Le guide complet du langage C

Claude Delannoy

EYROLLES

 ISBN : 978-2-212-14012-5

Table des matières

CHAPITRE 5

CHAPITRE 14

CHAPITRE 15

CHAPITRE 16

CHAPITRE 17

CHAPITRE 18

CHAPITRE 19

CHAPITRE 20

CHAPITRE 21

CHAPITRE 22

ANNEXE A

Avant-propos

À qui s'adresse ce livre ?

L'objectif de ce livre est d'offrir au développeur un outil de référence clair et précis sur le langage C tel qu'il est défini par la norme ANSI/ISO. Il s'adresse à un lecteur possédant déjà de bonnes notions de programmation, qu'elles aient été acquises à travers la pratique du C ou de tout autre langage. La vocation première de l'ouvrage n'est donc pas de servir de manuel d'initiation, mais plutôt de répondre aux besoins des étudiants avancés, des enseignants et des développeurs qui souhaitent approfondir leur maîtrise du langage ou trouver des réponses précises aux problèmes techniques rencontrés dans le développement d'applications professionnelles.

L'ouvrage a été conçu de façon que le lecteur puisse accéder efficacement à l'information recherchée sans avoir à procéder à une lecture linéaire. La tâche lui sera facilitée par un index très détaillé, mais aussi par la présence de nombreuses références croisées et de tableaux de synthèse servant à fois de résumé du contenu d'une section et d'aiguillage vers ses différentes parties. Il y trouvera également de nombreux encadrés décrivant la syntaxe des différentes instructions ou fonctions du langage, ainsi que des tableaux récapitulatifs et des canevas types.

Comme il se doit, nous couvrons l'intégralité du langage jusque dans ses aspects les plus marginaux ou les moins usités. Pour qu'une telle exhaustivité reste exploitable en pratique, nous l'avons largement assortie de commentaires, conseils ou jugements de valeur ; le lecteur pourra ainsi choisir en toute connaissance de cause la solution la plus adaptée à son objectif. Notamment, il sera en mesure de développer des programmes fiables et lisibles en évitant certaines situations à risque dont la connaissance reste malgré tout indispensable pour adapter d'anciens programmes. Il peut s'agir là de quelques rares cas où la norme reste elle-même ambiguë, ou encore d'usages syntaxiques conformes à la norme mais suffisamment « limites » pour être mal interprétés par certains compilateurs. Mais, plus souvent, il s'agira de possibilités désuètes, remontant à l'origine du langage, et dont la norme n'a pas osé se débarrasser, dans le souci de préserver l'existant. La plupart d'entre elles seront précisément absentes du langage C++ ; bon nombre de remarques (titrées *En C++*) visent d'ailleurs à préparer le lecteur à une éventuelle migration vers ce langage.

Une norme n'a d'intérêt que par la manière dont elle est appliquée. C'est pourquoi, au-delà de la norme elle-même, nous apportons un certain nombre d'informations pratiques. Ainsi, nous précisons le comportement qu'on peut attendre des différents compilateurs existants en cas de

non-respect de la syntaxe. Nous faisons de même pour les différentes situations d'exception qui risquent d'apparaître lors de l'exécution du programme. Il va de soi que ces connaissances se révéleront précieuses lors de la phase de mise au point d'un programme.

Malgré le caractère de référence de l'ouvrage, nous lui avons conservé une structure comparable à celle d'un cours. Il pourra ainsi être utilisé soit parallèlement à la phase d'apprentissage, soit ultérieurement, le lecteur retrouvant ses repères habituels. Accessoirement, il pourra servir de support à un cours de langage C avancé.

Toujours dans ce même souci pédagogique, nous avons doté l'ouvrage de nombreux exemples de programmes complets, accompagnés du résultat fourni par leur exécution. La plupart d'entre eux viennent illustrer une notion après qu'elle a été exposée. Mais quelques-uns jouent un rôle d'introduction pour les points que nous avons jugés les plus délicats.

Structure de l'ouvrage

La **première partie de l'ouvrage** est formée de 15 chapitres qui traitent des grandes composantes du langage suivant un déroulement classique.

Le chapitre 1 expose quelques notions de base spécifiques au C qui peuvent faire défaut au programmeur habitué à un autre langage : historique, préprocesseur, compilation séparée, différence entre variable et objet, classe d'allocation.

Le chapitre 2 présente les principaux constituants élémentaires d'un programme source : jeu de caractères, identificateurs, mots clés, séparateurs, espaces blancs, commentaires.

Le chapitre 3 est consacré aux types de base, c'est-à-dire ceux à partir desquels peuvent être construits tous les autres : entiers, flottants, caractères. Il définit également de façon précise l'importante notion d'expression constante.

Le chapitre 4 passe en revue les différents opérateurs, à l'exception de quelques opérateurs dits de référence (`[ ]`, `( )`, `->` et «.») qui trouvent tout naturellement leur place dans d'autres chapitres. Il étudie la manière dont sont conçues les expressions en C, ainsi que les différentes conversions qui peuvent y apparaître : implicites, explicites par `cast`, forcées par affectation.

Le chapitre 5 étudie l'ensemble des instructions exécutables du langage, après en avoir proposé une classification.

Le chapitre 6 traite de l'utilisation naturelle des tableaux à une ou à plusieurs dimensions. Leur manipulation, particulière au C, par le biais de pointeurs, n'est examinée que dans le chapitre suivant. De même, le cas des tableaux transmis en argument d'une fonction n'est traité qu'au chapitre 8.

Le chapitre 7 porte sur les pointeurs : déclarations, propriétés arithmétiques, lien entre tableau et pointeur, pointeur `NULL`, affectation, pointeurs génériques, comparaisons, conversions. Toutefois les pointeurs sur des fonctions ne sont abordés qu'au chapitre suivant.

Le chapitre 8 est consacré aux fonctions, tant sur le plan de leur définition que des différentes façons de les déclarer. Il examine notamment le cas des tableaux transmis en argument, en distinguant les tableaux à une dimension des tableaux à plusieurs dimensions. En outre, il fait

le point sur les variables globales et les variables locales, aussi bien en ce qui concerne leur déclaration que leur portée, leur classe d'allocation ou leur initialisation.

Le chapitre 9 étudie les entrées-sorties standard, que nous avons préféré séparer des fichiers, pour des questions de clarté, malgré le lien étroit qui existe entre les deux. Ce chapitre décrit en détail les fonctions `printf`, `scanf`, `puts`, `gets`, `putchar` et `getchar`.

Le chapitre 10 montre comment le C permet de manipuler des chaînes de caractères. Il passe en revue les différentes fonctions standard correspondantes : recopie, concaténation, comparaison, recherche. Il traite également des fonctions de conversion d'une chaîne en un nombre, ainsi que des fonctions de manipulation de suites d'octets.

Le chapitre 11 examine les types définis par l'utilisateur que sont les structures, les unions et les énumérations.

Le chapitre 12 est consacré à l'instruction `typedef` qui permet de définir des types synonymes.

Le chapitre 13 fait le point sur le traitement des fichiers : aspects spécifiques au langage C, traitement des erreurs, distinction entre opérations binaires et formatées… Puis il passe en revue les différentes fonctions standard correspondantes.

Le chapitre 14 montre comment mettre en œuvre ce que l'on nomme la gestion dynamique de la mémoire et en fournit quelques exemples d'application.

Le chapitre 15 étudie les différentes directives du préprocesseur.

La **seconde partie de l'ouvrage**, plus originale dans sa thématique, est composée de 11 chapitres qui traitent de sujets transversaux comme la récursivité ou les déclarations, des modalités d'application de certains éléments de syntaxe, ou encore de possibilités peu usitées du langage.

Le chapitre 16 propose un récapitulatif sur les déclarations aussi bien sur le plan de leur syntaxe, que sur la manière de les écrire ou de les interpréter. Sa présence se justifie surtout par la complexité et l'interdépendance des règles de déclaration en C : il aurait été impossible de traiter ce thème de manière exhaustive dans chacun des chapitres concernés.

Le chapitre 17 fait le point sur la manière de pallier les problèmes de manque de fiabilité que posent les lectures au clavier.

Le chapitre 18 montre comment le langage C distingue différentes catégories de caractères (caractères de contrôle, graphiques, alphanumériques, de ponctuation…) et examine les fonctions standard correspondantes.

Le chapitre 19 fournit quelques informations indispensables dans la gestion de gros programmes nécessitant un découpage en plusieurs fichiers source.

Le chapitre 20 explique comment, à l'image de fonctions standard telles que `printf`, écrire des fonctions à arguments variables en nombre et en type.

Le chapitre 21 examine les différentes façons dont un programme peut recevoir une information de l'environnement ou lui en transmettre, ainsi que des possibilités dites de *traitement de signaux*.

Le chapitre 22 montre comment, par le biais de ce que l'on nomme les *caractères étendus,* le langage C offre un cadre de gestion d'un jeu de caractères plus riche que celui offert par le codage sur un octet.

Le chapitre 23 traite du mécanisme général de *localisation*, qui offre à une implémentation la possibilité d'adapter le comportement de quelques fonctions standard à des particularités nationales ou locales.

Le chapitre 24 illustre les possibilités de récursivité du langage.

Le chapitre 25 traite d'un mécanisme dit de *branchements non locaux*, qui permet de s'affranchir de l'enchaînement classique : appel de fonction, retour.

Le chapitre 26 recense les incompatibilités qui ont subsisté entre le C ANSI et le C++, c'est-à-dire tout ce qui fait que le C++ n'est pas tout à fait un surensemble du C.

Une importante annexe fournit la syntaxe et le rôle de l'ensemble des fonctions de la bibliothèque standard. La plupart du temps, il s'agit d'un résumé d'informations figurant déjà dans le reste de l'ouvrage, à l'exception de quelques fonctions qui, compte tenu de leur usage extrêmement restreint, se trouvent présentées là pour la première fois.

Enfin, cette nouvelle édition tient compte des deux extensions de la norme publiées en 1999 et 2011 et connues sous les acronymes C99 et C11 :

- une importante annexe en présente la plupart des fonctionnalités ; sa situation tardive dans l'ouvrage se justifie par le fait que ces « nouveautés » ne sont pas intégralement appliquées par tous les compilateurs ;
- certains ajouts sont mentionnés au fil du texte, lorsque cela nous a paru utile.

À propos des normes ANSI/ISO

On parle souvent, par habitude, du C ANSI *(American National Standard Institute)* alors que la première norme américaine, publiée en 1989, a été rendue internationale en 1990 par l'ISO *(International Standardization Organisation)*, avant d'être reprise par les différents comités de normalisation continentaux ou nationaux sous la référence ISO/IEC 9899:1990. En fait, l'abus de langage se justifie par l'identité des deux documents, même si, en toute rigueur, le texte ISO est structuré différemment du texte ANSI d'origine.

Cette norme a continué d'évoluer. Certains « additifis » publiés séparemment ont été intégrés dans une nouvelle norme ISO/IEC 9899:1999 (nommée brièvement C99). Une nouvelle définition est apparue avec ISO/IEC 9899:2011 (C11). La première norme reste désignée par C ANSI ou par C90.

À propos de la fonction main

En théorie, selon la norme (C90, C99 ou C11), la fonction `main` qui, contrairement aux autres fonctions, ne dispose pas de prototype, devrait disposer de l'un des deux en-têtes suivants :

```
int main (void)
int main (int arg, char *argv)
```

En fait, tant que l'on ne cherche pas à utiliser les « arguments de la ligne de commande », les deux formes :

```
int main ()
main()
```

sont acceptées par toutes les implémentations, la seconde s'accompagnant toutefois fréquemment d'un message d'avertissement.

La première est la plus répandue et c'est celle qu'imposera la norme de C++. Nous l'utiliserons généralement.

Remerciements

Je tiens à remercier tout particulièrement Jean-Yves BROCHOT pour sa relecture extrêmement minutieuse de l'ouvrage, ainsi que pour les discussions nombreuses et enrichissantes qu'il a suscitées.

1

Généralités

A priori, cet ouvrage s'adresse à des personnes ayant déjà une expérience de la programmation, éventuellement dans un langage autre que le C. Bien qu'il puisse être étudié de manière séquentielle, il a été conçu pour permettre l'accès direct à n'importe quelle partie, à condition de disposer d'un certain nombre de notions générales, plutôt spécifiques au C, et qui sont examinées ici.

Nous commencerons par un bref historique du langage, qui permettra souvent d'éclairer certains points particuliers ou redondants. Puis nous verrons comment se présente la traduction d'un programme, à la fois par les possibilités de compilation séparée et par l'existence, originale, d'un préprocesseur. Nous expliquerons ensuite en quoi la classique notion de variable est insuffisante en C et pourquoi il est nécessaire de la compléter par celle, plus générale, d'objet. Enfin, nous verrons qu'il existe plusieurs façons de gérer l'emplacement mémoire alloué à une variable, ce qui se traduira par la notion de classe d'allocation.

1. Historique du langage C

Le langage C a été créé en 1972 par Denis Ritchie avec un objectif relativement limité : écrire un système d'exploitation (Unix). Mais ses qualités opérationnelles ont fait qu'il a très vite été adopté par une large communauté de programmeurs.

Une première définition rigoureuse du langage a été réalisée en 1978 par Kernighan et Ritchie avec la publication de l'ouvrage *The C Programming Language*. De nombreux compilateurs ont alors vu le jour en se fondant sur cette définition, quitte à l'assortir parfois de quelques extensions. Ce succès international du langage a conduit à sa normalisation, d'abord par l'ANSI *(American National Standard Institute)*, puis par l'ISO *(International Standardization Organisation)*, en 1993 par le CEN (Comité européen de normalisation) et enfin, en 1994, par

l'AFNOR. En fait, et fort heureusement, toutes ces normes sont identiques, et l'usage veut qu'on parle de C90 (autrefois de « C ANSI » ou de « C norme ANSI »).

La norme ANSI élargit, sans la contredire, la première définition de Kernighan et Ritchie. Pour la comprendre et pour l'accepter, il faut savoir qu'elle a cherché à concilier deux intérêts divergents :

- d'une part, améliorer et sécuriser le langage ;
- d'autre part, préserver l'existant, c'est-à-dire faire en sorte que les programmes créés avant la norme soient acceptés par la norme.

Dans ces conditions, certaines formes désuètes ou redondantes ont dû être conservées. L'exemple le plus typique réside dans les déclarations de fonctions : la première définition prévoyait de déclarer une fonction en ne fournissant que le type de son résultat ; la norme ANSI a prévu d'y ajouter le type des arguments, mais sans interdire l'usage de l'ancienne forme. Il en résulte que la maîtrise des différentes situations possibles nécessite des connaissances qui seraient devenues inutiles si la norme avait osé interdire l'ancienne possibilité. On notera, à ce propos, que la norme du C++, langage basé très fortement sur le C, supprime bon nombre de redondances pour lesquelles la norme du C n'a pas osé trancher ; c'est notamment le cas des déclarations de fonctions qui doivent obligatoirement utiliser la seconde forme.

Après cette première normalisation, des extensions ont été apportées, tout d'abord sous forme de simples additifs en 1994 (ISO/IEC 9899/COR1:1994) et en 1995 (ISO/IEC 9899/COR2 :1995), lesquels se sont trouvés intégrés dans la nouvelle norme ISO/IEC 9899:1999, désignée sous l'acronyme C99. Enfin, une dernière norme ISO/IEC 9899:2011 est apparue, plus connue sous l'acronyme C11.

Compte tenu du fait que tous les compilateurs ne respectent pas intégralement les dernières normes C99 et C11 (cette dernière comportant d'ailleurs des fonctionnalités « facultatives »), notre discours se fonde plutôt sur la norme C90. Mais, dans la suite de l'ouvrage, il nous arrivera souvent de préciser :

- ce qui constitue un apport de la norme C90 par rapport à la première définition du C ;
- ce qui dans la première définition est devenu désuet ou déconseillé, bien qu'accepté par la norme ; en particulier, nous ferons souvent référence à ce qui disparaîtra ou qui changera en C++ ;
- les points concernés par les apports des normes C99 et C11.

L'annexe en fin d'ouvrage récapitule les principaux apports de C99 et C11.

2. Programme source, module objet et programme exécutable

Tout programme écrit en langage évolué forme un texte qu'on nomme un « programme source ». En langage C, ce programme source peut être découpé en un ou plusieurs fichiers source. Notez qu'on parle de fichier même si, exceptionnellement, le texte correspondant, saisi en mémoire, n'a pas été véritablement recopié dans un fichier permanent.

Chaque fichier source est traduit en langage machine, indépendamment des autres, par une opération dite de « compilation », réalisée par un logiciel ou une partie de logiciel nommée « compilateur ». Le résultat de cette opération porte le nom de « module objet ». Bien que formé d'instructions machine, un tel module objet n'est pas exécutable tel quel car :

- il peut lui manquer d'autres modules objet ;
- il lui manque, de toute façon, les instructions exécutables des fonctions standards appelées dans le fichier source (par exemple `printf`, `scanf`, `strcat`...).

Le rôle de l'éditeur de liens est précisément de réunir les différents modules objet et les fonctions de la bibliothèque standard afin de constituer un programme exécutable. Ce n'est d'ailleurs que lors de cette édition de liens qu'on pourra s'apercevoir de l'absence d'une fonction utilisée par le programme.

3. Compilation en C : existence d'un préprocesseur

En C, la traduction d'un fichier source se déroule en deux étapes totalement indépendantes :

- un prétraitement ;
- une compilation proprement dite.

La plupart du temps, ces deux étapes sont enchaînées automatiquement, de sorte qu'on a l'impression d'avoir affaire à un seul traitement. Généralement, on parle du préprocesseur pour désigner le programme réalisant le prétraitement. En revanche, les termes de « compilateur » ou de « compilation » restent ambigus puisqu'ils désignent tantôt l'ensemble des deux étapes, tantôt la seconde.

L'étape de prétraitement correspond à une modification du texte d'un fichier source, basée essentiellement sur l'interprétation d'instructions très particulières dites « directives à destination du préprocesseur » ; ces dernières sont reconnaissables par le fait qu'elles commencent par le signe #.

Les deux directives les plus importantes sont :

- la directive d'inclusion d'autres fichiers source : `#include` ;
- la directive de définition de macros ou de symboles : `#define`.

La première est surtout utilisée pour incorporer le contenu de fichiers prédéfinis, dits « fichiers en-tête », indispensables à la bonne utilisation des fonctions de la bibliothèque standard, la plus connue étant :

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

La seconde est très utilisée dans les fichiers en-tête prédéfinis. Elle est également souvent exploitée par le programmeur dans des définitions de symboles telles que :

```
#define NB_COUPS_MAX 100
#define TAILLE 25
```

4. Variable et objet

4.1 Définition d'une variable et d'un objet

Dans beaucoup de langages, les informations sont manipulées par le biais de variables, c'est-à-dire d'emplacements mémoire portant un nom et dont le contenu est susceptible d'évoluer. En C, il existe bien entendu des variables répondant à une telle définition mais on peut également manipuler des informations qui ne sont plus vraiment contenues dans des variables ; le cas le plus typique est celui d'une information manipulée par l'intermédiaire d'un pointeur :

```
int *adi ;      /* adi est une variable destinée à contenir une adresse d'entier */
  ...
*adi = 5 ;      /* place la valeur 5 dans l'entier pointé par adi */
```

L'entier pointé par `adi` ne porte pas vraiment de nom ; d'ailleurs, au fil de l'exécution, `adi` peut pointer sur des entiers différents.

Pour tenir compte de cette particularité, il est donc nécessaire de définir un nouveau mot. On utilise généralement celui d'objet[1]. On dira donc qu'un objet est un emplacement mémoire parfaitement défini qu'on utilise pour représenter une information à laquelle on peut accéder à volonté (autant de fois qu'on le souhaite) au sein du programme.

Bien entendu, une variable constitue un cas particulier d'objet. Mais, dans notre précédent exemple, l'emplacement pointé à un instant donné par `adi` est lui-même un objet. En revanche, une expression telle `n+5` n'est pas un objet dans la mesure où l'emplacement mémoire correspondant n'est pas parfaitement défini et où, de plus, il a un caractère relativement fugitif ; on y accédera véritablement qu'une seule fois : au moment de l'utilisation de l'expression en question.

La question de savoir si des constantes telles que `34`, `'d'` ou `"bonjour"` sont ou non des objets est relativement ambiguë : une constante utilise un emplacement mémoire mais peut-on dire qu'on y accède à volonté ? En effet, on n'est pas sûr qu'une même constante se réfère toujours au même emplacement, ce point pouvant dépendre de l'implémentation. La norme n'est d'ailleurs pas totalement explicite sur ce sujet qui n'a guère d'importance en pratique. Dans la suite, nous conviendrons qu'une constante n'est pas un objet.

4.2 Utilisation d'un objet

4.2.1 Accès par une expression désignant l'objet

Lorsqu'un objet est une variable, son utilisation ne pose guère de problème, qu'il s'agisse d'en utiliser ou d'en modifier la valeur, même si, au bout du compte, le nom même de la variable possède une signification dépendant du contexte dans lequel il est employé. Par exemple, si `p` est une variable, dans l'expression :

```
p + 5
```

1. Attention, ce terme n'a ici aucun lien avec la programmation orientée objet.

`p` désigne tout simplement la valeur de la variable `p`, tandis que dans :

```
p = 5 ;
```

`p` désigne la variable elle-même.

Dans le cas des objets pointés, en revanche, on ne pourra pas recourir à un simple identificateur ; il faudra faire appel à des expressions plus complexes telles que `*adi` ou `*(adi+3)`. Ici encore, cette expression pourra intervenir soit pour utiliser la valeur de l'objet, soit pour en modifier la valeur. Par exemple, dans l'expression :

```
*adi + 5
```

`*adi` désigne la valeur de l'objet pointé par `adi`, tandis que dans :

```
*adi = 12 ;
```

`*adi` désigne l'objet lui-même.

4.2.2 Type d'un objet

Comme dans la plupart des langages, le type d'un objet n'est pas défini de façon intrinsèque : en examinant une suite d'octets de la mémoire, on est incapable de savoir comment l'information qui s'y trouve a été codée, et donc de donner une valeur à l'objet correspondant. En fait, le type d'un objet n'est défini que par la nature de l'expression qu'on utilise, à un instant donné, pour y accéder ou pour le modifier. Certes, dans un langage où tout objet est contenu dans une variable, cette distinction est peu importante puisque le type est alors défini par le nom de la variable, lequel constitue le seul et unique moyen d'accéder à l'objet[2]. Dans un langage comme le C, en revanche, on peut accéder à un objet par le biais d'un pointeur. Son type se déduira, là encore, de la nature du pointeur mais avec cette différence fondamentale par rapport à la variable qu'il est alors possible, volontairement ou par erreur, d'accéder à un même objet avec des pointeurs de types différents.

Certes, lorsque l'on est amené à utiliser plusieurs expressions pour accéder à un même objet, elles sont généralement de même type, de sorte qu'on a tendance à considérer que le type de l'objet fait partie de l'objet lui-même. Par souci de simplicité, d'ailleurs, il nous arrivera souvent de parler du « type de l'objet ». Il ne s'agira cependant que d'un abus de langage qui se réfère au type ayant servi à créer l'objet, en faisant l'hypothèse que c'est celui qu'on utilisera toujours pour accéder à l'objet.

> **Remarque**
>
> Parmi les différentes expressions permettant d'accéder à un objet, certaines ne permettent pas sa modification. C'est par exemple le cas d'un nom d'une variable ayant reçu l'attribut `const` ou d'un nom de tableau. Comme on le verra à la section 7.2 du chapitre 4, on parle généralement de `lvalue` pour désigner les expressions utilisables pour modifier un objet.

2. Bien que certains langages autorisent, exceptionnellement, d'accéder à un même emplacement par le biais de deux variables différentes, de types éventuellement différents.

5. Lien entre objet, octets et caractères

En langage C, l'octet correspond à la plus petite partie adressable de la mémoire, mais il n'est pas nécessaire, comme pourrait le faire croire la traduction française du terme anglais *byte*, qu'il soit effectivement constitué de 8 bits, même si cela est très souvent le cas.

Tout objet est formé d'un nombre entier d'octets et par conséquent, il possède une adresse, celle de son premier octet. La réciproque n'est théoriquement pas vraie : toute adresse (d'octet) n'est pas nécessairement l'adresse d'un objet ; voici deux contre-exemples :

- l'octet en question n'est pas le premier d'un objet ;
- l'octet en question est le premier octet d'un objet mais l'expression utilisée pour y accéder correspond à un type occupant un nombre d'octets différent.

Malgré tout, il sera souvent possible de considérer une adresse quelconque comme celle d'un objet de type quelconque. Bien entendu, cela ne préjugera nullement des conséquences plus ou moins catastrophiques qui pourront en découler.

Par ailleurs, la notion de caractère en C coïncide totalement avec celle d'octet. Dans ces conditions, on pourra toujours considérer n'importe quelle adresse comme celle d'un objet de type caractère. C'est d'ailleurs ainsi que l'on procédera lorsqu'on voudra traiter individuellement chacun des octets constituant un objet.

6. Classe d'allocation des variables

Comme on le verra, notamment au chapitre 8, l'emplacement mémoire attribué à une variable peut être géré de deux façons différentes, suivant la manière dont elle a été déclarée. On parle de « classe d'allocation statique » ou de « classe d'allocation automatique ».

Les variables de classe statique voient leur emplacement alloué une fois pour toutes avant le début de l'exécution du programme ; il existe jusqu'à la fin du programme. Une variable statique est rémanente, ce qui signifie qu'elle conserve sa dernière valeur jusqu'à une nouvelle éventuelle modification.

Les variables de classe automatique voient leur emplacement alloué au moment de l'entrée dans un bloc ou dans une fonction ; il est supprimé lors de la sortie de ce bloc ou de cette fonction. Une variable automatique n'est donc pas rémanente puisqu'on n'est pas sûr d'y trouver, lors d'une nouvelle entrée dans le bloc, la valeur qu'elle possédait à la sortie précédente de ce bloc.

On verra qu'une variable déclarée à un niveau global est toujours de classe d'allocation statique, tandis qu'une variable déclarée à un niveau local (à un bloc ou à une fonction) est, par défaut, seulement de classe d'allocation automatique. On pourra agir en partie sur la classe d'allocation d'une variable en utilisant, lors de sa déclaration, un mot-clé dit « classe de mémorisation ». Par exemple, une variable locale déclarée avec l'attribut `static` sera de classe d'allocation statique. On veillera cependant à distinguer la classe d'allocation de la classe de mémorisation, malgré le lien étroit qui existe entre les deux ; d'une part la classe de mémorisation est facultative et d'autre part, quand elle est présente, elle ne correspond pas toujours à la classe d'allocation.

Par ailleurs, par le biais de pointeurs, le langage C permet d'allouer dynamiquement des emplacements pour des objets. Il est clair qu'un tel emplacement est géré d'une manière différente de celles évoquées précédemment. On parle souvent de gestion dynamique (ou programmée). La responsabilité de l'allocation et de la libération incombant, cette fois, au programmeur. Comme on peut le constater, les notions de classe statique ou automatique ne concernent donc que les objets contenus dans des variables.

Remarque

En toute rigueur, il existe une troisième classe d'allocation, à savoir la classe registre. Il ne s'agit cependant que d'un cas particulier de la classe d'allocation automatique.

3

Les types de base

La manipulation d'une information fait généralement intervenir la notion de type, c'est-à-dire la manière dont elle a été codée en mémoire. La connaissance de ce type est nécessaire pour la plupart des opérations qu'on souhaite lui faire subir. Traditionnellement, on distingue les types simples dans lesquels une information est, à un instant donné, caractérisée par une seule valeur, et les types agrégés dans lesquels une information est caractérisée par un ensemble de valeurs.

Ce chapitre étudie tout d'abord les caractéristiques des différents types de base du langage, c'est-à-dire ceux à partir desquels peuvent être construits tous les autres, qu'il s'agisse de types simples comme les pointeurs ou de types agrégés comme les tableaux, les structures ou les unions. Nous les avons classés en trois catégories : entiers, caractères et flottants, même si, comme on le verra, les caractères apparaissent, dans une certaine mesure, comme des cas particuliers d'entiers. Nous présenterons ensuite l'importante notion d'expression constante, c'est-à-dire calculable par le compilateur. Enfin, nous terminerons sur la déclaration et l'initialisation des variables d'un type de base.

1. Les types entiers

Pour les différents types entiers prévus par la norme, nous étudions ici les différentes façons de les nommer et la manière dont les informations correspondantes sont codées en mémoire. Nous fournissons quelques éléments permettant de choisir le type entier le plus approprié à un objectif donné. Enfin, nous indiquons les différentes façons d'écrire des constantes entières dans un programme source.

1.1 Les six types entiers

En théorie, la norme ANSI prévoit qu'il puisse exister six types entiers différents caractérisés par deux paramètres :

- la taille de l'emplacement mémoire utilisé pour les représenter ;
- un attribut précisant si l'on représente des nombres signés, c'est-à-dire des entiers relatifs, ou des nombres non signés, c'est-à-dire des entiers naturels.

Le premier paramètre est assez classique et, comme dans la plupart des langages, il se traduit par l'existence de différents noms de type : à chaque nom correspond une taille qui pourra cependant dépendre de l'implémentation. Le second paramètre, quant à lui, est beaucoup moins classique et on verra qu'il est conseillé de ne recourir aux entiers non signés que dans des circonstances particulières telles que la manipulation de motifs binaires.

Pour chacun des six types, il existe plusieurs façons de les nommer, compte tenu de ce que :

- le paramètre de taille s'exprime par un attribut facultatif : `short` ou `long` ;
- le paramètre de signe s'exprime, lui aussi, par un attribut facultatif : `signed` ou `unsigned` ;
- le mot-clé `int`, correspondant à un entier de taille intermédiaire, peut être omis, dès lors qu'au moins un des qualificatifs précédents est présent.

Le tableau 3.1 récapitule les différentes manières de spécifier chacun des six types (on parle de « spécificateur de type »), ainsi que la taille minimale et le domaine minimal que leur impose la norme, quelle que soit l'implémentation concernée.

Tableau 3.1 : les six types entiers prévus par la norme

Dénomination usuelle	Spécificateurs de type possibles	Taille minimale (en octets)	Domaine minimal

Entier court signé	`short`
	`short int`
	`signed short`
	`signed short int`
		2	-32 767 à +32 767

Entier court non signé	`unsigned short` `unsigned short int`	2	0 à 65 535

Entier signé	`int`
	`signed int`
	`signed`
		2	-32 767 à +32 767

Entier non signé	`unsigned int` `unsigned`	2	0 à 65 535

Entier long signé	`long`
	`long int`
	`signed long`
	`signed long int`
		4	-2 147 483 647 à +2 147 483 647

Entier long non signé	`unsigned long` `unsigned long int`	4	0 à 4 294 967 295

Remarques

1. La norme impose qu'un type signé et un type non signé de même nom possèdent la même taille ; ce sera par exemple le cas pour `long int` et `unsigned long int`.
2. La norme impose seulement à une implémentation de disposer de ces six types ; en particulier, rien n'interdit que deux types différents possèdent la même taille. Fréquemment, d'ailleurs, soit les entiers courts et les entiers seront de même taille, soit ce seront les entiers et les entiers longs. Les seules choses dont on soit sûr sont que, dans une implémentation donnée, la taille des entiers courts est inférieure ou égale à celle des entiers et que celle des entiers est inférieure ou égale à celle des entiers longs.
3. C99 introduit les types supplémentaires : `long long int` et `unsigned long long int`.

1.2 Représentation mémoire des entiers et limitations

Comme le montre le tableau 3.2, la norme prévoit totalement la manière dont une implémentation doit représenter les entiers non signés ainsi que les valeurs positives des entiers signés. En revanche, elle laisse une toute petite latitude en ce qui concerne la représentation des valeurs négatives.

Tableau 3.2 : contraintes imposées à la représentation des entiers

Nature de l'entier	Contrainte imposée par la norme	Remarque
Non signé	Codage binaire pur	Un entier non signé de taille donnée est donc totalement portable.
Signé	Un entier positif doit avoir la même représentation que l'entier non signé de même valeur.	La plupart des implémentations utilisent, pour les nombres négatifs, la représentation dite du « complément à deux ».

Voyons cela plus en détail en raisonnant, pour fixer les idées, sur des nombres entiers représentés sur 16 bits. Il va de soi qu'il serait facile de généraliser notre propos à une taille quelconque.

1.2.1 Représentation des nombres non signés

La norme leur impose une notation binaire pure, de sorte que le codage de ces nombres est totalement défini : il s'agit simplement de la représentation du nombre en base 2. Voici des exemples de représentation de quelques nombres sur 16 bits : la dernière colonne reprend, sous forme hexadécimale classique, le codage binaire exprimé dans la colonne précédente :

Valeur décimale	Codage en binaire	Codage, exprimé en hexadécimal
1	0000000000000001	0001
2	0000000000000010	0002
3	0000000000000011	0003
16	0000000000010000	0010
127	0000000001111111	007F
255	0000000011111111	00FF
1 025	0000010000000001	0401
32 767	0111111111111111	7FFF
32 768	1000000000000000	8000
32 769	1000000000000001	8001
64 512	1111110000000000	FC00
65 534	1111111111111110	FFFE
65 535	1111111111111111	FFFF

1.2.2 Représentation des nombres entiers signés

Lorsqu'il s'agit d'un nombre positif, la norme impose que sa représentation soit identique à celle du même nombre sous forme non signée, ce qui revient à dire qu'on y trouve le nombre exprimé en base 2. Dans ces conditions, comme l'implémentation doit pouvoir distinguer entre nombres positifs et nombres négatifs, la seule possibilité qui lui reste consiste à se baser sur le premier bit, en considérant que 0 correspond à un nombre positif, tandis que 1 correspond à un nombre négatif[1].

Une latitude subsiste néanmoins dans la manière de coder la valeur absolue du nombre. Actuellement, la représentation dite « en complément à deux » tend à devenir universelle. Elle procède ainsi :

- on exprime la valeur en question en base 2 ;
- tous les bits sont « inversés » : 1 devient 0 et 0 devient 1 ;
- enfin, on ajoute une unité au résultat.

Voici des exemples de représentation de quelques nombres négatifs sur 16 bits, lorsqu'on utilise cette technique : la dernière colonne reprend, sous forme hexadécimale classique, le codage binaire exprimé dans la colonne précédente.

Valeur décimale	Codage en binaire	Codage, exprimé en hexadécimal
-1	1111111111111111	FFFF
-2	1111111111111110	FFFE
-3	1111111111111101	FFFD
-4	1111111111111100	FFFC
-16	1111111111110000	FFF0
-256	1111111100000000	FF00
-1 024	1111110000000000	FC00
-32 768	1000000000000000	8000

1. Même si la norme n'impose pas formellement l'existence d'un bit de signe.

Remarques

1. Dans la représentation en complément à deux, le nombre 0 est codé d'une seule manière, à savoir 0000000000000000.
2. Si l'on ajoute 1 au plus grand nombre positif (ici 0111111111111111, soit 7FFF en hexadécimal ou 32 767 en décimal) et que l'on ne tient pas compte du dépassement de capacité qui se produit, on obtient... le plus petit nombre négatif possible (ici 1000000000000000, soit 8 000 en hexadécimal ou -32 768 en décimal). C'est ce qui explique le phénomène de « modulo » bien connu de l'arithmétique entière dans le cas (fréquent) où les dépassements de capacité ne sont pas détectés.
3. Par sa nature, la représentation en complément à deux conduit à une différence d'une unité entre la valeur absolue du plus petit négatif et celle du plus grand positif. Dans les autres représentations (rares), on a souvent deux façons de coder le nombre 0 : avec bit de signe à 0 ou avec bit de signe à 1. Dans ce cas, on dispose d'autant de combinaisons possibles pour les positifs que pour les négatifs, la valeur absolue de la plus petite valeur négative étant alors égale à celle de la plus grande valeur positive qui se trouve être la même que dans la représentation en complément à deux. Autrement dit, la manière exacte dont on représente les nombres négatifs a une incidence extrêmement faible sur le domaine couvert par un nombre de bits donnés puisqu'elle ne joue que sur une unité.

1.2.3 Limitations

Le nombre de bits utilisés pour représenter un entier et le codage employé dépendent, bien sûr, de l'implémentation considérée. Il en va donc de même des limitations qui en découlent. Le tableau 3.3 indique ce que sont ces limitations relatives aux entiers dans le cas où ils sont codés sur 16 ou 32 bits, en utilisant la représentation en complément à deux.

Tableau 3.3 : limitations relatives aux entiers codés en complément à deux

Attribut	Taille (bits)	Mini		Maxi
Non signé	16	0	à	65 535
Signé	16	-32 768	à	32 767
Non signé	32	0	à	4 294 967 295
Signé	32	-2 147 483 648	à	2 147 483 647

Dans les implémentations respectant la norme ANSI, sans utiliser la représentation en complément à deux, la seule différence pouvant apparaître dans ces limitations concerne uniquement la valeur absolue des valeurs négatives. Ces dernières, comme expliqué dans la troisième remarque, peuvent se trouver diminuées de 1 unité. D'ailleurs, la norme tient compte de cette remarque pour imposer le domaine minimal des différents types entiers, comme nous l'avons présenté dans le tableau 3.1.

D'une manière générale, les limites spécifiques à une implémentation donnée peuvent être connues en utilisant le fichier en-tête `limits.h` qui sera décrit à la section 3 de ce chapitre. Ce fichier contient également d'autres informations concernant les caractéristiques des entiers et des caractères.

1.3 Critères de choix d'un type entier

Compte tenu du grand nombre de types entiers différents dont on dispose, voici quelques indications permettant d'effectuer son choix.

Utiliser les entiers non signés uniquement lorsque cela est indispensable

À taille égale, un type entier non signé permet de représenter des nombres deux fois plus grands (environ) qu'un type signé. Dans ces conditions, certains programmeurs sont tentés de recourir aux types non signés pour profiter de ce gain. En fait, il faut être prudent pour au moins deux raisons :

- dès qu'on est amené à effectuer des calculs, il est généralement difficile d'affirmer qu'on ne sera pas conduit, à un moment ou à un autre, à un résultat négatif non représentable dans un type non signé ;
- même s'il est permis, le mélange de types signés et non signés dans une même expression est fortement déconseillé, compte tenu du peu de signification que possèdent les conversions qui se trouvent alors mises en place (elles sont décrites à la section 3 du chapitre 4).

En définitive, les entiers non signés ne devraient pratiquement jamais être utilisés pour réaliser des calculs. On peut considérer que leur principale vocation est la manipulation de motifs binaires, indépendamment des valeurs numériques correspondantes. On les utilisera souvent comme opérandes des opérateurs de manipulation de bits ou comme membres d'unions ou de champs de bits. Dans certains cas, ils pourront également être utilisés pour échanger des informations numériques entre différentes machines, compte tenu du caractère entièrement portable de leur représentation. Toutefois, il sera alors généralement nécessaire de transformer artificiellement des valeurs négatives en valeurs positives (par exemple, en introduisant un décalage donné).

> **Remarque**
>
> Le mélange entre flottants et entiers non signés présente les mêmes risques que le mélange entre entiers non signés et entiers signés. En revanche, le mélange entre entiers signés et flottants ne pose pas de problèmes particuliers. Cela montre que l'arithmétique non signée constitue un cas bien à part.

Efficacité

En général, le type `int` correspond au type standard de la machine, de sorte que l'on est quasiment assuré que c'est dans ce type que les opérations seront les plus rapides. On pourra l'utiliser pour réaliser des programmes portables efficaces, pour peu qu'on accepte les limitations correspondantes.

Signalons que l'on rencontre actuellement des machines à 64 bits, dans lesquelles la taille du type `int` reste limitée à 32 bits, probablement par souci de compatibilité avec des machines antérieures. Le type `int` reste cependant le plus efficace car, généralement, des mécanismes spécifiques à la machine évitent alors la dégradation des performances.

Occupation mémoire

Le type `short` est naturellement celui qui occupera le moins de place, sauf si l'on peut se contenter du type `char`, qui peut jouer le rôle d'un petit entier (voir section 2 de ce chapitre). Toutefois, l'existence de contraintes d'alignement et le fait que ce type peut être plus petit que la taille des entiers manipulés naturellement par la machine, peuvent conduire à un résultat opposé à celui escompté. Par exemple, sur une machine où le type `short` occupe deux octets et où le type `int` occupe 4 octets, on peut très bien aboutir à la situation suivante :

- les informations de 2 octets sont alignées sur des adresses multiples de 4, ce qui peut annuler le gain de place escompté ;
- l'accès à 2 octets peut impliquer l'accès à 4 octets avec sélection des 2 octets utiles d'où une perte de temps.

Dans tous les cas, dans l'évaluation d'une expression, toute valeur de type `short` est convertie systématiquement en `int` (voir éventuellement le chapitre 4), ce qui peut entraîner une perte de temps.

En pratique, le type `short` pourra être utilisé pour des tableaux car la norme impose la contiguïté de leurs éléments : on sera donc assuré d'un gain de place au détriment éventuel d'une perte de temps.

> **Remarque**
>
> Tout ce qui vient d'être dit à propos du type `short` se transposera au petit type entier qu'est le type `char`.

Portabilité des programmes

Ici, le problème est délicat dans la mesure où le terme même de portabilité est quelque peu ambigu. En effet, s'il s'agit d'écrire un programme qui compile correctement dans toute implémentation, on peut effectivement utiliser n'importe quel type. En revanche, s'il s'agit d'écrire un programme qui fonctionne de la même manière dans toute implémentation, il n'en va plus de même étant donné qu'un spécificateur de type donné correspond à un domaine différent d'une implémentation à une autre. Par exemple, `int` pourra correspondre à 2 octets sur certaines machines, à 4 octets sur d'autres... Dans certains cas, on souhaitera disposer d'un type entier ayant une taille bien déterminée ; on pourra y parvenir en utilisant des possibilités de compilation conditionnelle (voir section 3.4.2 du chapitre 15).

1.4 Écriture des constantes entières

Lorsque vous devez introduire une constante entière dans un programme, le langage C vous laisse le choix entre trois formes d'écriture présentées dans le tableau 3.4 :

Tableau 3.4 : les trois formes d'écriture des constantes entières

Forme d'écriture	Définition	Exemples	Remarques
Décimale	Correspond à notre notation usuelle d'un nombre entier, avec ou sans signe.	+533 48 -273	Conseillée dans les cas usuels (arithmétique classique)
Octale	Nombre écrit en base 8, précédé du chiffre 0.	014 équivaut à 12 037 équivaut à 31	Peu portable, conseillée pour imposer un motif binaire (voir section suivante)
Hexadécimale	Nombre écrit en hexadécimal (base 16, les chiffres supérieurs à 9 sont représentés par les lettres A à F majuscules ou minuscules), précédées des deux caractères 0x (ou 0X).	0x1A (16+10) équivaut à 26 0x3F (3*16+15) équivaut à 63	Peu portable, conseillée pour imposer un motif binaire (voir section suivante)

1.5 Le type attribué par le compilateur aux constantes entières

Tant que l'on se limite à l'écriture de constantes sous forme décimale, au sein d'expressions ne faisant pas intervenir d'entiers non signés, on peut tranquillement ignorer la nature exacte du type que leur attribue le compilateur. Mais cette connaissance s'avère indispensable dans les situations suivantes :

- utilisation de constantes écrites en notation décimale dans une expression où apparaissent des quantités non signées ;
- utilisation de constantes écrites en notation hexadécimale ou octale.

Nous allons ici examiner les règles employées par le compilateur pour définir ce type.

1.5.1 Cas usuel de la notation décimale

Une constante entière écrite sous forme décimale est du premier des types suivants dont la taille suffit à la représenter correctement : `int`, `long int`, `unsigned long int`.

Ainsi, dans toute implémentation, la constante 12 sera de type `int` ; la constante 3 000 000 sera représentée en `long` si le type `int` est de capacité insuffisante et dans le type `int` dans le cas contraire. Il faut cependant noter que les valeurs positives non représentables dans le type `long` seront représentées dans le type `unsigned long` si ce dernier a une capacité suffisante. Cela va à l'encontre du conseil prodigué à la section 1.3 de ce chapitre, puisqu'on risque de se retrouver sans le vouloir en présence d'une expression mixte. Toutefois, cette anomalie ne se produit qu'avec des constantes qui, de toute façon, ne sont pas représentables dans l'implémentation.

1.5.2 Cas de la notation octale ou hexadécimale

Une constante entière écrite sous forme octale ou hexadécimale est du premier des types suivants dont la taille suffit à la représenter correctement : `int`, `unsigned int`, `long int`, `unsigned long int`.

Alors qu'une constante décimale n'est jamais non signée (excepté lorsqu'elle n'est pas représentable en `long`), un constante octale ou hexadécimale peut l'être, alors même qu'elle aurait été représentable en `long`. En outre, elle peut se voir représenter avec un attribut de signe différent suivant l'implémentation. Par exemple :

- OxFF sera toujours considérée comme un `int` (donc signée) car, dans toutes les implémentations, la capacité du type `int` est supérieure à 255.
- OxFFFF sera considérée comme un `unsigned int` dans les implémentations où le type `int` utilise 16 bits et comme un `int` dans celles où le type `int` utilise une taille supérieure.

Cette remarque trouvera sa pleine justification avec les règles utilisées dans l'évaluation d'expressions mixtes (mélangeant des entiers signés et non signés) puisque, comme l'explique le chapitre 4, ces dernières ont alors tendance à privilégier la conservation du motif binaire plutôt que la valeur. Quoi qu'il en soit, on ne perdra pas de vue que l'utilisation de constantes hexadécimales ou octales nécessite souvent la connaissance de la taille exacte du type concerné dans l'implémentation employée et qu'elle est donc, par essence, peu portable.

1.6 Exemple d'utilisation déraisonnable de constantes hexadécimales

La section 6 du chapitre 4, vous présentera des exemples d'utilisation raisonnable de constantes hexadécimales, notamment pour réaliser des « masques binaires ». Ici, nous nous contentons d'un programme qui illustre les risques que présente un usage non justifié de telles constantes. Il a été exécuté dans une implémentation où le type `int` est représenté sur 16 bits, les nombres négatifs utilisant la représentation en complément à deux :

Utilisation déraisonnable de constantes hexadécimales

```
int main()
{  int n ;
   n = 10 + 0xFF ;                                  premiere valeur : 265
   printf ("premiere valeur : %d\n", n) ;           seconde valeur  : 9
   n = 10 + 0xFFFF ;
   printf ("seconde valeur  : %d\n", n) ;
}
```

À première vue, tout se passe comme si `0xFF` était interprétée comme valant 255, tandis que `0xFFFF` serait interprétée comme valant -1, ce qui correspond effectivement à la représentation sur 16 bits de la constante -1 dans le type `int`. Or, comme indiqué précédement à la section 1.5.2, la norme prévoit que `0xFFFF` soit de type `unsigned int`, ce qui correspond à la valeur 65 565. En fait, comme on le verra au chapitre suivant, le calcul de l'expression `10 + 0xFFFF` se fait en convertissant 10 en `unsigned int`. Dans ces conditions, le résultat (65 545) dépasse la capacité de ce type ; mais la norme prévoit exactement le résultat (par une formule de modulo), à savoir 65 545 modulo 65 536, c'est-à-dire 9. La conversion en `int` qu'entraîne son affectation à *n* ne pose ensuite aucun problème.

Ainsi, dans cette implémentation, tout se passe effectivement comme si les constantes hexadécimales étaient signées : avec un type `int` représenté sur 16 bits, `0xFF` apparaît comme positive,

tandis que `xFFFF` apparaît comme négative. En revanche, avec un type `int` représenté sur 32 bits, `0xFFFF` apparaîtrait comme positive, alors que `0xFFFFFFFF` apparaîtrait comme négative.

1.7 Pour imposer un type aux constantes entières

Il est toujours possible, quelle que soit l'écriture employée (décimale, octale, hexadécimale), de forcer le compilateur :

- à utiliser un type `long` en ajoutant la lettre L (ou l) à la suite de l'écriture de la constante. Par exemple :

```
1L    045L    0x7F8L    25l    0xFFl
```

  Bien entendu, la constante correspondante sera de type `signed long` ou `unsigned long` suivant les règles habituelles présentées précédemment ;

- à utiliser un attribut `unsigned` en ajoutant la lettre U (ou u) à la suite de l'écriture de la constante. Par exemple :

```
1U    035U    0xFFFFu
```

  Là encore, la constante correspondante sera de type `unsigned int` ou `unsigned long` suivant les règles présentées précédemment ;

- à combiner les deux possibilités précédentes. Par exemple :

```
1LU    045LU    0xFFFFlu
```

  Cette fois, la constante correspondante est obligatoirement du type `unsigned long`.

> **Remarque**
>
> À simple titre indicatif, sachez que le programme précédent (utilisation de constantes hexadécimales), exécuté dans la même implémentation, fournit toujours les mêmes résultats, quelle que soit la façon d'écrire la constante utilisée dans la sixième ligne, à savoir : `0xFFFF`, `0xFFFFu`, `0xFFFFl` ou `0xFFFFlu`.

1.8 En cas de dépassement de capacité dans l'écriture des constantes entières

Comme le compilateur choisit d'office le type approprié, les seuls cas qui posent vraiment problème sont ceux où vous écrivez une constante positive supérieure à la capacité du type `unsigned long int` ou une constante négative inférieure au plus petit nombre représentable dans le type `long int`.

On notera que, dans ce cas, la norme du langage C, comme celle des autres langages, ne prévoit nullement comment doit réagir le compilateur[2]. Certains compilateurs fournissent alors un diagnostic de compilation, ce qui est fort satisfaisant. D'autres se contentent de fabriquer une constante fantaisiste (généralement par perte des bits les plus significatifs !). Ainsi, sur une

2. La norme ne dit jamais comment doit réagir le compilateur ou le programme en cas de situation d'exception, c'est-à-dire de non-respect de la norme.

machine où le type `longint` occupe 32 bits, une constante telle que 4 294 967 297 pourra être acceptée à la compilation et interprétée comme valant 2 !

Rappelons qu'il est déconseillé d'utiliser des constantes décimales dont la valeur est supérieure à la capacité du type `long`, tout en restant inférieure à la capacité du type `unsigned long` car on serait alors amené à créer une valeur de type non signé, sans nécessairement s'en apercevoir ; si cette constante apparaît dans une expression utilisant des types signés, le résultat peut être différent de celui escompté.

2. Les types caractère

Le langage C dispose non pas d'un seul, mais de deux types caractère, l'un signé, l'autre non signé. Cette curiosité est essentiellement liée à la forte connotation numérique de ces deux types. Ici, nous examinerons les différentes façons de nommer ces types, leurs caractéristiques et la façon d'écrire des constantes dans un programme.

2.1 Les deux types caractère

Les types caractère correspondent au mot-clé `char`. La norme ANSI prévoit en fait deux types caractère différents obtenus en introduisant dans le spécificateur de type, de façon facultative, un qualificatif de signe, à savoir `signed` ou `unsigned`. Cet attribut intervient essentiellement lorsqu'on utilise un type caractère pour représenter de petits entiers. C'est la raison pour laquelle la norme définit, comme pour les types entiers, le domaine (numérique) minimal des types caractère.

Contrairement à ce qui se passe pour les types entiers, l'absence de qualificatif de signe pour les caractères ne correspond pas systématiquement au type `signed char` mais plus précisément à l'un des deux types `signed char` ou `unsigned char`, ceci suivant l'implémentation et même, parfois, dans une implémentation donnée, suivant certaines options de compilation.

Tableau 3.5 : les deux types caractère du langage C

Dénomination usuelle	Noms de type possibles	Taille (en octets)	Domaine minimal
Caractère non signé	`unsigned char` `char` (suivant l'implémentation)	1	0 à 255
Caractère signé	`signed char` `char` (suivant l'implémentation)	1	-127 à 127

En C++

Alors que C dispose de deux types caractère, C++ en disposera de trois : `char` (malgré son ambiguïté, ce sera un type à part entière), `unsigned char` et `signed char`.

2.2 Caractéristiques des types caractère

Le tableau 3.6 récapitule les caractéristiques des types caractère. Ces derniers seront ensuite détaillés dans les sections suivantes de ce chapitre.

Tableau 3.6 : les caractéristiques des types caractère

Code associé à un caractère	– indépendant de l'attribut de signe ; – dépend de l'implémentation.	Voir section 2.2.1 de ce chapitre
Caractères existants	– au moins le jeu minimal d'exécution ; – ne pas oublier que certains caractères ne sont pas imprimables.	Voir section 2.2.2 de ce chapitre
Influence de l'attribut de signe	– en pratique, aucune, dans les simples manipulations de variables (type conseillé : `char` ou `unsigned char`) ; – importante si l'on utilise ce type pour représenter de petits entiers (type conseillé `signed char`).	Voir section 2.2.3 de ce chapitre
Manipulation d'octets	Possible par le biais de ce type, compte tenu de l'équivalence entre octet et caractère (type conseillé `unsigned char`).	Voir section 2.2.4 de ce chapitre

2.2.1 Code associé à un caractère

Les valeurs de type caractère sont représentées sur un octet, au sens large de ce terme, c'est-à-dire correspondant à la plus petite partie adressable de la mémoire. En pratique, le code associé à un caractère est indépendant de l'attribut de signe. Par exemple, on obtiendra exactement le même motif binaire dans `c1` et `c2` avec :

```
unsigned char c1 ;
signed char c2 ;
c1 = 'a' ;
c2 = 'a' ;
```

Cependant, on verra à la section 2.4, que les constantes caractère sont en fait de type `int`. Les instructions précédentes font donc intervenir des conversions d'entier en caractère dont les règles exactes sont étudiées à la section 9 du chapitre 4. Leur examen attentif montrera que, en théorie, cette conservation du motif binaire ne devrait être assurée que dans certains cas : caractères appartenant au jeu minimal d'exécution, variables caractère non signées (cas de `c1` dans notre exemple) dans les implémentations utilisant la représentation en complément à deux. En pratique, cette unicité se vérifie dans toutes les implémentations que nous avons rencontrées et d'ailleurs, beaucoup de programmes se basent sur elle.

Bien entendu, le code associé à un caractère donné dépend de l'implémentation. Certes, le code dit ASCII tend à se répandre, mais comme indiqué à la section 1.3 du chapitre 2, seul le code ASCII restreint a un caractère universel ; et nos caractères nationaux n'y figurent pas !

2.2.2 Caractères existants

La norme précise le jeu minimal de caractères d'exécution dont on doit disposer ; il est présenté à la section 1.1 du chapitre 2. Mais d'autres caractères peuvent apparaître dans une

implémentation donnée. Ainsi, lorsque l'octet occupe 8 bits (comme c'est presque toujours le cas), on est sûr de disposer d'un jeu de 256 caractères parmi lesquels figurent ceux du jeu minimal. Les caractères supplémentaires peuvent être imprimables ou de contrôle. À ce propos, signalons qu'il existe des fonctions standards permettant de connaître la nature (imprimable, alphabétique, numérique, de contrôle...) d'un caractère de code donné ; elles sont présentées au chapitre 18.

2.2.3 Influence de l'attribut de signe

Dans les manipulations de variables de type caractère

En pratique, tant que l'on se contente de manipuler des caractères en tant que tels, l'attribut de signe n'a pas d'importance. C'est le cas dans des situations telles que :

```
signed char c1 ;
unsigned char c2 ;
   .....
c2 = c1 ;
c1 = c2 ;
```

Le motif binaire est conservé par affectation. Cependant, là encore, si l'on examine la norme à la lettre, on constate que ces situations font intervenir des conversions (étudiées à la section 9 du chapitre 4) qui, en théorie, n'assurent cette conservation que dans certains cas : caractères appartenant au jeu minimal d'exécution, conversions de signé en non signé dans les implémentations utilisant la représentation en complément à deux. En pratique, ces conversions conservent le motif binaire dans toutes les implémentations que nous avons rencontrées.

Remarque

Si l'on vise une portabilité absolue, on pourra toujours éviter les conversions en évitant les mélanges d'attribut de signe. Cependant, dans ce cas, il faudra tenir compte du fait qu'une constante caractère est de type `int` (voir section suivante « Le type des constantes caractère »), ce qui pourra influer sur l'initialisation ou sur l'affectation d'une constante à une variable caractère. Si l'on suit la norme à la lettre, on verra que le seul cas de conservation théorique sera celui où l'on utilise le type `char`. En définitive, il faudra choisir entre :

- un type `char` qui assure la portabilité absolue du motif binaire mais qui présente une ambiguïté de signe (soit quand on s'intéresse à sa valeur numérique, soit lorsqu'on compare deux caractères, voir section 3.3.2 du chapitre 4) ;
- le type `unsigned char` qui, en théorie, n'assure la portabilité que dans les implémentations utilisant la représentation en complément à deux) mais qui ne présente plus l'ambiguïté précédente.

Dans les expressions entières

Comme on le verra à la section 9 du chapitre 4, il existe une conversion implicite de `char` en `int` qui pourra intervenir dans :

- des expressions dans lesquelles figurent des variables de type caractère ;
- des affectations du type caractère vers un type entier.

Ces conversions permettent aux types caractère d'être utilisés pour représenter des petits entiers. Dans ce cas, comme on peut s'y attendre, l'attribut de signe intervient pour définir le résultat de la conversion. On verra par exemple que, avec :

```
signed char c1 ;
unsigned char c2 ;
```

l'expression `c2+1` aura toujours une valeur positive, tandis que `c1+1` pourra, suivant la valeur de `c1`, être négative, positive ou nulle. De même, si `n1` et `n2` sont de type `int`, avec ces affectations :

```
n1 = c1 ;
n2 = c2 ;
```

la valeur de `n2` sera toujours positive ou nulle, tandis que celle de `n1` pourra être négative, positive ou nulle.

Les conseils fournis à la section 1.3 de ce chapitre, à propos des types entiers s'appliquent encore ici : si l'objectif est effectivement de réduire la taille de variables destinées à des calculs numériques classiques, il est conseillé d'utiliser systématiquement le type `signed char`.

2.2.4 Manipulations d'octets

Un des atouts du langage C est de permettre des manipulations dites « proches de la machine ». Parmi celles-ci, on trouve notamment les manipulations du contenu binaire d'un objet, indépendamment de son type. *A priori*, tout accès à un objet requiert un type précis défini par l'expression utilisée, de sorte que les manipulations évoquées semblent irréalisables. En fait, un objet est toujours formé d'une succession d'un nombre entier d'octets et un octet peut toujours être manipulé par le biais du type `char`.

Dans ces conditions, il est bien possible de manipuler les différents octets d'un objet quelconque, pour peu qu'on soit en mesure d'assurer la conservation du motif binaire. Cet aspect a été exposé à la section précédente. On a vu que cette conservation avait toujours lieu en pratique, même si en théorie, elle n'était absolue qu'avec le type `char`. Par ailleurs, les manipulations d'octets sont souvent associées à des manipulations au niveau du bit (masque, décalages...) pour lesquelles le type `unsigned char` sera plus approprié (voir section 6 du chapitre 4). Le type `unsigned char` constituera donc généralement le meilleur choix possible.

2.3 Écriture des constantes caractère

Il existe plusieurs façons d'écrire les constantes caractère dans un programme. Elles ne sont pas totalement équivalentes.

2.3.1 Les caractères « imprimables »

Les constantes caractère correspondant à des caractères imprimables peuvent se noter de façon classique, en écrivant entre apostrophes (ou *quotes*) le caractère voulu, comme dans ces exemples :

```
'a'  'Y'  '+'  '$'  '0'  '<'     /* caractères du jeu minimal d'exécution */
'é'  'à'  'ç'            /* n'existent que dans certaines implémentations */
```

On notera bien que l'utilisation de caractères n'appartenant pas au jeu minimal conduit à des programmes qu'on pourrait qualifier de « semi-portables ». En effet, une telle démarche présente les caractéristiques suivantes :

- elle est plus portable que celle qui consisterait à fournir directement le code du caractère voulu car on dépendrait alors de l'implémentation elle-même ;
- elle n'est cependant portable que sur les implémentations qui possèdent le graphisme en question (quel que soit son codage).

Par exemple, la notation `'é'` représente bien le caractère é dans toute implémentation où il existe, quel que soit son codage ; mais cette notation n'est pas utilisable dans les autres implémentations. À ce propos, il faut bien voir que cette notation du caractère imprimable n'est visible qu'à celui qui saisit ou qui lit un programme. Dès qu'on travaille sur des fichiers source, on a affaire à des suites d'octets représentant chacun un caractère. Par exemple, il n'est pas rare de saisir un caractère é dans une implémentation et de le voir apparaître différemment lorsqu'on exploite le même programme source dans une autre implémentation.

2.3.2 Les caractères disposant d'une « séquence d'échappement »

Certains caractères non imprimables possèdent une représentation conventionnelle dite « séquence d'échappement », utilisant le caractère \ (antislash)[3]. Dans cette catégorie, on trouve également quelques caractères qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteurs, ce qui les empêche d'être notés de manière classique entre deux apostrophes.

Tableau 3.7 : les caractères disposant d'une séquence d'échappement

Séquence d'échappement	Code ASCII (hexadécimal)	Abréviation usuelle	Signification
\a	07	BEL	Cloche ou bip (*alert* ou *audible bell*)
\b	08	BS	Retour arrière *(Backspace)*
\f	0C	FF	Saut de page *(Form Feed)*
\n	0A	LF	Saut de ligne *(Line Feed)*
\r	0D	CR	Retour chariot *(Carriage Return)*
\t	09	HT	Tabulation horizontale *(Horizontal Tabulation)*
\v	0B	VT	Tabulation verticale *(Vertical Tabulation)*
\\	5C	\	
\'	2C	‘	
\»	22	«	
\?	3F	?	

Si le caractère \ apparaît suivi d'un caractère différent de ceux qui sont mentionnés ici, le comportement du programme est indéterminé. Par ailleurs, une implémentation peut introduire d'autres séquences d'échappement ; il lui est cependant conseillé d'éviter les minuscules qui

3. Aussi appelé « barre inverse » ou « contre-slash » (*back-slash* en anglais).

sont réservées pour une future extension de la norme. Par essence, l'emploi de cette notation est totalement portable, quel que soit le code correspondant dans l'implémentation. En revanche, tout recours à un caractère de contrôle n'appartenant pas à cette liste nécessite l'introduction directe de son code, ce qui n'assure la portabilité qu'entre implémentations utilisant le même codage.

Remarque

La notation sous forme d'une séquence d'échappement ne dispense nullement de l'utilisation des apostrophes dans l'écriture d'une constante caractère. Ainsi, il faudra bien écrire `'\n'` et non simplement `\n`. Bien entendu, quand cette même séquence d'échappement apparaîtra dans une chaîne constante, ces apostrophes n'auront plus aucune raison d'être. Par exemple, on écrira bien :

```
"bonjour\nmonsieur"
```

2.3.3 Écriture d'un caractère par son code

Il est possible d'utiliser directement le code du caractère, en l'exprimant, toujours à la suite du caractère \ :

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

Voici quelques exemples de notations équivalentes (sur une même ligne), dans le code ASCII restreint :

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

La notation octale doit comporter de 1 à 3 chiffres ; la notation hexadécimale n'est pas soumise à des limites. Ainsi, `'\4321'` est incorrect, tandis que `'x4321'` est correct. Toutefois, dans le dernier cas, le code obtenu en cas de dépassement de la capacité d'un octet, n'est pas précisé par la norme. Il est donc recommandé de ne pas utiliser cette tolérance[4].

D'une manière générale, ces notations, manifestement non portables, doivent être réservées à des situations particulières telles que :

- besoin d'un caractère de contrôle non prévu dans le jeu d'exécution, par exemple `ACK`, `NACK`... Dans ce cas, on minimisera le travail de portage d'une machine à une autre en prenant bien soin de définir une seule fois, au sein d'un fichier en-tête, chacun de ces caractères par une instruction de la forme :

```
#define ACK 0x5
```

- besoin de décrire le motif binaire contenu dans un octet ; c'est notamment le cas lorsqu'on doit recourir à un masque binaire.

4. D'autant plus que certaines implémentations se permettent de limiter d'office (souvent à 3) le nombre de caractères pris effectivement en compte dans la notation hexadécimale.

Remarques

1. Le caractère \ suivi d'un caractère autre que ceux du tableau 3.7 ou d'un chiffre de 0 à 7, est simplement ignoré. Ainsi, dans le cas du code ASCII, \9 correspond au caractère 9 (de code ASCII 57), tandis que \7 correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».
2. Avec la notation hexadécimale ou octale, comme avec la notation sous forme d'une séquence d'échappement présentée à la section précédente 2.3.2, il ne faut pas oublier les apostrophes délimitant une constante caractère. Bien entendu, cette remarque ne s'appliquera plus au cas des constantes chaînes.

2.4 Le type des constantes caractère

Assez curieusement, la norme prévoit que :

Toute constante caractère est de type `int` et la valeur correspondante est obtenue comme si l'on convertissait une valeur de type `char` (dont l'attribut de signe dépend de l'implémentation) en `int`.

L'explication réside probablement dans le lien étroit existant en C entre caractères et entiers. Si l'on admet que les types caractère correspondent à des types entiers de petite taille, il n'est alors pas plus choquant de dire qu'une notation comme `'a'` est de type `int` que de dire qu'une « petite constante numérique » comme +43 était de type `int` (et non `short` !).

Dans ces conditions, on peut s'interroger sur le fait que, suivant les implémentations, le type `char` peut être signé ou non, de sorte que, suivant les règles de conversion étudiées dans le chapitre suivant, le résultat peut être parfois négatif. Nous allons examiner deux situations :

- on utilise les constantes caractère de façon naturelle, c'est-à-dire pour représenter de vrais caractères ;
- on utilise les constantes caractère pour leur valeur numérique.

2.4.1 Utilisation naturelle des constantes caractère

En pratique, les trois instructions suivantes placeront le même motif dans `c1`, `c2` et `c3` (la notation α désignant un caractère quelconque) :

```
char c1 = 'α' ;
unsigned char c2 = 'α' ;
signed char c3 = 'α' ;
```

Il en ira de même si la constante caractère est exprimée sous forme octale ou hexadécimale.

Cependant, si l'on examine la norme à la lettre, on constate que ces situations, hormis la première, font intervenir une suite de deux conversions de `char` en `int`, puis en `char`. En théorie (voir section 9 du chapitre 4), elles n'assurent la conservation du motif binaire que dans certains cas : caractères appartenant au jeu minimal d'exécution, conversions de signé en non signé dans les implémentations utilisant le complément à deux. En pratique, ces conversions conservent le motif binaire dans toutes les implémentations que nous avons rencontrées. Si toutefois on cherche une portabilité absolue, on pourra se limiter à l'utilisation du type `char`, à condition que l'ambiguïté correspondante ne s'avère pas gênante (voir section précédente 2.2.3).

2.4.2 Utilisation des constantes caractère pour leur valeur numérique

Il s'agit du cas où une telle constante apparaît dans une expression numérique ou dans une affectation à une variable entière, ce qui peut se produire lorsqu'on s'intéresse à la valeur du code du caractère correspondant. Dans ce cas, l'implémentation intervient, non seulement sur la valeur obtenue, mais éventuellement sur son signe. Voici un exemple dans lequel on suppose que le code du caractère é est supérieur à 127, dans une implémentation codant les caractères sur 8 bits et utilisant la représentation en complément à deux :

```
int n = 'é' ;   /* la valeur de n sera >0 si char est signé       */
                /*                 et >0 si char n'est pas signé  */
```

On peut simplement affirmer que l'effet de cette déclaration sera équivalent à :

```
char c = 'é' ;  /* char est signé ou non suivant l'implémentation */
int n ;
   .....
n = c ;
```

D'une manière comparable, dans la même implémentation (octets de 8 bits, représentation en complément à deux) :

```
int n ='\xFE' ;  /* -2 si le type char est signé par défaut         */
                 /* 254 si le type char n'est pas signée par défaut */
```

On notera bien que, alors qu'on pouvait choisir l'attribut de signe d'une variable de type `char`, il n'en va plus de même pour une constante. On peut toutefois faire appel à l'opérateur de `cast` comme dans :

```
int n = (signed char)   '\xfE' ;   /* conv char -> signed char -> int          */
                                   /*  -2 dans le cas du complément à deux     */
int n = (unsigned char) '\xfE' ;   /* conv char -> unsigned signed char -> int */
                                   /*  254 dans le cas du complément à deux    */
```

Remarques

Ici, il est important de ne pas confondre valeur et motif binaire. Le motif binaire associé à une constante caractère est bien constant après conversion dans un type `char` de type quelconque (du moins, en pratique) ; en revanche, la valeur numérique correspondante de type `int`, ne l'est pas.

La norme autorise des constantes caractère de la forme `'xy'` voire `'xyz'`, contenant plusieurs caractères imprimables. Certes, une telle particularité peut se justifier par le fait que la constante produite est effectivement de type `int` et non de type `char` ; elle reste cependant d'un emploi malaisé et, de toute façon, la valeur ainsi obtenue dépend de l'implémentation.

En C++

En C++, les constantes caractère seront effectivement de type `char`, et non plus de type `int`.

3. Le fichier limits.h

3.1 Son contenu

Ce fichier en-tête contient, sous forme de constantes ou macros (définies par la directive `#define`), de nombreuses informations concernant le codage des entiers et les limitations qu'elles imposent dans une implémentation donnée.

En voici la liste, accompagnée de la valeur minimale qu'on est assuré d'obtenir pour chaque type d'entier quelle que soit l'implémentation. On notera la présence d'une constante `MB_LEN_MAX` relative aux caractères dits « multi-octets », d'usage assez peu répandu, et dont nous parlerons au chapitre 22.

Tableau 3.8 : les valeurs définies dans le fichier limits.h

Symbole	Valeur minimale	Signification
CHAR_BIT	8	Nombre de bits dans un caractère
SCHAR_MIN	-127	Plus petite valeur (négative) du type `signed char`
SCHAR_MAX	+127	Plus grande valeur du type `signed char`
UCHAR_MAX	255	Plus grande valeur du type `unsigned char`
CHAR_MIN		Plus petite valeur du type `char` (`signed char` ou `unsigned char` suivant l'implémentation, ou même suivant les options de compilation)
CHAR_MAX		Plus grande valeur du type `char` (`signed char` ou `unsigned char` suivant l'implémentation, ou même suivant les options de compilation)
MB_LEN_MAX	1	Nombre maximal d'octets dans un caractère multi-octets (quel que soit le choix éventuel de localisation)
SHRT_MIN	- 32 767	Plus petite valeur du type `short int`
SHRT_MAX	+ 32 767	Plus grande valeur du type `short int`
USHRT_MAX	65 535	Plus grande valeur du type `unsigned short int`
INT_MIN	- 32 767	Plus petite valeur du type `int`
INT_MAX	+ 32 767	Plus grande valeur du type `int`
UINT_MAX	65 535	Plus grande valeur du type `unsigned int`
LONG_MIN	- 2 147 483 647	Plus petite valeur du type `long int`
LONG_MAX	+ 2 147 483 647	Plus grande valeur du type `long int`
ULONG_MAX	4 294 967 295	Plus grande valeur du type `unsigned long int`

3.2 Précautions d'utilisation

On notera bien que la norme impose peu de contraintes au type des constantes définies dans `limits.h`. En général, on trouvera des définitions de ce genre :

```
#define INT_MAX +32767
```

Le symbole `INT_MAX` sera donc remplacé par le préprocesseur par la constante +32 767, laquelle sera de type `int`. Dans ces conditions, on évitera certains calculs arithmétiques risquant de conduire à des dépassements de capacité dans le type `int`. Le programme suivant montre ce qu'il ne faut pas faire, puis ce qu'il faut faire, pour calculer la valeur de `INT_MAX+5` :

Exemple de mauvais et de bon usage de la constante INT_MAX

```
#include <stdio.h>
#include <limits.h>
int main()
{
  int n ;
  long q ;
  q = INT_MAX + 5 ;                    /* calcul incorrect */
  printf ("INT_MAX+5 = %ld\n", q) ;
  q = (long)INT_MAX + 5 ;              /* calcul correct   */
  printf ("INT_MAX+5 = %ld\n", q) ;
}
```

```
INT_MAX+5 = -32764
INT_MAX+5 = 32772
```

4. Les types flottants

Nous commencerons par rappeler brièvement en quoi consiste le codage en flottant d'un nombre réel et quels sont les éléments caractéristiques d'un codage donné : précision, limitations, *epsilon machine*... Puis nous examinerons les trois types de flottants prévus par la norme, leur nom et leurs caractéristiques respectives. Nous terminerons par les différentes façons d'écrire des constantes flottantes dans un programme source.

4.1 Rappels concernant le codage des nombres en flottant

Les types flottants (appelés parfois, un peu à tort, réels) servent à représenter de manière approchée une partie des nombres réels. Ils s'inspirent de la notation scientifique des calculettes, dans laquelle on exprime un nombre sous forme d'une mantisse et d'un exposant correspondant à une puissance de 10, comme dans 0.453 E 15 (mantisse 0,453, exposant 15) ou dans 45.3 E 13 (mantisse 45,3, exposant 13). Le codage en flottant se distinguera cependant de cette notation scientifique sur deux points :

- le codage de la mantisse : il est généralement fait en binaire et non plus en base 10 ;

- la base utilisée pour l'exposant : on n'a guère de raison d'employer une base décimale. En général, des bases de 2 ou de 16 seront utilisées car elles facilitent grandement les calculs de la machine (attention, l'utilisation d'une base 16 n'est pas incompatible avec le codage en binaire des valeurs de la mantisse et de l'exposant) .

D'une manière générale, on peut dire que la représentation d'un nombre réel en flottant revient à l'approcher par une quantité de la forme

$$s.\ m\ .\ b^{\ e}$$

dans laquelle :

- *s* représente un signe, ce qui revient à dire que *s* vaut soit -1, soit +1 ;
- *m* représente la mantisse ;
- *e* représente l'exposant, tel que : $e_{min} <= e <= e_{max}$;
- *b* représente la base.

La base *b* (en pratique 2 ou 16) est fixée pour une implémentation donnée[5]. Il n'en reste pas moins qu'un même nombre réel peut, pour une valeur *b* donnée, être approché de plusieurs façons différentes par la formule précédente. Par exemple, si un nombre est représentable par le couple de valeurs (*m, e*), il reste représentable par le couple de valeurs (*b/m*, *e+1*) ou (*mb*, *e-1*)...

Pour assurer l'unicité de la représentation, on fait donc appel à une contrainte dite de « normalisation ». En général, il s'agit de :

$$1/b <= m< 1$$

Dans le cas de la notation scientifique des calculettes, l'application de cette contrainte conduirait à une mantisse commençant toujours par 0 et dont le premier chiffre après le point décimal est non nul. Par exemple, 0.2345 et 0.124 seraient des mantisses normalisées, tandis que 3.45 ou 0.034 ne le seraient pas[6].

4.2 Le modèle proposé par la norme

Contrairement à ce qui se passe, en partie du moins, pour les nombres entiers, la norme ANSI n'impose pas de contraintes précises quant à la manière dont une implémentation représente les types flottants. Elle se contente de proposer un modèle théorique possible, dont le principal avantage est de donner une définition formelle d'un certain nombre de paramètres caractéristiques tels que la précision ou l'epsilon machine. Ce modèle correspond à la formule précédente (voir section 4.1), dans laquelle on explicite la mantisse de la façon suivante, le coefficient f_1 étant non nul si le nombre est non nul :

$$m = \sum_{k=1}^{p} f_k b^{-k}$$

5. En toute rigueur, rien n'interdirait à une implémentation d'utiliser une base (par exemple, 2) pour un type (par exemple, `float`) et une autre base (par exemple, 16) pour un autre type (par exemple, `long double`). Cela conduirait toutefois à complexifier inutilement l'unité centrale, de sorte qu'en pratique, cette situation ne se rencontre pas !
6. Attention à ne pas confondre la contrainte de normalisation utilisée pour coder un nombre flottant en mémoire avec celle qu'utilise `printf` pour afficher un tel nombre.

Certes, cette formule définit la mantisse de façon unique, dès lors que p, b et les limites e_{min} et e_{max} sont fixées. Comme on s'en doute, elles dépendront de l'implémentation et du type de flottant utilisé (`float`, `double` ou `long double`). Mais il ne s'agit que d'un modèle théorique de comportement et, même si la plupart des implémentations s'en inspirent, quelques petits détails de codage peuvent apparaître :

- élimination de certains bits superflus de la mantisse ; par exemple, lorsque la base est égale à 2, le premier bit est toujours à 1 et certaines implémentations ne le conservent pas, ce qui double la capacité ;
- réservation, comme le propose la norme IEEE 754, de certains motifs binaires pour représenter des nombres infinis ou des quantités non représentables.

4.3 Les caractéristiques du codage en flottant

Le codage en entier n'a guère d'autres conséquences que de limiter le domaine des valeurs utilisables. Dans le cas du codage en flottant, les conséquences sont moins triviales et nous allons les examiner ici, en utilisant parfois le modèle théorique présenté précédemment.

4.3.1 Représentation approchée

Le codage en flottant permet de représenter un nombre réel de façon approchée, comme on le fait dans la vie courante en approchant le nombre réel *pi* par 3.14 ou 3.14159... La notion de représentation approchée paraît alors relativement naturelle. En revanche, lorsqu'on a affaire à un nombre décimal tel que 0.1, qui s'exprime de manière exacte dans notre système décimal, on peut être surpris de ce qu'il ne s'exprime plus toujours de façon exacte une fois codé en flottant[7]. Voici un petit programme illustrant ce phénomène, dans une implémentation où la base b de l'exposant est égale à 2 :

Conséquences de la représentation approchée des nombres flottants

```
include <stdio.h>
int main()
{ float x = 0.1 ;
  printf ("x avec 1  decimale  : %.1e\n", x) ;
  printf ("x avec 10 decimales : %.10e\n", x) ;
}
```

```
x avec 1  decimale  : 1.0e-01
x avec 10 decimales : 1.0000000149e-01
```

Cependant, la norme impose aux entiers dont la valeur absolue est inférieure à une certaine limite d'être représentés de façon exacte en flottant, de façon à ce qu'un cycle de conversion

7. Pour s'en convaincre, il suffit d'exprimer la valeur 0,1 dans le modèle de comportement proposé par la norme : on s'aperçoit que la conversion en binaire (exprimée en puissances négatives de b) conduit dans les cas usuels (b=2 ou b=16) à une mantisse m ayant un nombre infini de décimales, et donc obligatoirement à une approximation, quel que soit le nombre de bits réservés à m.

entier → flottant → entier permette de retrouver la valeur d'origine. Ces limites dépendent à la fois du type de flottant concerné et de l'implémentation ; elles sont précisées dans le tableau 3.8, qui montre qu'elles sont toujours au moins égales à 1E6.

4.3.2 Notion de précision

On peut définir la précision d'une représentation flottante :

- soit en considérant le nombre de chiffres en base b, c'est-à-dire finalement la valeur de p dans le modèle défini par la norme et présenté à la section 4.2 ; cette valeur est définie de façon exacte ;
- soit en cherchant à exprimer cette précision en termes de chiffres décimaux significatifs ; en théorie, on peut montrer que p chiffres exacts en base b conduisent toujours à au moins q chiffres décimaux exacts, avec q = (p-1).log10 b ; autrement dit, tout nombre entier d'au plus q chiffres s'exprime sans erreur en flottant.

Ces différentes valeurs sont fournies dans le fichier `float.h` décrit à la section 5.

4.3.3 Limitations des valeurs représentables

Dans le cas du type entier, les valeurs représentables appartenaient simplement à un intervalle de l'ensemble des entiers relatifs. Dans le cas des flottants, on a affaire à une limitation des valeurs de l'exposant e_{min} et e_{max}, lesquelles conduisent en fait à une limitation de l'amplitude de la valeur absolue du nombre. Les valeurs réelles représentables appartiennent donc à deux intervalles disjoints, de la forme :

$$[-x_{max}, -x_{min}] \quad [x_{min}, x_{max}] \qquad \text{avec } x_{min} = b^{e_{min}} \quad \text{et} \quad x_{max} = b^{e_{max}}$$

En outre, la valeur 0 (qui n'appartient à aucun de ces deux intervalles) est toujours représentable de façon exacte ; l'unicité de sa représentation nécessite l'introduction d'une contrainte conventionnelle, par exemple, mantisse nulle, exposant 1.

4.3.4 Non-respect de certaines règles de l'algèbre

La représentation approchée des types flottants induit des différences de comportement par rapport à l'algèbre traditionnelle.

Certes, la commutativité des opérations est toujours respectée ; ainsi les opérations *a+b* ou *b+a* donneront-elles toujours le même résultat (même si celui-ci n'est qu'une approximation de la somme).

En revanche, si *a* et *b* désignent des valeurs réelles et si *x'* désigne l'approximation en flottant de l'expression *x*, on n'est pas assuré que les conditions suivantes soient vérifiées :

$$(a + b)' = a' + b'$$
$$(a' + b')' = a' + b'$$

Par exemple :

```
float x = 0.1, y = 0.1
   .....
if (x + y == 0.2)          /* peut être vrai ou faux */
```

De façon comparable, on n'est pas assuré que ces conditions soient vérifiées[8] :

$$(3 * a)' = 3 * a'$$

$$(3 * a')\ ' = 3 * a'$$

Par exemple :

```
float x = 0.1 ;
if (3*x == 0.3)          /* peut être vrai ou faux */
```

Par ailleurs, l'associativité de certaines opérations n'est plus nécessairement respectée. On n'est plus assuré que :

$$a' + (b'+c')' = (a' + b')' + c'$$

Tout ceci se compliquera encore un peu plus avec l'incertitude qui règne en C sur l'ordre d'évaluation des opérateurs commutatifs, comme nous le verrons à la section 2.1.4 du chapitre 4.

4.3.5 Notion d'epsilon machine

Compte tenu de la représentation approchée du type flottant, on peut aisément trouver des nombres `eps` tels que la représentation de la somme de `eps+1` soit identique à celle de 1, autrement dit que la condition suivante soit vraie :

```
1 + eps == 1
```

La plus grande de ces valeurs se nomme souvent « l'espilon machine ». On peut montrer qu'elle est définie par :

$$eps = b^{1-p}$$

où *b* et *p* sont définis par le modèle de comportement ANSI présenté à la section 4.2.

On en trouvera la valeur pour chacun des types flottants dans le fichier `float.h` décrit à la section 5.

4.4 Représentation mémoire et limitations

La norme ANSI prévoit les trois types de flottants suivants :

Tableau 3.9 : les trois types flottants prévus par la norme

Spécificateur de type	Domaine minimal	Précision minimale	Propriétés imposées par la norme
`float`	10^{-37} à 10^{+37}	6	Les entiers d'au plus 6 chiffres sont toujours représentés de façon exacte.
`double`	10^{-37} à 10^{+37}	10	Les entiers d'au plus 10 chiffres sont toujours représentés de façon exacte.

8. Ici, il n'est pas nécessaire de considérer 3' car, dans toute implémentation, 3' = 3 puisque tout nombre entier d'au plus *q* chiffres s'exprime exactement en flottant (voir section 4.3.2) et que la valeur de *q* est toujours supérieure ou égale à 6 (voir section 5).

Spécificateur de type	Domaine minimal	Précision minimale	Propriétés imposées par la norme
`long double`	10^{-37} à 10^{+37}	10	Les entiers d'au plus 10 chiffres sont toujours représentés de façon exacte.

Bien entendu, les caractéristiques exactes de chacun de ces types dépendent à la fois de l'implémentation et du type concerné. Un certain nombre d'éléments sont cependant généralement communs, dans une implémentation donnée, aux trois types de flottants :

- la technique d'approximation (arrondi par défaut, par excès, au plus proche...) ;
- la valeur de la base de l'exposant *b* ;
- la manière dont la mantisse est normalisée.

D'autres éléments, en revanche, dépendent effectivement du type de flottant (et aussi de l'implémentation) à savoir :

- le nombre de bits utilisés pour coder la mantisse *m* ;
- le nombre de bits utilisés pour coder l'exposant *e*.

D'une manière générale, le fichier `float.h` contient bon nombre d'informations concernant les caractéristiques des flottants.

Remarques

1. La première définition du langage C (Kernighan et Ritchie) ne comportait pas le type `long double`. En outre, `long float` y apparaissait comme un synonyme de `double` ; cette possibilité a disparu de la norme.
2. Certaines implémentations acceptent des valeurs flottantes non normalisées, c'est-à-dire des valeurs dans lesquelles la mantisse comporte un ou plusieurs de ses premiers chiffres (en base *b*) nuls. Dans ce cas, il devient possible de manipuler des valeurs inférieures en valeur absolue au minimum imparti au type, moyennant, alors une perte de précision...

4.5 Écriture des constantes flottantes

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

- décimale ;
- exponentielle.

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

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

En revanche, la constante `47` serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance[9], compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons au chapitre suivant).

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

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

4.6 Le type des constantes flottantes

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

- d'être du type `float`, en faisant suivre son écriture de la lettre F (ou f), comme dans 1.25E+03f ; cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision ;
- d'être du type `long double`, en faisant suivre son écriture de la lettre L (ou l), comme dans 1.0L ; cela permet de gagner en précision, en contrepartie d'une perte de place mémoire ; c'est aussi le seul moyen de représenter les valeurs très grandes ou très petites (bien qu'un tel besoin soit rare en pratique).

4.7 En cas de dépassement de capacité dans l'écriture des constantes

Contrairement à ce qui se produit pour les entiers, la manière dont est écrite une constante flottante impose son type, de manière unique : `float`, `double` ou `long double`. Dans chacun de ces trois cas, vous avez affaire à des limitations propres, à la fois :

- vers « le bas » : une constante de valeur absolue trop petite ne peut être représentée (avec une erreur relative d'approximation raisonnable) ; on parle alors de sous-dépassement de capacité (en anglais *underflow*) ;
- vers « le haut » : une constante de valeur absolue trop grande ne peut être représentée (avec une erreur relative d'approximation raisonnable) : on parle alors de dépassement de capacité (en anglais *overflow*).

Là encore, suivant les compilateurs, on pourra obtenir : un diagnostic de compilation, une valeur fantaisiste ou une utilisation des conventions IEEE 754 (présentées à la section 2.2.2) en cas de dépassement de capacité, une valeur fantaisiste ou une valeur nulle en cas de sous-dépassement de capacité.

9. Si ce n'est au niveau du temps d'exécution.

5. Le fichier float.h

Ce fichier contient, sous forme de constantes ou de macros (définies par la directive `#define`), de nombreuses informations concernant :

- les caractéristiques du codage des flottants tel qu'il est défini par le modèle théorique de comportement proposé par la norme et présenté à la section 4.2 : base b, précision p en base b ou précision en base 10, epsilon machine...
- les limitations correspondantes : e_{min}, e_{max}, x_{min}, x_{max}...

En voici la liste. On notera que, à l'exception des symboles `FLT_ROUNDS` et `FLT_RADIX` qui concernent les trois types flottants (`float`, `double` et `long double`), les autres symboles sont définis pour les trois types avec le même suffixe et un préfixe indiquant le type concerné :

- `FLT` : le symbole correspondant (par exemple, `FLT_MIN_EXP`) concerne le type `float` ;
- `DBL` : le symbole correspondant (par exemple, `DBL_MIN_EXP`) concerne le type `double` ;
- `LDBL` : le symbole correspondant (par exemple, `LDBL_MIN_EXP`) concerne le type `long double`.

Tableau 3.10 : le contenu du fichier float.h

Symbole	Valeur minimale	Signification
`FLT_RADIX`	2	Base *b* telle que définie à la section 4.2
`FLT_ROUNDS`		Méthode utilisée pour déterminer la représentation d'un nombre réel donné : -1 : indéterminée 0 : arrondi vers zéro 1 : arrondi au plus proche 2 : arrondi vers plus l'infini 3 : arrondi vers moins l'infini autre : méthode définie par l'implémentation

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
		Précision (*p* dans la formule de la section 4.2)

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
	6
	10
	10
		Valeur *q*, telle que tout nombre décimal de *q* chiffres puisse être exprimé sans erreur en notation flottante ; on peut montrer que :
		$q = (p-1).\log_{10}b$ (+1 si *b* est puissance de 10)

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
		Plus petit nombre négatif *n* tel que $\texttt{FLT\_RADIX}^{n-1}$ soit un nombre flottant normalisé ; il s'agit de e_{min} tel qu'il est défini à la section 4.2

`FLT_MIN_10_EXP`
`DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`
	-37
	-37
	-37
		Plus petit nombre négatif *n* tel que 10^n soit dans l'intervalle des nombres flottants normalisés ; on peut montrer que :
		$n = \log_{10} b^{e_{min-1}}$

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
		Plus grand nombre *n* tel que $\texttt{FLT\_RADIX}^{n-1}$ soit un nombre flottant fini représentable ; il s'agit de e_{max} tel qu'il est défini à la section 4.2

Symbole	Valeur minimale	Signification

`FLT_MAX_10_EXP`
`DBL_MAX_10_EXP`
`LDBL_MAX_10_EXP`
	+37
	+37
	+37
		Plus grand entier *n* tel que 10^n soit dans l'intervalle des nombres flottants finis représentables ; on peut montrer que :
		$n = \log_{10}((1-b^{-p})b^{e_{max}})$

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX`
	1e37
	1e37
	1e37
		Plus grande valeur finie représentable ; on peut montrer qu'il s'agit de :
		$(1-b^{-p})b^{e_{max}}$

`FLT_EPSILON`
`DBL_EPSILON`
`LDBL_EPSILON`
	1e-5
	1e-9
	1e-9
		Écart entre 1 et la plus petite valeur supérieure à 1 qui soit représentable ; il s'agit de ce que l'on nomme généralement l'epsilon machine dont on montre qu'il est égal à : b^{1-p}

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
	1e-37
	1e-37
	1e-37
		Plus petit nombre flottant positif normalisé. On montre qu'il est égal à :
		$b^{e_{min-1}}$

6. Déclarations des variables d'un type de base

Le tableau 3.11 récapitule les différents éléments pouvant intervenir dans la déclaration des variables d'un type de base. Ils seront détaillés dans les sections indiquées.

Tableau 3.11 : déclaration de variables d'un type de base

Rôle d'une déclaration	– associe un spécificateur de type (éventuellement complété de qualifieurs et d'une classe de mémorisation), à un déclarateur ; – dans le cas des types de base, le déclarateur se limite au nom de la variable.	voir section 6.1
Initialisation variables classe automatique (ou registre)	– aucune initialisation par défaut ; – initialisation explicite par expressions quelconques.	voir section 6.2
Initialisation variables classe statique	– initialisation par défaut à zéro ; – initialisation explicite par des expressions constantes.	voir section 6.2
Qualifieurs (**`const`**, **`volatile`**)	– `const` : la variable ne peut pas voir sa valeur modifiée ; – `volatile` : la valeur de la variable peut changer, indépendamment des instructions du programme ; – une variable constante doit être initialisée (il existe deux rares exceptions – voir remarque 3 de la section 6.3.2).	voir section 6.3
Classe de mémorisation	– `extern` : pour les redéclarations de variables globales ; – `auto` : pour les variables locales (superflu) ; – `static` : variable rémanente ; – `register` : demande de maintien dans un registre.	voir chapitre 8

6.1 Rôle d'une déclaration

6.1.1 Quelques exemples simples

Comme on s'y attend, la déclaration d'une variable d'un type de base permet de préciser son nom et son type, par exemple :

```
unsigned int n ;     /* n est de type unsigned int */
```

On peut déclarer plusieurs variables dans une seule instruction. Par exemple :

```
unsigned int n, p ;
```

est équivalente à :

```
unsigned int n ;
unsigned int p ;
```

Un même type peut être défini par des spécificateurs de type différents. Par exemple, ces quatre instructions sont équivalentes :

```
short int p
signed short p
signed short int p ;
short p ;
```

Les tableaux 3.1, 3.5 et 3.9 fournissent les différents spécificateurs de type qu'il est possible d'utiliser pour un type donné.

6.1.2 Les déclarations de variables en général

Tant qu'on se limite à des variables d'un type de base, les déclarations restent relativement simples puisque, comme dans les précédents exemples, elles associent un simple identificateur à un spécificateur de type. On peut cependant y trouver quelques informations supplémentaires, parmi les suivantes :

- une valeur initiale de la variable, comme dans :

```
int n, p=5, q ;       /* n, p et q sont des int; p est initialisée à 5 */
```

- un ou plusieurs qualifieurs (`const` ou `volatile`) associés au spécificateur de type et qui concernent donc l'ensemble des variables de la déclaration, par exemple :

```
const float x, y ;    /* x et y sont des float constants */
```

- une classe de mémorisation associée, elle aussi, à l'ensemble des variables de la déclaration, par exemple :

```
static int n, q ; /* n et q sont déclarés avec la classe de mémorisation static */
```

Les deux premiers points (valeur initiale et qualifieurs) sont examinés ici, dans le seul cas cependant des variables d'un type de base. Pour les autres types, on trouvera des compléments d'information dans les chapitres correspondants (tableaux, pointeurs, structures, unions,

énumérations). Quant à la classe de mémorisation, dont on a dit au chapitre 1 qu'elle pouvait influer sur la classe d'allocation des variables, elle est étudiée en détail au chapitre 8.

Remarque

D'une manière générale, les déclarations en C sont complexes et parfois peu naturelles. Ainsi, une même instruction peut déclarer des variables de types différents, par exemple un entier, un pointeur sur un entier et un tableau d'entiers, comme dans :

```
unsigned int n, *adi, t[10] ;
```

Pour connaître le type correspondant à un identificateur donné, on considère qu'une telle déclaration associe un spécificateur de type (ici `unsigned int`) non pas simplement à des identificateurs, mais à des déclarateurs (ici `n`, `*adi` et `t[10]`). Il existe trois formes de déclarateurs (tableaux, pointeurs, fonctions) qui peuvent se composer à volonté. Chacun de ces déclarateurs sera étudié dans le chapitre correspondant, tandis que le chapitre 16 récapitulera tout ce qui concerne les déclarations.

6.2 Initialisation lors de la déclaration

Une variable peut être initialisée lors de sa déclaration comme dans :

```
int n = 5 ;
```

Cela signifie que la valeur 5 sera placée dans l'emplacement correspondant à *n*, avant le début de l'exécution de la fonction ou du bloc contenant cette déclaration (pour les variables de classe automatique) ou avant le début de l'exécution du programme (pour les variables de classe statique).

On notera bien qu'une variable ainsi initialisée reste une « vraie variable », c'est-à-dire que son contenu peut tout à fait être modifié lors de l'exécution du programme ou de la fonction correspondante.

Dans une même déclaration, on peut initialiser certaines variables et pas d'autres :

```
int n=5, p, q=3 ;
```

L'expression utilisée pour initialiser une variable porte le nom d'initialiseur. Le chapitre8 fait le point sur les différents initialiseurs qu'il est possible d'utiliser pour tous les types de variables. Pour résumer ce qui concerne les variables d'un type de base, disons qu'un initialiseur peut être :

- une expression quelconque pour les variables de classe automatique ;
- une expression constante, c'est-à-dire calculable par le compilateur, pour les variables de classe statique ; la notion d'expression constante est étudiée en détail à la section 14 du chapitre 4.

Le type de l'expression servant d'initialiseur n'est pas obligatoirement du type de la variable à initialiser ; il suffit qu'il soit d'un type autorisé par affectation (voir section 7 du chapitre 4).

Voici quelques exemples :

```
float x = 5 ;       /* la valeur entière 5 sera convertie en float        */
                    /* comme elle le serait dans une affectation          */
int n = 8.23 ;      /* la valeur flottante (environ 8,23) sera convertie  */
                    /* en int comme elle serait dans une affectation      */
                    /* ici, il serait plus raisonnable d'écrire :         */
                    /*     int n = 8 ;                                    */
float x = 40.73 ;
int n = x/2.3 ;     /* l'expression x/2.3 est évaluée en flottant ;       */
                           /*  son résultat est converti en entier        */
```

6.3 Les qualifieurs const et volatile

La norme ANSI a introduit la possibilité d'ajouter dans une déclaration des qualifieurs choisis parmi les mots-clés `const` et `volatile`. Le premier est de loin le plus utilisé, et son rapprochement avec le second n'est qu'une pure affaire de syntaxe. Ici, nous étudions la signification de ces qualifieurs lorsqu'ils sont appliqués à une variable d'un type de base.

6.3.1 Le qualifieur const

Considérons la déclaration :

```
const int n = 5, p = 12 ;
```

Elle précise que `n` et `p` sont de type `int` et que, de plus, leur valeur ne devra pas varier au fil de l'exécution du programme. Cependant, la norme ne précise pas de façon exhaustive les situations que le compilateur devrait interdire. Dans la plupart des implémentations, il rejettera alors une instruction telle que les suivantes, dès lors qu'elles figurent dans la portée de la déclaration de `n` (la fonction ou le bloc pour une variable locale, la partie du fichier source suivant sa déclaration pour une variable globale) :

```
n = 6 ;       /* généralement rejeté puisque n est qualifié de constant */
n++ ;         /* généralement rejeté puisque n est qualifié de constant */
```

Cependant, quelle que soit la bonne volonté du compilateur, des modifications indirectes de ces variables restent possibles, notamment :

- par appel d'une fonction de lecture, par exemple :

```
scanf ("%d", &n) ;     /* toujours accepté car le compilateur n'a aucune  */
                       /* connaissance du rôle de scanf                   */
```

- par l'utilisation d'un pointeur sur ces variables (pour peu que le qualifieur `const` n'ait pas été attribué à l'objet pointé, comme on le verra au chapitre 7...).

Il n'en reste pas moins que l'usage systématique de `const` améliore la lisibilité des programmes.

Remarques

1. Comme on le verra au chapitre 4, une variable déclarée d'un type qualifié par `const` n'est pas une expression constante ; il s'agit là d'une lacune importante de la norme ANSI du C, à laquelle le langage C++ a d'ailleurs remédié.
2. La norme ANSI laisse l'implémentation libre d'allouer les emplacements destinés à des objets constants dans une mémoire protégée contre toute modification pendant l'exécution du programme. Dans ce cas, les tentatives de modification de tels objets, lorsqu'elles ne sont pas rejetées à la compilation, provoqueront obligatoirement une erreur d'exécution.

6.3.2 Le qualifieur volatile

Il s'emploie de la même manière que `const` ; il sert à préciser au compilateur qu'une variable (ou un objet pointé) peut voir sa valeur évoluer, indépendamment des instructions du programme. Un tel changement peut par exemple être provoqué :

- par une interruption ;
- par un périphérique qui agit sur des emplacements particuliers de la mémoire ; dans ce cas, `volatile` s'appliquera généralement à un objet pointé plutôt qu'à une variable sauf si l'on dispose dans l'implémentation concernée d'un moyen permettant d'imposer une adresse à une variable.

L'intérêt de ce qualifieur `volatile` est d'interdire au compilateur d'effectuer certaines optimisations. Par exemple, avec :

```
volatile int etat ;
int n ;
  .....
while (...)
  { n = etat + 1 ;
    .....
  }
```

le compilateur ne sortira jamais l'instruction `n = etat + 1` de la boucle, comme il pourrait le faire si `etat` n'avait pas été déclarée avec le qualifieur `volatile` et qu'elle n'était pas modifiée à l'intérieur de la boucle.

Remarques

1. Les qualifieurs `const` ou `volatile` s'appliquent à toutes les variables mentionnées dans l'instruction de déclaration : il n'est pas possible, dans une même instruction, de déclarer une variable ayant le qualifieur `const` et une autre ne l'ayant pas. Cette remarque ne s'appliquera cependant pas aux variables de type pointeur, compte tenu de la manière dont ces qualifieurs sont alors utilisés.
2. En théorie, il est possible d'utiliser conjointement les deux qualifieurs `const` et `volatile` (l'ordre est alors indifférent) :

```
const volatile int n ;      /* la valeur de n ne peut pas être modifiée par le   */
                            /* programme mais elle peut l'être "de l'extérieur" */
                            /* c'est pourquoi elle peut ne pas être initialisée */
volatile const int p = 5 ; /* même chose mais ici, p a été initialisée          */
                            /* lors du déroulement du programme, sa valeur      */
                            /* pourra devenir différente de 5                   */
```

3. Une variable ayant reçu l'attribut `const` doit être initialisée lors de sa déclaration, à deux exceptions près :
   - elle possède en plus le qualifieur `volatile` : elle pourra donc être modifiée indépendamment du programme ; son initialisation n'est donc pas indispensable mais elle reste possible ;
   - il s'agit de la redéclaration d'une variable globale (par `extern`) ; l'initialisation a dû être faite par ailleurs ; qui plus est, l'initialisation est alors interdite à ce niveau.

7

Les pointeurs

Comme certains autres langages, C permet de manipuler des adresses d'objets ou de fonctions, par le biais de ce que l'on nomme des pointeurs. Pour ce faire, on peut définir des variables dites de type pointeur, destinées à contenir des adresses. Plus généralement, on peut faire appel à des expressions de type pointeur, dont l'évaluation fournit également une adresse. Le terme de pointeur tout court peut indifféremment s'appliquer à une variable ou à une expression. Il existe différents types pointeur, un type précis se définissant par le type des objets ou des fonctions pointés. Ce dernier aspect sera d'autant plus important que certaines opérations ne seront possibles qu'entre pointeurs de même type.

En matière de pointeurs, une fois de plus, C fait preuve d'originalité à la fois au niveau du typage fort (type défini à la compilation) qu'il impose aux pointeurs, des opérations arithmétiques qu'on peut leur faire subir et du lien qui existe entre tableau et pointeur.

Les pointeurs sont fortement typés, ce qui signifie qu'on ne pourra pas implicitement placer par exemple dans un pointeur sur des objets de type `double`, l'adresse d'un entier. Cette interdiction sera toutefois pondérée par l'existence de « l'opérateur de cast » qui autorise, avec plus ou moins de risques, des conversions d'un pointeur d'un type donné en un pointeur d'un autre type. En outre, les « pointeurs génériques » permettent de véhiculer des adresses sans type.

Par ailleurs, on peut, dans une certaine mesure, effectuer des opérations arithmétiques sur des pointeurs : incrémentation, décrémentation, soustraction. Ces opérations vont plus loin qu'un simple calcul d'adresse, puisqu'elles prennent en compte la taille des objets pointés.

Enfin, il existe une corrélation très étroite entre la notion de tableau et celle de pointeur, un nom de tableau étant assimilable à un pointeur (constant) sur son premier élément. Ceci a de nombreuses conséquences :

- au niveau de l'opérateur d'indiçage `[]` qui pourra indifféremment s'appliquer à des tableaux ou à des pointeurs ;

- dans la transmission de tableau en argument d'une fonction.

Après une introduction générale à la notion de pointeur, nous examinerons en détail la déclaration des pointeurs. Nous étudierons ensuite l'arithmétique des pointeurs, ce qui permettra d'établir le lien avec la notion de tableau et de faire le point sur les opérateurs `&` et `[]`. Après la présentation de la valeur pointeur particulière `NULL`, nous aborderons l'affectation à des `lvalue` de type pointeur. Puis nous étudierons ce que l'on nomme les pointeurs génériques (type `void *`). Nous verrons enfin comment comparer des pointeurs et comment les convertir.

Signalons que C permet de manipuler des pointeurs sur des fonctions. Cet aspect sera étudié à la section 11 du chapitre 8. Il apparaîtra cependant, par souci d'exhaustivité, dans certains tableaux récapitulatifs de ce chapitre.

1. Introduction à la notion de pointeur

Considérons cette déclaration :

```
int *adr ;   /*    on peut aussi écrire :  int * adr;   */
```

Elle précise que `adr` est une variable de type pointeur sur des objets de type `int`.

1.1 Attribuer une valeur à une variable de type pointeur

Puisque la déclaration précédente de `adr` ne comporte pas d'initialiseur, la valeur de `adr` est, pour l'instant, indéfinie. Pour assigner une valeur à `adr` et, donc, la faire « pointer » sur un entier précis, il existe deux démarches de nature très différente.

La première consiste à affecter à `adr` l'adresse d'un objet ou d'une partie d'objet existant (à ce moment-là). Par exemple, si l'on suppose ces déclarations :

```
int n ;
int t[5] ;
```

on peut écrire :

```
adr = &n ;
```

ou :

```
adr = &t[2] ;
```

La seconde démarche consiste à utiliser des fonctions d'allocation dynamique pour créer un emplacement pour un objet du type voulu dont on affecte l'adresse à `adr`, par exemple (pour plus d'informations concernant ces possibilités, on se reportera au chapitre 14) :

```
adr = malloc (sizeof (int)) ;
```

Mais on pourra aussi :

- affecter une valeur d'une variable pointeur à une autre variable pointeur :

```
int *ad1, *ad2 ;
   .....
ad1 = ad2 ;       /* ad1 et ad2 pointent maintenant sur le même objet */
```

  Cette démarche apparaît comme une variante de l'une ou l'autre des précédentes, suivant la nature de l'objet pointé par `ad2` ;

- utiliser, comme nous le verrons à la section 3, les propriétés de l'arithmétique des pointeurs pour décrire différentes parties d'un même objet ou encore pour accéder à des objets voisins, pour peu qu'ils soient de même type.

1.2 L'opérateur * pour manipuler un objet pointé

En C, le pointeur peut bien sûr servir à manipuler simplement des adresses. Toutefois, son principal intérêt est de permettre d'appliquer à l'objet pointé des opérations comparables à celles qu'on applique à un objet contenu dans une variable. Ceci est possible grâce à l'existence de l'opérateur de déréférenciation noté `*`. Plus précisément, si `ad` est un pointeur, `*ad` désigne l'objet pointé par `ad`. Voyez ces instructions :

```
int * ad ;
int n = 20 ;
  .....
ad = &n ;            /* ad pointe sur l'entier n qui contient actuellement la valeur 20 */
printf ("%d", *ad) ; /* affiche la valeur 20 */
*ad = 30 ;           /* l'entier pointe par ad prend la valeur 30 */
printf ("%d", *ad) ; /* affiche la valeur 30 */
```

Bien sûr, ici, l'avant dernière instruction est équivalente à :

```
n = 30 ;
```

et elle n'a donc que peu d'intérêt dans le présent contexte. En revanche, elle peut en avoir dès lors que la valeur de `ad` n'est pas toujours la même, comme dans cet exemple :

```
if (...) ad = &n ;
    else ad = &p ;
   .....
*ad = 30 ;    /* place la valeur 30 dans n ou dans p suivant le cas */
```

ou encore dans celui-ci :

```
ad = malloc (sizeof (int)) ;
*ad = 30 ;    /* place la valeur 30 dans l'emplacement préalablement alloué */
```

Voici quelques autres exemples d'utilisation de l'opérateur * :

```
*ad += 3          /* équivaut à :      *ad = *ad + 3                  */
                  /* augmente de 3 la valeur de l'entier pointé par ad */

(*ad)++  ;        /* équivaut à :      *ad = *ad + 1                  */
                  /* augmente de 1 la valeur de l'entier pointé par ad */
                  /* attention aux parenthèses :                      */
                  /*    *ad++ serait équivalent à *(ad++)             */
```

Là encore, ces possibilités seront enrichies par les propriétés arithmétiques des pointeurs qui permettront d'accéder non seulement à l'objet pointé, mais à d'éventuels voisins.

Remarques

1. Il faut éviter de dire qu'une expression telle que `*ad` désigne la valeur de l'objet pointé par `ad`, pas plus qu'on ne dit, par exemple, que le nom d'une variable `n` désigne sa valeur. En effet, suivant le cas, la même notation peut désigner la valeur ou l'objet :

   ```
   int n, p ;
   int *ad ;
   n = p ;          /* ici, p désigne la valeur de p, n désigne la variable n */
   n = n + 12 ;     /* à gauche, n désigne la variable n, à droite, sa valeur */
   p = *ad + 3 ;    /* *ad désigne la valeur de l'entier pointé par ad        */
   *ad = p+5 ;      /* *ad désigne l'entier pointe par ad                     */
   ```

   Cette différence d'interprétation en fonction du contexte ne pose aucun problème pour les variables usuelles. En revanche, l'expérience montre qu'elle est souvent source de confusion dans le cas des objets pointés. En particulier, dans le cas de pointeurs de pointeurs, elle peut amener à oublier ou à ajouter une déréférenciation.
2. En général, une expression telle que `*ad` est une `lvalue` dans la mesure où elle est utilisable comme opérande de gauche d'une affectation. Il existe cependant une exception, à savoir lorsque l'objet pointé est constant. Nous y reviendrons en détail à la section suivante.

2. Déclaration des variables de type pointeur

Le tableau 7.1 récapitule les différents éléments intervenant dans la déclaration des pointeurs. Ils seront ensuite décrits de façon détaillée dans les sections indiquées.

Tableau 7.1 : déclaration des pointeurs

Type des éléments pointés	Objet de type quelconque (type de base, structures, unions, tableaux, pointeurs...) ou fonction.	Voir section 2.2
Déclarateur de pointeur	De la forme : `* [qualifieurs] declarateur`	– rappels sur les déclarations à la section 2.1 ; – description du déclarateur et exemples à la section 2.3.

Classe de mémorisation	Concerne la variable pointeur, pas l'objet pointé : – `extern` : pour les redéfinitions de pointeurs globaux ; – `auto` : pour les pointeurs globaux (superflu) ; – `static` : pointeur rémanent ; – `register` : demande d'attribution de registre.	– étude détaillée de la classe de mémorisation aux sections 8, 9 et 10 du chapitre 8 ; – discussion à la section 2.4.
Qualifieurs (`const`, `volatile`)	– peuvent s'appliquer à la fois à la variable pointeur et à l'objet pointé ; – un pointeur constant doit être initialisé, sauf s'il s'agit de la redéclaration d'une variable globale ou si l'objet pointé est volatile.	– définition des qualifieurs à la section 6.3 du chapitre 3 ; – discussion à la section 2.5.
Nom de type d'un pointeur	Utile pour `sizeof`, pour le prototype d'une fonction ayant un argument de type pointeur, pour l'opérateur de `cast`.	Voir section 2.6

2.1 Généralités

La déclaration d'une variable de type pointeur permet de préciser tout ou partie des informations suivantes :

- son nom, il s'agit d'un identificateur usuel ;
- le type des éléments pointés avec d'éventuels qualifieurs (`const`, `volatile`) destinés aux objets pointés ;
- éventuellement, une classe de mémorisation, destinée à la variable pointeur elle-même ;
- éventuellement, un qualifieur, destiné cette fois à la variable pointeur elle-même.

Cependant, la nature même des déclarations en C disperse ces différentes informations au sein d'une même instruction de déclaration. Par exemple, dans :

```
static const unsigned long p, *adr1, * const adr2 ;
```

- `adr1` est un pointeur sur des objets constants de type `unsigned long` ;
- `adr2` est un pointeur constant sur des objets constants de type `unsigned long`.

D'une manière générale, on peut dire qu'une déclaration en C associe un ou plusieurs déclarateurs (ici `p`, `*adr1`, `* const adr2`) à une première partie commune à tous ces déclarateurs et comportant effectivement :

- un spécificateur de type (ici, il s'agit de `unsigned long`) ;
- un éventuel qualifieur (ici, `const`) ;
- une éventuelle classe de mémorisation (ici, `static`).

Les déclarations en C peuvent devenir complexes compte tenu de ce que :

- un même spécificateur de type peut être associé à des déclarateurs de nature différente ;
- les déclarateurs peuvent se « composer » : il existe des déclarateurs de tableaux, de pointeurs et de fonctions ;

- la présence d'un déclarateur de type donné ne renseigne pas précisément sur la nature de l'objet déclaré ; par exemple, un tableau de pointeurs comportera, entre autres, un déclarateur de pointeur ; ce ne sera pas pour autant un pointeur.

Pour tenir compte de cette complexité et de ces dépendances mutuelles, le chapitre 16 fait le point sur la syntaxe des déclarations, la manière de les interpréter et de les rédiger. Ici, nous nous contenterons d'examiner, de manière moins formelle, des déclarations correspondant aux situations les plus usuelles.

2.2 Le type des objets désignés par un pointeur

Le langage C est très souple puisqu'il permet de définir des pointeurs sur n'importe quel type d'objet, aussi complexe soit-il, ainsi que des pointeurs sur des fonctions. Comme certains types sont eux-mêmes construits à partir d'autres types, et ce d'une façon éventuellement récursive, on voit qu'on peut créer des types relativement complexes, même si ces derniers ne sont pas toujours indispensables.

En dehors des pointeurs sur des objets d'un type de base, le C fait largement appel à des pointeurs sur des structures ; l'une des raisons est qu'ils accélèrent l'échange d'informations de ce type avec des fonctions, en évitant d'avoir à les recopier. On en trouvera des exemples d'utilisation aux chapitres 11 et 14. En ce qui concerne les autres agrégats que sont les tableaux, les pointeurs sont moins utilisés pour la principale raison qu'un nom de tableau est déjà un pointeur. Ils peuvent cependant intervenir dans le cas de tableaux à plusieurs indices. Ppar exemple, on peut être amené à créer un tableau de pointeurs sur les lignes d'un tableau à deux indices. On en rencontrera un exemple à la section 7.2 du chapitre 8. Enfin, on peut avoir besoin de disposer de pointeurs sur des pointeurs. On en trouvera un exemple dans la gestion de liste chaînée présentée au chapitre 14.

2.3 Déclarateur de pointeur

Comme indiqué au paragraphe 2.1, le type d'un pointeur (donc celui des objets ou des fonctions pointées) est défini par une déclaration qui associe un déclarateur à un spécificateur de type, éventuellement complété par des qualifieurs.

Ce déclarateur, quelle que soit sa complexité, est toujours de la forme suivante :

Déclarateur de forme pointeur

* [qualifieurs] declarateur		
`declarateur`	Déclarateur quelconque	

`qualifieurs`	`const`
	`volatile`
	`const volatile`
	`volatile const`
		– attention, ce qualifieur s'applique au pointeur ; il ne doit pas être confondu avec celui qui accompagne éventuellement le spécificateur de type et qui s'applique, quant à lui, aux objets pointés ;
		– discussion à la section 2.5.

N.B : les crochets signifient que leur contenu est facultatif.

Notez que nous parlons de « déclarateur de forme pointeur » plutôt que « déclarateur de pointeur », car la présence d'un tel déclarateur de pointeur ne signifie pas que l'identificateur correspondant est un pointeur. Elle prouve simplement que la définition de son type fait intervenir un pointeur. Par exemple, il pourra s'agir d'un tableau de pointeurs, comme nous le verrons dans les exemples ci-après.

Par ailleurs, on notera bien que la récursivité de la notion de déclarateur fait que le déclarateur mentionné dans cette définition peut être éventuellement complexe, même si, dans les situations les plus simples, il se réduit à un identificateur.

Exemples

Voici des exemples de déclarateurs que, par souci de clarté, nous avons introduit dans des déclarations complètes. Lorsque cela est utile, nous indiquons en regard les règles utilisées pour l'interprétation de la déclaration, telles que vous les retrouverez à la section 4 du chapitre 16. La dernière partie constitue un contre-exemple montrant que la présence d'un déclarateur de pointeur ne correspond pas nécessairement à la déclaration d'un pointeur. Ici, aucune de ces déclarations ne comporte de qualifieur. Vous trouverez de tels exemples à la section 2.5.

Cas simples : pointeurs sur des objets d'un type de base, structure, union ou défini par typedef

Les déclarations sont faciles à interpréter lorsque le déclarateur concerné correspond à un simple identificateur :

`unsigned int n, *ptr, q, *ad;`	– `ad` et `ptr` sont des pointeurs sur des éléments de type `unsigned int` ; – notez qu'il est nécessaire de répéter le symbole * dans chaque déclarateur de pointeur ; ici `n` et `q` sont de simples variables de type `unsigned int`.

`struct point { char nom;`
`  int x;`
`  int y;`
`  };`
`struct point *ads, s;`
	– `ads` est un pointeur sur des structures de type `struct point` ;
	– `s` est de type `struct point`.

`union u { float x;`
`  char z[4];`
`  };`
`union u *adu;`
	`adu` est un pointeur sur des unions de type `union u`.

`typedef  int vecteur [3];` `vecteur  *adv;`	`vecteur` est synonyme de `int [3]`. `adv` est un pointeur sur des tableaux de 3 `int`.

Pointeur de pointeur

On peut naturellement composer deux fois de suite le déclarateur de pointeur :

`float x, *adf, * *adadf;`	(pour mémoire : `x` est un `float`, `adf` est un pointeur sur un `float`) `* *adadf` est un `float` → `*adadf` est un pointeur sur un `float` → `adadf` est un pointeur sur un pointeur sur un `float`

`int (* *chose)[10];`	`(* *chose)[10]` est un `int`
	`(* *chose)` est un tableau de 10 `int`
	`* *chose` est un tableau de 10 `int`
	`*chose` est un pointeur sur un tableau de 10 `int`
	`chose` est un pointeur sur un pointeur sur un tableau de 10 `int`
	Notez la présence des parenthèses ; en leur absence, la signification serait différente (voir plus loin).

Pointeurs sur des tableaux

Les déclarateurs peuvent se composer à volonté. On peut ainsi composer un déclarateur de tableau (voir chapitre 6) et un déclarateur de pointeur. Cependant, il faut alors tenir compte de l'ordre dans lequel se fait la composition des déclarateurs et le recours aux parenthèses est indispensable :

`int n, *ad, (*chose)[10];`	`(*chose)[10]` est un `int` → `(*chose)` est un tableau de 10 `int` → `*chose` est un tableau de 10 `int` (on s'est contenté de supprimer les parenthèses) → `chose` est un pointeur sur un tableau de 10 `int`

Un déclarateur de forme pointeur ne correspond pas toujours à un pointeur

`int *chose [10];`	`*chose[10]` est un `int` → `chose[10]` est un pointeur sur un `int` (on doit interpréter le déclarateur pointeur en premier) → `chose` est un tableau de 10 pointeurs sur un `int`
`int * *chose[10];`	`**chose[10]` est un `int` → `*chose[10]` est un pointeur sur un `int` (on doit interpréter le déclarateur pointeur en premier) → `chose[10]` est un pointeur sur un pointeur sur un `int` → `chose` est un tableau de 10 pointeurs sur un pointeur sur un `int`
`int * f(float);`	`*f (float)` est un `int` → `f(float)` est un pointeur sur un `int` (on doit interpréter le déclarateur pointeur en premier) → `f` est une fonction recevant un `float` et renvoyant un pointeur sur un `int`

2.4 Classe de mémorisation associée à la déclaration d'un pointeur

Comme toute déclaration, une déclaration faisant intervenir un déclarateur pointeur peut commencer par un mot-clé dit « classe de mémorisation ». Ce dernier peut être choisi parmi : `extern`, `static`, `auto` ou `register`. Il faut bien noter que ce mot-clé est associé à tous les déclarateurs d'une même déclaration, comme dans :

```
static int n, *ad, t[10] ;
```

Par ailleurs, dans les rares cas où ce mot-clé est présent, il sert généralement à modifier la classe d'allocation de la variable correspondante, mais ce n'est pas toujours le cas. En particulier, l'application du mot-clé `static` à une variable globale la « cache » à l'intérieur d'un fichier source, tout en la laissant de classe statique.

D'une manière générale, le rôle et la signification de ces différents mots-clés dans les divers contextes possibles sont étudiés en détail aux sections 8, 9 et 10 du chapitre 8. Ici, nous nous contentons d'apporter une précision relative aux pointeurs, à savoir que :

La classe de mémorisation concerne la variable pointeur elle-même, non les objets pointés.

Ainsi, dans la déclaration précédente, si la variable `ad` est locale à une fonction, elle sera de classe d'allocation `static`. En revanche, aucune restriction ne pèse sur les entiers pointés par `ad` qui pourront être tantôt de classe automatique, tantôt de classe dynamique[1]. Cette dernière possibilité présente d'ailleurs le risque de voir une variable pointer sur un objet dont l'emplacement mémoire a été désalloué.

2.5 Les qualifieurs const et volatile

La signification générale des qualifieurs `const`, `volatile`, `const volatile` ou `volatile const` a été présentée à la section 6.3 du chapitre 3. Ici, nous apportons quelques précisions concernant les pointeurs, car ces qualifieurs peuvent intervenir à deux niveaux différents :

- à un niveau collectif, c'est-à-dire associé au spécificateur de type et concernant donc tous les déclarateurs, qu'il s'agisse ou non de pointeurs ; dans le cas des pointeurs, il concerne alors l'objet pointé et non la variable pointeur ;
- au niveau d'un déclarateur pointeur ; dans ce cas, il concerne alors la variable pointeur et non l'objet pointé.

Rappelons que `const` est, de très loin, le qualifieur le plus utilisé et que son rapprochement avec `volatile` n'est qu'une pure affaire de syntaxe.

2.5.1 Qualifieur utilisé à un niveau collectif

Le qualifieur const

Avec la déclaration :

1. Ils ne pourront cependant pas être de classe `registre`, car un objet placé dans un registre ne possède pas d'adresse.

```
const int n, *p ;
```

le qualifieur `const` s'applique à la fois à `n` et à `*p`. Dans le second cas, il s'interprète ainsi :

- `*p` est un `int` constant ;
- donc `p` est un pointeur sur un `int` constant.

Cela signifie notamment que la modification de l'objet pointé par `p` est interdite :

```
*p = ... ;   /* interdit */
```

En revanche, la modification de `p` reste permise :

```
p = ... ;    /* OK */
```

Ces règles s'appliquent également aux pointeurs constants reçus en argument. L'usage de `const`, dans ce cas permet une certaine protection dans l'écriture de la fonction correspondante :

```
void fct (const int * adr)
{ ...  /* ici, la modification de *adr est interdite            */
}
```

La protection d'un objet constant pointé n'est cependant pas plus absolue que ne l'est celle d'un objet constant en général. Hormis dans le cas des machines implémentant les objets constants dans une zone protégée (et qui déclenchent alors une erreur d'exécution en cas de tentative de modification), il reste toujours possible de transmettre l'adresse d'un tel objet (ici `p`), à une fonction qui modifie l'objet qui s'y trouve, ne serait-ce que `scanf` :

```
scanf ("%d", p) ;    /* toujours accepté car le compilateur n'a aucune */
                     /* connaissance du rôle de scanf                  */
```

Cette dernière remarque plaide d'ailleurs en faveur de l'usage des prototypes, dès que cela est possible. Ainsi, dans l'exemple suivant, on voit qu'il n'est pas possible de transmettre l'adresse d'un objet constant à une fonction attendant une adresse d'un objet non constant :

```
void f (int * adr ) ; /* f attend un pointeur sur un int qu'elle peut modifier */
   .....
int * adi ;
const int * adci ;
f (adi) ;     /* OK : adi est du type attendu */
f (adci) ;    /* incorrect : la conversion de const int * en int *   */
              /* n'est pas autorisée par affectation                 */
```

Le qualifieur volatile

Le qualifieur `volatile` peut apparaître de la même façon que `const`. Dans :

```
volatile int n, *p ;
```

le qualifieur `volatile` s'applique à la fois à `n` et à `p`. Dans le second cas, il signifie que `p` est un pointeur sur un entier volatile, autrement dit que la valeur d'un tel objet peut être modifiée,

indépendamment des instructions du programme. Par exemple, en présence d'une construction comme la suivante, dans laquelle `n` est supposée être une variable de type `int` :

```
while (...)
 { .....
   n = *adr ;
   .....
 }
```

l'usage de `volatile` devrait interdire à un compilateur de « sortir » l'affectation de la boucle `while`. En pratique, même en l'absence du qualifieur `volatile`, il est peu probable qu'un compilateur réalise ce genre d'optimisation, dans la mesure où il lui est difficile d'être sûr que l'objet pointé n'est pas éventuellement modifié par le biais d'un autre pointeur...

2.5.2 Qualifieur associé au déclarateur

Dans ce cas, le qualifieur concerne la variable pointeur elle-même et non plus l'objet pointé.

Le qualifieur const

Avec la déclaration :

```
int n, * const p ;          /* en général, p devra être initialisé */
```

- `* const p` est un `int` ;
- `const p` est un pointeur sur un `int` ;
- `p` est un pointeur constant sur un `int`.

Cela signifie, de façon usuelle cette fois, que la modification de la valeur de `p` est interdite :

```
p = ... ;   /* interdit */
```

En revanche, la modification de l'objet pointé par `p` reste permise :

```
*p = ... ;    /* OK */
```

Remarque

La plupart du temps, une variable pointeur ayant reçu le qualifieur `const` devra, comme n'importe quelle autre variable constante, être initialisée lors de sa déclaration. On utilisera une expression constante de type adresse telle qu'elle a été définie à la section 14 du chapitre 4, comme dans :

```
float x ;
float * const adf1 = &x ;     /* initialisation de adf1 avec l'adresse de x  */
float * const adf2 = NULL ;   /* initialisation de adf2 à une valeur nulle   */
                              /*        (voir &5)                            */
```

Il existe cependant des exceptions :

- La variable possède en plus le qualifieur `volatile` : elle pourra donc être modifiée indépendamment du programme. Son initialisation n'est donc pas indispensable mais elle reste possible.
- Il s'agit de la redéclaration d'une variable globale (par `extern`). L'initialisation a dû être faite par ailleurs, l'initialisation est alors interdite à ce niveau.

Le qualifieur volatile

De même, avec cette déclaration :

```
int n, * volatile p ;
```

`p` est un pointeur volatile sur un `int`.

> **Remarque**
>
> La place dans la déclaration d'un qualifieur destiné à un identificateur de type pointeur n'est pas la même que pour les autres identificateurs, puisqu'il est alors associé au déclarateur et non plus à l'ensemble de la déclaration :
>
> ```
> const int n ;           /* n est constant */
> const float t[12] ;     /* t (donc ses éléments) est constant */
> int * const p ;         /* p est constant */
> ```

2.5.3 Utilisation des deux sortes de qualifieurs

Bien entendu, rien n'empêche de faire appel à une déclaration telle que :

```
const int n, * const p ;       /* en général, p devra être initialisé */
```

Dans ce cas, hormis le fait que `n` est un entier constant :

- `* const p` est un `int` constant ;
- `const p` est un pointeur sur un `int` constant ;
- `p` est un pointeur constant sur un `int` constant.

Cette fois, ni `p`, ni l'objet pointé par `p` ne peuvent être modifiés ;

```
p  = ...   /* interdit */
*p = ...   /* interdit */
```

Bien entendu, on peut combiner à loisir `const` et `volatile` (voire les deux), ce qui conduit théoriquement à 16 combinaisons différentes (en comptant l'absence de qualifieur)[2] ! Mais seules celles utilisant `const` sont vraiment employées.

2.5.4 Cas des pointeurs de pointeurs

Dans le cas de pointeurs de pointeurs, on aura affaire à trois sortes de qualifieurs. Dans le cas de pointeurs de pointeurs de pointeurs, on aura affaire à quatre sortes de qualifieurs et ainsi de suite. Par exemple, avec :

```
const int n, *const ad1, *const *const ad2, **const ad3 ;
```

2. Et à 256 dans le cas des pointeurs de pointeurs, etc.

- `n` est un `int` constant ;
- `ad1` est un pointeur constant sur un `int` constant ;
- `ad2` est un pointeur constant sur un pointeur constant sur un `int` constant ;
- `ad3` est un pointeur constant sur un pointeur (non constant) sur un `int` constant : `ad3` n'est pas modifiable, pas plus que `**ad3` ; en revanche, `*ad3` est modifiable.

2.6 Nom de type correspondant à un pointeur

Dans le cas des pointeurs, il est nécessaire de disposer de ce que l'on nomme le « nom de type », dans les trois cas suivants :

- comme opérande de l'opérateur de `cast` ;
- comme opérande de l'opérateur `sizeof`, lorsqu'on l'applique à un type pointeur (et non à une variable de type pointeur) ;
- dans un prototype de fonction.

Dans le cas des types de base (voir section 8 du chapitre 4), ce nom de type n'est rien d'autre que le spécificateur de type, éventuellement précédé de qualifieurs. Par exemple, avec :

```
unsigned int p ;
const float x ;
```

le nom de type de la variable `p` est tout simplement `unsigned int` ; celui de `x` est `const float`. Toutefois, le qualifieur ne joue aucun rôle dans ce cas et quel que soit l'usage qu'on fasse du nom de type de `x` (`cast`, `sizeof`, argument), on peut indifféremment utiliser `unsigned int` ou `const unsigned int`.

Dans le cas d'un pointeur, les choses sont moins simples. Par exemple, avec :

```
const unsigned long q, * const ad ;
```

le nom de type de `ad` s'obtient en juxtaposant les éventuels qualifieurs (ici `const`), le spécificateur de type (ici `unsigned long`) et le déclarateur (ici `* const ad`) privé de l'identificateur correspondant (soit, ici, `* const`). Le nom de type correspondant au pointeur `ad` sera donc, en définitive :

```
const unsigned long * const
```

Là encore, le qualifieur associé au déclarateur ne joue aucun rôle, pour les mêmes raisons que le qualifieur associé à une variable d'un type de base[3] ne jouait aucun rôle dans son nom de type (il n'intervient que dans les opérations applicables à une `lvalue`), de sorte que ce nom de type peut aussi s'écrire :

```
const unsigned long *   /* équivalent à  const unsigned long * const      */
```

En revanche, le qualifieur associé au spécificateur de type fait bien partie intégrante du type, donc aussi du nom de type.

3. Attention aux emplacements des qualifieurs (voir la remarque de la section 2.5.2.

Remarques

1. Dans le cas de pointeurs de pointeurs, seul le dernier qualifieur pourra être omis, sans que cela ne modifie le type. Par exemple, avec cette déclaration, dans laquelle `adadf` est un pointeur constant sur un pointeur constant sur un `float` constant :

   ```
   const float x, * const * const adadf ;
   ```

   Le nom de type de `adadf` est `const float * const * const`, mais on pourra tout aussi bien utiliser `const float * const *`.
2. Lorsque le nom de type est utilisé comme opérande de `sizeof`, les qualifieurs, quel que soit leur niveau, n'ont aucun rôle et ils peuvent être omis.

3. Les propriétés des pointeurs

Non seulement le langage C autorise la manipulation de pointeurs, mais il permet également d'effectuer des calculs basés sur des pointeurs. Plus précisément, on peut ajouter ou soustraire un entier à un pointeur ou faire la différence de deux pointeurs. Dans ces calculs, on adopte alors comme unité, non pas l'octet, mais la taille des objets pointés. On traduit souvent cela en parlant des propriétés arithmétiques des pointeurs. Par ailleurs, il existe un lien très étroit entre l'opérateur `[]` et les pointeurs. Ce sont ces différents aspects que nous étudions ici. Ils nous permettront, à la section suivante, de faire le point sur tout ce qui concerne les opérateurs faisant intervenir des pointeurs : `+`, `-`, `[]`, `&` et `*`.

Tableau 7.2 : les propriétés des pointeurs

Propriétés arithmétiques	– unité utilisée : taille de l'objet pointé ; – on peut ajouter ou soustraire un entier à un pointeur ; les qualifieurs de l'objet pointé sont répercutés sur l'expression ; – on peut soustraire deux pointeurs ; – l'ordre des pointeurs ne coïncide pas obligatoirement avec celui des adresses.	Voir section 3.1 Voir section 3.3
Lien pointeur tableau	– une référence à un tableau est convertie en un pointeur constant sur son premier élément (excepté avec `&` ou `sizeof`) ; – l'opérateur `[]` reçoit un opérande pointeur et un opérande entier (ordre indifférent) et : `exp1[exp2] ó * (exp1 + exp2)`	Voir section 3.2
Restrictions	Dans une expression faisant intervenir des pointeurs, les différents objets pointés doivent pouvoir être considérés comme éléments d'un même tableau (l'un des deux pouvant être situé juste au-delà de la fin).	Voir section 3.4

3.1 Les propriétés arithmétiques des pointeurs

3.1.1 Addition ou soustraction d'un entier à un pointeur

Si une variable `ad` a été déclarée ainsi :

```
int *ad ;
```

une expression telle que :

```
ad + 1
```

a un sens en C. Elle est de même type que `ad` (ici, pointeur sur `int`) et sa valeur est celle de l'adresse de l'entier suivant l'entier pointé actuellement par `ad`.

On voit donc que la notion de pointeur diffère de celle d'adresse. En effet, la différence entre les adresses correspondant à `ad` et `ad + 1` est de `sizeof (int)` octets. Si `ad` était déclaré :

```
double *ad ;
```

la différence entre `ad` et `ad + 1` serait de `sizeof(double)` octets. Une autre différence, plus subtile, apparaîtra au niveau du sens dans lequel se fait la progression, lequel peut être différent de celui de la progression des adresses. Nous y reviendrons à la section 3.3.

D'une manière comparable, avec la déclaration :

```
int *ad ;
```

l'expression :

```
ad + 3
```

pointerait sur un emplacement situé « 3 entiers plus loin » que celui pointé actuellement par `ad`.

Enfin, une expression telle que :

```
ad++
```

incrémente l'adresse contenue dans `ad`, de façon qu'elle désigne l'entier suivant.

3.1.2 Qualifieurs et expressions pointeur

Les éventuels qualifieurs relatifs à l'objet pointé sont répercutés dans le type de l'expression. Par exemple, avec :

```
const int * ad ;
```

une expression telle que `ad + 3` est bien de type `const int *`, et non seulement de type `int *`. En particulier, l'affectation suivante resterait interdite :

```
*(ad+3) = ...  /* interdit :   ad + 3 est de type const int *    */
```

Cette remarque s'avérera particulièrement précieuse dans le cas de pointeurs sur des tableaux et notamment dans le cas des pointeurs sur des chaînes de caractères.

En revanche, les qualifieurs relatifs à la variable pointeur elle-même n'ont aucune influence sur le type de l'expression. Par exemple, avec :

```
int * const adci ;     /* adci est un pointeur constant sur un int */
```

`adci+3` est une expression de type pointeur sur un `int`, et non pointeur constant sur un `int`. Notez qu'il en va exactement de même pour une variable usuelle. Par exemple, avec :

```
const int n = 5 ;
```

la notion de constance de l'expression `n+3` n'aurait aucun sens.

3.1.3 Soustraction de deux pointeurs

Il est possible de soustraire les valeurs de deux pointeurs de même type. La signification d'une telle opération découle de celle de l'addition d'un pointeur et d'un entier présentée précédemment.

Le résultat est un entier correspondant au nombre d'éléments (du type commun) situés entre les deux adresses en question. Par exemple, avec :

```
int t[10] ;
int * ad1 = &t[3] ;
int * ad2 = &t[8] ;
```

l'expression :

```
ad2 - ad1
```

a pour valeur 5.

En fait, on peut dire que cette soustraction n'est rien d'autre que l'opération inverse de l'incrémentation : si `p1` et `p2` sont des pointeurs de même type tels que l'on ait `p2 = p1+ i`, alors `p2 - p1 = i` ou encore `p1 - p2 = -i`.

3.2 Lien entre pointeurs et tableaux

Soit ces déclarations :

```
int t[10] ;
int *adr = ...     /* on suppose que adr pointe sur une suite */
                   /*    d'objets de type int                 */
```

En général, on accède aux différents éléments de `t` par des expressions de la forme `t[i]` et aux objets pointés par `adr` par des expressions de la forme `*adr` ou `*(adr+i)`.

Cependant, les concepteurs du C ont fait en sorte que :

- une notation telle que `*(t+i)` soit totalement équivalente à `t[i]` ;
- une notation telle que `adr[i]` soit totalement équivalente à `*(adr+i)`.

Cette équivalence est en fait la conséquence de deux choix fondamentaux concernant d'une part les noms de tableaux (et même, plus généralement, les références à des tableaux, c'est-à-dire les

expressions de type tableau) et d'autre part, l'opérateur `[]`. Ce sont ces choix que nous allons étudier maintenant en détail.

3.2.1 Équivalence entre référence à un tableau et pointeur

Cette équivalence se manifeste par une règle générale comportant deux exceptions.

Règle générale

Lorsqu'un nom de tableau apparaît dans un programme, il est converti en un pointeur constant sur son premier élément. Ainsi, avec :

```
int t[10] ;
```

lorsque `t` apparaît dans une instruction, il est remplacé par l'adresse de `t[0]`. Par exemple, cette instruction serait correcte (même si elle est rarement utilisée ainsi) :

```
scanf ("%d", t) ;    /* équivaut à scanf ("%d", &t[0]);    */
```

En outre, `t` étant un pointeur (constant) de type `int *`, des expressions telles que celles-ci ont un sens :

```
t + 1      /* équivaut à &t[1]  */
t + i      /* équivaut à &t[i]  */
```

Voici un exemple d'instructions utilisant ces remarques :

```
int * adr ;
int t[10] ;
  .....

adr = t + 3 ;     /* adr pointe sur le troisième élément de t */
*adr = 50 ;       /* équivaut à :    t[3] = 50;               */
                  /*  ou encore à :    *(t+3) = 50;           */
```

D'une manière générale, ce sont non seulement les noms de tableaux qui sont convertis en un pointeur, mais également toutes les références à un tableau. Ceci aura surtout des conséquences dans le cas des tableaux à plusieurs indices que nous examinerons en détail à la section 3.2.4. Signalons que les tableaux apparaissant en argument d'une fonction seront soumis à cette règle (voir section 6 du chapitre 8).

Les exceptions à la règle

Il existe deux exceptions naturelles à la règle précédente qui demande de remplacer une référence à un tableau par un pointeur sur son premier élément : lorsque le nom de tableau apparaît comme opérande de l'un des deux opérateurs `sizeof` et `&`.

Avec `sizeof` (voir section 3.4 du chapitre 6), l'opérateur `sizeof` appliqué à un nom de tableau fournit bien la taille du tableau, non pas celle du pointeur, ce qui est, somme toute, plus satisfaisant !

Avec `&` : si `t` est un tableau d'éléments de type `T`, l'expression `&t` fournit bien l'adresse du premier élément de `t`, non l'adresse de cette adresse, ce qui n'aurait aucun sens. Elle est du type « pointeur sur T ».

Remarque

Les qualifieurs des éléments du tableau se retrouvent dans le type du pointeur correspondant à son nom. Par exemple, avec :

```
int t1[10] ;
const int t2[10] ;
```

`t1` est du type `int *`, tandis que `t2` est du type `const int *`. Comme en réalité, `t1` et `t2` sont des pointeurs constants, on pourrait dire également que `t1` est du type `int const *`, tandis que `t2` est du type `const int const *`, ce qui ne change rien au type effectif.

En résumé

Toute référence à un tableau d'objets de type `T`, à l'exception de l'opérande de `sizeof` ou de `&`, est convertie en un pointeur constant (du type pointeur sur T) sur le premier élément du tableau.

3.2.2 L'opérateur []

Pour assurer l'équivalence entre `t[i]` et `*(t+i)` annoncée en introduction, les concepteurs du C ont prévu que l'accès par indice à un élément de tableau s'effectue en fait à l'aide d'un opérateur binaire noté `[]` recevant :

- un opérande de type pointeur (un identificateur de tableau adéquat, puisque remplacé par un pointeur...) ;
- un opérande de type entier.

Il fournit comme résultat la référence de l'objet ayant comme adresse la valeur de la somme des deux opérandes (somme d'un pointeur et d'un entier).

Par exemple, avec :

```
int t[10] ;
```

`t[i]` est en fait interprété comme suit[4] :

- `t` est converti en un pointeur sur `t[0]` ;
- on ajoute `i` au résultat ;
- on considère l'objet ainsi pointé ;
- `t[i]` est donc bien équivalent à `*(t+i)`.

De même, avec :

```
int * adr ;
```

4. Attention, on ne doit pas considérer l'opérateur `[]` comme formé d'un seul symbole, mais bien de deux symboles différents `[` et `]`, entre lesquels vient se glisser le second opérande.

`adr[i]` est en fait interprété comme suit :

- on ajoute `i` à la valeur de `adr` (notez qu'il n'y a plus de conversion de tableau en pointeur cette fois) ;
- on considère l'objet ainsi pointé ;
- `adr[i]` est donc bien, là encore, équivalent à `*(t+i)`.

Remarques

1. L'ordre des deux opérandes de l'opérateur `[]` est indifférent. Ainsi, dès lors que la notation `t[i]` est légale, la notation `i[t]` l'est aussi et elle représente la même chose. En pratique, la seconde forme est très rarement utilisée, ne serait-ce qu'à cause de son aspect inhabituel et déroutant.
2. Le résultat de l'opérateur `[]` n'est pas toujours une `lvalue`. En effet, l'objet correspondant peut ne pas être une `lvalue`. L'exemple le plus typique étant (en dehors des tableaux constants) le cas où ces objets sont eux-mêmes des tableaux. Nous en verrons des exemples à la section 3.2.4.

En résumé

L'opérateur `[]` reçoit deux opérandes (dans un ordre indifférent), l'un de type pointeur sur des objets, l'autre de type entier. Son résultat est l'objet pointé par la somme de ses deux opérandes, de sorte qu'il y a équivalence entre `exp1[exp2]` et `*(exp1+exp2)`.

3.2.3 Application à des tableaux à un indice

Examinons maintenant en détail toutes les conséquences des règles précédentes, en supposant que `t` est un tableau d'éléments de type `T`, `T` étant un type quelconque, autre que tableau : il peut donc s'agir d'un type de base, d'un type structure ou union, ces agrégats pouvant éventuellement comporter des tableaux. En fait, tout cela revient à dire qu'on se place dans le cas où les éléments de `t` sont des `lvalue`. Quant au cas des tableaux de tableaux ou tableaux à plusieurs indices, il sera examiné à la section suivante.

Dans la suite du programme (aux deux exceptions près évoquées précédemment), la notation `t` est donc convertie en un pointeur sur le premier élément de `t`. Autrement dit, elle est identique à `&t[0]`[5] et elle est de type « pointeur sur T ».

Voici, sur une même ligne, quelques notations équivalentes, même si la première paraît plus naturelle :

```
&t[0]          t               &0[t]       /* type pointeur sur T */
&t[i]          t + i           &i[t]       /* type pointeur sur T */
t[1]           *(t + 1)        1[t]        /* type T              */
t[i]           *(t + i)        i[t]        /* type T              */
```

5. En fait, nous utilisons cette notation parce que son interprétation intuitive ne pose pas de problème. En toute rigueur, si l'on s'en tient aux règles précédentes, cette dernière s'interprète comme : l'adresse de l'élément ayant comme adresse celle de `t`, augmentée de 0...

Rappelons que cette équivalence entre notation pointeur et notation indicielle reste valable lorsque l'on a affaire à un banal pointeur et non plus à un nom de tableau. Ainsi, si `p` est une variable de type « pointeur sur T », les notations suivantes sont équivalentes, même si la première paraît parfois la plus naturelle dans ce nouveau contexte :

```
p              &p[0]          &0[p]         /* type pointeur sur T */
p + i          &p[i]          &i[p]         /* type pointeur sur T */
*(p + 1)       p[1]           1[p]          /* type T              */
*(p + i)       p[i]           i[p]          /* type T              */
```

Bien entendu, cela ne préjuge pas de l'existence effective des objets pointés, qui ne faisait aucun doute dans le cas précédent puisque `t` avait fait l'objet d'une allocation mémoire. Ici, on doit supposer qu'une telle allocation a bien été faite, indépendamment de la déclaration du pointeur `p`.

Par ailleurs, on ne perdra pas de vue que, bien que `t` et `p` soient de même type, `t` est une constante alors que `p` est une variable. En particulier, `t` n'est pas une `lvalue`, tandis que `p` en est une. En revanche, les expressions `*(t+i)` et `*(p+i)` sont bien toutes les deux des `lvalue`, du moins tant que `t` n'est pas un tableau constant et que `p` n'est pas un pointeur sur des objets constants.

Exemple

Pour illustrer ces différentes possibilités de notation, voici plusieurs façons de placer la valeur 1 dans chacun des 10 éléments d'un tableau `t` de 10 entiers :

```
int t[10], i ;
for (i=0 ; i<10 ; i++)
  t[i] = 1 ;

int t[10], i ;
for (i=0 ; i<10 ; i++)
  *(t+i) = 1 ;

int t[10], i ;
int *ad :    /* pointeur courant */
for (ad=t, i=0 ; i<10 ; i++, ad++)
  *ad = 1 ;
```

Dans la troisième façon, nous avons dû recopier la « valeur » représentée par `t` dans un pointeur nommé `ad`. En effet, il ne faut pas perdre de vue que le symbole `t` représente une adresse constante (`t` est une constante de type pointeur sur des `int`). Autrement dit, une expression telle que `t++` aurait été invalide, au même titre que, par exemple, `3++`. Un nom de tableau est un pointeur constant, ce n'est pas une `lvalue`.

En revanche, ce problème ne se poserait plus si l'on travaillait avec une variable pointeur `p`. On pourrait incrémenter directement la valeur de `p` (ce qui ne serait pas nécessairement judicieux, dans la mesure où la valeur initiale serait perdue) :

```
int * p ;     /* p est l'adresse d'un entier supposé appartenir à un tableau */
int i ;
for (i=0 ; i<10 ; i++, p++)
  *p = 1 ;
```

3.2.4 Application à des tableaux à deux indices

Supposons, cette fois, que `t` est un tableau à deux indices d'éléments de type `T`, `T` étant un type quelconque autre que tableau. Pour fixer les idées, nous admettrons que le déclarateur de `t` est de la forme :

```
t[3][4]
```

Là encore, dans la suite du programme (aux deux exceptions près évoquées précédemment), la notation `t` est donc convertie en un pointeur sur le premier élément de `t`. Autrement dit, elle est identique à `&t[0]` et elle est du type « pointeur sur des tableaux de 4 éléments de type `T` ». Une expression telle que `t+1` est évaluée en incrémentant cette adresse d'une taille égale à celle d'un tableau de 4 `T`.

Ces deux points n'apportent rien de plus à ce qui a été dit précédemment sur des tableaux à un indice. En revanche, des nouveautés apparaissent si l'on considère une expression telle que :

```
t[1]
```

Certes, son évaluation se passe comme auparavant. Autrement dit :

- `t` est converti en un pointeur sur un tableau de `4 T` ;
- on ajoute 1 au résultat, ce qui fournit un pointeur sur le deuxième élément de `t` ; son type est « pointeur sur des tableaux de 4 `T` ».

Comme précédemment donc, `t[1]` désigne bien le deuxième élément du tableau `t`. Mais cette fois, il s'agit à nouveau d'une référence à tableau. Ce résultat est donc à son tour converti en un pointeur sur le premier élément, c'est-à-dire en une valeur de type « pointeur sur `T` » pointant sur l'élément `t[1][0]`. C'est précisément l'aspect nouveau par rapport au cas des tableaux à un indice : on obtient un résultat de type « pointeur sur `T` », alors que l'on s'attendait (peut-être) à un résultat de type « pointeur sur des tableaux de 4 `T` ».

D'une manière semblable, l'expression :

```
*(t+1)
```

représente, elle aussi, une référence à un tableau (dont les éléments sont de type `T`). Elle est donc convertie en un pointeur sur son premier élément.

En définitive, les notations `t[1]` et `*(t+1)` sont donc toujours équivalentes, comme elles l'étaient dans le cas des tableaux à un indice, mais leur type n'est plus celui qu'on attendait. Qui plus est, ce ne sont plus des `lvalue` puisqu'il s'agit de pointeurs constants.

Par ailleurs, comme `t[1]` est l'adresse de `t[1][0]`, il est clair que `*t[1]` est équivalent à `t[1][0]`. En outre, comme `*(t+1)` est équivalent à `t[1]`, `**(t+1)` est équivalent à `*t[1]`, c'est-à-dire finalement à `t[1][0]`. Ces expressions restent bien des `lvalue`, tant que que `t` n'a pas été déclaré constant.

Enfin, et fort heureusement, une expression telle que :

```
t[i][j]
```

s'interprète comme `(t[i])[j]`, c'est-à-dire, au bout du compte, comme :

```
*( *(t+i) + j)
```

En résumé, voici, sur une même ligne, quelques notations équivalentes (compte tenu de ce que l'ordre des opérandes de `[]` est quelconque, lorsque cet opérateur apparaît deux fois, il existe quatre façons différentes de les combiner. Manifestement, seule la notation usuelle est compréhensible !) :

```
&t[0]    t          &0[t]                  /* pointeur sur des tableaux de 4T */
&t[i]    t + i      &i[t]                  /* pointeur sur des tableaux de 4T */
t[1]     *(t + 1)   1[t]                                  /* pointeur sur T */
t[i]     *(t + i)   i[t]                                  /* pointeur sur T */
t[i][0]  *t[i]      **(t+i)  i[t][0]  0[t[i]]                      /* type T */
t[i][j]  (t[i])[j]  (*(t+i))[j]  j[*(t+i)]  j[i[t]]  *(*(t+i)+j)   /* type T */
```

Là encore, ces notations resteraient équivalentes si `t` était, non plus un nom de tableau, mais un pointeur sur des tableaux de 4 `T`.

3.3 Ordre des pointeurs et ordre des adresses

Parler, par exemple, de l'entier suivant un entier donné, sous-entend que l'on progresse en mémoire selon une direction donnée. Certes, il n'y a que deux directions possibles : suivant les adresses décroissantes ou croissantes.

La norme ne précise nullement quel est l'ordre choisi. En général, ce n'est pas gênant, tant qu'on ne s'intéresse pas véritablement aux adresses correspondantes. On notera que la même incertitude existe quant à l'arrangement des éléments d'un tableau en mémoire : ils peuvent être placés selon l'ordre des adresses croissantes ou décroissantes.

Cependant, dans tous les cas, la norme assure que si `ad` pointe sur l'entier de rang `i` d'un tableau, alors `ad + 1` pointera sur l'entier de rang `i+1`, que ce dernier ait une adresse supérieure ou inférieure au précédent. Autrement dit, il y a bien cohérence entre l'arithmétique des pointeurs et l'arrangement des tableaux.

3.4 Les restrictions imposées à l'arithmétique des pointeurs

Supposons que l'on ait déclaré :

```
int *adi ;
```

Si `adi` pointe sur un tableau comportant suffisamment d'éléments de type `int`, une expression telle que `adi+3` pointe sur un élément de ce tableau, donc sur un entier précis.

En revanche, si tel n'est pas le cas, par exemple :

```
int n ;
int *adi = &n ;
```

il n'est pas du tout certain qu'à l'adresse contenue dans `adi+3`, on trouve un entier. Dans le cas le plus défavorable, il se peut même que l'adresse correspondante ne soit pas valide.

D'une manière similaire, si les adresses correspondantes sont situées au sein d'un même tableau, la différence de deux pointeurs `p2 - p1`, ramenée à l'unité utilisée (ici, la taille d'un

int) a bien un sens. Dans le cas contraire, rien ne dit que cette différence soit un multiple de la taille d'un int.

Pour tenir compte de ces difficultés, la norme impose une contrainte théorique aux expressions de type pointeur, à savoir l'appartenance à un même tableau des objets concernés. Comme on peut s'y attendre, une telle contrainte sera rarement vérifiable à la compilation et d'ailleurs la norme prévoit un comportement indéterminé lors de l'exécution. Examinons cela plus en détail en considérant séparément le cas des expressions de type pointeur et celui des soustractions de pointeurs.

3.4.1 Cas des expressions de type pointeur

La règle

Considérons des expressions de la forme :

```
pointeur + entier
pointeur - entier
```

La norme précise que le comportement du programme est indéterminé si pointeur et l'expression à évaluer ne pointent pas sur des objets appartenant à un même tableau. Il s'agit cependant d'une condition à la fois floue et restrictive.

Cette condition est floue car elle ne précise pas vraiment que le tableau en question a besoin d'avoir été déclaré comme un tableau. Heureusement d'ailleurs, puisque c'est ce qui permettra de travailler correctement avec des tableaux dont l'emplacement aura été alloué dynamiquement :

```
int * adr ;
   .....
adr = malloc (100 * sizeof (int)) ;
   .....
/* adr[i] ou *(adr+i) sont utilisables dans ce contexte */
```

Par ailleurs, cette condition est restrictive dans la mesure où, par définition, en C, tout objet de type T peut être considéré comme une suite de sizeof (T) octets, c'est-à-dire finalement comme un tableau de sizeof (T) caractères. Or il est tout à fait licite de parcourir les différents octets d'un objet comme s'il s'agissait d'un tableau de caractères !

En fait, nous préférons exprimer cette contrainte en disant que :

> **Les objets concernés doivent pouvoir être considérés comme des éléments d'un même tableau.**

Cette condition est notamment vérifiée dans les cas évoqués précédemment.

Remarque

Considérons une construction telle que :

```
int t[10] ;
int *p ;
  .....
for (p=t, i=0  ; i<=10  ; i++, p++)
```

À la fin du dernier tour, la valeur de `p` dépasse de une unité l'adresse du dernier élément du tableau `t`. La construction proposée ne respecte pas la contrainte évoquée précédemment, concernant l'appartenance à un même tableau. Pour rendre légale une construction aussi répandue, la norme a dû faire une exception pour le cas des éléments situés un élément plus loin que la fin du tableau.

En fait, cette règle n'est utile qu'aux réalisateurs de compilateurs. Il leur faut en effet accepter le calcul d'une telle adresse, même dans le cas où elle serait invalide, c'est-à-dire dans le cas (rare) où le tableau se trouverait juste en limite de l'espace mémoire alloué au programme.

Comportement du programme en cas d'exception

Examinons les différentes situations d'exception qui peuvent se produire dans les calculs liés aux pointeurs.

En pratique, la plupart des compilateurs ne cherchent pas à vérifier que la contrainte d'appartenance à un tableau est vérifiée, même quand cela est possible. Ce n'est pas pour autant qu'il est possible d'utiliser impunément n'importe quelle expression de type pointeur.

En effet, il ne faut pas oublier qu'un calcul d'adresse de la forme `pointeur + entier` ou `pointeur - entier` peut toujours conduire à une adresse invalide, c'est-à-dire à une adresse inexistante ou, pour le moins, située en dehors de la mémoire allouée au programme. Dans ce cas, le comportement du programme n'est pas défini. Dans la plupart des implémentations, on obtient une erreur d'exécution. Cependant, il existe des implémentations où un tel calcul peut conduire non pas à une adresse invalide, mais à une adresse fausse ; dans ce cas, les conséquences peuvent être plus graves...

Par ailleurs, même si l'expression concernée fournit une adresse existante, il est possible que dans sa déréférenciation, c'est-à-dire dans un calcul de la forme :

```
*(adresse + entier)     adresse [entier]     *(adresse - entier)
```

on aboutisse à une erreur d'exécution liée au fait que le motif binaire trouvé à cette adresse n'est pas légal pour le type correspondant ; par exemple, il peut s'agir d'un flottant non normalisé.

D'une manière générale, par le biais de l'équivalence entre tableau et pointeur, ces risques sont identiques à ceux évoqués en cas de débordement d'indice dans un tableau (voir section 4 du chapitre 6).

Exemple

```
int main()
 { float x[5] ;
   float *adf = x  ;      /* adf pointe sur le premier élément de x */
   while (1) printf ("%f", *adf++) ;
 }
```

Ce programme peut générer plusieurs sortes de situations d'exception :

- L'adresse résultant du calcul `adf++` correspond à un élément situé au-delà de l'élément suivant le dernier élément du tableau `x`. Il y a non respect de la norme et donc, en théorie, comportement indéterminé. En pratique, le programme se poursuivra, tant que l'une des exceptions suivantes n'apparaîtra pas.
- L'adresse obtenue par `adf++` est invalide ou se situe en dehors de l'emplacement alloué au programme. La plupart du temps, cette anomalie sera convenablement détectée et conduira à une erreur d'exécution. Si tel n'est pas le cas, les conséquences peuvent être diverses et conduire à un « plantage » de la machine.
- La valeur située à l'adresse `adf` (et qu'on cherche à imprimer) ne correspond pas à un flottant normalisé. Selon les implémentations, on pourra obtenir un message et/ou un arrêt de l'exécution.

3.4.2 Cas de la soustraction de pointeurs : le type ptr_difft

De façon comparable à ce qui se passe pour la somme d'un pointeur et d'un entier, la norme ANSI prévoit que le comportement de l'évaluation de l'expression `p2 - p1` n'est pas défini si les objets pointés n'appartiennent pas à un même tableau (avec une exception pour l'élément suivant immédiatement le dernier). En pratique, le résultat de cet opérateur est évalué en calculant la différence entre les adresses de `p2` et de `p1`, et en divisant la valeur obtenue par la taille des objets pointés (avec une incertitude sur la manière dont se fait l'arrondi lorsque l'on n'aboutit pas à un multiple exact de la taille).

La norme demande à l'implémentation de définir un type entier signé nommé `ptrdiff_t` (synonyme défini par `typedef` dans le fichier en-tête `stddef.h`) et de l'utiliser pour recueillir le résultat de la soustraction. On notera que la norme n'impose pas à ce type d'être assez grand pour contenir la différence de deux pointeurs quelconque. Elle n'impose d'ailleurs même pas qu'il suffise à recueillir la différence de pointeurs sur deux éléments d'un même tableau de taille quelconque ! En pratique, on peut raisonnablement supposer qu'un tel problème ne se pose pas au sein d'un « vrai » tableau, quelle que soit sa taille, dans la mesure où `ptrdiff_t` devrait au moins être défini en fonction de la taille du plus grand objet manipulable par un programme. En revanche, rien n'empêche qu'on rencontre ce problème lorsque l'on est amené à soustraire deux pointeurs sur des objets, certes de même type, mais n'ayant aucun lien entre eux. Dans ce cas, on risque d'aboutir soit à une erreur d'exécution, liée à un dépassement de capacité, soit, plus fréquemment, à une valeur fausse.

4. Tableaux récapitulatifs : les opérateurs +, -, &, * et []

Cette section n'apporte pas d'éléments nouveaux. Elle récapitule simplement tout ce qui concerne les opérateurs faisant intervenir des pointeurs sur des objets, le cas des pointeurs sur des fonctions étant, quant à lui, examiné au chapitre 8.

Tableau 7.3 : les opérateurs + et - dans un contexte pointeur

Opération	Résultat et type	Contrainte(s) théorique(s) (invérifiables en compilation)	Comportement si contrainte(s) non vérifiée(s)

`pointeur + entier`
`entier + pointeur`
`pointeur - entier`
	– résultat : pointeur incrémenté ou décrémenté de `entier` ;
	– type : celui de `pointeur`, y compris les qualifieurs de l'objet pointé.
		Le pointeur et l'expression doivent pointer sur des objets pouvant être considérés comme éléments d'un même tableau (l'un des éléments pouvant être situé immédiatement après le dernier élément du tableau).	ANSI : indéfini
			En pratique :
			– si l'adresse est invalide, erreur d'exécution si la machine la gère, plantage potentiel sinon ;
			– erreur retardée possible lors de l'utilisation de la valeur pointée si le motif binaire ne convient pas pour le type.

`pointeur1 - pointeur2`	– résultat : différence entre les deux pointeurs (obligatoirement de même type), en se fondant sur l'unité correspondant aux objets pointés ; – type : `ptrdif_t`.	1 - `pointeur1` et `pointeur2` doivent pointer sur des objets pouvant être considérés comme éléments d'un même tableau (l'un des éléments pouvant être situé immédiatement après le dernier élément du tableau).	ANSI : indéfini *En pratique* : pas de problème si la contrainte 2 (ci-dessous) est satisfaite.
		2 - La valeur résultante doit être représentable dans le type `ptrdif_t`.	ANSI : indéfini *En pratique* : conséquences habituelles d'un dépassement de capacité dans un type entier signé.

Tableau 7.4 : les opérateurs réciproques & et * (cas des objets)

Opérateur	Opérande	Résultat et comportement
`&exp`	`exp` est une expression désignant un objet de type `T` quelconque, autre que champ de bits, et n'étant pas de classe `register` (il peut s'agir d'un tableau).	– résultat : pointeur, de type « pointeur constant sur T » (avec les éventuels qualifieurs de `T`) contenant l'adresse de l'objet.
`*adr`	`adr` est un pointeur sur un objet de type quelconque.	– résultat : l'objet pointé ; s'il n'a pas le type prévu, la norme prévoit un comportement indéterminé ; en pratique, si l'on place quelque chose à l'adresse indiquée, on court les mêmes risques qu'avec `scanf` et un code de format incorrect ; si l'on utilise la valeur concernée, on cours les mêmes risques qu'avec `printf` et un code de format incorrect ; – comportement : si l'opérande a la valeur nulle ou s'il correspond à une adresse invalide, la norme prévoit un comportement indéterminé ; en pratique, on obtient généralement une erreur d'exécution.

N.B : le cas des pointeurs sur des fonctions est étudié à la section 11 du chapitre 8.

Tableau 7.5 : l'opérateur []

Opérateur	Type des opérandes	Résultat	Contrainte théorique (non vérifiable en compilation)	Comportement si contrainte non vérifiée
`op1 [op2]`	L'un des deux de type pointeur sur un objet, l'autre de type entier.	L'objet pointé par `op1+op2`, c'est-à-dire `*(op1+op2)`.	L'objet résultat et l'objet pointé par celui des deux opérandes qui est un pointeur doivent pouvoir être considérés comme éléments d'un même tableau (l'un des éléments pouvant être situé immédiatement après le dernier élément du tableau.	ANSI : indéfini *En pratique*[1] : erreur d'exécution si l'adresse correspondante est invalide ou si le motif binaire ne convient pas pour le type.

1. Comme `exp1[exp2]` est équivalent à `*(exp1+exp2)`, on cumule ici les deux possibilités d'erreur, d'une part pour l'addition d'un entier à un pointeur et d'autre part, pour l'opération `*`.

5. Le pointeur NULL

Il existe un symbole noté `NULL`, défini dans certains fichiers en-tête (`stddef.h`, `stdio.h` et `stdlib.h`), dont la valeur représente conventionnellement un pointeur ne pointant sur rien, c'est-à-dire auquel n'est associée aucune adresse. Cette valeur peut être affectée à un pointeur de n'importe quel type, par exemple :

```
int *adi ;              /* pointeur sur des int */
double (* adt) [10]     /* pointeur sur des tableaux de 10 double */
  .....
adi = NULL ;
adt = NULL ;
```

Cette valeur `NULL` ne doit pas être confondue avec une valeur de pointeur indéfinie, même si on l'utilise conventionnellement pour indiquer qu'un pointeur ne pointe sur rien. Il s'agit au contraire d'une valeur bien définie. En particulier, on peut tester l'égalité (opérateur `==`) ou la non-égalité (opérateur `!=`) de n'importe quel pointeur avec `NULL`. En revanche, les comparaisons d'inégalité (`<`, `<=`, `>`, `>=`) sont théoriquement interdites, même si certaines implémentations les acceptent. Dans ce cas, le résultat obtenu, probablement basé sur l'ordre des adresses, n'est pas portable.

Exemples d'utilisation de **NULL**

Pour l'initialisation d'une variable pointeur

Dès qu'on utilise des variables de type pointeur, il est raisonnable :

- d'initialiser avec `NULL` tous les pointeurs pour lesquels il n'existe pas de meilleure initialisation :

```
int * adi = NULL ;        /* par précaution */
```

- de tester, au moment de son utilisation, tout pointeur qui risque de n'avoir pas reçu de valeur (autre que NULL) :

```
if (adi != NULL) *adi = ...  /* on est sûr que adi a reçu une valeur */
                             /* on peut aussi écrire :               */
                             /*     if (adi) *adi = ...              */
```

Dans des listes chaînées

Dans des listes chaînées, pour indiquer qu'un pointeur ne pointe sur rien, la valeur de NULL s'avère parfaitement adaptée. Ce sera notamment le cas d'un pointeur de fin de liste. On en trouvera un exemple au chapitre 14.

En valeur de retour d'une fonction

Les fonctions standard qui fournissent un pointeur comme résultat renvoient souvent NULL en cas de problème. C'est notamment le cas de toutes les fonctions d'allocation dynamique telles que malloc ou calloc. Il est conseillé de procéder de la même manière avec ses propres fonctions.

Remarque

On pense souvent que la valeur associée à NULL est l'entier 0. Si l'on examine la norme à ce propos, elle n'est pas aussi précise :

- d'une part, elle dit que la macro NULL fournit un pointeur nul, dont la valeur dépend de l'implémentation ;
- d'autre part, elle indique que l'entier 0, converti en void *, est un pointeur nul.

En toute rigueur donc, la valeur associée à NULL n'est pas totalement définie : on est sûr que c'est (void *) 0, mais on n'est pas sûr que ce soit un objet avec tous les bits à 0 (comme l'est l'entier 0). En pratique, hormis peut-être dans des situations de mise au point un peu particulières, il n'est pas indispensable d'en savoir plus. En effet, quelle que soit la façon d'employer NULL, la valeur entière 0 reste utilisable grâce aux règles concernant l'affectation, les conversions et la comparaison de pointeurs présentées sections 6 à 9. Ainsi, avec :

```
int *adi ;
```

vous pouvez indifféremment écrire :

```
adi = NULL ;          /* écriture conseillée */
adi = 0 ;             /* 0 sera converti en NULL par conversion implicite */
                      /*   de int en int *                                */
adi = (void *) 0 ;
adi = (int *) 0 ;
```

De même, dans le test de nullité d'un pointeur, les deux écritures :

```
if (adi != NULL)
if (adi)
```

restent identiques et portables, car l'interprétation logique d'une expression de type pointeur se fait bien par rapport au pointeur nul (sous-entendu ayant la valeur NULL) et non par rapport à l'entier 0.

6. Pointeurs et affectation

On peut affecter à une variable de type pointeur sur un objet la valeur d'une expression de même type, par exemple :

```
int *ad1, *ad2 ;
   .....
ad1 = ad2 + 1 ;        /* ad2 est de type int *, ad2+1 est de type int *    */
```

Il faut cependant préciser comment interviennent les qualifieurs des objets pointés. En outre, il existe quelques rares possibilités de conversion par affectation :

- d'une expression de type `void *` en un pointeur quelconque et réciproquement ;
- de la valeur entière 0 en un pointeur quelconque.

6.1 Prise en compte des qualifieurs des objets pointés

Comme indiqué dans la section 3.1.2, les qualifieurs de l'objet pointé sont répercutés sur le type d'une expression de type pointeur. Par exemple, avec :

```
const int *ad ;
```

la variable `ad` est du type « pointeur sur un int constant ». Une expression telle que `ad+3` sera également du type « pointeur sur un int constant », et pas seulement du type « pointeur sur int ». La même remarque s'appliquerait à `volatile`.

En cas d'affectation d'une expression de type pointeur, on pourrait penser qu'il est nécessaire que la `lvalue` réceptrice possède les mêmes qualifieurs pour les objets pointés. En fait, la règle est un peu moins restrictive :

> **En cas d'affectation d'une expression de type pointeur à une `lvalue`, cette dernière doit posséder au moins les mêmes qualifieurs que l'objet pointé par l'expression.**

Nous allons justifier cette règle en examinant séparément le cas de `const` et celui de `volatile`.

6.1.1 Cas de const

Considérons par exemple ces déclarations :

```
const int *adc ;      /* adc est un pointeur sur un int constant */
int *ad ;             /* ad est un pointeur sur un int           */
```

D'après la règle évoquée, les affectations suivantes sont interdites :

```
ad = adc ;         /* incorrect */
ad = adc + 3 ;     /* incorrect */
```

Ceci est logique car si elles étaient acceptées, on risquerait par la suite de modifier un objet constant par une simple instruction telle que :

```
*ad = ...
```

Rappelons, en effet, qu'une telle instruction ne pourrait plus être rejetée par le compilateur puisqu'il fonde sa décision sur le type de `ad` et en aucun cas sur le type effectif de l'objet pointé par `ad` au moment de l'exécution (C est un langage à typage statique).

En revanche, ces affectations seront acceptées :

```
adc = ad ;          /* correct */
adc = ad + 5 ;      /* correct */
```

Là encore, les choses sont logiques, dans la mesure où aucun risque de modification intempestive n'existe ici. En effet, il n'est pas possible d'écrire une affectation de la forme :

```
*adc = ...
```

et quand bien même elle le serait, l'objet pointé n'est de toute façon pas constant !

On peut interpréter la règle en disant qu'on peut donner l'adresse d'un objet variable, là où on s'attend à l'adresse d'un objet constant, mais non l'inverse. On peut aussi dire qu'on peut faire à un objet non constant tout ce qu'on a prévu de faire sur un objet constant, mais non l'inverse.

Remarque

Il ne faut pas confondre les qualifieurs des objets pointés avec les éventuels qualifieurs de la `lvalue`. Dans nos précédents exemples, avec :

```
const int * const adc ;
int *adi ;
```

l'instruction `adc = ad` serait rejetée car `adc` n'est plus une `lvalue`.

6.1.2 Cas de volatile

Considérons par exemple ces déclarations :

```
volatile int *adv ;      /* adv est un pointeur sur un int volatile */
int *ad ;                /* ad est un pointeur sur un int           */
```

D'après la règle évoquée, les affectations suivantes sont interdites :

```
ad = adv ;          /* incorrect, mais accepté par certaines implémentations */
ad = adv + 3 ;      /* incorrect, mais accepté par certaines implémentations */
```

Ceci est logique, car si elles étaient acceptées, le compilateur risquerait, par exemple dans certains cas d'optimisation, de sortir d'une boucle une instruction telle que :

```
... = *ad ;
```

sous prétexte que cette valeur ne change pas. Cette conclusion peut se révéler erronée puisque la valeur réellement pointée est en fait `volatile`. Rappelons, là encore, qu'une telle instruction ne peut plus être rejetée par le compilateur, qui fonde sa décision uniquement sur le type de `ad` et en aucun cas sur le type effectif de l'objet pointé par `ad` au moment de l'exécution (C est un langage à typage statique).

En revanche, ces affectations seront acceptées :

```
adv = ad ;          /* correct */
adv = ad + 5 ;      /* correct */
```

Là encore, les choses sont logiques, dans la mesure où aucun risque d'optimisation abusive n'existe puisque, cette fois, le compilateur ne risque plus de sortir d'une boucle une instruction telle que :

```
... = *adv ;
```

En effet, cette fois `adv` est censé pointer sur des objets volatiles et, qui plus, l'objet réellement pointé n'est pas `volatile`.

On peut interpréter la règle en disant qu'on peut donner l'adresse d'un objet non volatile, là où on s'attend à l'adresse d'un objet `volatile`, mais non l'inverse. On peut aussi dire qu'on peut faire à un objet non `volatile` tout ce qu'on a prévu de faire sur un objet `volatile`, mais non l'inverse.

Remarque

Là encore, il ne faut pas confondre les qualifieurs des objets pointés avec les éventuels qualifieurs de la `lvalue`. Par exemple, avec :

```
volatile int * adv ;   /* adv est un pointeur sur un int volatile */
int * volatile ad ;    /* ad est un pointeur volatile sur un int  */
```

l'instruction `adv = ad` serait correcte, bien que la `lvalue adv` ne possède pas le qualifieur `volatile` que possède `ad`.

6.2 Les autres possibilités d'affectation

Le type void * et les affectations

Le type générique `void *` est présenté à la section 7. Il s'agit du seul type compatible par affectation avec tous les types de pointeurs : un pointeur de type `void *` peut être affecté à n'importe quel autre et, réciproquement, un pointeur de n'importe quel type peut être affecté à un pointeur de type `void *`. On verra toutefois que seules les affectations d'un pointeur de type `void *` à un pointeur quelconque fournissent l'assurance de conserver l'adresse d'origine.

Le pointeur NULL

Il a été présenté à la section 5. Il correspond à une valeur particulière ne coïncidant jamais avec une véritable adresse en machine et il a été conçu pour pouvoir être affecté à n'importe quelle variable pointeur.

L'entier 0

D'une manière générale, les conversions d'entier en pointeur ne sont pas permises par affectation (elles pourront s'obtenir par l'opérateur de `cast`, comme indiqué à la section 9). Une

exception existe cependant pour l'entier 0. Elle est simplement justifiée par le fait que, quel que soit le type de pointeur concerné, sa conversion fournit le pointeur `NULL`. D'ailleurs, comme vu à la section 5, il y a équivalence entre `NULL` et `(void *) 0` :

```
int *adi ;
float *adf ;
   .....
adi = 0 ;    /* équivaut à la forme conseillée :  adi = NULL;   */
adf = 0 ;    /* équivaut à la forme conseillée :  adf = NULL;   */
```

Les pointeurs sur des fonctions

Ces possibilités sont étudiées à la section 11.2 du chapitre 8.

6.3 Tableau récapitulatif

Le tableau 7.6 récapitule tout ce qui concerne l'affectation de pointeurs, y compris dans le cas des pointeurs sur des fonctions. Il s'agit en fait d'un extrait du tableau de la section 7.4 du chapitre 4, concernant l'affectation en général, et il n'est fourni ici qu'à titre de commodité.

Tableau 7.6 : les affectations à des lvalue de type pointeur

Opérande de gauche	Opérande de droite	Remarques
`lvalue` de type pointeur sur un objet, autre que `void *`	Une des deux possibilités suivantes : – expression du même type pointeur ou de type `void *`, la `lvalue` concernée devant dans tous les cas posséder **au moins** les mêmes qualifieurs `const` ou `volatile` que le type des objets pointés ; – `NULL` ou entier 0.	– justification de la règle des qualifieurs à la section 6.1 ; – présentation du pointeur `NULL` à la section 5.
`lvalue` de type `void *`	Une des deux possibilités suivantes : – expression d'un type pointeur quelconque, y compris `void *`, la `lvalue` concernée devant dans tous les cas posséder **au moins** les mêmes qualifieurs `const` ou `volatile` que le type des objets pointés ; – `NULL` ou entier 0.	– justification de la règle des qualifieurs à la section 6.1 ; – présentation du type `void *` à la section 7 ; – présentation du pointeur `NULL` à la section 5.
`lvalue` de type pointeur sur une fonction	Valeur d'un type compatible au sens de la redéclaration des fonctions.	Voir section 11.2 du chapitre 8

6.4 Les affectations élargies += et -= et les incrémentations ++ et --

Bien entendu, les opérateurs `+=` et `-=` restent utilisables dans un contexte pointeur, leur signification se déduisant de celle de l'affectation et de la somme (ou de la différence) d'un pointeur et d'un entier. Quant à `++` et `--`, ils incrémentent ou décrémentent tout simplement de une unité la valeur d'un pointeur. Voici quelques exemples :

```
int *ad ;
   .....
ad += 3 ;   /* équivaut à :  ad = ad + 3;     */
ad -= 8 ;   /* équivaut à :  ad = ad - 8;     */
ad++ ;      /* équivaut à :  ad += 1;  ou encore :  ad = ad + 1; */
ad-- ;      /* équivaut à :  ad -= 1;  ou encore :  ad = ad - 1; */
```

7. Les pointeurs génériques

7.1 Généralités

En C, un pointeur possède un type défini par le type des objets pointés. On peut dire, en quelque sorte, que la valeur d'un pointeur est formée de l'association d'une adresse et d'un type. La connaissance de ce type est indispensable dans des situations telles que :

- calculs arithmétiques : l'unité utilisée étant définie par la taille de l'objet pointé ;
- déréférenciation de pointeur, c'est-à-dire utilisation d'expression de la forme `*p` (`p` étant un pointeur) ; le type est ici utile pour connaître la taille de l'objet à considérer et, éventuellement, pour utiliser la valeur correspondante au sein d'une expression.

Néanmoins, la connaissance de ce type n'est pas toujours indispensable. En particulier, l'adresse contenue dans un pointeur a toujours un sens, indépendamment de la nature de l'objet pointé. Dans certains cas, il peut être utile, voire indispensable de pouvoir manipuler de simples adresses, sans avoir à se préoccuper d'un quelconque type. C'est ce qui se produit lorsque :

- une fonction doit traiter des objets de différents types, alors qu'elle en a reçu l'adresse en argument ;
- on souhaite effectuer un traitement sur des pointeurs, sans avoir à tenir compte (ou sans connaître) le type des objets pointés ; c'est par exemple le cas lorsqu'on souhaite simplement échanger les valeurs de deux pointeurs ;
- on doit traiter une suite d'octets, à partir d'une adresse donnée.

Dans la première définition du langage C, la seule solution à ce type de problème consistait à faire appel au type `char *`, lequel permet en théorie de représenter n'importe quelle adresse d'octet, donc, *a fortiori*, n'importe quelle adresse d'objet. Cependant, cette démarche s'avérait peu satisfaisante dans certains cas, notamment lorsqu'on souhaitait transmettre une telle adresse à une fonction. En effet, il fallait alors prévoir des conversions explicites du pointeur concerné dans le type `char *`, à l'aide d'un opérateur de `cast`.

La norme ANSI a introduit un type particulier souvent appelé « pointeur générique » et noté :

```
void *
```

Ce type `void *` présente les avantages suivants sur le type `char *` :

- il évite les confusions : dans la première version du langage C, le type `char *` pouvait aussi bien désigner :
  - un « vrai pointeur » sur un caractère ou, ce qui revient au même, sur une chaîne de caractères ;
  - la représentation artificielle d'un pointeur générique ;
- il interdit l'arithmétique et la déréférenciation, afin de rendre les programmes plus sûrs.

Néanmoins, même dans sa version normalisée, le C continue à souffrir de certaines lacunes en matière de pointeurs. En effet, il lui manque toujours un pointeur sur des octets, le type `void *` ne pouvant pas être utilisé, par exemple, pour parcourir les différents octets d'un objet. Dans ce cas, il faudra encore recourir au type `char *`.

7.2 Déclaration du type void *

On déclare des pointeurs de ce type, comme n'importe quel autre pointeur, avec d'éventuels qualifieurs. Par exemple, avec :

```
const void *p1, * const p2 ;
```

- `p1` est un pointeur générique sur des objets constants de type quelconque. On notera que le fait que les objets en question soient constants n'apporte rien de plus au compilateur puisque, le pointeur `p1` ne pouvant pas être déréférencé, il est, de toutes façons, impossible d'écrire, par exemple :

```
*p1 = ...   /*     interdit car p1 est de type void *             */
            /*     que les objets pointés soient constants ou non */
```

- `p2` est un pointeur générique constant sur des objets constants. Là encore, comme pour `p1`, le fait que les objets pointés par `p2` soient constants n'apporte rien de plus au compilateur. En revanche, le fait que `p2` soit lui même constant en interdit la modification.

On notera que l'utilisation du mot `void` peut prêter à confusion. En effet, si l'on peut affirmer que `p1` est de type `void *`, on ne peut pas pour autant affirmer, comme on le ferait pour n'importe quel autre type de pointeur, que `*p1` est de type `void`, car il n'existe pas de type `void`. En fait, l'ambiguïté réside essentiellement dans le désir des concepteurs du C de limiter le nombre de mots clés, ce qui les a conduits à employer le mot `void` dans des contextes différents, avec des significations différentes : dans les en-têtes de fonctions, il signifie « absence de », tandis que, associé à un déclarateur de pointeur, il signifie « n'importe quel type » !

Remarque

Pour se convaincre de ce que, dans une déclaration, `void` est vraiment un spécificateur de type différent des autres, on peut aussi comparer les deux déclarations suivantes :

```
void *ad, n ;    /* incorrect : n ne peut pas être du type void */
int *ad, n ;     /* ad est de type int *, n et de type int      */
```

7.3 Interdictions propres au type void *

Contrairement aux autres types de pointeurs, un pointeur de type `void *` ne peut pas :

- être soumis à des calculs arithmétiques ;
- être déréférencé (utilisation de l'un des opérateurs `*` ou `[]`).

7.3.1 Le type void * ne peut pas être soumis à des calculs arithmétiques

Par exemple :

```
void *ad1, *ad2 ;
int dif ;
     .....
ad1++ ;             /* interdit                              */
ad2 = ad1 + 5 ;     /* l'expression ad1 + 5 est illégale     */
dif = ad2 - ad1 ;   /* l'expression ad2 - ad1 est illégale */
```

On notera que ces interdictions sont justifiées si l'on part du principe qu'un pointeur générique est simplement destiné à être manipulé en tant que tel, par exemple pour être transmis d'une fonction à une autre. En revanche, elles le sont moins si l'on considère qu'un tel pointeur peut aussi servir à manipuler des octets successifs ; dans ce cas, il faudra quand même recourir au type `char *`.

7.3.2 Le type void * ne peut pas être déréférencé

Par exemple, avec :

```
void *adr ;
```

il est impossible d'utiliser l'expression `*adr`. On notera que, cette fois, une telle interdiction est logique puisque la valeur de l'expression en question ne peut être déterminée que si l'on connaît le type de l'objet pointé.

Bien entendu, il reste toujours possible de convertir par `cast` la valeur de `adr` dans le type approprié, pour peu qu'on le connaisse effectivement ! Par exemple, si l'on sait qu'à l'adresse contenue dans `adr`, il y a un entier, on pourra utiliser l'expression :

```
* (int *) adr     /* équivautt à :    * ( (int *) adr)         */
                  /*  valeur de l'objet pointé par adr         */
                  /* en supposant qu'il s'agit d'un entier     */
```

Si l'adresse contenue dans `adr` n'a pas véritablement été obtenue comme celle d'un entier, il est possible que sa conversion en `int *` la modifie pour tenir compte de certaines contraintes d'alignement (voir section 9.1.1).

7.4 Possibilités propres au type void *

Par rapport aux autres types de pointeurs, le type `void *` dispose de possibilités supplémentaires, au niveau :

- des comparaisons d'égalité ;
- des conversions par affectation.

7.4.1 Comparaisons d'égalité ou d'inégalité

Comme nous le verrons à la section 8 on ne peut pas tester l'égalité ou l'inégalité de deux pointeurs de types différents. Une exception a lieu pour le type `void *`, qui peut être comparé par `==`

ou != avec n'importe quel autre type de pointeur (voir section 8.2). Nous verrons alors que cette comparaison se ramène finalement à la comparaison des adresses correspondantes.

```
void * ad ;
int * adi ;
   .....
if (ad == adi) .....   /* ad et adi contiennent la même adresse       */
          else .....   /* ad et adi ne contiennent pas la même adresse */
```

7.4.2 Conversions par affectation

Tout d'abord, un pointeur de n'importe quel type peut être converti par affectation en `void *`, pour peu que la règle relative aux qualifieurs, présentée à la section 6, soit vérifiée. Cette possibilité n'a rien de très surprenant puisqu'elle revient à ne conserver du pointeur d'origine que l'information d'adresse, ce qui correspond bien à la notion de pointeur générique.

Réciproquement, un pointeur de type `void *` peut être converti par affectation en un pointeur sur un objet de type quelconque, moyennant le respect de la règle relative aux qualifieurs. Cette fois, une telle possibilité est relativement discutable, dans la mesure où elle présente certains risques. En effet, il est possible que l'adresse d'origine soit modifiée pour tenir compte d'éventuelles contraintes d'alignement du type d'arrivée.

On notera toutefois que la conversion de `void *` en `char *` ne présente pas les risques évoqués dans la mesure où aucune contrainte d'alignement ne pèse sur le type caractère, lequel correspond à un octet.

En C++

En C++, seule la conversion par affectation d'un pointeur en `void *` sera légale ; la conversion inverse ne sera possible qu'en recourant explicitement à un opérateur de `cast`, y compris dans le cas sans risque de la conversion de `void *` en `char *`.

8. Comparaisons de pointeurs

Le langage C permet de comparer des pointeurs sur des objets. Il faut distinguer :

- les comparaisons basées sur un ordre, c'est-à-dire celles qui font appel aux opérateurs <, <=, > et >= ;
- les comparaisons d'égalité ou d'inégalité, c'est-à-dire celles qui font appel aux opérateurs == ou !=.

Quant à la comparaison de pointeurs sur des fonctions, elle ne peut se faire que par égalité ou inégalité (voir section 11 du chapitre 8). Elle ne figurera que par souci d'exhaustivité dans les tableaux récapitulatifs.

8.1 Comparaisons basées sur un ordre

Tout d'abord, ces comparaisons ne sont théoriquement définies par la norme que pour des pointeurs de même type, les qualifieurs du pointeur ou de l'objet pointé n'intervenant pas. Si cette condition n'est pas vérifiée, on obtient une erreur de compilation[6].

En outre, la norme prévoit une contrainte théorique assurant que ces comparaisons ont une signification, en induisant une contrainte théorique d'appartenance des objets pointés à un même agrégat (tableau, structure ou union). Le cas des structures et des unions présente peu d'intérêt, il sera étudié à la section 4 du chapitre 11.

En ce qui concerne les tableaux, la contrainte imposée correspond à celle exposée à la section 3.4, à propos des expressions de type pointeur et que nous préférons traduire en disant que les éléments doivent pouvoir être considérés comme éléments d'un même tableau. Dans le cas contraire, le résultat de la comparaison est simplement indéterminé. On notera bien qu'ici, il ne s'agit plus de comportement indéterminé, ce qui signifie que, pour peu que les expressions comparées soient de même type et valides, on obtiendra toujours un résultat.

Exemple 1

Avec :

```
int t[10] ;
int *adr1, *adr2, *adr3 ;
   .....
adr1 = &t[1] ;
adr2 = &t[4] ;
adr3 = &t[10] ;    /* pointe sur l'élément suivant le dernier du tableau */
```

on peut assurer que les deux conditions suivantes sont vraies :

```
adr1 < adr2     /* vrai */
adr1 <= adr2    /* vrai */
adr1 < adr3     /* vrai */
```

En revanche, si l'on considère ces trois conditions après une affectation telle que :

```
adr2 = adr1 + 100 ;     /* ou encore :  adr2 = &t[101]    */
```

deux situations anormales apparaissent :

- Tout d'abord, au niveau du calcul de l'expression `adr1+100` elle-même, dont la norme prévoit que le comportement du programme est alors indéfini. En pratique, dès lors que l'adresse correspondante est valide, on obtiendra bien un résultat.
- Ensuite, au niveau des comparaisons elles-mêmes. La norme précise simplement que le résultat est alors indéterminé. En pratique cependant, dès lors que le calcul de l'expression a pu se faire, il est fort probable que le résultat des comparaisons restera le même que précédemment.

6. Toutefois, certaines implémentations se contentent d'un message d'avertissement n'interdisant pas l'exécution.

Exemple 2

Avec :

```
int n, p ;
int *adr1 = &n, *adr2 = &p ;
```

les comparaisons suivantes fournissent un résultat indéfini :

```
adr1 <  adr2     /* indéfini */
adr1 <= adr2     /* indéfini */
```

En pratique, le résultat de la comparaison dépendra simplement de l'implémentation des variables `n` et `p` en mémoire (en tenant compte de l'ordre imposé aux pointeurs qui n'est pas nécessairement celui des adresses, comme l'explique la section 3.3)[7]. Une telle information aura cependant généralement peu d'intérêt.

8.2 Comparaisons d'égalité ou d'inégalité

Les comparaisons précédentes imposaient des restrictions permettant de donner un sens à une relation d'ordre. Ici, en revanche, le langage va s'avérer plus tolérant puisqu'il ne s'agit que de détecter l'identité d'emplacement de deux objets pointés et non plus leur position relative.

C'est ainsi qu'on pourra tester l'égalité de deux pointeurs de même type (aux qualifieurs près). Il y aura égalité s'ils pointent sur les mêmes objets ; ils contiendront donc la même adresse. On notera bien que, réciproquement, deux pointeurs contenant la même adresse ne pourront être trouvés égaux que s'ils sont de même type. Ils ne pourront pas, de toute façon, être comparés directement, c'est-à-dire sans conversion par `cast`.

De plus, tout pointeur peut être comparé à `NULL`. Par exemple, avec :

```
long *adl ;
char *adc ;
```

les comparaisons suivantes sont légales et ont un sens (certaines sont utilisées à la section 5) :

```
adl == NULL     /* correct */
adc != NULL     /* correct */
adl == 0        /* correct : il y aura conversion de 0 en (void *) 0  */
                /* c'est-à-dire en NULL                               */
```

Un pointeur de type `void *` peut être comparé par égalité ou inégalité avec n'importe quel autre pointeur, quel que soit son type, la comparaison se faisant après conversion en `void *`. Au bout du compte, cela revient donc simplement à comparer les adresses correspondantes, sans tenir compte de la véritable nature de l'objet pointé par le premier pointeur.

7. En fait, on peut penser que les contraintes imposées par la norme permettent à une implémentation qui le souhaiterait d'utiliser pour les éléments de tableaux un ordre différent de celui utilisé pour les structures.

8.3 Récapitulatif : les comparaisons dans un contexte pointeur

Tableau 7.7 : les comparaisons de pointeurs basées sur un ordre

Opération	Contrainte vérifiée en compilation	Contrainte théorique (norme ANSI) non vérifiable en compilation	Comportement si contrainte théorique non vérifiée

`p1 < p2`
`p1 <= p2`
`p1 > p2`
`p1 >= p2`
	`p1` et `p2` sont des pointeurs de même type (aux qualifieurs près).	Les objets pointés appartiennent à la même structure ou peuvent être considérés comme des éléments d'un même tableau.	ANSI : résultat indéfini
			En pratique : résultat basé sur les valeurs relatives des adresses correspondantes, sachant que l'ordre des pointeurs peut différer de celui des adresses (voir section 3.3).

Tableau 7.8 : les comparaisons d'égalité et d'inégalité des pointeurs

Opération	L'une des contraintes suivantes doit être satisfaite (vérifiée en compilation)	Remarques
`p1 == p2` `p1 != p2`	`p1` et `p2` sont des pointeurs sur des objets de même type (aux qualifieurs près).	
	Un des deux pointeurs au moins est de type `void *`.	Conversion de l'autre opérande en `void *`.
	Un des deux opérandes est `NULL` ou 0.	
	`p1` et `p2` sont des pointeurs sur des fonctions de type compatible.	Étude détaillée à la section 11.6 du chapitre 8.

9. Conversions de pointeurs par cast

Comme nous l'avons vu dans la section 6, le langage C est assez restrictif en ce qui concerne les conversions implicites autorisées lors d'une affectation entre pointeurs : hormis celle de `void *` en un pointeur quelconque, ces conversions ne présentent aucun risque. En revanche, il est beaucoup plus tolérant en ce qui concerne les conversions forcées par l'opérateur de `cast`. En effet, il est possible de convertir :

- tout type pointeur sur un objet en n'importe quel autre type pointeur sur un objet ;
- un entier en un pointeur ;
- un pointeur en un entier ;
- tout type pointeur sur une fonction en n'importe quel autre type pointeur sur une fonction.

Le dernier point est étudié à la section 11.7 du chapitre 8. Nous nous contenterons de le citer dans les tableaux récapitulatifs.

9.1 Conversion d'un pointeur en un pointeur d'un autre type

D'une manière générale, on pourrait penser que ce genre de conversion revient à conserver l'adresse du pointeur initial, en se contentant de modifier la nature de l'objet pointé et, donc l'arithmétique correspondante. En fait, il n'en va pas toujours ainsi, compte tenu de l'existence, sur certaines machines, de contraintes d'alignement, dont nous allons parler ici.

Par ailleurs, l'opérateur de `cast` autorise la conversion d'un pointeur sur un objet constant en un pointeur sur un objet non constant, de sorte qu'il devient possible, par ce biais, de modifier la valeur d'un objet constant ! Nous vous en proposerons un exemple plutôt dissuasif.

9.1.1 Contraintes d'alignement et conversions de pointeurs

Pour des questions d'efficacité, il est fréquent qu'une implémentation impose à certains types de données des contraintes sur leurs adresses. Voici des exemples de situations usuelles :

- alignement des entiers de deux octets sur des adresses paires, ce qui, sur des machines à 16 bits, permet d'accéder en une fois à l'entier correspondant ;
- alignement d'objets de 4 octets sur des adresses multiples de 4, ce qui, sur des machines à 32 bits, permet d'accéder en une seule fois à l'objet correspondant.

Dans ces conditions, l'utilisation comme adresse d'un objet de type donné d'une adresse ne respectant pas ces contraintes d'alignement pose parfois problème. C'est pourquoi la norme autorise qu'une conversion de pointeur puisse modifier l'adresse correspondante, afin que le résultat vérifie toujours la contrainte du nouvel objet pointé.

Par exemple, si on suppose que l'implémentation aligne les `int` sur des adresses paires, avec :

```
char *adc ;
int  *adi ;
```

l'affectation :

```
adi = (int *) adc ;
```

est légale, mais l'adresse figurant dans `adr` pourra être :

- celle de `adc` si cette dernière était paire ;
- celle de `adc` augmentée ou diminuée de un, si cette dernière était impaire, de façon que le résultat soit pair.

Ainsi, le cycle de conversions suivant peut, dans certains cas, modifier de une unité la valeur initiale figurant dans `adc` :

```
adi = (int *)  adc ;
adc = (char *) adi ;
```

D'une manière générale, dans une implémentation donnée, les contraintes d'alignement des différents types d'objets peuvent être classées :

- de la plus faible, c'est-à-dire en fait de l'absence de contrainte ; les caractères sont obligatoirement dans ce cas car tout objet doit pouvoir être décrit comme une succession continue d'octets, c'est-à-dire de `char` ;
- à la plus forte.

La norme impose que, dans le cas où `T` et `U` sont deux types de données tels que la contrainte sur `T` soit égale ou plus forte que la contrainte sur `U`, la conversion de « pointeur sur `T` en pointeur sur U » sera acceptable et qu'elle ne dénaturera pas l'adresse correspondante. Autrement dit, la conversion inverse permettra de retrouver l'adresse d'origine[8].

En particulier, on est toujours certain que les conversions dans le type `char *` ou dans le type générique `void *` ne dénatureront jamais l'adresse correspondante. Ce sont d'ailleurs des pointeurs de ce type qui sont utilisés lorsqu'il s'agit de transmettre l'adresse d'un objet dont ne se préoccupe pas du type (ou dont on ne connaît pas le type).

Remarque

Dans certaines implémentations, les contraintes d'alignement sont paramétrables. Cette souplesse possède une contrepartie notoire : un même code, exécuté sur une même implémentation, peut produire des résultats différents, selon la manière dont il a été compilé ! Par ailleurs, la norme C11 introduit des outils de gestion de ces contraintes d'alignement (voir l'annexe B consacrée aux normes C99 et C11).

9.1.2 Qualifieurs et conversions de pointeurs

Comme indiqué dans la section 2.5, il existe deux types de qualifieurs (voire davantage en cas de pointeurs de pointeurs) concernant les variables de type pointeur :

- le qualifieur appartenant au déclarateur de pointeur lui-même ; il concerne la variable pointeur ; par exemple :

```
int p, const *adi ;    /* adi est un pointeur constant sur des int    */
```

- le qualifieur accompagnant le spécificateur de type dans la déclaration du pointeur ; il concerne l'objet pointé ; par exemple :

```
const int n, *adic ;    /* adic est un pointeur sur des int constants */
                        /* alors que n est un int constant            */
```

8. Compte tenu des technologies actuelles, les contraintes d'alignement sont « emboîtées » les unes dans les autres. Par exemple, on rencontre des alignements sur des multiples de 2, 4, 8... Il est donc assez naturel de satisfaire à la condition dictée par la norme. En revanche, les choses seraient moins simples pour le concepteur du compilateur si, dans une même implémentation, on trouvait, par exemple, à la fois des alignements sur des multiples de 2 et des alignements sur des multiples de 3.

Le qualifieur d'une variable pointeur joue le même rôle que celui des variables usuelles ; il n'intervient donc pas dans les conversions et il ne fait pas partie du nom de type correspondant.

Le qualifieur de l'objet pointé, en revanche, fait partie intégrante du nom de type et il intervient donc dans l'opérateur de `cast`. C'est ainsi qu'il est possible de réaliser des conversions :

- de `int *` en `const int *`, c'est-à-dire de « pointeur sur int » en « pointeur sur int constant » ;
- de `const int *` en `int *`, c'est-à-dire de « pointeur sur int constant » en « pointeur sur int ».

La première conversion fait partie des conversions autorisées par affectation et ne présente guère de risque : elle permettra de traiter un entier comme un entier constant, ce qui revient à dire qu'elle interdira certaines affectations. En revanche, la seconde conversion, non autorisée par affectation, n'est à utiliser qu'avec précaution. En effet, elle permettra de traiter un entier constant comme un entier non constant et, par suite, d'en modifier peut-être la valeur. Signalons qu'une telle modification ne sera cependant pas possible dans une implémentation qui place les objets constants dans une zone protégée en écriture puisqu'alors une tentative de modification provoquera une erreur d'exécution.

De façon semblable, il est possible de réaliser des conversions :

- de `int *` en `volatile int *`, c'est-à-dire de « pointeur sur int » en « pointeur sur int volatile » ;
- de `volatile int *` en `int *`, c'est-à-dire de « pointeur sur int volatile » en « pointeur sur int ».

Là encore, la première conversion, déjà autorisée par affectation, ne présente pas de risque particulier, tandis que la seconde doit être utilisée avec précaution.

9.2 Conversions entre entiers et pointeurs

Hormis les conversions déjà autorisées de façon implicite (`NULL` ou 0 en pointeur), les conversions entre entiers et pointeurs ont un caractère relativement désuet et nous ne les exposons que par souci d'exhaustivité.

La norme accepte les conversions d'entier en pointeur et de pointeur en entier. Néanmoins, elle reste floue sur un certain nombre de points.

En particulier, elle laisse à l'implémentation toute liberté dans le choix d'un type entier de taille suffisante pour recevoir le résultat de toute conversion d'un pointeur en un entier, et elle indique que le comportement du programme sera indéfini si l'on tente une conversion d'un pointeur dans un entier trop petit. De plus, quand l'entier est de taille suffisante, elle prévoit que la valeur obtenue dépend de l'implémentation.

En ce qui concerne les conversions inverses, c'est-à-dire d'entier en pointeur, la norme se contente de préciser que le résultat dépend de l'implémentation ; autrement dit, il ne peut y avoir de comportement indéfini dans ce cas.

D'une manière générale, nous conseillons de n'utiliser ce genre de conversions que dans des circonstances exceptionnelles.

9.3 Récapitulatif concernant l'opérateur de cast dans un contexte pointeur

Cette section n'apporte pas d'éléments nouveaux. Elle récapitule simplement tout ce qui concerne l'opérateur de `cast`, utilisé dans un contexte pointeur, y compris certains éléments qui seront examinés en détail au chapitre 8.

Tableau 7.9 : les conversions autorisées par cast dans un contexte pointeur

Type initial	Type résultant	Remarques
Pointeur sur un objet	Pointeur sur un objet de type quelconque	Si la contrainte d'alignement du type résultant est supérieure à celle du type initial, l'adresse obtenue peut être différente de l'adresse initiale.
Pointeur sur une fonction	Pointeur sur une fonction de type quelconque	– possibilités étudiées à la section 11.7 du chapitre 8 ; – adresse toujours conservée ; – effet indéterminé si le pointeur résultant est utilisé pour appeler une fonction d'un type différent (en pratique, conséquences usuelles de non-correspondance d'arguments).
Pointeur	Entier	Résultat dépendant de l'implémentation (si taille entier insuffisante → comportement indéfini)
Entier	Pointeur	Résultat dépendant de l'implémentation