;    // c vaut aussi true
d = false ; // d vaut false
```

Si vous affichez les valeurs des variables précédentes, vous aurez d'ailleurs une surprise, car l'affichage des booléens utilise par défaut la représentation numérique.

```
std::cout <<  a << " " << b << " " << c << " " << d << std::endl ;
```

```
1 1 1 0
```

Toutefois, il est facile de configurer le mode d'affichage[3], de manière à ce que les booléens soient imprimés d'une manière spécifique :

```
std::cout << std::boolalpha ; // configure le mode d'affichage des bool
std::cout <<  a << " " << b << " " << c << " " << d << std::endl ;
```

```
true true true false
```

Pour des raisons de rétrocompatibilité, il est possible d'utiliser une expression de type `int` là ou une expression logique est attendue : celle-ci est automatiquement convertie vers le type `bool`. Et ceci est également valable pour les pointeurs, un pointeur nul étant considéré comme *faux*. Voici un exemple où un entier est utilisé comme une expression logique :

```
int k = f() ;  // f est une fonction à résultat entier
if(k)          // contexte d'un test : k est automatiquement converti en bool
{
  // ...
}
```

4.4.4 Réels

Les nombres réels manipulés dans les ordinateurs ne sont qu'une approximation des nombres réels du monde mathématique. Ils sont représentés par un procédé appelé *virgule flottante*, que je ne détaillerai pas ici. Pour cette raison, ces nombres s'appellent *nombres à virgule flottante*, ou plus simplement *flottants*. Comme pour les entiers, le résultat de la représentation est imparfait : on ne sait représenter qu'un nombre limité de réels. Aussi bizarre que cela paraisse, un flottant permet par exemple de représenter exactement le nombre $\frac{1}{4}$, mais pas le nombre $\frac{1}{3}$.
Le schéma de la figure 4.4 montre l'axe des nombres réels, sur lequel les zones grisées contiennent les nombres représentables. Le nombre 0 est représentable, mais les autres nombres situés dans l'intervalle $[-min \cdots + min]$ ne le sont pas, tout comme ceux dont la valeur absolue est $> max$. Il faut noter que les réels se trouvant dans les zones grisées ne sont pas tous représentables, puisqu'il y en a une infinité !

3. Les manipulateurs d'entrées/sorties, tels que `boolalpha` sont décrits en section 29.11, page 360

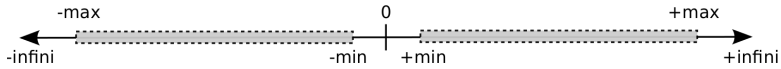

Figure 4.4 – Zones de l'axe des réels dans lesquelles les nombres flottants sont représentables.

Les limites de ces différentes zones dépendent du type de nombre flottant utilisé. Trois variantes de flottants existent en langage C et C++ : les `float`, les `double` et les `long double`. Leur représentation est dépendante du compilateur, mais il est fréquent qu'ils soient codés sur 4, 8 et 16 octets, selon la norme IEEE-754 créée par l'*Institute of Electrical and Electronics Engineers*.

À partir de C++23 , vous pouvez utiliser des types flottants dont la taille et la représentation sont imposées par la norme [4] : il s'agit de `float16_t`, `float32_t`, `float64_t`, `float128_t` et `bfloat16_t`. Ces nouveaux types utilisent respectivement *toujours* 2, 4, 8, 16 et 2 octets de mémoire [5].

Le tableau de la figure 4.5 montre les limites des intervalles de valeurs représentables.

Type C++	taille (octets)	min	max
`float` (ou `float32_t`)	4	$\approx 1.2 \times 10^{-38}$	$\approx 3.4 \times 10^{38}$
`double` (ou `float64_t`)	8	$\approx 2.2 \times 10^{-308}$	$\approx 1.7 \times 10^{308}$
`long double` (ou `float128_t`)	16	$\approx 3.4 \times 10^{-4932}$	$\approx 1.2 \times 10^{4932}$

Figure 4.5 – Exemples d'intervalles de valeurs absolues représentables par des nombres flottants, selon la norme IEEE-754. Ces valeurs sont arrondies.

La précision des nombres représentables est limitée. Par exemple, le type `float` permet de coder des nombres avec 6 digits décimaux significatifs, ce qui est souvent insuffisant. Nous disposons de 15 digits significatifs avec un `double`, et de 18 avec un `long double`. Comme le type `float` est assez peu précis, il est rarement utilisé : c'est pour cela que le type flottant standard des langages C et C++ est le `double`.

4.4.4.1 Déclaration des variables réelles

La déclaration d'une variable réelle est analogue à celle d'un entier.

type_de_la_variable nom_de_la_variable;

Par exemple :

```
double x ;  /* x est un double */
float k ;   /* k est un float */
```

4.4.4.2 Exemple d'utilisation de réels

Calculs utilisant des nombres flottants

```
1
2 #include <iostream>
3
4 int main()
```

4. Utiliser ces types permet de garantir que vos données resteront représentables lors du portage des programmes.
5. Le type `bfloat16_t` est une version de `float16_t` offrant moins de précision et davantage d'amplitude

```
{
  int a,b ;
  double x,y,z ;
  a = 10 ; b = 3 ;
  x = 10 ; y = 3 ;
  z = a / b ;
  std::cout << z << std::endl ;
  z = x / y ;
  std::cout << z << std::endl ;

  return 0 ;
}
```

Le résultat obtenu sur ma machine est le suivant :

```
3
3.33333
```

Cet exemple ressemble au précédent, mais réalise des divisions. Cette opération est appliquée de la même manière à des `int` et des `double`. Dans les deux cas, le résultat est affiché grâce à l'opérateur `<<` , comme dans l'exemple concernant les nombres entiers.

Ce code a été conçu pour donner un résultat surprenant. En effet, la valeur du quotient `a/b` est `3`, et celle du quotient `x/y` est `3.333333`. Le premier résultat vient du fait qu'en C++, un calcul entre entiers donne *toujours* un entier. Le calcul fournit donc la partie entière du quotient, qui est ensuite stockée dans une variable de type `double`.

4.4.5 Types des constantes

Les exemples précédents utilisent des *variables* (`x`, `y`, etc.) et des *constantes littérales* (`3`, `10`, etc.). Le type de chaque variable est obligatoirement fourni à sa déclaration, ce qui permet au compilateur de la manipuler correctement. Mais comment le type des constantes littérales est-il spécifié ? La réponse est simple : la façon de les écrire détermine non seulement leur valeur, mais aussi leur type. Il existe deux familles de constantes numériques : les *entiers* et les *flottants*. Toute constante numérique qui comporte un point ou un exposant décimal est de type flottant. Sinon elle est de type entier, comme dans les exemples ci-dessous.

```
i = 3 ;     /* 3 est un entier (int) */
a = 4.5 ;   /* 4.5 est un flottant (double) */
b = 5E3 ;   /* 5E3 est un flottant (5000.0, double)  */
c = 5e3 ;   /* 5e3 est un flottant (5000.0, double)  */
d = 4.5E3 ; /* 4.5E3 est un flottant (4500.0, double)  */
```

4.4.5.1 Écriture des constantes entières sous forme numérique

Les littéraux entiers peuvent être écrits en base 10, en base 2, en base 8 ou en base 16. Cette base est spécifiée en utilisant un préfixe. Le tableau de la figure 4.6 indique les différents préfixes disponibles, et les écritures correspondantes.

Remarquez que ces préfixes commencent par la lettre `0` (zéro) et non `O` (o majuscule).

En pratique, les bases d'écriture décimales et hexadécimales sont très utiles. L'écriture en base 2 peut aussi être utile dans certains domaines, tels que celui des logiciels qui accèdent directement au matériel. Par contre, l'écriture en base 8 est peu utile, et elle est une source classique de confusion. Par exemple `010` signifie 10_8, c'est-à-dire 8_{10} et non 10_{10} comme on pourrait le croire [6].

6. La notation N_b signifie N en base b.

Base d'écriture	Préfixe	Exemples
10 (*décimal*)	*sans préfixe*	`-34`, `0`, `4`
2 (*binaire*) à partir de C11	`0b`	`0b111`, `0b10110`
8 (*octal*)	`0`	`07`, `010`, `0377`
16 (*hexadécimal*)	`0x`	`0x100`, `0x1C`, `0x1BAFFE`

Figure 4.6 – Expression d'une constante entière dans quatre bases de numération.

N'utilisez donc l'écriture octale que si vous avez de sérieuses raisons de le faire.

Voici différentes façons d'écrire la constante `255`, en l'exprimant dans différentes bases de numération.

```
i = 255 ;         /* 255, decimal */
j = 0377 ;        /* aussi 255, octal */
k = 0xFF ;        /* encore 255, hexadécimal */
l = 0b11111111 ;  /* toujours 255, binaire */
```


La base utilisée pour écrire une constante n'a aucune incidence sur la représentation en machine de cette constante : compte tenu de la technologie des ordinateurs actuels, elle est presque toujours représentée en base 2. Les contenus des quatre variables de l'exemple précédent sont donc physiquement identiques, pour une machine donnée.

4.4.5.2 Écriture des entiers sous forme « caractère »

Les constantes entières peuvent également être écrites sous forme non numérique : par exemple `'A'` est un entier de type `char` dont la valeur est celle du code [7] de la lettre `A`. Ce genre de constante doit être vu comme un moyen portable d'obtenir le code d'un caractère particulier.


En langage C, le type des constantes écrites de cette manière est `int`, et non `char`.

Les codes des caractères non imprimables [8] peuvent être obtenus grâce à une séquence d'échappement prédéfinie commençant par `\`. Par exemple, la séquence `'\n'` correspond au caractère « saut de ligne » (*newline* en anglais). Le tableau de la figure 4.7 donne l'écriture des caractères non imprimables les plus fréquents.

Le petit morceau de programme suivant permet d'afficher les valeurs de quelques constantes écrites sous forme caractère. L'utilisation de l'opérateur `sizeof` [9], qui permet d'obtenir la taille (en octets) du résultat d'une expression permet de montrer que le type de la constante `'x'` est bien `char`, qui occupe 1 octet sur ma machine.

```
int code  = 'A' ;
std::cout << code << std::endl ;
code = '\n' ;
std::cout << code << std::endl ;
/* Quelle est la taille du résultat de l'expression 'x' ? */
std::cout << sizeof 'x' << std::endl ;
```

Ce qui affiche sur ma machine :

7. Le code d'une lettre est simplement un entier qui représente conventionnellement cette lettre. Sur les machines utilisant le code ASCII ou UNICODE (c'est-à-dire presque toutes), la valeur correspondant à la lettre `A` est 65

8. Un caractère non imprimable est un caractère auquel ne correspond aucun dessin (ou glyphe).

9. Bien que `sizeof` ne ressemble pas aux opérateurs auxquels nous sommes habitués, il en est réellement un, au même titre que `+`, `-`, etc.

```
65
10
1
```

Écriture	Signification de la constante
`'\n'`	saut de ligne
`'\r'`	retour en début de ligne
`'\f'`	saut de page
`'\a'`	bip sonore
`'\t'`	tabulation horizontale
`'\v'`	tabulation verticale
`'\0'`	nul (caractère de code zéro)
`'\\'`	authentique caractère « \ » (*backslash*)

Figure 4.7 – Écriture des constantes correspondant aux caractères non imprimables les plus fréquents

4.4.5.3 Écriture des constantes flottantes

Les nombres flottants sont exprimés en base 10, en notation *classique* ou *scientifique*. Le tableau de la figure 4.8 donne quelques exemples d'écriture.

Écriture classique	Écriture scientifique	Signification
`0.0` ou `0.` ou `.0`		0.0
	`1.e-2`	1.0×10^{-2}
	`-45.33E3`	-45.33×10^{3}
	`1.0e3`	1.0×10^{3}

Figure 4.8 – Différentes écritures de nombres flottants

4.4.5.4 Constantes avec modificateur de type

Le type par défaut des constantes flottantes, est **`double`** (et non **`float`** comme on pourrait l'imaginer) et celui des constantes entières est **`int`**. Pour préciser qu'une constante est d'un autre type, il faut lui adjoindre un *suffixe* spécifique, comme indiqué dans le tableau de la figure 4.9.

Voici quelques exemples de constantes dont le type est précisé par un suffixe :

```
a = 1.F ;       /* constante de type float */
b = 3L ;        /* constante de type long */
c = 0LL ;       /* constante de type long long */
d = 10ULL ;     /* constante de type unsigned long long */
```

Il est fortement déconseillé d'utiliser les suffixes `l` et `ll` à cause de leur trop grande ressemblance avec `1` et `11`. Dans le tableau de la figure 4.9, il est par exemple aisé de confondre la constante `-1ll` (`-1 long long`) avec `-111` (moins cent-onze). La confusion est aussi possible entre `23l` et `231`. Pour éviter ce type de situation, utilisez systématiquement `L` et `LL`.

Type désiré	Suffixe	Exemple
`char`	*inexistant*	
`short`	*inexistant*	
`int`	*inutile*	`123456`
`long`	`l` ou `L`	`0x1BABAC00L`, `23l`
`long long`	`ll` ou `LL`	`0LL`, `-1ll`
`float`	`f` ou `F`	`-44.61f`, `4.0F`
`double`	*inutile*	`1.23e-1`, `3.E4`
`long double`	`l` ou `L`	`0.0L`
`unsigned int`, `long` ou `long long`	`u` ou `U`	`5U`, `100UL`, `99ULL`
`float16_t` (c++23)	`f16` ou `F16`	`0.0f16`
`float32_t` (c++23)	`f32` ou `F32`	`0.0f32`
`float64_t` (c++23)	`f64` ou `F64`	`1.5f64`
`float128_t` (c++23)	`f128` ou `F128`	`0.5f128`
`bfloat16_t` (c++23)	`bf16` ou `BF16`	`9.5BF16`

Figure 4.9 – Les différents suffixes modificateurs permettant de préciser le type d'une constante.

Si la valeur d'une constante n'est pas représentable par le type spécifié par son suffixe, elle sera du type qui permet de la représenter occupant le moins de place en mémoire, s'il existe. Par exemple, si les `int` sont codés sur 32 bits, la valeur `4294967295U` sera de type `unsigned int`, mais la valeur `4294967296U` sera de type `unsigned long` car elle ne peut pas être représentée sur 32 bits.

4.4.5.5 Constantes littérales numériques décorées

Les constantes numériques réelles ou entières peuvent être décorées d'apostrophes simples qui permettent d'améliorer leur lisibilité. Cette décoration n'a aucun autre effet.

```
double x = 1'000'000.5 ; // 1000000.5
int k = 100'000 ;        // 100000
```

4.4.6 Pointeurs

Le type *pointeur* sert à représenter l'adresse d'une entité. L'existence des pointeurs découle de ce qui est mentionné en annexe A.1.1.1 (page 789) : accéder à une entité (donnée ou code) stockée en mémoire nécessite de connaître son emplacement, c'est-à-dire son adresse. La plupart du temps, cette nécessité est insoupçonnée. Pourtant, si l'on écrit

```
vitesse = 0 ;
```

c'est bel et bien la connaissance de l'adresse de la variable `vitesse` qui permet d'en modifier la valeur. La notion d'adresse est donc fondamentale en programmation, même lorsqu'elle n'apparaît pas explicitement, comme c'est le cas dans beaucoup de langages.

En C++ (et en C), cette notion apparaît explicitement sous la forme du type pointeur. Le pointeur est l'outil le plus explicite du langage nous permettant d'accéder à une donnée ou à du code.

Les pointeurs sont utilisables dans un très grand nombre de contextes et il est donc impossible d'en montrer tous les usages dans cette section. Je me contenterai donc de présenter le concept et son usage de base. D'autres usages, d'ailleurs beaucoup plus intéressants, seront présentés au fur et à mesure des besoins dans les chapitres suivants.


Cet outil étant très souple, il stimule beaucoup la créativité des programmeurs en leur permettant de faire des choses difficiles à réaliser autrement. Mais comme il est également *très proche de la machine*, il ne pardonne guère les utilisations maladroites. Ceci explique en partie que l'usage des pointeurs soit souvent considéré comme difficile. Je ne partage pas cette opinion : les pointeurs sont au contraire des entités extrêmement simples, qu'il est facile de maîtriser à condition de bien savoir de quoi on parle dès le départ. N'hésitez donc pas à passer du temps sur l'acquisition de ces bases.


Il existe en C++ des alternatives aux pointeurs qui permettent d'en limiter l'usage. Il s'agit des *références* (section 17.4, page 173) et des *pointeurs intelligents* (chapitre 49, page 645).

4.4.6.1 Déclaration d'une variable de type pointeur sur ...

Comme les autres variables, les variables de type pointeur doivent être déclarées. Cette déclaration est de la forme :

type_de_la_donnée_pointée `*`*nom_du_pointeur* `;`

Par exemple, si l'on désire déclarer un pointeur `pi` destiné à contenir l'adresse d'un `int`, il suffira de le déclarer par :

```
int *pi ;  /* pi est un pointeur sur un int (Noter la lettre *) */
```

Sauriez-vous dire maintenant à quoi correspondent les déclarations suivantes ?

```
double *a ;  /* facile */
char   *b ;  /* encore facile */
short  **c ; /* moins facile */
```


Les deux premières déclarations sont simples : elles concernent respectivement un pointeur sur un `double` et un pointeur sur un `char`. La troisième est un peu plus complexe, car elle permet de déclarer un *pointeur sur un pointeur* sur un `short`. En effet, les variables de type pointeur ont, comme les autres, une adresse en mémoire. Par conséquent, cette adresse peut être mémorisée dans un (autre) pointeur, qui doit être déclaré. Pour l'instant, ne vous focalisez pas trop sur la syntaxe de déclaration des pointeurs, car nous verrons en section 12.2.1 (page 128) comment déclarer sans erreur n'importe quelle entité (même la plus complexe) ayant un sens en langage C ou C++. Pour l'instant, il suffit de savoir déclarer des pointeurs simples tels que `a`, `b` ou `c`.


On peut remarquer ici que le type pointeur (tout court) n'existe pas. En effet, un pointeur sur un `int` n'est pas du même type qu'un pointeur sur un `double`. C'est en cela que les pointeurs ont un type composé.

4.4.6.2 Obtenir l'adresse d'une entité : opérateur « & »

Maintenant que nous savons déclarer un pointeur, nous allons voir comment lui donner une valeur. La valeur d'un pointeur étant une adresse, le problème revient à trouver l'adresse d'une entité [10]. Le langage C++ propose pour cela l'opérateur « `&` ». Placé devant une expression correspondant à une entité stockée en mémoire, cet opérateur fournit son adresse. Par exemple :

```
char u ;      /* u est un char */
char *pu ;    /* pu est un pointeur sur un char */
char **ppu ;  /* ppu est un pointeur sur un pointeur sur un char */

u = 34 ;      /* u contient maintenant 34 */
pu = &u ;     /* pu contient maintenant l'adresse de u */
ppu = &pu ;   /* ppu contient maintenant l'adresse de pu */
```

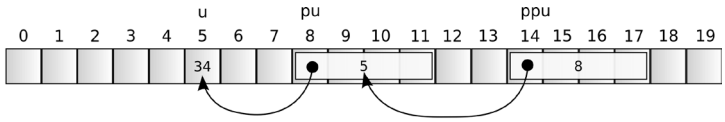

Figure 4.10 – Exemple d'implantation d'un `char` et de deux pointeurs en mémoire

Le schéma de la figure 4.10 montre une implantation imaginaire des variables `u`, `pu` et `ppu` dans les 20 premiers octets de la mémoire. Dans cette zone, la variable `u` a été placée arbitrairement à l'adresse `5` alors que `pu` et `ppu` sont placés respectivement aux adresses `8` et `14`. Je n'ai pas représenté les valeurs des autres octets, car elles n'ont ici pas d'importance. Les flèches indiquent quelle est la variable pointée par chaque pointeur, mais cette information est également fournie par la valeur du pointeur.

Vous pouvez remarquer que `pu` et `ppu` occupent chacun quatre octets. Nous sommes donc en présence d'un processeur utilisant des pointeurs codés sur 32 bits. Remarquez aussi que lorsqu'une entité occupe plusieurs octets, son adresse est celle de son premier octet.

Lorsqu'on parle de pointeurs, les notions de *valeur* et d'*adresse* ont tendance à se télescoper dans l'esprit des débutants parce que, précisément, *la valeur d'un pointeur est une adresse*. Je vais donc préciser les choses pour cet exemple :

- `u` est un `char`, son adresse est `5` et sa valeur est `34`,
- `pu` est un pointeur sur un `char`, son adresse est `8` et sa valeur est l'adresse de `u`, c'est-à-dire `5`,
- `ppu` est un pointeur sur un pointeur sur un `char`, son adresse est `14` et sa valeur est l'adresse de `pu`, c'est-à-dire `8`,
- si `pu` contient l'adresse de `u`, on dit que `pu` *pointe sur* `u`,

Bien que les adresses `5`, `8` et `14` soient écrites ici comme des entiers, *ce ne sont pas des entiers*. Nous verrons pourquoi à la section 14.1, page 141.

Ne pensez surtout pas que tous les types pointeurs sont équivalents. Par exemple, le code suivant ne pourra pas être compilé.

```
ppu = &u ; /* FAUX */
```

En effet, dans cette expression, `ppu` est de type « pointeur sur un pointeur sur un `char` », alors que `&u` est de type « pointeur sur un `char` ». L'affectation n'a donc pas de sens.

Enfin, soyez conscient que toutes les données manipulées ne résident pas forcément en mémoire, et donc qu'elles n'ont pas forcément d'adresse. C'est souvent le cas pour le résultat des calculs :

```
pu = &(u+u) ; /* FAUX */
```

4.4.6.3 Accéder à une entité via un pointeur : opérateur « * »

Lorsqu'il est appliqué à un pointeur, l'opérateur `*` permet d'obtenir l'entité pointée par celui-ci. Appliquer cet opérateur à un pointeur s'appelle *déréférencer* ce pointeur. Dans l'exemple de la section précédente, `pu` pointe sur `u`. Par conséquent, l'expression `*pu` est exactement équivalente à `u`, et les deux lignes de code qui suivent ont le même effet :

```
u = 0 ;    // accès à u
*pu = 0 ;  // accès indirect à u, via pu
```

10. On utilise ici le terme entité pour désigner une variable ou une fonction.

Souvenez-vous également que dans cet exemple, `ppu` pointait sur `pu` comme le montre la figure 4.10. Par conséquent `*ppu` et `pu` désignent la même entité, ce qui nous donne un autre moyen d'affecter `0` à `u` :

```
**ppu = 0 ; // accès doublement indirect à u, via ppu et pu
```

Un pointeur peut également pointer sur une fonction. Dans ce cas, il est possible d'exécuter cette fonction au travers du pointeur. Supposons par exemple que `pfi` soit un pointeur sur une fonction qui reçoit et renvoie un entier. L'expression `(*pfi)` correspond donc à la fonction pointée, et il est par conséquent possible d'exécuter cette fonction comme ceci :

```
k = (*pfi)(5) ; // appel de la fonction pointée par pfi
k = pfi(5) ;    // idem: déréférencement automatique des pointeurs sur fonction
```

Les fonctions seront traitées au chapitre 6, page 71.

4.4.6.4 Pointeur nul

Déréférencer un pointeur non initialisé est une erreur grave qui conduit à accéder à un emplacement mémoire indéterminé. Deux situations peuvent se produire :

(a) Si cet emplacement est illégal, le programme sera immédiatement arrêté par le système d'exploitation. En langage courant, il s'agit d'un *plantage* (ou *crash*).

(b) S'il correspond à une zone mémoire légale, le programme exploitera ou modifiera des données n'ayant rien à voir avec le code en cours d'exécution. Il aura un comportement erratique, donc très difficile à analyser.

Pour signaler qu'un pointeur ne pointe (provisoirement) sur rien d'utilisable, il est possible de le faire pointer sur l'adresse zéro. En effet, celle-ci correspond à un emplacement mémoire où, par convention, le compilateur et l'éditeur de liens ne placent rien. Pour rendre un pointeur nul, vous devez lui donner la valeur `nullptr`. Évitez d'utiliser `0` ou `NULL` pour cet usage, car cela peut provoquer des ambiguïtés dans certaines situations.

```
char *p = 0 ;       // OK, mais non recommandé, style C++ classique
char *q = NULL ;    // OK, mais non recommandé, style C
char *r = nullptr ; // OK, recommandé, style C++ moderne
```

Initialiser un pointeur de cette manière présente un gros avantage : si celui-ci est déréférencé accidentellement avant de contenir une valeur valide, le programme sera arrêté immédiatement, ce qui exclut les comportements imprévisibles.

D'une manière générale, il est *absolument interdit* d'affecter un entier à un pointeur. L'entier `0` est un cas particulier, qui signifie conventionnellement « pointeur nul ».

4.4.6.5 Mais à quoi servent les pointeurs ?

On a présenté ici les éléments de base permettant l'utilisation des pointeurs : leur déclaration, la génération d'un pointeur avec l'opérateur `&` et son déréférencement avec l'opérateur `*`. Ayant lu les exemples précédents, vous pourriez penser que les pointeurs servent juste à faire de façon compliquée ce qu'il est possible de faire plus simplement par d'autres moyens. Mais il n'en est rien : combinés avec d'autres outils du langage, les pointeurs rendent au contraire d'énormes services. On présentera au fil des chapitres suivants toutes sortes d'applications de cet outil.

Résumé 1

(a) Chaque entité possède un type. La notion de type permet au compilateur de fabriquer un code manipulant correctement cette entité. Mais les types permettent aussi d'introduire du sens dans les programmes.

(b) Les déclarations sont des instructions qui permettent d'indiquer au compilateur le type de chaque entité manipulée. Elles doivent figurer *avant* l'usage de ces entités et sont *obligatoires* en C++.

(c) Les types primitifs les plus simples sont les entiers, les réels et les pointeurs.

(d) Les entiers se déclinent en cinq sous-types. Par taille croissante : `char`, `short`, `int`, `long` et `long long`. Il existe une version non signée de chaque sous-type : `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long`.

(e) Les réels se déclinent en trois sous-types. Par taille et précision croissantes : `float`, `double` et `long double`, qui sont tous signés.

(f) Le type entier le plus utilisé est `int`. Le type réel le plus utilisé est `double`.

(g) Les constantes ont aussi un type, qui dépend de la manière dont elles sont écrites. Dans leur écriture la plus simple, les constantes entières ont le type `int` (ex : `324`), et les constantes réelles ont le type `double` (ex : `23.75`).

(h) Le type pointeur se décline potentiellement en une multitude de variantes. En effet, un pointeur sur une entité de type `X` n'est pas du même type qu'un pointeur sur une entité de type `Y`.

(i) Les pointeurs sont liés aux opérateurs suivants : l'opérateur `&` qui permet d'obtenir l'adresse d'une entité, et l'opérateur dual `*` qui permet d'obtenir une entité dont on connaît l'adresse.

4.5 Données constantes

4.5.1 Qu'est-ce qu'une donnée constante ?

Si vous voulez protéger des données contre une modification intempestive, vous avez la possibilité de les déclarer comme étant *constantes*. Une donnée constante est déclarée à l'aide du mot-clé `const`, qui signifie que la donnée gardera sa valeur initiale durant toute sa vie. Vous devez donc voir la spécification `const` comme signifiant *non modifiable*. Par exemple :

```
const int    max = 1023 ;
const double pi  = 3.141592653 ;
```

Le compilateur garantit que ces données ne seront jamais modifiées. Le code suivant sera par exemple rejeté à la compilation :

```
pi = 4.0 ; /* FAUX (pi est déclarée constante ci-dessus) */
```

Vous devez être conscient qu'une donnée constante *doit être initialisée*. En effet, étant par définition non modifiable, toute affectation à une donnée constante n'a pas de sens et sera rejetée [11].

```
const double pi ;   /* FAUX (pas d'initialisation) */
pi = 3.141592653 ;  /* FAUX (tentative de modification) */
```

4.5.2 Données constantes et constantes littérales

Une variable déclarée avec `const` (telle que `pi` dans la section précédente) ne doit pas être confondue avec une constante littérale telle que `3.141592653`. En effet, `pi` est manipulée comme une variable, même si vous ne pouvez pas changer sa valeur. Comme les autres variables, elle est stockée en mémoire : elle a donc une adresse, que vous pouvez obtenir avec l'opérateur `&`. Par opposition,

11. Cette remarque démontre qu'en programmation, *l'affectation* et *l'initialisation* sont des opérations différentes.

une constante littérale n'est pas forcément stockée dans l'espace mémoire des données : elle est souvent encodée dans l'instruction qui l'utilise, et parfois de manière implicite[12]. Par conséquent, une expression telle que `&3.141592653` n'a aucun sens, alors que `&pi` en a un.

4.5.3 Constantes énumérées

L'instruction `enum` permet de définir des constantes de types `int`, et de leur attribuer un identifiant. Les constantes énumérées sont équivalentes à des constantes littérales. Il est possible de préciser les valeurs de ces constantes, ou de laisser le compilateur les générer lui-même.

```
enum { faux, vrai } ;                    /* faux: 0, vrai: 1 */
enum { eteint=10, allume, casse=0 } ;    /* eteint: 10, allume: 11, casse: 0 */
enum { inf=-1, egal=0, sup=1 } ;         /* inf: -1, egal: 0, sup: 1 */
```

Si le type `int` ne vous convient pas, il est possible de spécifier un autre type entier.

```
enum : char { la='a', lz='z' } ;         /* la: 'a' lz: 'z' */
```

Enfin, une énumération peut être nommée, ce qui permet de définir des variables pouvant prendre une des valeurs énumérées.

```
enum lettre : char { la='a', lz='z' } ;  /* la: 'a' lz: 'z' */
lettre k = la ;                          /* La valeur de k est 'a' */
```

4.5.4 `const` et les pointeurs

Lorsque vous utilisez `const` dans la déclaration d'un pointeur, vous pouvez déclarer constants *le pointeur*, *la donnée pointée*, ou *les deux*.

- Déclarer constant un pointeur signifie que sa valeur ne peut pas être modifiée, c'est-à-dire qu'il désignera toujours la même zone de mémoire.
- Déclarer un pointeur sur une donnée constante signifie que ce pointeur ne pourra pas être utilisé pour modifier la donnée pointée.

La position de `const` dans la déclaration du pointeur va nous permettre de préciser au compilateur ce l'on veut faire, selon les règles suivantes :

(a) Si `const` est placé juste avant le nom du pointeur, celui-ci sera constant.

(b) Si `const` est juste avant ou après le nom du type de l'objet pointé, ce dernier sera constant.

(c) Fort logiquement, si `const` figure aux deux emplacements, l'objet pointé et le pointeur seront constants.

Voici quelques exemples permettant de comprendre ces règles. Nous supposerons tout d'abord que la variable `x` est déclarée comme étant non constante, et `y` comme étant constante :

```
double x ;                /* x n'est pas constante */
const double y = 0 ;      /* y est constante */
```

Si nous voulons dire que la donnée pointée par `p` est constante, nous écrirons :

```
const double *p ; /* const est à côté du nom de type */
```

ou bien

```
double const *p ; /* const est aussi à côté du nom de type : même sens */
```

Si maintenant nous voulons dire que le pointeur `q` est constant, nous devrons écrire :

```
double * const q = &x ; /* const est à côté du nom du pointeur */
```

12. L'expression `x=0` peut par exemple être traduite par le compilateur par l'instruction `clear x` en langage machine. La constante `0` n'existe donc pas en mémoire dans cet exemple.

Remarquez ici que `q` étant constant, nous *devons* lui donner une valeur initiale, qui devra être obligatoirement l'adresse d'une donnée non constante du bon type.

Et si nous voulons dire que `r` *et* la donnée pointée sont constants, nous écrirons :

```
const double * const r = &y ; /* Désolé, c'est dur à lire */
```

Ici aussi, ce pointeur doit être initialisé, mais l'adresse utilisée pour cela peut être celle d'une donnée constante ou non. En effet le pointeur `r` ne pouvant pas être utilisé pour modifier la donnée pointée, il importe peu que celle-ci soit modifiable ou non.

Voici maintenant quelques exemples d'expressions qui illustrent la protection offerte par `const`.

```
double x ;                    /* x n'est pas constante */
const double y = 0 ;          /* y est constante */
const double *p ;             /* const est à coté du nom de type */
double * const q = &x ;       /* const est à coté du nom du pointeur */
const double * const r = &y ; /* const est partout */

x = 0 ;       /* OK : x n'est pas constant */
y = 1 ;       /* FAUX : y est constant */
p = &x ;      /* OK : p pointe sur une donnée non constante */
p = &y ;      /* OK : p ne peut pas être utilisé pour modifier y */
p = p+1 ;     /* OK : p n'est pas constant */
q = q+1 ;     /* FAUX : q est constant  */
*p = 4.0;     /* FAUX : la donnée pointée est constante */
r = p ;       /* FAUX : r est constant */
*r = *r+2 ;   /* FAUX : la donnée pointée est constante */
```

4.5.5 Quand utiliser `const` ?

À quoi sert réellement `const` ? Vous devez vous demander pourquoi il faudrait interdire la modification d'une variable, alors que, par essence, une variable est faite pour être modifiée.

En dehors du cas trivial des constantes dont la modification n'a aucun sens (telles que `pi` dans les exemples précédents), le véritable intérêt de `const` est de faire en sorte qu'une portion de code ne puisse pas modifier certaines données, *qui pourraient toutefois être modifiables ailleurs*. En effet, la modification d'une variable réalisée à l'insu d'un morceau de code qui l'utilise peut provoquer des résultats désastreux. Dans ce cadre, le qualificatif `const` agit comme un contrat par lequel une portion de programme s'engage à ne pas modifier des données : il permet donc d'expliciter les responsabilités de ce code.

En pratique, `const` qualifie très souvent des paramètres de fonctions, surtout lorsque ceux-ci sont des pointeurs [13]. En effet, dans ce cas, la fonction a la possibilité matérielle de modifier les données pointées, ce qui peut compromettre le fonctionnement du code appelant. Dans ce contexte, utiliser `const` permet de demander au compilateur de rejeter tout code qui pourrait aboutir à la modification des données pointées. Cet aspect sera spécifiquement développé à la section 6.5, page 76.

En attendant, détendez-vous : il n'est pas indispensable d'avoir compris toutes les subtilités liées à l'usage de `const` pour écrire des programmes intéressants en langage C++.

13. Ceci est aussi vrai pour les références en langage C++. Les références seront présentées en section 17.4, page 173.

4.6 Données volatiles

4.6.1 Qu'est-ce qu'une donnée volatile ?

On a l'habitude de penser que la valeur d'une variable ne peut pas changer si aucune instruction ne la modifie, et c'est effectivement généralement le cas. Mais lorsqu'on écrit des programmes manipulant des dispositifs matériels, on est souvent amené à utiliser des variables qui sont *physiquement* connectées à ces dispositifs.

Une variable de ce type peut par exemple représenter un *port d'entrée-sortie*, c'est-à-dire un ensemble de fils sortant du boîtier du processeur. Lorsque cette technique est utilisée, on parle d'*entrées-sorties projetées en mémoire* (ou *memory mapped*). Cette façon de réaliser les entrées-sorties est assez commode, puisqu'il suffit de modifier une variable pour agir sur un dispositif physique. Réciproquement, un changement d'état de ce dispositif peut se traduire par un changement de la valeur de la variable correspondante. On dit alors que cette variable est *volatile*, ce qui indique bien que sa valeur peut changer à tout moment, en dehors de toute directive logicielle.

4.6.2 Données volatiles et optimisations

Cette façon de dialoguer avec le matériel a une conséquence : elle rend inapplicables beaucoup d'optimisations ordinairement pratiquées par les compilateurs. Par exemple, dans le code

```
PORT0 = 3 ;   /* Instruction inutile ? */
PORT0 = 4 ;
```

il est extrêmement fréquent que le compilateur supprime purement et simplement la première affectation, qui est manifestement inutile puisque son effet est annulé par la seconde. Malheureusement, si la variable `PORT0` désigne un dispositif matériel, l'optimisation ne doit pas être appliquée, parce que dans ce cas, la valeur `3` ne serait jamais transmise au dispositif.

Il est donc important que les données volatiles soient explicitement déclarées comme telles, ce qui interdit au compilateur de réaliser des optimisations potentiellement nuisibles. Les langages C et C++ proposent pour cela le modificateur `volatile`, dont l'usage est syntaxiquement identique à celui de `const`.

```
/*  On suppose ici que PORT0 est déclarée volatile */
PORT0 = 3 ;  /* Cette instruction ne sera pas supprimée */
PORT0 = 4 ;
```

Résumé 2

(a) `const` et `volatile` sont deux qualificatifs incompatibles entre eux, car antagonistes.

(b) `const` signifie que la valeur d'une variable ne peut pas changer. Ceci permet aux compilateurs de pratiquer certaines optimisations.

(c) `volatile` signifie que la valeur d'une variable peut changer à tout moment, sans qu'elle soit explicitement modifiée. Ceci interdit aux compilateurs de pratiquer certaines optimisations.

5 | Opérateurs

Le langage C++ propose un grand nombre d'opérateurs permettant de réaliser des calculs sur des données de type entier, flottant, pointeur ou logique. Certains de ces opérateurs sont classiques et ne vous surprendront pas, mais d'autres sont plutôt originaux. Certains travaillent à partir d'un seul opérande, d'autres à partir de deux opérandes. Il existe même un opérateur qui travaille sur trois opérandes, et des opérateurs utilisables avec un ou deux opérandes, avec des significations totalement différentes. Enfin, comble de la bizarrerie, il existe même un opérateur qui ne fait rien, mais qui s'avère néanmoins très utile.

Certains opérateurs (tels que « `[]` », « `()` », « `->` », « `.` », « `->*` » ou « `.*` ») ne sont pas traités dans ce chapitre, car ils sont liés à des éléments du langage qui n'ont pas encore été présentés. Ils seront traités lorsque ces éléments seront introduits. De la même manière, on ne parlera ici que des opérateurs applicables aux entités de types primitifs provenant du langage C. Les autres seront décrits au fil de la présentation des concepts du langage C++ qu'ils supportent.

5.1 Opérateurs arithmétiques

5.1.1 Opérateurs +, −, *, /

Ils permettent de réaliser les quatre opérations arithmétiques élémentaires : addition, soustraction, multiplication, division. Il faut noter que si les deux opérandes sont entiers, le résultat est toujours entier. Si au moins un des deux opérandes est flottant, le résultat est toujours flottant. Cette caractéristique peut s'avérer surprenante dans le cas de la division, comme le montre l'extrait de code suivant, où la valeur finale de la variable affectée est mentionnée en commentaire :

```
int k = 10 , n ;
double x = 5.0 , y  ;

n = k * 2 ;     /* 20 */
n = k / 2 ;     /* 5 */
n = k / 3 ;     /* 3 */
n = 1 / 2 ;     /* 0 */

y = x * 2 ;     /* 10.0 */
y = x * 2.0 ;   /* 10.0 */
y = x / 2 ;     /* 2.5 */
y = 1.0 / 2 ;   /* 0.5 */
y = 1.0 / 2.0 ; /* 0.5 */
y = 1 / 2 ;     /* 0.0 */
```

L'opérateur `-` a la particularité de pouvoir être appliqué à deux opérandes (soustraction) ou à un seul (calcul de l'opposé), comme dans l'expression « `x = -x ;` ».

Les opérateurs `+` et `-` permettent aussi de faire des calculs sur des pointeurs, mais ce point sera spécifiquement traité en section 14.1, page 141.

5.1.2 Opérateur %

L'opérateur `%` (dit aussi *opérateur modulo*) permet de calculer le reste d'une division entière : ses opérandes doivent donc être des entiers. Compte tenu du comportement de l'opérateur de division, l'opérateur modulo peut donc être défini par :

```
a % b  ≡  a - ((a / b) * b)
```

L'extrait de code suivant illustre le comportement de cet opérateur.

```
int k , n ;
k = 10 ;

n = k % 5 ;      /*  0 */
n = k % 3 ;      /*  1 */
n = -k % 3 ;     /* -1 */
n = -k % -3 ;    /* -1 */
```

5.1.3 Opérateurs ++ et --

Ces deux opérateurs jouent un rôle similaire : `++` permet *d'ajouter 1* à son opérande (qui peut être de type numérique ou pointeur), alors que `--` permet *d'enlever 1*. Ces deux opérations s'appellent respectivement *incrémentation* et *décrémentation*[1]. Pour simplifier les explications, on ne traite donc ici que de l'opérateur `++`, mais l'opérateur `--` s'utilise de façon identique.

Cet opérateur est très particulier pour deux raisons : la première est qu'il modifie son opérande. La seconde particularité est que son effet exact dépend de la façon dont on l'utilise. Par exemple, il peut être placé *avant* son opérande (usage préfixé, comme dans `++k`), ou *après* (usage postfixé, comme dans `k++`).

Voici en quoi consistent ces deux modes d'utilisation.

- `++` préfixé

  Dans ce mode, l'opérande est incrémenté avant d'être utilisé. Par conséquent,

  ```
  int j, k = 3 ;
  j = ++k ;
  ```

  sera équivalent à :

  ```
  int j, k = 3 ;
  k = k + 1 ; j = k ;
  ```

- `++` postfixé

  Dans ce mode, l'opérande est incrémenté après avoir été utilisé. Par conséquent,

  ```
  int j, k = 3 ;
  j = k++ ;
  ```

  sera équivalent à :

  ```
  int j, k = 3 ;
  j = k ; k = k + 1 ;
  ```

Dans les deux versions du code, la valeur finale de `k` sera 4, mais celle de `j` sera 4 pour la version préfixée (c'est-à-dire : `++k`), et 3 pour la version postfixée (c'est-à-dire : `k++`).

Naturellement, si la valeur de la variable incrémentée n'est pas utilisée (c'est-à-dire que l'opérateur est uniquement employé pour modifier son opérande) les formes pré et postfixées sont équivalentes. Les trois lignes du code suivant produisent donc exactement le même effet.

```
++k ;
k++ ;
k = k + 1 ;
```

1. `++` et `--` sont supportés au niveau des processeurs par des instructions-machine simples et efficaces qui s'appellent généralement `INC` et `DEC`. Le code résultant est donc très efficace, même si le compilateur utilisé est peu optimisant.

5.2 Opérateurs sur bits

Bien que les opérateurs sur les bits travaillent sur des opérandes entiers, ils n'ont pas de signification arithmétique. Leur nom vient du fait que ces opérandes sont considérés comme une juxtaposition de bits ne représentant pas forcément un nombre. Avec des entiers codés sur N bits, ces opérateurs réalisent donc N calculs indépendants portant sur chaque bit de leurs opérandes. Leur intérêt est de permettre la réalisation d'opérations élémentaires qui, sans eux, devraient être confiées à du code écrit en langage d'assemblage. Ils sont donc très utiles pour écrire des logiciels manipulant directement le matériel, tels que les systèmes d'exploitation ou les pilotes de périphériques, mais c'est loin d'être leur seul usage.

Pour des raisons de concision, les exemples de cette section portent sur des entiers codés sur 8 bits (`char` ou `unsigned char`), mais sont facilement transposables aux autres types d'entiers.


Vous devez vous demander pourquoi les opérateurs bit à bit ne pourraient pas être appliqués à des flottants qui sont, eux aussi, des « assemblages » de bits. La réponse courte est que ça n'a pas d'intérêt évident. La réponse longue est que, si on y trouve un intérêt, il est possible de le faire. Cela demande toutefois un peu de magie, dont vous trouverez un exemple en section 41.3, page 552.

5.2.1 Opérateur ~

L'opérateur ~ (prononcez « tilde ») calcule le « complément vrai » (ou « complément à 1 ») de son opérande : ceci signifie que tous ses bits à 1 sont transformés en 0, et ses bits à 0 sont transformés en 1. Le tableau de la figure 5.1 donne quelques exemples dans lesquels l'opérande est de type `unsigned char` (entier non signé codé sur 8 bits).

x	00000000_2	00100010_2
~x	11111111_2	11011101_2

Figure 5.1 – Effet de l'opérateur ~ (opérandes exprimés en base 2)

Les valeurs figurant dans ce tableau sont exprimées en base 2 (binaire) pour mieux démontrer l'effet de l'opérateur, et aussi parce que c'est réellement sous cette forme que les données sont représentées dans l'ordinateur. Si vous les affichez, elles seront exprimées par défaut en base 10 (décimal) comme dans le tableau de la figure 5.2. Mais il ne s'agit là que d'une représentation externe conforme à nos habitudes.

x	0_{10}	34_{10}
~x	255_{10}	221_{10}

Figure 5.2 – Effet de l'opérateur ~ (opérandes exprimés en décimal)

5.2.2 Opérateur &

L'opérateur & (prononcez « ET ») réalise une conjonction entre les bits de même rang de ses opérandes. Autrement dit, chaque bit du résultat sera 1 si les bits de même rang des opérandes sont tous les deux 1, et sera 0 sinon. Le tableau de la figure 5.3 donne le résultat de cette opération dans différents cas.

x	00000000_2	00111001_2
y	00111001_2	00101010_2
x & y	00000000_2	00101000_2

Figure 5.3 – Effet de l'opérateur `&` (opérandes exprimés en binaire)

5.2.3 Opérateur |

L'opérateur `|` (prononcez « OU ») réalise une disjonction entre les bits de même rang de ses opérandes. Autrement dit, chaque bit du résultat sera 1 si au moins un des deux bits de même rang des opérandes est 1. Le tableau de la figure 5.4 donne le résultat de cette opération dans différents cas.

x	00000000_2	00111001_2
y	00111001_2	00101010_2
x | y	00111001_2	00111011_2

Figure 5.4 – Effet de l'opérateur `|` (opérandes exprimés en binaire)

5.2.4 Opérateur ^

L'opérateur `^` (prononcez « OU exclusif ») réalise un *ou exclusif* entre les bits de même rang de ses opérandes. Le « ou exclusif » est une opération de l'algèbre de Boole telle que :

`a^b = (a&~b) | (~a&b)`.

Autrement dit, chaque bit du résultat sera 1 si (et seulement si) les bits de même rang des deux opérandes sont différents. Le tableau de la figure 5.5 donne le résultat de cette opération dans différents cas.

x	00000000_2	00111001_2
y	00111001_2	00101010_2
x ^ y	00111001_2	00010011_2

Figure 5.5 – Effet de l'opérateur `^` (opérandes exprimés en binaire)

5.3 Opérateurs de décalage

Comme les opérateurs bit à bit, ces opérateurs considèrent leurs opérandes comme un ensemble de bits. Ils permettent de transférer des bits d'une position à une autre, par « glissement » vers les bits de poids forts ou faibles.

5.3.1 Opérateur <<

Cet opérateur calcule un entier obtenu par décalage de son opérande de gauche d'un nombre de bits fourni par son opérande de droite. Le décalage a lieu vers la gauche, c'est-à-dire vers les bits de poids forts. Le tableau de la figure 5.6 donne quelques exemples de mise en œuvre de cet opérateur sur des entiers non signés codés sur 8 bits.

On remarquera sur ces exemples que chaque décalage à gauche est accompagné de l'introduction à droite d'un bit nul. Le décalage à gauche de n bits peut avoir une signification arithmétique : il s'agit d'une multiplication par 2^n. Ceci résulte de la manière dont sont codés les entiers en machine. Le décalage peut donc être une manière efficace de coder certaines multiplications lorsque

a	00100011_2	00111001_2
n	1_{10}	3_{10}
a << n	01000110_2	11001000_2

Figure 5.6 – Effet de l'opérateur **<<** (opérandes exprimés en binaire)

le processeur ne possède pas d'instruction de multiplication[2]. Le résultat d'une telle multiplication sera faux en cas de débordement (perte de bits de poids forts), mais cela peut aussi être le cas pour toutes les autres opérations arithmétiques.

5.3.2 Opérateur >>

Cet opérateur calcule un entier obtenu par décalage de son opérande de gauche d'un nombre de bits fourni par son opérande de droite. Ce décalage a lieu vers la droite, c'est-à-dire vers les bits de poids faibles. Le tableau de la figure 5.7 donne quelques exemples de mise en œuvre de cet opérateur sur des entiers *non signés* codés sur 8 bits.

a	00100011_2	10011100_2
n	1_{10}	3_{10}
a >> n	00010001_2	00010011_2

Figure 5.7 – Effet de l'opérateur **>>** (opérandes *non signés* exprimés en binaire)

Comme **<<** , l'opérateur **>>** peut avoir un sens arithmétique (division par 2^n). Par rapport à son homologue, il a toutefois une particularité s'il est appliqué à un entier signé : il introduit à gauche une *copie du bit de poids le plus fort*, au lieu d'introduire un bit nul. Cette particularité permet de garantir que le décalage à droite d'un nombre négatif sera un nombre négatif. Par exemple, la valeur de l'expression `-4>>1` est égale à **-2**. Ce point est illustré par l'exemple de la dernière colonne du tableau de la figure 5.8, que vous pouvez comparer à celui de la figure 5.7.

a	00100011_2	10011100_2
n	1_{10}	3_{10}
a >> n	00010001_2	11110011_2

Figure 5.8 – Effet de l'opérateur **>>** (opérandes *signés* exprimés en binaire)

5.4 Opérateurs logiques

Les opérateurs logiques travaillent sur des données logiques et génèrent un résultat logique, c'est-à-dire de type **`bool`**.

> ⚠ Par compatibilité avec le langage C, les opérateurs logiques peuvent aussi travailler sur des entiers, qui sont alors automatiquement convertis en **`bool`**. La convention utilisée pour cela est que tout entier non nul sera considéré comme **`true`**, et tout entier nul comme **`false`**.

Utilisés avec les opérateurs relationnels (décrits en section 5.5.1, page 61) les opérateurs logiques permettent de construire des *prédicats*, c'est-à-dire des expressions dont le résultat est *vrai* ou *faux*. Les prédicats sont principalement utilisés lorsque les programmes doivent prendre une décision basée sur la valeur d'un résultat.

2. Ceci est très rare, et concerne uniquement de très petits microcontrôleurs.

5.4.1 Opérateur !

L'opérateur ! (prononcez « NON ») calcule l'opposé logique de son opérande, qui peut être logique ou entier. Son résultat est toujours de type `bool`, et il ne doit pas être confondu avec l'opérateur `~`.

x	0	1	34	true	false
!x	true	false	false	false	true
!(!x)	false	true	true	true	false

Figure 5.9 – Effet de l'opérateur ! (opérandes `int` ou `bool`)

5.4.2 Opérateur &&

L'opérateur `&&` (prononcez « ET ») calcule la conjonction logique de ses deux opérandes logiques ou entiers. En d'autres termes, il renvoie *vrai* si ses deux opérandes sont *vrais* (ou non nuls), et *faux* dans les autres cas. Cet opérateur ne doit pas être confondu avec `&`, qui réalise la même opération au niveau de chaque bit.

x	10	0	0	true	true
y	2	10	0	true	false
x && y	true	false	false	true	false

Figure 5.10 – Effet de l'opérateur `&&` (opérandes `int` ou bool)

Cet opérateur est évalué de manière *progressive*, ce qui signifie qu'il est garanti que l'opérande de gauche sera évalué avant l'opérande de droite. De plus, si l'opérande de gauche est évalué comme faux, l'autre ne sera pas évalué car il est certain que le résultat final sera faux. L'influence de ce type d'évaluation sur le code effectivement exécuté est très grande, comme le montre le code suivant, où `f` et `g` sont des fonctions [a] à résultat entier ou logique :

```
x = f() && g() ;
```

Dans cet exemple, la fonction `g` ne sera pas exécutée si la fonction `f` a retourné un résultat faux ou nul. Pour bien comprendre les implications de ce fonctionnement, comparez-le à celui de l'opérateur `+` dans une expression telle que : `x = f() + g()`. Dans celle-ci, il n'est pas garanti que `f` soit exécutée avant `g`. De plus, les deux fonctions sont systématiquement exécutées, quels que soient leurs résultats.

a. L'utilisation des fonctions sera décrite en section 6 (page 71).

5.4.3 Opérateur ||

L'opérateur `||` (prononcez « OU ») calcule la disjonction logique de ses deux opérandes logiques ou entiers. En d'autres termes, il renvoie `true` si au moins un de ses opérandes est vrai (ou non nul), et `false` dans les autres cas. Cet opérateur ne doit pas être confondu avec `|`, qui réalise la même opération au niveau de chaque bit.

x	10	0	0	true	false
y	2	10	0	false	false
x || y	true	true	false	true	false

Figure 5.11 – Effet de l'opérateur `||` (opérandes `int` ou `bool`)

Cet opérateur est évalué de manière *progressive*, ce qui signifie qu'il est garanti que l'opérande de gauche sera évalué avant l'opérande de droite. De plus, si l'opérande de gauche est évalué comme vrai, l'autre ne sera pas évalué car il est certain que le résultat final sera vrai. L'influence de ce type d'évaluation sur le code effectivement exécuté est très grande, comme le montre le code suivant, où `f` et `g` sont des fonctions à résultat logique ou entier :

```
x = f() || g() ;
```

Ici, la fonction `g()` ne sera pas exécutée si la fonction `f()` a retourné un résultat vrai ou non nul.

5.5 Opérateurs relationnels

5.5.1 Opérateurs <, >, <=, >=, ==, !=

Les opérateurs relationnels (table de la figure 5.12) permettent de comparer deux grandeurs en définissant une relation d'ordre entre elles. Ils peuvent être appliqués à des grandeurs entières ou flottantes, mais permettent aussi sous certaines contraintes de comparer des pointeurs. Ils renvoient tous *vrai* si la relation représentée par l'opérateur est vérifiée, et faux sinon.

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

Figure 5.12 – Signification des opérateurs relationnels en C++

Le tableau de la figure 5.13 montre le résultat de différentes expressions comportant un opérateur relationnel.

Expression	Valeur
a	3
b	8
`a > b`	`false`
`a < b`	`true`
`a <= b`	`true`
`a >= b`	`false`
`a != b`	`true`
`a == b`	`false`

Figure 5.13 – Exemples d'application d'opérateurs relationnels à des données entières.

5.6 Opérateurs d'affectation

5.6.1 Opérateur =

L'opérateur = permet de transférer la valeur qui est à sa droite dans la variable qui est à sa gauche. Notez bien dans cette phrase les termes *valeur* et *variable*. Fondamentalement, une variable est quelque chose dont on peut changer la valeur. Par exemple dans

```
x = 3+y ;
```

`x` est une variable, mais `3+y` est une valeur calculée comme étant la somme de la constante `3` et de la valeur de la variable `y`. Bien entendu, l'expression

```
3+y = x; /* FAUX */
```

n'a pas de sens, car l'expression `3+y` n'est pas une variable (elle n'est donc pas affectable). Une telle expression sera donc rejetée par le compilateur.

Dans la terminologie du langage C++, on appelle *lvalue* ce qui peut être placé à gauche (left) de l'opérateur `=`, et *rvalue* ce qui doit être placé à droite (right). Dans les exemples précédents, `x` est donc une lvalue, alors que `3+y` est une rvalue. N'en déduisez pas que le résultat d'une expression ne peut pas être une lvalue. Par exemple dans

```
*p = 3 ;
```

l'expression `*p` correspond bien à une variable obtenue en déréférençant le pointeur `p` au moyen de l'opérateur `*` : c'est une lvalue.

En outre, il est important de comprendre que `=` est réellement un *opérateur*, et non une *instruction* comme dans certains langages. Ceci signifie qu'en plus de son effet principal (la copie de valeur), il génère un résultat. Ce résultat est la valeur de son opérande de gauche, après l'affectation. L'opérateur `=` peut donc être utilisé dans n'importe quelle expression arbitrairement complexe. Par exemple, dans le code qui suit, les valeurs finales de `a`,`b` et `c` seront respectivement `4`, `3` et `3`.

```
int a = 1, b = 2 , c = 3;
a = (b = c) + 1 ;
```

L'opérateur `=` ne doit pas être confondu avec `==`. Cette erreur est très fréquente et ne peut malheureusement pas être détectée par le compilateur, car elle mène presque toujours à une instruction légale. Par exemple :

```
int a = 1, b = 2 , c = 3;
/* Douteux mais légal : le résultat est 4 */
a = (b = c) + 1 ;
/* Douteux mais légal : le résultat est 1 */
a = (b == c) + 1 ;
/* Probablement faux mais légal : condition toujours vraie */
if( a = b )   // ...
```

Dans la dernière situation, les compilateurs émettent généralement un avertissement et vous invitent à corriger votre code. Si vous voulez réellement réaliser une affectation dans le prédicat de ce test, vous devez confirmer votre intention en ajoutant une paire de parenthèses.

```
/* Si l'on désire réellement réaliser une affectation dans le test */
if( (a = f(b)) )  // ...
```

L'instruction `if` est décrite en section 9.2, page 97.

5.6.2 Opérateurs de calcul « en place »

Il arrive assez souvent qu'on soit amené à écrire des expressions dans lesquelles une variable reçoit une valeur dépendant de sa valeur antérieure. Voici un exemple de cette situation :

```
x = x * (2+y);
```

Lorsque ceci se produit, il est possible d'utiliser une expression équivalente plus simple en utilisant un opérateur qui combine le calcul et l'affectation du résultat de ce calcul, de la manière suivante :

```
x *= (2+y);
```

Ce type d'opérateur permet d'écrire un code plus concis. Ce code est facile à compiler (même pour un compilateur peu optimisant), car il correspond bien aux instructions disponibles sur les processeurs.

Il existe un opérateur de calcul « en place »[3] pour chaque opérateur à deux opérandes de type arithmétique, bit à bit ou de décalage. Le tableau de la figure 5.14 donne quelques exemples d'utilisation d'opérateurs de calcul en place. On peut remarquer que les expressions :

```
x =  x + 1;
x += 1 ;
x++ ;
++x ;
```

ont exactement le même effet. En théorie, ces expressions n'ont pas la même efficacité (la moins efficace étant la première). Cependant, en pratique, tous les compilateurs modernes génèrent exactement le même code pour chaque ligne de cet exemple, si bien que le choix d'une écriture plutôt qu'une autre est essentiellement une question de goût personnel.

Expression	Signification	Ancienne valeur de `a`	Nouvelle valeur de `a`
`a += 2`	`a = a + 2`	2	4
`a <<= 2`	`a = a << 2`	2	8
`a >>= 2`	`a = a >> 2`	2	0
`a |= 2`	`a = a | 2`	4	6
`a &= 2`	`a = a & 2`	6	2

Figure 5.14 – Exemples d'application d'opérateurs de calcul en place.

5.7 Opérateurs relatifs aux pointeurs

Il n'existe que deux opérateurs spécifiquement liés aux pointeurs. Si vous n'êtes pas à l'aise avec la notion de pointeur, relisez la section 4.4.6 (page 47) où ce concept est présenté de manière plus détaillée avant de lire la suite.

5.7.1 Opérateur de prise d'adresse (&)

En langage C++, l'opérateur `&` doté d'un seul opérande permet d'obtenir l'adresse de n'importe quelle variable, fonction ou objet. Pour une entité `e`, de type `E`, l'expression `&e` fournit une valeur du type « pointeur sur un `E` », que l'on peut aussi noter `E*`. Lorsque l'entité pointée est constante, c'est-à-dire de type `const E`, le pointeur généré par l'opérateur sera de type `const E*`.

```
int *pi ;      /* pi est un pointeur sur un int */
double *pd ;   /* pd est un pointeur sur un double */
int k ;
pi = &k ;      /* OK */
pd = &k ;      /* FAUX: pointeurs de types différents */
```

Ne confondez pas le `&` à un seul opérande avec le `&` à deux opérandes qui n'a pas du tout le même rôle. Par exemple :

```
int *pi ;
int j,k ;
pi = &k ;      /* un seul opérande : prise d'adresse */
j = j & k ;    /* deux opérandes : ET bit à bit */
```

3. Le terme *en place* provient du fait que les calculs sont directement réalisés sur les opérandes, sans variable intermédiaire.

Rappelez-vous qu'une constante littérale ou le résultat d'un calcul ne résident pas forcément en mémoire. Leur appliquer l'opérateur `&` n'a donc pas de sens. Voici quelques exemples :

```
int *pi, **ppi ;  /* pi et ppi sont des pointeurs */
int k ;
pi  = &k ;        /* OK */
ppi = &(&k) ;     /* FAUX : l'opérande est une rvalue */
pi  = &(k+1) ;    /* FAUX : l'opérande est une rvalue */
pi  = &0 ;        /* FAUX : l'opérande est une constante littérale */
```

5.7.2 Opérateur de déréférencement (*)

L'opérateur `*` doté d'un seul opérande permet de réaliser un déréférencement.
Cette opération consiste à accéder à une donnée dont on connaît l'adresse : elle ne peut donc être appliquée qu'à une expression de type pointeur. Voici quelques exemples de déréférencement :

```
int i ;
int *pi ;
pi = &i;          /* pi pointe sur i */
*pi = 10 ;        /* OK: met 10 dans i */
*i = 0 ;          /* FAUX : i n'est pas un pointeur */
```

Vous ne devez pas confondre le `*` à un seul opérande avec le `*` à deux opérandes qui n'a pas du tout le même rôle. L'exemple suivant montre une expression qui utilise les deux types d'opérateur `*`.

```
*pi = *pi * *pi ; /* Calcule i = i * i ,  si pi pointe sur i */
```

5.8 Opérateurs divers

5.8.1 Opérateur conditionnel (? :)

L'opérateur conditionnel est le seul opérateur du langage C++ faisant intervenir trois opérandes. Il est en outre composé de deux lettres séparées : « `?` » et « `:` » . Une expression conditionnelle est de la forme :

```
A ? B : C
```

où `A`, `B` et `C` représentent des sous-expressions. Dans une expression conditionnelle, la valeur de `A` est évaluée la première. Cette valeur est un prédicat. Si elle est *vraie* (ou non nulle), le résultat de l'expression est celui de `B`, sinon c'est celui de `C`. Notez que dans une expression conditionnelle, *une seule* des deux sous-expressions `B` ou `C` est évaluée.

Cet opérateur permet donc de calculer un résultat qui dépend d'une condition arbitraire, et autorise une notation très compacte. À titre d'exemple, le code suivant réalise l'opération $max(x, y) \rightarrow z$.

```
z = (x > y) ? x : y ;
```

5.8.2 Opérateur de séquencement (,)

Voici un opérateur hors du commun, puisqu'il ne réalise aucun calcul, et n'en est pas moins extrêmement utile ! Il s'utilise sous la forme

```
A , B
```

où `A` et `B` sont des sous-expressions quelconques. L'opérateur « `,` » réalise l'exécution séquentielle de `A` *et* de `B`, *dans cet ordre* [4]. Le résultat final est celui de `B`, tandis que le résultat de `A` est simplement ignoré. Cet opérateur est extrêmement utile dans toutes les situations où le langage exige une (et une seule) expression, mais où le programmeur voudrait en évaluer plusieurs.

L'exemple suivant montre comment ajouter une minute à une durée exprimée en heures et minutes, et stockée dans deux variables **`heures`** et **`minutes`**. Grâce aux opérateurs « `,` » et « `?:` », une seule expression suffit à résoudre le problème.

```
minutes = (minutes >= 59) ? (++heures, 0) : minutes+1 ;
```

Ce code fonctionne de la façon suivante : si le nombre de minutes courant est inférieur à 59, nous rajoutons simplement une minute à la durée (et le nombre d'heures demeure inchangé) comme si l'on avait exécuté :

```
minutes = minutes+1 ;
```

Sinon, nous incrémentons le nombre d'heures, *et* rendons nul le nombre de minutes. Ces deux opérations sont ici réalisées grâce à l'expression `(++heures, 0)`, qui est en réalité composée de deux sous-expressions connectées par un opérateur « `,` ». Tout se passe donc comme si on avait exécuté :

```
minutes = (++heures, 0) ;
```

qui signifie

```
heures = heures + 1 ;
minutes = 0 ;
```

Pour des raisons de priorité relative entre opérateurs, les parenthèses sont indispensables dans cette expression. Reportez-vous à la section 5.9 (page 66), qui traite la question des priorités.

Dans l'expression `x=f(A,B,C)` où `A`,`B` et `C` sont des expressions quelconques, la virgule sert uniquement à séparer les arguments de la fonction `f` ; *elle n'est donc pas un opérateur*. Dans cet exemple, la fonction `f` reçoit donc trois paramètres.

Si vous désirez dans un tel contexte que la virgule soit considérée comme un opérateur, il suffit d'ajouter une paire de parenthèses, comme suit : `x=f((A,B,C))`. Dans ce cas, les expressions `A`, `B` et `C` sont évaluées dans cet ordre, et la fonction `f` ne reçoit qu'un seul paramètre, qui est le résultat de la sous-expression `C`.

5.8.3 Opérateur de transtypage (cast)

Pour traiter les calculs entre données de types différents, le langage C++ propose un mécanisme automatique de conversion de type qui est décrit en section 5.12, page 69. Il existe toutefois des situations dans lesquelles on peut avoir besoin de convertir explicitement le type d'une donnée avant de l'utiliser. L'opérateur de *transtypage* (qu'on appelle aussi *cast*) ne ressemble pas aux opérateurs habituels. En effet, il comporte un nom de type dans une paire de parenthèses :

```
x = (double)(k) ; /* la valeur de k est explicitement convertie en double */
```

Supposons par exemple que nous ayons envie de calculer le taux de voyelles dans un texte. Pour cela, il suffit de parcourir le texte et de compter le nombre de voyelles **`nbVoyelles`** et le nombre total de lettres **`nbLettres`**. Le taux de voyelles peut ensuite être calculé par :

```
double tauxVoyelles = nbVoyelles / nbLettres ; /* Hum... */
```

4. Les opérateurs « `,` » « `&&` » et « `||` » sont les seuls opérateurs du langage C à garantir une évaluation de l'opérande de gauche avant celui de droite.

Hélas, ce code est faux, car le taux obtenu sera `1.0` si le texte ne comporte que des voyelles, et `0.0` dans les autres cas. Ceci provient du fait qu'une division entre entiers produit toujours un entier, qui est la partie entière du résultat. Pour éviter ceci, il suffit de convertir un des deux opérandes de la division en `double` : le calcul sera alors réalisé en arithmétique flottante.

```
double tauxVoyelles = (double)(nbVoyelles) / nbLettres ;
```

Soyez bien conscients que l'opérateur de transtypage ne peut pas convertir une citrouille en carrosse. Même autorisé, un transtypage est toujours fait sous la responsabilité du programmeur. En résumé, considérez que *ce genre de transtypage est réservé aux conversions entre types numériques*, et qu'il faut s'assurer de l'absence de débordement numérique ou de troncature. La conversion de pointeurs à l'aide de cet opérateur est également possible, *mais toujours risquée*.

N'utilisez *jamais* un transtypage pour « faire plaisir » au compilateur lorsqu'il se plaint qu'une expression est mal typée. Demandez-vous plutôt *pourquoi* cette expression est mal typée.

Ce transtypage provient du langage C, et il est toujours disponible en C++, qui est un sur-ensemble du C. Il est toutefois *très fortement déconseillé* de l'utiliser, car C++ propose une méthode de conversion plus sûre, qui n'est pas basée sur un opérateur. Dans notre exemple, il aurait fallu écrire :

```
double tauxVoyelles = static_cast<double>(nbVoyelles)/nbLettres ; // Transtypage C++
```

Cette question importante est rapidement traitée en section 17.7 (page 181), puis en détail au chapitre 41 (page 551), qui présente également d'autres transtypages C++ répondant à des problématiques différentes.

5.9 Priorité et associativité des opérateurs

5.9.1 Priorité des opérateurs

La notion de priorité est bien connue en arithmétique. Si, par exemple, on vous demande la valeur de l'expression

```
2 + 3 * 4
```

vous allez probablement répondre `14`, ce qui est la bonne réponse, compte tenu du fait que l'on considère généralement que les multiplications doivent être exécutées avant les additions. Le langage C++ respecte cette convention, et attribue à chaque opérateur une priorité permettant de décider de l'ordre d'évaluation. Toutefois, il n'est pas facile de retenir la priorité de chaque opérateur, car il existe 17 niveaux de priorité pour une soixantaine d'opérateurs.

La table de la figure 5.15 indique la priorité relative des opérateurs. Elle débute par les plus prioritaires, et les opérateurs de même priorité sont placés dans la même case. Vous noterez que certains opérateurs figurent deux fois dans cette table : il s'agit de ceux qui admettent un ou deux opérandes, avec des sens différents.

Si vous avez un doute concernant la priorité des opérateurs, il suffit d'en spécifier explicitement l'ordre d'application en ajoutant des parenthèses à l'expression, comme dans le code suivant :

```
x = 2 + 3 * 4 ;    /* 14 */
x = (2 + 3) * 4 ;  /* 20 */
```

Opérateurs *(par priorité décroissante)*	**Associativité**
`::`	→
`() [] -> . type() type{}` `++ --` (*postfixés*)	→
`! ~ (type) sizeof` `++ --` (*préfixés*) `+ - * &` (*1 op.*) `new new[] delete delete[]` (*1 op.*) `co_await` *(C++20)*	←
`.* ->*`	→
`*` (*2 op.*) `/ %`	→
`+ -` (*2 op.*)	→
`<< >>`	→
`<=>` (*C++20*)	→
`< <= > >=`	→
`== !=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→

`?:  throw`
`co_yield` *(C++20)*
`= += -= *= /= %= &= ^= |= <<= >>=`
	←
	←

`,`	→

Figure 5.15 – Priorité et associativité des opérateurs en C++

5.9.2 Associativité des opérateurs

La priorité des opérateurs ne suffit pas à lever toutes les ambiguïtés liées à l'ordre d'évaluation. Considérons par exemple le code suivant :

```
int a = 3 , b = 4 , c = 5 , d ;
d = a - b - c ;   /* Quel est le résultat ? */
```

Les deux opérateurs de calcul intervenant ici ayant la même priorité, il est important de savoir si les opérateurs associent à gauche ou à droite. En effet, dans le premier cas, le résultat vaudra -6, alors que dans le second il vaudra 4, comme dans le code ci-dessous.

```
d = (a - b) - c ;   /* Association à gauche : -6 */
d = a - (b - c) ;   /* Association à droite :  4 */
```

En C++, la grande majorité des opérateurs (dont l'opérateur « - » utilisé dans l'exemple) associent à gauche : le résultat de notre calcul sera donc -6 en l'absence de parenthèses. Les opérateurs qui associent à droite sont les opérateurs réalisant une affectation, les opérateurs à un seul opérande, et l'opérateur conditionnel, comme l'indique la seconde colonne du tableau de la figure 5.15.
L'association à droite se révèle intéressante dans le cas de l'opérateur =, puisque cela permet les affectations multiples, comme dans le code suivant :

```
a = b = c = 0 ;  /* équivalent à a = (b = (c = 0)) ; */
```

qui a le même effet que

```
c = 0 ; b = c ; a = b ;
```

5.10 Ordre d'évaluation des opérandes avant C++17

L'ordre d'évaluation des opérandes est une notion qui est souvent confondue avec l'ordre d'application des opérateurs. Par exemple, dans l'expression

```
x = f1() + (f2() - f3()) ;
```

nous savons que la soustraction est appliquée avant l'addition, mais cela ne nous dit rien de l'ordre d'évaluation de leurs opérandes. Est-ce que la fonction `f2` est exécutée avant `f3` ou `f1` ? Avant C++17, on ne pouvait pas répondre à cette question parce que le compilateur était libre d'évaluer les opérandes dans l'ordre permettant d'optimiser le code.

Dans beaucoup de situations, l'ordre d'évaluation n'a pas d'importance, mais il arrive qu'il en ait. C'est le cas lorsque le calcul d'un opérande produit un effet de bord qui impacte le calcul des autres opérandes. Par exemple dans

```
int k = 3, j = 2 ;
x = k++ + (k * j) ; /* x reçoit-il 9 ou 11 ? */
```

la valeur affectée sera `3+6` si la sous-expression de droite est évaluée la première, ou `3+8` si c'est celle de gauche.

Dans cet exemple, l'effet produit par l'opérateur `++` est visible, mais ce n'est pas toujours le cas. Par exemple, les appels de fonctions peuvent aussi provoquer des effets de bord inattendus.

```
/* Résultat imprévisible si g ou f modifient un état commun */
x = f() + g() ;
```

De la même manière, l'ordre d'évaluation des paramètres d'une fonction n'est pas spécifié. Par conséquent, dans le code suivant il est impossible de prédire si `f` reçoit les valeurs `3,3` ou `4,3`.

```
int k = 3 ;
x = f(k, k++) ;
```

En résumé, *une variable subissant un effet ne doit pas apparaître plusieurs fois dans la même expression*. Il suffit pour cela d'exprimer clairement son intention en décomposant les expressions complexes en plusieurs expressions simples. Par exemple :

```
int k = 3, j = 2 ;
x = k + (k * j) ; /* x reçoit 9 */
++k ;             /* k reçoit 4 */
```

De plus, il existe quatre opérateurs qui ne peuvent pas poser de problème, car leurs opérandes sont évalués *par construction* dans un ordre défini :

(a) `&&` et `||` L'évaluation progressive (section 5.4.2 et 5.4.3 (page 60)) implique que l'opérande de gauche doit être évalué le premier.

(b) `? :` La logique de cet opérateur (section 5.8.1, page 64) implique que la condition doit être évaluée la première.

(c) `,` L'*opérateur de séquencement* (section 5.8.2, page 64) a justement été créé pour garantir une évaluation de ses opérandes de gauche à droite.

5.11 Ordre d'évaluation des opérandes à partir de C++17

Afin de minimiser les risques liés à l'ordre d'évaluation des opérandes, celui-ci est maintenant fixé *pour les opérateurs figurant dans le tableau de la figure 5.16*. Dans celui-ci, les opérandes sont évalués dans l'ordre `e1`, `e2`, puis `e3`. L'opérateur `@=` est ici une notation générale représentant tous les opérateurs de calcul « en place » tels que `+=`, `-=`, `*=`, etc.

Opérateur	Ordre d'évaluation garanti par C++17
	De gauche à droite
`.`	`e1.e2`
`->`	`e1->e2`
`->*`	`e1->*e2`
`()`	`e1(x, y, z)`
`[]`	`e1[e2]`
`<<`	`e1<<e2<<e3`
`>>`	`e1>>e2>>e3`
	De droite à gauche
`=`	`e2=e1`
`@=`	`e2@=e1`

Figure 5.16 – Ordre d'évaluation des opérandes en C++ 17

Cette table montre que l'opérande de gauche est généralement évalué avant celui de droite (ce qui limite les surprises), sauf en ce qui concerne les affectations et leurs variantes.

Même en C++17 et après, *l'ordre d'évaluation des arguments d'une fonction n'est pas défini*. Il en va de même des opérandes des opérateurs qui ne figurent pas dans la table. Il faut donc rester attentif aux expressions comportant des effets de bord.

5.12 Expressions hétérogènes et conversion automatique de type

Il est fréquent d'utiliser en langage C ou C++ des expressions faisant intervenir des données de types différents. Or, ces expressions ne peuvent pas être évaluées directement parce que les processeurs ne comportent généralement pas d'instructions dont les opérandes sont de types différents.

Pour cette raison, le compilateur est amené à introduire des *opérations de conversion de type* dans beaucoup d'expressions. L'objet de cette section est de préciser les règles selon lesquelles ces opérations sont appliquées. Il faut tout d'abord faire trois remarques :

- Certaines conversions n'ont pas de sens ou ne peuvent pas être réalisées. Dans ce cas le compilateur signalera une erreur de programmation. Pour simplifier, partez du principe que seules les conversions entre types numériques peuvent être faites de manière automatique.
- Les règles de conversion visent à éviter ou à minimiser les pertes d'information. Cette remarque constitue une grille de lecture qui permet de comprendre facilement ces règles.
- Certaines conversions sont appliquées uniquement pour optimiser l'usage du processeur.

Concernant les conversions automatiques, la norme est trop technique pour être entièrement reproduite ici. Voici toutefois ce que vous devez retenir :

(a) Si un des opérandes est entier et l'autre flottant, l'opérande entier est converti dans le type de l'opérande flottant.

(b) Si les deux opérandes sont flottants, l'opérande ayant le type le moins précis est converti dans le type de l'autre opérande. Dans l'ordre de précision croissante, les types flottants sont : `float`, `double` et `long double`.

(c) Si les deux opérandes sont entiers, l'opérande dont la représentation a la plus petite taille est converti dans le type de l'autre opérande. Dans l'ordre des tailles croissantes, les types entiers sont : `char`, `short`, `int`, `long int` et `long long int`. Les versions non signées de ces types (ex : `unsigned int`) font la même taille que leurs équivalents signés.

(d) Finalement, si les deux opérandes entiers ont la même taille, mais que l'un est signé et l'autre pas, l'opérande signé est converti dans le type de l'autre opérande.

Afin de fixer les idées, le tableau suivant donne quelques exemples de différentes situations, et la conversion qui est effectuée dans chaque cas [5].

Cas	Type op. 1	Type op. 2		Type op. 1	Type op. 2
(a)	`double`	`char`	→	*inchangé*	`double`
	`float`	`long`	→	*inchangé*	`float`
(b)	`double`	`float`	→	*inchangé*	`double`
	`long double`	`double`	→	*inchangé*	`long double`
(c)	`int`	`char`	→	*inchangé*	`int`
	`unsigned int`	`unsigned char`	→	*inchangé*	`unsigned int`
	`long`	`unsigned int`	→	*inchangé*	`long`
	`unsigned int`	`short`	→	*inchangé*	`unsigned int`
	`short`	`char`	→	`int`	`int`
(d)	`unsigned int`	`int`	→	*inchangé*	`unsigned int`

Figure 5.17 – Conversions automatiques de types dans diverses situations

⚠ Il faut être conscient du fait que, malgré les précautions, certaines conversions *peuvent* provoquer des pertes d'information. Par exemple, la conversion de `1234567890L` (qui est de type `long`) en `float` donnera la valeur `1.23457e+09`, dont l'ordre de grandeur est correct, mais dont les derniers digits décimaux ont été tronqués [a].

Par ailleurs, la conversion d'un entier signé en entier non signé ne sera correcte que si la valeur d'origine était positive. En effet cette conversion n'est qu'une réinterprétation du profil binaire du nombre d'origine, qui n'est absolument pas modifié. Par exemple, si les `int` sont codés en complément à deux sur 32 bits, l'entier `-1` est codé par une succession de 32 bits à 1. L'interprétation de ce profil comme un nombre non signé donne la valeur $2^{32} - 1$, soit `4294967295`.

a. Cet exemple suppose que les `long` sont codés sur 32 bits.

5.13 Promotion entière

Indépendamment de la question des conversions automatiques, tous les opérandes d'une expression entière dont la taille est inférieure à celle des `int` sont systématiquement convertis en `int` (ou `unsigned int` s'ils sont non signés) avant le calcul. Par exemple, la somme de deux `char` est un `int`. Ce mécanisme, nommé *promotion entière*, implique que le résultat d'un calcul entre entiers ne peut pas être plus petit qu'un `int`. Ceci provient du fait que le type `int` est celui qui, par définition, garantit les meilleures performances sur le processeur cible.

L'exemple suivant montre que la différence de deux `char` n'est pas de type `char` : elle de type `int`, et codée sur 4 octets sur ma machine.

```
std::cout << sizeof 'A' << std::endl ;
std::cout << sizeof 'B' << std::endl ;
std::cout << sizeof ('B'-'A') << std::endl ;
```

```
1
1
4
```

5. On suppose ici que les `long` utilisent davantage de mémoire que les `int`.

6 | Fonctions

6.1 Fonctions et sous-programmes

6.1.1 Contexte informationnel des fonctions

Les fonctions ont déjà été décrites en section 3.1 (page 35) comme étant des morceaux de code que l'on assemble pour construire des programmes. En C++, chaque fonction dispose de son propre contexte de travail, c'est-à-dire que les données qu'elle manipule n'ont, *a priori*, aucun rapport avec les données manipulées par les autres fonctions. L'isolation des contextes des fonctions est généralement considérée comme une bonne propriété, car elle garantit que l'activité d'une fonction ne perturbera pas celles des autres. Toutefois, il est évident que pour que deux « morceaux » de programme puissent coopérer, il faut qu'ils puissent *aussi* échanger des informations.

Nous allons donc affiner la présentation des fonctions en nous intéressant aux échanges de données entre une fonction et le code qui l'utilise. Cet échange est réalisé par un mécanisme basé sur des *paramètres*. Un paramètre est une variable qui sert de « porte d'entrée » aux données manipulées par la fonction. Dans certaines circonstances, ce paramètre peut aussi servir de « porte de sortie », c'est-à-dire qu'il peut servir à transmettre des informations de la fonction vers le code qui l'utilise.

Les paramètres sont à l'origine de la souplesse d'utilisation des fonctions, puisqu'ils permettent de réaliser très simplement des calculs identiques portant sur des données différentes.

6.1.2 Définition d'une fonction

Une fonction est définie de la manière suivante :

```
type_du_résultat   nom_de_la_fonction(  liste_des_paramètres  )
{
    code_de_la_fonction
}
```

À titre d'exemple, la fonction suivante calcule la valeur d'un polynôme du second degré $ax^2 + bx + c$ à partir d'une valeur x et de coefficients a, b et c.

```
1 double poly2(double x, double a, double b, double c)
2 {
3   double resultat ;
4   resultat = a*x*x + b*x + c ;
5   return resultat ;
6 }
```

Cette fonction s'appelle **poly2**, et génère un résultat de type **double**. Elle reçoit quatre paramètres (**x**, **a**, **b**, **c**) qui sont également de type **double**. Elle utilise une variable locale **resultat** qui sert au stockage temporaire de la valeur du polynôme au point **x**. Le calcul lui-même est trivial.

L'instruction **return** nouvellement introduite ici a deux rôles : elle permet à la fois *d'indiquer la valeur du résultat* calculé par la fonction (qui est celle de la variable **resultat**), et *provoque la sortie de la fonction,* c'est-à-dire un retour au code appelant [1].

1. Vous trouverez à la section A.1.1.5, page 794 des explications techniques sur le mécanisme d'appel de fonctions.

La variable `resultat` permet d'expliciter le fait que l'on est en train de calculer le résultat de la fonction, mais n'est techniquement pas utile. On aurait donc pu écrire :

```
1 double poly2(double x, double a, double b, double c)
2 {
3   return  a*x*x + b*x + c ;
4 }
```

Une fonction comportant plusieurs instructions `return` est légale [2], mais la première instruction `return` exécutée provoquera le retour au code appelant. Par exemple, la version suivante de la fonction `poly2` renverra toujours `0`, quelle que soit la valeur de la variable `resultat`. Notez que ce code provoquera l'émission d'un avertissement à la compilation, car le compilateur va détecter que le second `return` ne peut pas être exécuté.

```
1 double poly2(double x, double a, double b, double c)
2 {
3   double resultat ;
4   resultat = a*x*x + b*x + c ;
5   return 0.0 ;      /* la fonction retourne ici .. */
6   return resultat ; /* .. donc ce code ne sera jamais exécuté */
7 }
```

6.1.3 Utilisation d'une fonction

Une fois définie, une fonction peut être invoquée (on dit aussi *appelée*) pour réaliser le calcul prévu à partir des données passées en paramètres. Les instructions

```
y = poly2(z, 0.5, 3.0, -6.0) ;
u = poly2(w, 4.0, 2.0, k*k) ;
```

réalisent par exemple les opérations $0.5z^2 + 3z - 6 \to y$ puis $4w^2 + 2w + k^2 \to u$ grâce à la *même* fonction `poly2`. Ceci montre que le mécanisme des paramètres permet de réaliser facilement des calculs portant sur des données différentes à partir d'un code unique.

6.2 Paramètres réels et formels

Dans les sections précédentes, on utilise le terme *paramètre* pour désigner indifféremment deux types de paramètres qu'il faut cependant absolument distinguer :

- Les *paramètres formels* sont ceux qui figurent dans la *définition de la fonction* (`x`, `a`, `b` et `c` pour la fonction `poly2`).
- Les *paramètres réels* sont ceux qui sont fournis lors de *l'appel de la fonction* (`z`, `0.5`, `3.0`, `-6.0` pour le premier appel de `poly2`). On les appelle aussi *arguments* de la fonction.

Remarquez que les paramètres formels sont toujours des variables, alors que les paramètres réels (ou arguments) peuvent être n'importe quelle expression dont le résultat a un type compatible avec la définition de la fonction.

2. Une fonction comportant plusieurs points de sortie peut être difficile à maintenir. Ce style de programmation doit donc être évité.

6.2.1 Mécanisme de passage des paramètres

On appelle passage (ou transmission) de paramètres l'opération consistant à mettre en correspondance les paramètres réels et formels. Il existe en C++ deux mécanismes de passage de paramètres : le passage *par valeur* (qui est hérité du langage C), et le passage *par référence*, qui est spécifique à C++. On ne traitera que du premier dans cette section, car le second s'appuie sur la notion de référence, qui n'a pas encore été présentée. Le passage par référence est présenté en détail en section 17.4 (page 173), qui couvre aussi les autres questions liées aux références.

Le mécanisme de passage par valeur est plutôt simple : la *valeur* de chaque paramètre réel est recopiée dans le paramètre formel correspondant juste avant l'exécution de la fonction. Par exemple, pour l'appel

```
y = poly2(z, 0.5, 3.0, -6.0) ;
```

le paramètre formel `x` prendra la valeur du paramètre réel `z`. Tout se passe donc *comme si* l'on avait écrit `x=z` juste avant d'exécuter le code de la fonction [3]. Les autres paramètres sont traités de manière identique.

Il est très important de comprendre que pendant l'exécution de la fonction de cet exemple, `x` et `z` sont *vraiment deux variables distinctes*. Elles sont donc situées à des adresses différentes. Le seul lien entre ces deux paramètres est donc que leurs valeurs sont identiques lorsque la fonction commence son exécution.

Il n'existe aucun lien entre le nom d'un paramètre formel et celui du paramètre réel correspondant. Ils peuvent être identiques ou non, cela n'a aucune importance.

6.2.2 Paramètres d'entrée et de sortie

Le paragraphe précédent permet de comprendre pourquoi une fonction peut modifier un paramètre formel sans que le paramètre réel correspondant en soit affecté. Pour illustrer ceci, nous allons réaliser une fonction dont le rôle est de multiplier la valeur d'une variable par 10.
Voici une version naïve de cette fonction de multiplication. Pour mettre en évidence ce qui se passe, nous affichons les valeurs des paramètres réels et formels en différents points du code.

```
1 /* Je sens que ça ne va pas fonctionner .. */
2 void mul10(double x)
3 {
4   x = x * 10 ;
5   std::cout << "x=" << x << std::endl ;
6 }
```

Utilisons maintenant la fonction `mul10` :

```
double z = 3 ;
mul10(z) ;
std::cout << "z=" << z << std::endl ;
```

Voici le résultat de l'exécution :

```
x=30
z=3
```

3. En réalité, il n'est pas possible d'écrire cela, car `x` est une variable locale à `poly2` : elle n'est donc pas visible depuis l'extérieur de cette fonction.

Ceci montre que la fonction `mul10` a bien reçu la valeur `3` par l'intermédiaire du paramètre `x` et a bien multiplié cette variable par `10`.
La deuxième valeur affichée montre que le paramètre réel `z` n'a pas été modifié par l'exécution de la fonction. Ceci n'a rien d'étonnant, car la variable `x` n'étant qu'une *copie* de `z`, il n'y a aucune raison pour que la modification de cette copie affecte l'original. Comme annoncé, cette expérience montre qu'une fonction C ne peut pas modifier un de ses paramètres réels. On dit aussi que les paramètres d'une fonction passés par valeur *sont toujours des paramètres d'entrée*, puisqu'ils permettent d'introduire des informations dans la fonction, mais pas d'en extraire.

Cette caractéristique est intéressante, car elle permet de garantir qu'une fonction ne peut pas détruire accidentellement des données qui lui sont externes. Mais elle semble aussi terriblement limitative, puisqu'elle interdit d'écrire des fonctions aussi simples que `mul10`. Heureusement, il n'en est rien.

Comment permettre à `mul10` de modifier une variable quelconque, par exemple la variable `z` ?
La réponse est simple : il suffit de lui fournir l'adresse de cette variable. Autrement dit, le paramètre de la fonction `mul10` doit être un *pointeur* :

```
1 /* Le paramètre est maintenant un pointeur */
2 void mul10V2(double *px)
3 {
4   *px = *px * 10 ;   // Ou bien : *px *= 10 ;
5   std::cout << *px << std::endl ;
6 }
```

Le paramètre de `mul10V2` (`mul10` version 2) est maintenant un pointeur sur un `double`. En déréférençant ce pointeur (avec l'opérateur `*`), nous pouvons accéder à la donnée pointée pour utiliser sa valeur, mais aussi pour la modifier comme à la ligne 4.

Bien entendu, il faudra aussi passer à la fonction `mul10V2` l'adresse de la variable à modifier :

```
double z = 3 ;
mul10V2(&z) ; /* Noter le & */
```

Ce code ne diffère du précédent que par l'usage de l'opérateur de prise d'adresse (`&`) qui permet de générer un pointeur sur `z`. La fonction `mul10V2` rend bien le service attendu, car les expressions `z` (à l'extérieur de la fonction) et `*px` (à l'intérieur de la fonction) *désignent physiquement la même variable*.


On dit parfois que ce genre de technique constitue un *passage de paramètre par adresse*. Nous avons en réalité simulé ce passage par adresse en passant un pointeur, qui est lui-même passé par valeur. Mais ce qui compte en définitive est que le service requis soit rendu. Nous verrons en section 17.4 (page 173) que le langage C++ propose un authentique mécanisme de passage de paramètres par adresse, qui s'appuie sur la notion de *référence*.


Résumé 3

Si vous désirez qu'une fonction vous fournisse une information via un de ses paramètres *passé par valeur*, ce paramètre doit être un pointeur sur la variable (ou plus généralement sur la zone mémoire) dans laquelle vous désirez que cette information soit stockée.

6.2.3 Validité des paramètres réels

Utiliser une fonction de façon incorrecte peut avoir des conséquences catastrophiques. C'est pourquoi le compilateur vérifie chaque appel, et signale les erreurs qu'il est capable de détecter. En particulier, un appel de fonction dans lequel le nombre de paramètres réels n'est pas le même que le