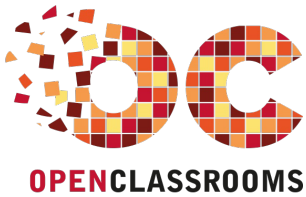
La vérité sur les tableaux et pointeurs en C

Par Marc Mongenet



www.openclassrooms.com

Dernière mise à jour le 16/06/2011

Sommaire

La vérité sur les tableaux et pointeurs en C

Par Marc Mongenet
Mise à jour : 16/06/2011
Difficulté : Difficile Durée d'étude : 3 jours

Explications détaillées sur les tableaux et pointeurs en C.
Sommaire du tutoriel :

Introduction

Bienvenue

Ainsi, tu es à la recherche de la vérité sur les tableaux et pointeurs en C?
Mais laisse-moi d'abord deviner ce qui te conduit à lire ce tutoriel...

Peut-être as-tu présomptueusement affirmé qu'un tableau est un pointeur? Et quelqu'un qui connait la vérité t'aura dirigé ici...

Ou alors tu auras essayé de passer un tableau de tableaux en argument d'une fonction qui attend un pointeur de pointeur, comme ceci:

Code : C

```
void f(int **ppint);

void g(void) {
 int tab[3][4];
 f(tab); /* grossière erreur ! */
}
```

Et le compilateur aura émis un avertissement que tu ne comprends pas:

Code : Console

```
ex1.c: In function 'g':
ex1.c:5:2: attention : passing argument 1 of 'f' from incompatible pointer type
ex1.c:1:6: note: expected 'int **' but argument is of type 'int (*)[4]'
```

Et tu auras peut-être même fait fi de l'avertissement du compilateur, lancé ton programme, et il plante lamentablement à la première utilisation de `ppint` dans la fonction `f`...

On m'aurait menti?

Oui, les tutoriels mentent, car les débutants ne pourraient pas affronter la vérité toute nue sur les tableaux et les pointeurs.

Mais pour commencer, revoyons ce qui est à l'origine des malentendus, c'est-à-dire le pieux mensonge enseigné aux débutants. Il est généralement écrit dans des termes ressemblant à ceux-ci: «Un tableau se comporte comme un pointeur constant sur son élément initial.»

Notons d'abord qu'un (bon) tutoriel ne prétendra jamais qu'un tableau *est* un pointeur, mais qu'il se *comporte comme*, ou *se convertit en*, etc. Affirmer qu'un tableau *est* un pointeur est abusif. 

Récapitulation de ce qui devrait être déjà connu

Revoyons en quoi «un tableau se comporte comme un pointeur constant sur son élément initial»:

Code : C

```c
#include <stdio.h>
int main(void) {
    int tab[3];
    int *p;
    p = tab;
    p = tab + 2;
    tab[1] = 5;
    p[-1] = 6;
    printf("hello, world\n");
/* tab = p; /* ne compile pas */
    return 0;
}
```

Explications ligne par ligne
`int tab[3];`
Un tableau de 3 `int` appelé `tab` est défini.

`int *p;`
Un pointeur de `int` appelé `p` est défini.

`p = tab;`
`tab` se comporte comme un pointeur sur son élément initial, qui est affecté à `p`. Le pointeur `p` pointe donc sur `tab[0]`.

Essaie de deviner la suite des explications avant d'ouvrir la partie secrète qui suit...

Secret (cliquez pour afficher)

`p = tab + 2;`
`tab` se comporte comme un pointeur sur son élément initial, `2` est ajouté à ce pointeur (arithmétique de pointeur), et le résultat est affecté à `p`. Le pointeur `p` pointe donc sur `tab[2]`.

`tab[1] = 5;`
L'instruction `tab[1] = 5;` est équivalente à `*(tab + 1) = 5;`, car par définition `T[N]` est équivalent à `*((T)+(N))`. `tab` est donc converti en un pointeur sur son élément initial, puis `1` est ajouté (arithmétique de pointeur), puis le pointeur résultant est déréférencé pour affecter `5` à l'objet pointé.

`p[-1] = 6;`
C'est le cas simplifié de `tab[1] = 5;` car p est un pointeur, il n'y a même pas besoin de conversion. À noter que malgré l'indice négatif, on pointe toujours dans le même tableau car `p` pointe sur `tab[2]`.

`printf("hello, world\n");`
La chaine `"hello, world\n"` est un tableau de 14 `char`. Ce tableau se comporte comme un pointeur sur son élément initial, soit la lettre 'h'. Ce pointeur est passé à la fonction `printf`, qui attend justement un pointeur de `char`.
Eh oui, au cas où tu ne t'en serais pas rendu compte, on tombe sur une conversion de tableau en pointeur dès le *hello world*!

`tab = p;`
Cette ligne ne compile pas car `tab` est constant. Il est donc impossible d'affecter une valeur à `tab`.

Déclaration de tableau et pointeur

Ce chapitre repasse en revue ce qui est mis en mémoire lorsqu'on déclare un tableau, et lorsqu'on déclare un pointeur. Ça devrait être connu, mais sait-on jamais.

Quand on déclare `int t1[3];` on obtient trois `int` en mémoire, et rien d'autre.
Quand on déclare `int *p1;` on obtient un pointeur de `int` en mémoire, et rien d'autre.
Quand on déclare `int t2[3][4];` on obtient trois tableaux de quatre `int`, soit douze `int` en mémoire, et rien d'autre.
Quand on déclare `int **p2;` on obtient un pointeur de pointeur de `int` en mémoire, et rien d'autre.
Quand on déclare `int *t3[3];` on obtient trois pointeurs de `int` en mémoire, et rien d'autre.
Quand on déclare `int (*p3)[3];` on obtient un pointeur de tableau de 3 `int` en mémoire, et rien d'autre.

Initialisation et représentation en mémoire

Logiquement (si si, c'est logique, les trucs illogiques, c'est pour les derniers chapitres), on peut écrire:

Code : C

```
int t1[3] = { 1, 2, 3 };
int *p1 = &t1[0];
int t2[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
int **p2 = &p1;
int *t3[3] = { &t1[1], &t2[0][0], &t2[1][2] };
int (*p3)[3] = &t2[3];
```

Voici ce que ça pourrait donner en mémoire, avec des `int` de 4 bytes et des pointeurs de 8 bytes:

Secret (cliquez pour afficher)

`int t1[3] = { 1, 2, 3 };`

Code : Autre

```
adresses ...|8   |12  |16  |20
données  ...|0001|0002|0003|...
```

`int *p1 = &t1[0];`

Code : Autre

```
adresses ...|24      |32
données  ...|00000008|...
```

`int t2[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};`

Code : Autre

```
adresses ...|32  |36  |40  |44  |48  |52  |56  |60  |64  |68  |72  |76  |80
données  ...|0001|0002|0003|0004|0005|0006|0007|0008|0009|0010|0011|0012|...
```

`int **p2 = &p1;`

Code : Autre

```
adresses ...|80      |88
données  ...|00000024|...
```

`int *t3[3] = { &t1[1], &t2[0][0], &t2[1][2] };`

Code : Autre

```
adresses ...|88      |96      |104     |112
```

```
données  ...|00000012|00000032|00000052|...
```

```
int (*p3)[3] = &t2[3];
```

Code : Autre

```
adresses ...|112     |120
données  ...|00000068|...
```

Enfin, on ne peut évidemment pas écrire ce qui suit, puisqu'un tableau n'est pas un pointeur, et un pointeur n'est pas un tableau:

Code : C

```c
int t1[3] = 0;         /* FAUX */
int *p = { 1, 2, 3 }; /* FAUX */
```

Déclarations externes

Comme un tableau n'est pas un pointeur, lorsqu'on déclare un tableau ou un pointeur sans le définir (avec le mot clé **extern**), il faut que la définition soit respectivement celle d'un tableau ou d'un pointeur. Sinon, apparaît un bogue qui révèle bien les différences de représention mémoire entre tableau et pointeur.

Observons le comportement du programme bogué suivant, composé des fichiers source `main.c` et `text.c`:

Code : C

```c
/* main.c */
#include <stdio.h>

extern char *p1;
extern char tab1[];
extern char p2[];  /* déclaration de tableau alors que p2 est
défini en pointeur */
extern char *tab2; /* déclaration de pointeur alors que tab2 est
défini en tableau */

int main(void) {
 puts(p1);
 puts(tab1);
 puts(p2);
 puts(tab2);
 return 0;
}
```

Code : C

```c
/* text.c */
char *p1 = "p1";
char tab1[] = "tab1";
char *p2 = "p2";
char tab2[] = "tab2";
```

Code : Console

```
$ gcc main.c text.c -Wall
$ ./a.out
```

```
p1
tab1
ÿ@
Erreur de segmentation
$
```

Les deux premières chaines s'affichent parfaitement, comme il se doit, puisque les déclarations externes correspondent bien aux définitions.

Pour la troisième chaine, l'affichage est incohérent; en effet, en compilant `main.c` mon compilateur a cru que `p2` est un tableau, et a produit du langage machine qui affiche les bytes constituant ce qui est en fait un pointeur. D'où le `ÿ@` sur mon PC. Selon la norme C, il s'agit d'un comportement indéterminé *(**C99 §6.2.7** All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.)*

Enfin, le `puts(tab2)` crée une erreur de segmentation, car les bytes de la chaine `"tab2"` sont pris pour un pointeur, qui pointe sans surprise à une adresse invalide. Évidemment, encore un comportement indéterminé.

La vérité vraie

La source de la vérité

La vérité sur les tableaux et les pointeurs se trouve actuellement dans le chapitre *6.3.2.1 Lvalues, arrays, and function designators*, paragraphe 3, de la norme internationale ISO/IEC 9899:1999 (C99 pour les intimes):

Citation : C99

> Except when it is the operand of the **sizeof** operator or the unary **&** operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

Eh oui, la vérité nue est en anglais. Ça permet d'amortir le choc. 
Voici une traduction:

Citation : C99 traduit

> Sauf quand elle est l'opérande de l'opérateur **sizeof** ou de l'opérateur unaire **&**, ou est une chaine de caractères littérale utilisée pour initialiser un tableau, une expression de type "tableau de *type*" est convertie en une expression de type "pointeur de *type*" qui pointe sur l'élément initial de l'objet tableau et n'est pas une lvalue. Si le tableau est d'une classe de stockage registre, le comportement est indéterminé.

Le mot clé, c'est le mot expression.
Une expression n'est pas un objet en mémoire, c'est un bout de code source, comme `2+3`, `&i`, ou simplement `tab`.
Quand il lit l'expression `2+3`, le compilateur peut directement calculer le résultat: 5 (de type `int`).
Quand il lit l'expression `&i`, le compilateur génère les instructions machine qui calculent l'adresse de `i` (de type `int*` si `i` est de type `int`).
Quand il lit l'expression `tab`, le compilateur génère les instructions machine qui calculent l'adresse de l'élément initial de `tab` (de type `int*` si `tab` est de type `int[]`). Sauf avec `sizeof`, `&`, et `"chaine d'initialisation"`, mais c'est le chapitre suivant.

Enfin, je rappelle qu'une déclaration n'est pas une expression. Déclarer un tableau ne revient donc évidemment pas du tout à déclarer un pointeur. Quand on écrit `int*p,tab[3];` il y a deux déclarations, mais aucune expression, donc aucune conversion. (Et le premier de classe qui lève la main pour me dire que, si, le `3` tout seul dans `tab[3]` est une expression constante, je lui demanderai de se taire pour ne pas embrouiller les autres.)

Je dois encore signaler que la vérité a un peu évolué. Du temps de C89 (trouvé sur http://flash-gordon.me.uk/ansi.c.txt), c'était:

Citation : C89

> Except when it is the operand of the sizeof operator or the unary & operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with wchar_t, an lvalue that has type "array of type" is converted to an expression that has type "pointer to type" that points to the initial member of the array object and is not an lvalue.

En C89, ce ne sont pas toutes les expressions qui sont converties, mais seulement les *lvalues*. Étant donné qu'un tableau est une lvalue, c'est une différence subtile qui ne se révèle que dans des cas assez tordus. Et pour ne pas décourager les lecteurs qui ont suivi jusqu'ici, je n'illustrerai cette différence qu'au dernier chapitre du tutoriel.

Bon, maintenant que la vérité est dévoilée, j'espère que tout est devenu beaucoup plus clair. Sinon, des exemples sont donnés dans la suite du tutoriel...

sizeof tab, &tab, et char tab[] = "chaine"

Citation

> Sauf quand elle est l'opérande de l'opérateur **sizeof** ou de l'opérateur unaire **&**, ou est une chaine de caractères littérale utilisée pour initialiser un tableau, une expression de type "tableau de *type*" est convertie en une expression de type "pointeur de *type*" qui pointe sur l'élément initial de l'objet tableau et n'est pas une lvalue.

Comme revu dans l'introduction, lorsqu'une expression de type tableau est l'opérande de l'opérateur d'affectation (`p=tab`), d'addition (`tab+2`), d'appel de fonction (`printf("hello, world\n")`), et de beaucoup d'autres opérateurs, alors elle se comporte comme un pointeur constant sur l'élément initial du tableau. En fait, il n'existe que trois cas où une expression de type tableau se comporte clairement comme un tableau. Ils sont soulignés dans l'extrait de la norme ci-dessus, et illustrés ci-dessous.

sizeof tab

Code : C

```c
#include <stdio.h>

int main(void)
{
    char tab[3];
    char *p = tab;

    printf("sizeof tab vaut %d.\n", (int)sizeof tab);
    printf("sizeof p vaut %d.\n", (int)sizeof p);

    return 0;
}
```

Ce code affiche le texte suivant sur mon système:

Code : Console

```
sizeof tab vaut 3.
sizeof p vaut 8.
```

Il se peut que la taille de `p` soit différente de 8 sur ton système (4 est une valeur courante).

Ce qu'il est important de constater, c'est que la taille du tableau `tab` est différente de celle du pointeur `p`. En effet, lorsque l'opérande de `sizeof` est de type tableau, alors le résultat est la taille du tableau, et pas la taille d'un pointeur sur un élément du tableau.

J'ai écrit `(int)sizeof tab` et `(int)sizeof p` avec conversion en `int`, afin de correspondre au format `"%d"` de `printf`. Sinon il y a un risque de bogue, car `sizeof` crée un entier de type `size_t`. Si le compilateur supporte C99, alors on peut écrire `printf("sizeof p vaut %zu.\n", sizeof p);` sans conversion.

On peut se demander pourquoi, une fonction `size_t my_strlen(char s[]){return sizeof s;}` retourne la taille d'un pointeur de `char`, et pas la taille du tableau contenant la chaine de caractères `s`. La réponse est très simple: parce que dans ce cas, `s` est un pointeur de `char`. C'est une particularité des déclarations de paramètres de fonction sur laquelle je reviendrai plus tard...

&tab

Code : C

```c
int main(void)
{
    int tab[3];
    int *p;
    int **pp;
    int (*pt)[3];

    p = &tab;
    pp = &tab;
    pt = &tab;

    return 0;
}
```

Dans le code ci-dessus, deux des trois affectations de pointeur provoquent un avertissement de la part du compilateur:

Code : Console

```
$ gcc ex3.c
ex3.c: In function ‘main’:
ex3.c:8:7: attention : assignment from incompatible pointer type
ex3.c:9:8: attention : assignment from incompatible pointer type
$
```

Seule l'affectation `pt=&tab` ne provoque aucun avertissement. En effet, le type de `pt` est le même que celui de l'expression `&tab` : «pointeur de tableau de 3 `int` », ce qui s'écrit `int(*)[3]` en C.

On remarque en particulier que le type de l'expression `&tab` n'est pas `int**` . Ce serait le cas si l'expression `tab` était convertie en pointeur de `int` (soit `int*`) avant l'opération unaire `&` . Mais ce n'est pas le cas; le type de `&tab` est donc un pointeur sur un tableau, et pas un pointeur sur un pointeur.

Attention à la paire de parenthèses en notant le type `int(*)[3]` . Si on écrit `int*[3]` , cela signifie «tableau de 3 pointeurs de `int` », car l'opérateur `*` a une précédence inférieure à l'opérateur `[]` .

char tab[] = "chaine"

Code : C

```c
int main(void)
{
    char *p = "hello";
    char t1[] = "world";
    char t2[] = p;              /* FAUX */
    char t3[] = (char*)"foo"; /* FAUX */

    return 0;
}
```

Pourquoi les initialisations de `t2` et `t3` ne compilent pas, alors que celle de `t1` est parfaitement valable?
Car un tableau ne peut pas être initialisé avec une valeur scalaire, or un pointeur est une valeur scalaire, et les initialisateurs de `t2` et `t3` sont des pointeurs.
En revanche, la chaine `"world"` qui sert à initialiser `t1` n'est pas convertie en pointeur. La chaine `"world"` est de type tableau de 6 `char` , et l'initialisation fait de `t1` un tableau de 6 `char` . C'est équivalent à `char t1[]={'w','o','r','l','d',0};`
Enfin pour initialiser `p` , le tableau `"hello"` est classiquement converti en un pointeur sur son élément initial.

Pour être propre, il faudrait écrire `const char *p = "hello";` mais ça compliquerait inutilement les explications.

char tab[] et char *P

Note: Cette partie est inspirée du topic *Différence entre char *tab et char tab[]* d'Arthurus.

Code : C

```
char t1[] = { 'h', 'e', 'l', 'l', 'o', 0 };
char t2[] = "hello";
char *p = "hello"; /* BOF */

t1[0] = 'H'; /* OK */
t2[0] = 'H'; /* OK */
p[0] = 'H';  /* FAUX */
p = t1;      /* OK */
p[0] = 'H'   /* OK */
```

Comme on l'a vu précédemment, `t1` et `t2` sont des tableaux de 6 `char`. On peut librement changer la valeur de leurs éléments. Mais pourquoi l'instruction commentée `/* FAUX */` est-elle fausse, alors qu'elle compile sans avertissement ?

C'est faux car dans la déclaration `char *p = "hello";` il y a deux objets en mémoire: le pointeur `p`, et un tableau constant anonyme qui contient la chaine `"hello"`. Les tableaux anonymes sont courants, on en trouve par exemple un dans `printf("hello, world\n")`...

Voilà ce qui arrive avec `char *p = "hello";`: le compilateur crée un tableau de `char` statique quelque-part, et ce quelque-part peut être une zone de mémoire où les modifications sont interceptées par le système d'exploitation. Et en cas d'interception, un système comme Linux écrit «erreur de segmentation» et stoppe le programme intercepté.

Bien sûr, le pointeur `p` lui-même est modifiable, comme le montre l'expression `p = t1`.

Pourquoi ai-je commenté la déclarationde `p` avec `/* BOF */` ? Car lorsqu'on pointe vers un objet que l'on n'est pas censé modifier, il vaut mieux utiliser le mot clé **`const`** afin que le compilateur émette un message en cas de tentative de modification de l'objet pointé.

Code : C

```
const char *p = "hello";  /* bien */
p[0] = 'H';   /* FAUX et message: erreur: assignment of read-only
location '*p' */
```

On peut encore noter que le compilateur a le droit de réutiliser les chaines littérales. Par exemple le programme suivant ne nécessite qu'une seule chaine `"hello, world\n"` en mémoire:

Code : C

```
#include <stdio.h>

int main(void) {
 printf("hello, world\n");
 printf("hello, world\n");
 printf("world\n");
 return 0;
}
```

Accessoirement, la réutilisation signifie que si la chaine littérale n'est pas protégée contre l'écriture, la modifier peut introduire des bogues plus subtils qu'une erreur de segmentation.

lvalue et register

Citation

Sauf quand elle est l'opérande de l'opérateur **sizeof** ou de l'opérateur unaire **&**, ou est une chaîne de caractères littérale utilisée pour initialiser un tableau, une expression de type "tableau de *type*" est convertie en une expression de type "pointeur de *type*" qui pointe sur l'élément initial de l'objet tableau et n'est pas une lvalue. Si le tableau est d'une classe de stockage registre, le comportement est indéterminé.

lvalue

Sous ce terme technique se cache une réalité simple: un tableau ne peut pas être affecté, autrement dit, il ne peut pas être la valeur à gauche *(left value)* d'une affectation. Et bien sûr, il ne peut pas non plus être incrémenté ni changé d'une quelconque manière.

Pour être tout à fait rigoureux, à la base, un tableau est une lvalue non modifiable. Et la conversion qui donne un pointeur sur l'élément initial produit une simple valeur, qui n'est pas une lvalue.

Code : C

```c
void f(void)
{
    int t1[3], t2[3], *p;
    t1 = 0;  /* erreur */
    t1 = t2; /* erreur */
    t1++;    /* erreur */
    ++t1;    /* erreur */
    t1--;    /* erreur */
    --t1;    /* erreur */
    t1 += 1; /* erreur */
    t1 -= 1; /* erreur */
}
```

Notons que si l'on remplace `t1` par `p` dans chaque instruction ci-dessus, alors il n'y a plus d'erreur, car `p` est une lvalue modifiable.

register int t[3]

Ce cas est exotique. Il faut savoir qu'un objet de classe de stockage registre n'existe pas forcément en mémoire : il peut être entièrement contenu dans un registre du processeur. Prendre l'adresse d'un objet de classe de stockage registre n'a donc pas de sens déterminé. Cela rend donc les règles de conversion de tableau en pointeur indéterminée...

Un petit test avec GCC donne:

Code : C

```c
/* ex.c */
#include <stdio.h>

int main(void)
{
 register int t[3] = { 1, 2, 3 };
 printf("%d\n", (int)sizeof t);
 printf("%d\n", t[1]);
 return 0;
}
```

Code : Console

```
$ gcc ex.c -pedantic
ex.c: In function ‘main’:
ex.c:8:18: attention : ISO C forbids subscripting ‘register’ array
$
```

GCC est plutôt permissif, puisque sans l'option `-pedantic`, il compile ce source. Toutefois, selon la norme, la seule chose bien définie qu'on puisse faire avec un tableau **`register`**, c'est prendre sa taille avec **`sizeof`** *(C99 §6.7.1.4 note 103: Thus, the only operator that can be applied to an array declared with storage-class specifier register is sizeof.)*

Pointeur de pointeur et tableau de tableaux

Cette sous-partie explique en détail le bogue présenté en introduction: ce qui arrive lorsqu'on passe un tableau de tableaux à une fonction attendant un pointeur de pointeur.

Note: Si tu veux voir des fonctions non boguées manipulant des tableaux, tableaux de tableaux, pointeurs sur tableaux, statiquement et dynamiquement, avec de jolis schémas, alors je t'invite à lire le tutoriel Tableaux, pointeurs et allocation dynamique d'Uknow.

Revenons à nos moutons, voici un code source bogué:

Code : C

```c
/* ex8.c */
#include <stdio.h>

void f(int **p, int x, int y)
{
 if (p[x][y] == 42)
  puts("coucou");
}

int main(void)
{
 int t[2][3] = { { 0, 1, 2 }, { 3, 4, 5 } };
 f(t, 0, 1);
 return 0;
}
```

Compilons-le:

Code : Console

```
$ cc ex8.c
ex8.c: In function 'main':
ex8.c:13:2: attention : passing argument 1 of 'f' from incompatible pointer type
ex8.c:4:6: note: expected 'int **' but argument is of type 'int (*)[3]'
$
```

Mmmh, un avertissement... soyons fou, lançons-le:

Code : Console

```
$ ./a.out
Erreur de segmentation
$
```

D'où vient ce bon gros plantage? Quelle instruction cause une erreur de segmentation?

Il s'agit de **`if`** `(p[x][y] == 42)`. Voyons plus précisément ce qui se passe (sortez vos crayons, il va falloir prendre des notes):

- à l'appel `f(t, 0, 1)` on passe un pointeur sur l'élément initial de `t`, soit un pointeur sur t[0], qui est un tableau de 3 `int`. Or la fonction attend un pointeur sur un pointeur de `int`, pas un pointeur sur un tableau de 3 `int`. Ceci cause l'avertissement du compilateur.
- Arrivé dans `void f(int **p, int x, int y)` on a `p`, un pointeur de pointeur de `int`, qui pointe en fait sur le tableau `t[0]` initialisé avec `{ 0, 1, 2 }`.
- Quand on écrit `p[x][y]` (équivalent à `*(p+x)[y]`), l'exécutable charge d'abord `p[x]`.

- Je rappelle que si `p` est de type `int**` , alors `p[x]` est de type `int*` .
  - Admettons que `p[0]` soit chargé
    - Admettons que **`sizeof`**`(int)` égale **`sizeof`**`(int*)` , alors ce qui est chargé est le `0` de `{0, 1, 2}` , ce qui donne le pointeur nul!
    - Admettons que **`sizeof`**`(int)` égale la moitié de **`sizeof`**`(int*)` , alors ce qui est chargé est le `0` et le `1` de `{0, 1, 2}` , ce qui donne un pointeur qui a fort peu de chance d'être valide!
  - Admettons donc que `p[0]` soit nul. Alors la suite de l'évaluation de l'expression `p[x][y]` évalue `((int*)0)[y]` , soit `*(((int*)0)+y)` . Et le déréférencement de ce pointeur cause sans surprise une erreur de segmentation. CQFD 

Paramètres formels: char*argv[] vs char**argv

Pour connaître toute la vérité sur les tableaux et les pointeurs, il faut encore connaître le cas des déclarations de paramètre de fonction. Dans ce cas, lorqu'on utilise la syntaxe de déclaration de tableau, on déclare en fait un pointeur!
Il s'ensuit qu'il est impossible de déclarer un paramètre formel de type tableau.

Dans une déclaration de paramètre de fonction, une paire de crochets à droite de l'identificateur déclare un pointeur. S'il y a une expression constante entre les crochets, alors elle est ignorée. J'avais prévenu dans un précédent chapitre que l'on sortirait de la logique sur la fin... On dira qu'il y a des raisons historiques à ce bazar.

Comme démonstration, voici quatre prototypes apparemment différents pour la même fonction. On constate que le compilateur ne dit rien, car en fait ils sont tous équivalents:

Code : C

```c
/* c.c */
void f(int *p);
void f(int p[]);
void f(int p[3]);

void g(int *p) { f(p); }
void f(int p[8]) { g(++p); }
```

Code : Console

```
$ gcc -c -Wall c.c
$
```

On remarque en outre que `p` est bien un pointeur dans `f`, et l'on peut donc logiquement l'incrémenter.

Les déclarations bien connues
`int main(int argc, char *argv[])`
et
`int main(int argc, char **argv)`
sont donc parfaitement équivalentes.

Il faut ensuite noter que cette règle n'est pas récursive:

Code : C

```c
void h(int p[2][3]);
void h(int p[][3]);
void h(int (*p)[3]);
/* void h(int (*p)[4]); /* incompatible */
/* void h(int **p); /* incompatible */
void h(int (*)[]); /* compatible, pointeur de tableau de type
incomplet */
```

Le cas du tableau de type incomplet est compatible. Je pense que le type est complété par les autres déclarations. Toutefois, si cela semble le cas avec GCC, Clang rend le type incomplet en vidant le sens des précédents prototypes. On arrive un peu aux

limites des compilateurs.

C89 et C99

Le dernier pour la fin... La différence entre les normes C89 et C99:

Citation : C89

> ...an lvalue that has type "array of type" is converted to an expression that has type "pointer to type"...

Citation : C99

> ...an expression that has type "array of type" is converted to an expression that has type "pointer to type"...

Pour comprendre cette différence, il faut reprendre la définition d'une lvalue selon C89:

Citation : C89

> An lvalue is an expression (with an object type or an incomplete type other than void) that designates an object.

Je ne connais qu'un cas où un tableau n'est pas une lvalue: lorsqu'une structure contenant un tableau est retournée par valeur par une fonction.

Code : C

```c
/* lvalue.c */

#include <stdio.h>

struct s {
 int tab[3];
 const char *str;
};

struct s f(void)
{
 struct s res = { { 1, 2, 3 }, "foo" };
 return res;
}

int main(void)
{
 printf("%d\n", f().tab[0]);
 printf("%c\n", f().str[0]);
 return 0;
}
```

Code : Console

```
$ gcc lvalue.c -Wall -std=c89 -pedantic
lvalue.c: In function ‘main’:
lvalue.c:18:24: attention : ISO C90 interdit d'indicer de tableau n'étant pas membr
$ gcc lvalue.c -Wall -std=c99 -pedantic
$ gcc lvalue.c
```

Comme on le voit avec le compilateur GCC, ça ne compile pas en C89 (ou C90, c'est un peu pareil); mais ça compile en C99. Et en C89, seul le tableau `tab` pose problème, pas le pointeur `str` .

A noter que ça compile aussi sans option de compilation; c'est une des nombreuses extensions de GCC activées par défaut.