Le guide complet du langage C

Claude Delannoy

EYROLLES

 ISBN : 978-2-212-14012-5

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