**Université Paris 13**
**Institut Galilée**
**Licence 1ère année**
**2006-2007**

# Programmation Impérative
# Polycopié de cours n° 3

**C. Recanati**

**L.I.P.N.**

**http : //www-lipn.univ-paris13.fr/~recanati**

# Table des matières

**8 STRUCTURE DES PROGRAMMES ..... 3**

8.1 LES PARAMETRES DE LA FONCTION MAIN ..... 3
8.2 VARIABLES GLOBALES (OU EXTERNES) ..... 4
8.3 DEFINITION ET DECLARATION DE VARIABLES GLOBALES ..... 6
8.4 CLASSES DE VARIABLES ..... 7
8.5 PORTEE DES VARIABLES ET COMPILATION SEPAREE ..... 8

**9 POINTEURS ..... 9**

9.1 DEFINITION DE VARIABLES DE TYPE POINTEUR ..... 9
9.2 INITIALISATION DES POINTEURS ..... 11
9.3 OPERATEUR * (OPERATEUR DE DEREFERENCE) ..... 15
9.4 PASSAGE DE PARAMETRE PAR REFERENCE ..... 15

**10 POINTEURS ET TABLEAUX (A UNE DIMENSION) ..... 18**

10.1 OPERATIONS ARITHMETIQUES SUR LES POINTEURS ..... 18
10.2 IDENTIFICATEUR DE TABLEAU ET CONSTANTE DE TYPE POINTEUR ..... 21
10.3 LES VARIABLES POINTEURS COMME PARAMETRES DE FONCTION ..... 22
10.4 TABLEAU (OU CHAINES) DE CARACTERES ET POINTEUR SUR DES CARACTERES ..... 23
*10.4.1 Les chaînes de caractères ..... 23*
*10.4.2 La bibliothèque `<string.h>` ..... 24*

**11 POINTEURS ET TABLEAUX A PLUSIEURS DIMENSIONS ..... 28**

11.1 TABLEAUX A DEUX DIMENSIONS ..... 28
11.2 TABLEAU DE POINTEURS ..... 30

# 8 Structure des programmes

## 8.1 Les paramètres de la fonction main

L'exécution d'un programme C commence par un appel à la fonction `main`.

Il est possible de passer des arguments à la fonction `main` en les spécifiant (sous forme de chaîne de caractères) à la console, lors du lancement de la commande exécutable compilée à partir du programme.

Le prototype de la fonction `main` peut donc être celui-ci[1] :

```
int main(int argc, char *argv[]) ;
```

`argc` : nombre de paramètres effectifs (`argc` pour *argument count*)

`argv` : tableau des paramètres effectifs sous forme de chaînes de caractères (`argv` pour *argument vector*).

Comme nous le verrons bientôt, la déclaration du type de `argv` est en fait équivalente à `char argv[][]`, c'est-à-dire que l'on a un tableau dont les éléments sont des chaînes de caractères. Les noms des paramètres formels de la fonctions main ne peuvent pas être modifiés et sont donc nécessairement `argc` et `argv` si l'on déclare deux paramètres.

Ainsi, si la commande de lancement du programme est

```
prog  arg1 toto 10 arg4
```

la fonction `main` sera exécutée avec les valeurs suivantes :

```
argc == 5, argv[0]== ″prog″ ,
argv[1]== ″arg1″,argv[2]== ″toto″ ,argv[3]== ″10″,
arg[4]== ″arg4″.
```

Si on souhaite récupérer des arguments sous une forme autre que celle de chaîne de caractères, il faudra utiliser des fonctions de conversion (`atoi` par exemple, pour *ascii to integer*, qui convertit une chaine de caractères en entier) ou la fonction `sscanf` qui permet de lire une chaîne de caractères de manière formattée. De cette manière, on pourra traduire des chaînes comme `″10″` en entier.

**Exemple**

```
#include <stdio.h>
int main(int argc,char *argv[]) {
/* Afficher les paramètres effectifs */
int i ;
for(i=0;i<argc;i=i+1)
  printf(″Paramètre effectif %d : %s\n″ , i, argv[i]);
```

[1] En réalité, on peut passer un 3ème argument supplémentaire permettant d'avoir des informations sur les variables de l'environnement dans lequel sera interprété le lancement de la commande du programme.

```
return(0) ;
}
```

A l'exécution si on lance l'exécutable `affiche` suivi des arguments : `numero 222`

```
affiche numero 222
```

On aura l'affichage suivant :

```
Paramètre effectif 0 : affiche
Paramètre effectif 1 : numero
Paramètre effectif 2 : 222
```

### 8.2 Variables globales (ou externes)

Rappelons qu'une variable globale (ou externe) est une variable définie en dehors de tout bloc, c'est-à-dire en dehors de tout bloc de définition de fonction, et en particulier, en dehors de la fonction `main`. En C, toutes les fonctions sont des variables globales, car, (contrairement à beaucoup d'autres langages, comme Pascal, java, etc.), on ne peut pas définir de fonctions à l'intérieur du bloc d'une autre fonction pour en restreindre la portée. Toutes les fonctions C sont donc définies dans un fichier au même niveau que la fonction `main` et sont des variables globales.

Mais on peut aussi déclarer des variables globales d'autres types, pas nécessairement fonction, à ce niveau le plus haut. Ces variables (globales) sont accessibles à l'intérieur du corps de toutes les fonctions qui les déclarent ensuite[2], avec la mention **`extern`**, parmi leurs variables locales. Elles peuvent ainsi jouer un rôle de communication entre fonctions.

C'est ce qui se passe dans le programme qui suit. Ce programme lit sur la console des lignes une à une, et recopie dans un tableau nommé `lapluslongue` la chaîne qu'il vient de lire si elle se trouve être la plus longue de celles qui ont été lues jusqu'alors. A la fin, il imprime la chaine la plus longue qu'il a rencontré en imprimant ce tableau.

```
#include <stdio.h>
#define MAXLIGNE 1000

          /* déclarations de variables globales */
int max ;
char ligne[MAXLIGNE] ;         /* la ligne courante */
char lapluslongue[MAXLIGNE] ;  /* la ligne la plus longue
*/

int getligne(void) ; /* lit la chaine ligne[] et
                      * retourne sa longueur */
void recopie(void) ; /* recopie la chaine ligne[] dans
                      * la chaine lapluslongue[] */

int main()
{
      int len ;
      extern int max ;
```

[2] On verra plus loin que cette déclaration n'est pas toujours nécessaire pour que la variable globale soit visible (en paticulier au sein d'un même fichier).

```
        extern char lapluslongue[] ;

        max = 0 ;
        while ((len = getligne()) > 0)
             if (len > max) {
                  max = len ;
                  recopie() ;
             }
        if (max > 0)  /* il y avait au moins une ligne */
             printf("%s", lapluslongue) ;
        return 0 ;
}

/* la fonction getline lit la chaine ligne[] (globale)
 * et renvoie sa longueur
 */
int getligne(void)
{
        int c, i ;
        extern char ligne[] ;

        for (i = 0 ; i < MAXLIGNE – 1
                        &&  (c=getchar()) != EOF
                        &&   c != '\n'           ;  ++i)
             ligne[i] = c ;
        if (c == '\n') {
             ligne[i] = c ;
             ++i ;
        }
        ligne[i] = '\0' ;
        return i ;
}

/* la fonction recopie copie la chaine ligne[] (globale)
 * dans la chaine lapluslongue[] (egalement globale)
 */
void recopie(void)
{       int i ;
        extern char ligne[], lapluslongue[] ;

        i = 0 ;
        while ((lapluslongue[i] = ligne[i]) != '\0')
             ++i ;
}
```

Les variables `ligne` et `lapluslongue` sont des tableaux de caractères déclarées à l'extérieur de toute fonction : ce sont des variables globales. De même, les fonctions `recopie` et `getligne` déclarées[3] au début du fichier, sont des variables globales. Elles

[3] Le mot `void` dans la déclaration des paramètres des fonctions `recopie` et `getligne` est nécessaire pour indiquer que la liste de paramètres sera effectivement vide sans que cela

doivent ici être déclarées en tête de fichier, parce qu'elles sont utilisées dans la fonction `main` avant d'être définies.

Les déclarations `extern` des variables `ligne` et `laplusl ongue` , à l'intérieur des blocs de définition des fonctions `getligne` et `recopie`, permettent de mettre en évidence le fait que ces variables seront modifiées par un appel à ces fonctions (mais, en réalité, ces déclarations sont ici optionnelles). Syntaxiquement, leur déclaration ressemble à une déclaration de variable locale, mais en réalité, aucune allocation mémoire ne sera effectuée pour ces variables, car une adresse et une zone mémoire leur aura déjà été attribuées lors de leur définition en tête de fichier.

Ici, leur déclaration avec mention `extern` dans chaque fonction est optionnelle, car ces variables sont par défaut visibles depuis leur déclaration dans le fichier source où elles figurent jusqu'à la fin de ce fichier. (L'usage est de les déclarer en tête de fichier, et de ne pas les redéclarer ensuite dans les fonctions qui suivent). Ce sont des variables globales qu'on peut qualifier de « globales au programme ». Mais, si leur déclaration est optionnelle à l'intérieur des fonctions figurant dans le même fichier (en réalité, à partir de l'endroit de leur déclaration), il faut par contre les déclarer avec la mention `extern`, et cette fois, de manière obligatoire, si l'on veut qu'elles soient connues dans un autre fichier, ou de manière plus restreinte encore, dans une fonction particulière (définie dans un autre fichier).

### 8.3 Définition et déclaration de variables globales

On distingue donc la **déclaration** d'une variable de sa **définition**. Une définition de variable est une déclaration de variable qui cause, au moment de la compilation du programme, l'allocation d'une zone mémoire (et d'une adresse) pour cette variable. Une variable ne peut avoir qu'une seule adresse et ne doit donc être définie qu'une seule fois.

Dans les programmes n'utilisant qu'un seul fichier, les déclarations sont toujours des définitions, et la distinction n'est pas très pertinente. Mais si l'on fait plusieurs modules de programmes (en définissant des fonctions dans des fichiers séparés), une variable ne doit être définie que dans un seul fichier, et simplement déclarée dans les autres fichiers qui l'utilisent. Pour que ces déclarations extérieures ne soient pas confondues avec leur définition, la déclaration d'une variable dans un module externe est préfixée du qualificatif « `extern` » .

Lorsqu'il rencontre une définition de variable globale, le compilateur alloue à cette variable un emplacement mémoire qui sera constant le temps d'une exécution de programme, mais qui pourra varier d'une exécution à l'autre (lors d'une simple déclaration, il se contente d'enregistrer le type de la variable pour vérifier la cohérence des déclarations).

L'éditeur de liens vérifie ensuite que chaque variable globale n'est bien définie qu'une fois - i.e. que sa définition, s'il s'agit d'une fonction, ne figure que dans un seul fichier, ou bien, s'il s'agit d'une variable d'un autre type, qu'elle n'est déclarée sans la mention « extern » que dans un seul fichier (sa déclaration portant toujours ailleurs la mention « extern »). Si tel est bien le cas, l'éditeur de liens peut alors « effectuer les liens » entre les différents noms des variables déclarées externes, et leurs adresses effectives dans le module de code où elles sont

---

puisse être confondu avec l'ancienne syntaxe du C (qui ne requierait pas la liste des paramètres dans les déclarations de fonctions).

définies. Sinon, le processus de compilation est interrompu car il n'est pas possible de produire un code exécutable cohérent à partir des différents codes objet.

## 8.4 Classes de variables

On a pour l'instant distingué deux sortes de variables : les variables locales (à un bloc), et les variables globales (au programme) — les paramètres formels des fonctions ayant un statut analogue à celui des variables locales. Mais les variables se distinguent principalement par le mode d'allocation de place en mémoire qui leur est associé. On parle alors de **classe de stockage** des variables. Ainsi, il existe quatre qualificatifs qui peuvent être utilisés lors d'une déclaration pour spécifier le mode d'allocation des variables :

**auto** **static** **register** et **extern**

Mais ce sont les termes **auto** et **static** qui définissent réellement la classe de stockage d'une variable (=son mode d'allocation), car le terme **extern** indique simplement que le mode d'allocation est défini ailleurs, et le terme register ne s'emploie en réalité qu'en combinaison avec le terme auto. Il n'y a donc que deux classes grandes de stockage.

Le mode **auto**, pour mode d'allocation « automatique », est le mode par défaut. Sans mention particulière, une variable est de type auto. Cela signifie que son emplacement mémoire lui est attribué dynamiquement à chaque exécution du programme, et que la durée de vie de cette adresse coïncide avec la portée de la variable : elle dure le temps de l'exécution du programme si la variable est une variable globale, et seulement le temps de l'exécution du bloc (ou de la fonction) où elle est déclarée, s'il s'agit d'une variable locale.

A l'inverse, le mode **static** (pour allocation « statique ») est un mode qui alloue un emplacement permanent à la variable. L'adresse d'une variable de type `static` reste la même durant l'exécution de tout le programme, quelle que soit sa portée.

S'il s'agit d'une variable statique locale à un bloc (ou à une fonction), cette propriété fait que la valeur initiale de la variable lors d'une seconde exécution du bloc (ou de la fonction) est la même que celle qu'avait cette variable à la fin de l'exécution précédente. Ce mécanisme est utilisé pour mémoriser des valeurs et éviter le coût d'un passage de paramètre supplémentaire. L'initialisation d'une variable statique au niveau de sa déclaration ne sera exécutée qu'une seule fois en début de programme, et la variable gardera ensuite les valeurs qui lui seront attribuées au cours des différentes exécutions du bloc d'instructions où elle figure.

S'il s'agit d'une variable statique globale définie dans un module (ou fichier) compilé séparément, la portée de cette variable sera alors restreinte à celle qu'elle a dans ce module, et son nom, utilisé dans un autre fichier source, ne fera pas référence à cette variable. Le terme static vient donc dans ce cas particulier, restreindre la portée de la variable au module où elle est définie, et les variables de ce type ne sont pas globales au programme, mais seulement globales au module (ou fichier) dans lequel elles sont définies.

Le terme **register** se combine (uniquement) à auto pour indiquer un usage fréquent de la variable. Le compilateur dans ce cas, essaye d'attribuer un registre interne de l'unité de calcul de l'ordinateur à cette variable (s'il en a suffisamment à disposition le moment venu). On utilise généralement ce qualificatif pour les variables locales de fonctions dont on souhaite optimiser le temps d'exécution.

Enfin, il existe un dernier qualificatif qui peut être utilisé pour indiquer qu'une variable ne changera pas de valeur : le qualificatif **const**. Dans ce cas, la variable déclarée doit être

initialisée au moment de sa déclaration, et cette variable ne pourra être modifiée (le compilateur vérifie qu'elle ne subit jamais d'affectation). Il s'agit donc d'une variable constante et l'intérêt de ce type de déclaration serait bien mince si l'on ne pouvait aussi l'utiliser dans des déclarations de paramètres de fonctions. En effet, dans ce dernier cas, cela permettra au programmeur de se prémunir contre lui-même, s'il manipule des paramètres de type tableaux dont il ne souhaite pas voir les cellules modifiées.

## 8.5 Portée des variables et compilation séparée

Diverses possibilités restreignant la portée des variables existent. Nous savons déjà qu'une variable n'est connue dans un fichier qu'à partir de sa déclaration, et que si elle est déclarée localement dans un bloc, elle n'est alors connue que dans ce bloc. De plus, une variable globale déclarée **static** n'est pas considérée comme globale au programme, mais comme globale à l'unité de compilation (i.e. le fichier considéré) : sa portée est en effet alors restreinte à ce module. Une variable globale de même nom figurant dans un autre fichier ne désigne pas le même emplacement mémoire (ce sera soit une autre variable statique, dont la portée sera limitée à cet autre fichier, soit une variable globale au programme, partagée par d'autres modules l'ayant déclarée extern).

Il existe donc a priori, trois grands niveaux de visibilité de variables. Les variables peuvent être locales (à un bloc ou à une fonction) ; globales à un fichier (ou à une unité de compilation), ce sont les variables globales statiques ; ou globales à tout le programme (c'est-à-dire globales mais non statiques, et déclarées externes dans certaines unités de compilation). A ce dernier niveau, toute variable globale de même nom doit désigner la même entité, même dans des modules de programme compilés séparément. Mais il n'est pas recommandé d'utiliser des variables définies à ce niveau, car rien ne garantit que les différents modules qui les utilisent ont été pensés pour fonctionner ensemble. On essayera donc, dans la mesure du possible, de n'utiliser que des variables globales statiques.

Le tableau de la Figure qui suit récapitule les différentes portées des variables selon leur classe de stockage et leur niveau de déclaration.

| Classe d'allocation | Déclarée (locale) dans une fonction | Déclarée (globale) dans un fichier |
|---|---|---|
| `static` | Portée locale à la fonction | **Portée locale au fichier** |
| [`auto`] ou `register` | Portée locale à la fonction | Portée globale au programme |

Figure 3. Portées des variables selon leur classe d'allocation

Pour rendre les choses plus simples, les variables globales statiques (locales à un fichier) et les autres variables globales (éventuellement externes) seront préférentiellement déclarées en tête de fichier. Les déclarations externes sont généralement regroupées dans un fichier de déclarations ayant une extension en « .h », fichier que l'on pourra inclure, s'il est nécessaire pour la compilation, en tête de fichier avec une directive `#include`.

# 9 Pointeurs

Les pointeurs sont des variables permettant de stocker des adresses mémoire. Les pointeurs sont très utilisés en C car ils permettent un code concis et efficace et sont parfois la seule solution pour l'expression d'un calcul. En particulier :

- On a vu qu'en C, le mode de passage des paramètres des fonctions se fait par valeur (sauf pour les tableaux). D'où la nécessité de passer par un pointeur pour pouvoir passer des adresses de variables en paramètres, dans le but de les modifier. On parle alors parfois, par abus de langage, de *passage de paramètre par référence ou par adresse*, (par opposition au passage de paramètre par valeur). En effet, si c'est l'adresse d'une variable qui est transmise au paramètre d'une fonction, la modification de la valeur contenue à cette adresse dans la fonction engendrera une modification effective de la variable passée en argument. C'est comme si on avait passé directement la variable en paramètre à la fonction. Ce mécanisme est souvent utilisé pour simuler des fonctions renvoyant plusieurs valeurs résultats.

- Les pointeurs sont les outils qui permettent la gestion de suites d'objets en mémoire, et en particulier des tableaux, qui sont constitués d'objets de même taille rangés successivement en mémoire. L'identificateur d'un tableau est en réalité l'équivalent d'un pointeur. (Ce pointeur indique l'adresse de début en mémoire où sont stockés les éléments du tableau). Les pointeurs et les tableaux sont donc intimement reliés et souvent utilisés simultanément.

- Les pointeurs permettent de compenser le fait que la valeur de retour d'une fonction ne peut être de type tableau. En effet, elle ne peut pas être de type tableau, mais elle peut être de type pointeur. On pourra donc retourner un pointeur sur le premier élément d'un tableau, au lieu de retourner ce tableau.

- De la même façon, la valeur de retour d'une fonction ne peut pas être de type fonction. Mais l'identificateur d'une fonction est aussi l'équivalent d'un pointeur. (Ce pointeur indique l'adresse en mémoire de l'espace alloué pour l'exécution du corps de la fonction). On peut donc ici encore, contourner la règle, et renvoyer un pointeur correspondant à l'identificateur d'une fonction définie du programme, pour simuler le renvoi d'une fonction.

## 9.1 Définition de variables de type pointeur

Un **pointeur** est une variable dont la valeur est une adresse de variable d'un type donné. On dit que le pointeur **pointe** vers la valeur rangée à cette adresse. Les pointeurs sont donc typés, et on distingue différents types de pointeurs selon les types de valeurs susceptibles d'être trouvées à l'adresse vers laquelle ils pointent. Ainsi, il existe des pointeurs sur des `int`, des pointeurs sur des `char`, etc. – ce qui signifie que les valeurs rangées à l'adresse fournie par ces pointeurs, seront de type `int`, de type `char`, etc.

La définition d'une variable pointeur est donc une déclaration mentionnant un type, et qui a la forme suivante

*type* `*`*identificateur* `;`

Le fait que la variable est de type pointeur est indiqué par la présence d'une étoile. L'identificateur d'une variable pointeur est un identificateur quelconque. Sa déclaration

indique que la valeur du pointeur sera une adresse où sera stockée une valeur de ce type. On peut accoler l'étoile à l'identificateur ou la séparer avec des blancs. On peut aussi la coller au début au type (au lieu de la coller à l'identificateur). Toutes ces formes sont en réalité équivalentes, car le préprocesseur du compilateur élimine de toutes manières les espaces lorsqu'ils ne sont pas nécessaires comme séparateurs:

*type*`*` *identificateur* `;`

Le type *type* peut avoir été indifféremment défini par l'utilisateur (avec `typedef`) ou prédéfini.

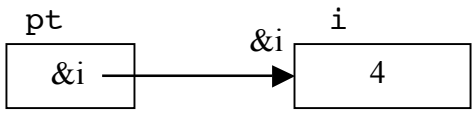
**Exemple**

```
int *pt;                // pt est un pointeur sur un int
char * string ;                 // string est un pointeur sur un char
```

On dira dans ce cas que `pt` est un pointeur sur un `int` et `string` un pointeur sur un `char`, ce qui signifie que la valeur pointée par `pt` est de type `int` et celle pointée par `string`, de type `char`.

La notation pour la déclaration d'un type de pointeur est cohérente avec celle de l'opérateur qui donne accès à la valeur pointée, car cet opérateur, appelé **opérateur de dé-référence** (ou déréférençage), est précisément noté par une étoile. En effet, si l'on a effectué les déclarations et initialisations suivantes

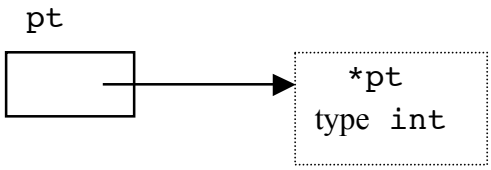
```
int *pt ;
int i=4 ;
pt = &i ;
```




alors, l'expression `*pt` désigne la valeur « pointée » par `pt`, i.e. l'entier de type `int` qui se trouve stocké dans la variable `i`, ici `4`.

Déclarer une variable pointeur revient donc en quelque sorte à déclarer le type de sa valeur pointée (ou « dé-référencée »), c'est-à-dire, celle qui subsiste une fois qu'on aura franchi sa référence (le pointeur – i.e., l'adresse qui y donne accès). Ainsi, quand on déclare

```
int *pt ;
```




l'étoile indique que `pt` (sans étoile) est un pointeur, et le mot `int` devant indique que `*pt` (avec étoile) sera un entier. Mais attention, du point de vue de l'allocation de place en mémoire, le compilateur aura juste réservé la place nécessaire au stockage d'une adresse dans la variable pt, et pas spécifiquement de place pour un entier. Il est donc particulièrement important de bien initialiser les variables de type pointeur, par des adresses mémoires correctement allouées.

Du point de vue des types, si l'on a un entier `j`, et un pointeur `pt` sur un entier (correctement initialisé), on pourra affecter `j` avec la valeur pointée par `pt` :

```
int i = 7, j, *pt;
```

```
pt = &i ; /* initialisation de pt */
j = *pt ;
```

Pour définir un nouveau type pointeur nommé `TypePointeur`, on pourra utiliser une déclaration de type avec `typedef` :

```
typedef   type  *TypePointeur ;
```

`TypePointeur` est alors un nouveau type pointeur vers une zone mémoire pouvant contenir des valeurs de type *type*. `TypePointeur` peut bien entendu être remplacé ici par n'importe quel identificateur valide.

**Exemple**

```
typedef int *PtInt;
PtInt pt;       /* pt est un pointeur sur un entier */
```

### 9.2 Initialisation des pointeurs

Comme nous l'avons indiqué, l'initialisation correcte des pointeurs est une étape fondamentale qui, si elle est oubliée, sera la cause de *bugs* particulièrement difficiles à détecter et à corriger[4]. Il est donc très important de toujours veiller à l'initialisation correcte de ces variables un peu spéciales. Il existe plusieurs méthodes permettant d'initialiser une variable pointeur, que nous allons expliquer en détail.

1ère méthode : on lui attribue l'adresse d'une variable déjà définie grâce à l'opérateur `&` qui donne l'adresse en mémoire centrale de son opérande. (L'opérateur & ne s'applique qu'à des variables). On dit alors que le pointeur « pointe » vers cette variable.

**Exemple**

```
int *po;
int c=10;

po = &c; /* initialisation de po par l'adresse de c */
printf ("valeur de c = %d, valeur pointee = %d", c, *po) ;
```

Une exécution de ces deux lignes afficherait

```
valeur de c = 10, valeur pointee = 10
```

Si on représente la mémoire centrale d'un ordinateur comme un tableau de mots mémoire (d'un octet par exemple) disposant chacun d'une adresse unique, les définitions précédentes de `po` et `c` donneraient par exemple :

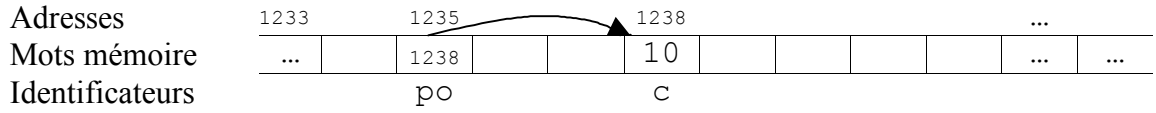
| Adresses | 1233 | | 1235 | | | 1238 | | | | | … | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mots mémoire | … | | ? | | | 10 | | | | | … | … |
| Identificateurs | | | po | | | c | | | | | | |

[4] Le programme pourra « marcher » par hasard, ou « planter » à l'exécution, en un endroit variable et quelconque.

L'instruction

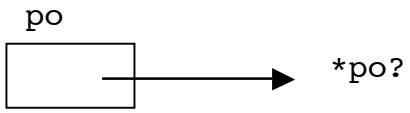
```
po = &c;
```

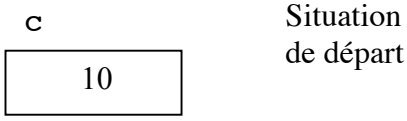
affecte l'adresse mémoire de `c` (ici `1238`) au pointeur `po`. On dit que `po` pointe sur `c`. Le fait d'affecter l'adresse de `c` à `po` peut se représenter comme suit :




Mais on n'a pas besoin en réalité de penser que les adresses sont des entiers, ni pour l'instant que certaines sont consécutives. On peut utiliser des schémas comme le premier que nous avions donné, n'utilisant que des flèches. Les flèches symbolisent alors des valeurs quelconques d'adresses vers lesquelles « pointent » les pointeurs :

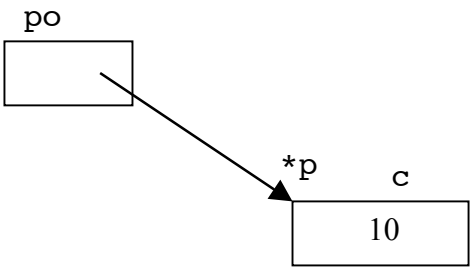
```
int *po;
int c=10;
```




Situation de départ

```
po = &c;
```




après

```
po = &c;
```

2ème méthode : on lui affecte la valeur d'une autre variable pointeur (correctement initialisé) de même type.

**Exemple**

```
int *p,*q;
int c = 2;

p = &c; /* p pointe sur c */
q = p; /* q pointe sur la même adresse que p */
printf ("valeur pointee par p : %d, par q : %d\n", *p,
*q) ;
```

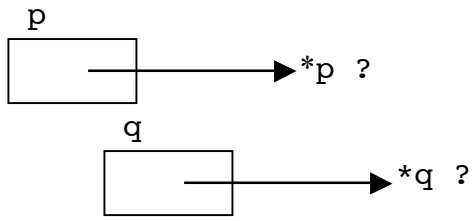
Une exécution de ces lignes afficherait

```
valeur pointee par p : 2, par q : 2
```

On peut se représenter la chose comme suit :

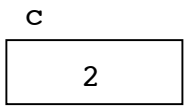
Pour les déclarations

```
int *p,*q;
```



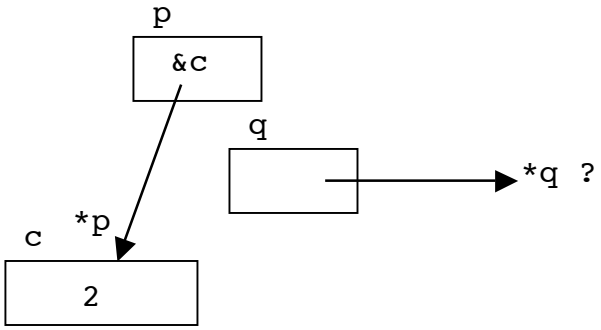

Situation de départ

```
int c = 2;
```




Ensuite, les instructions :

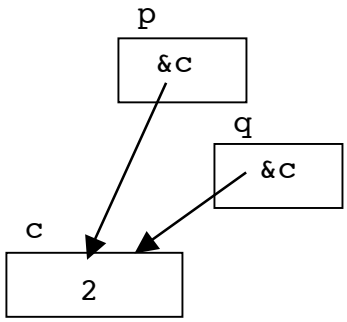
```
p = &c ;
```




Enfin quand q reçoit la valeur de p, il reçoit donc en fait l'adresse de c, ce qui fait qu'avec

```
q = p ;
```

les deux pointeurs pointent vers la même adresse. `*p` et `*q` désignent donc la même valeur, celle de la variable `c`, quelque soit la valeur que l'on aura donné à cette variable. Or, ici, on l'a initialisée à 2, d'où les impressions du programme, puisque l'on a :




**Mais attention :**

Si on initialise une variable pointeur avec l'adresse d'une variable *automatique* dont la durée de vie n'est pas permanente(comme les variables locales à un bloc, ou les paramètres formels des fonctions) , on peut avoir de curieuses surprises, comme le montre le programme suivant :

```
char * initialise_mal() ;
void Passe_Derriere() ;

int main ()
{
      char * pc ;
      pc = initialise_mal() ;
      printf("Avant Passe_derriere, *pc vaut %c\n", *pc) ;
      Passe_derriere() ;
      printf("Apres Passe_derriere, *pc vaut %c\n", *pc) ;
}
```

```
char * initialise_mal()
{     char c ;  /* variable locale allouee dynamiquement */

      c = 'A' ;
      return (&c) ;
}
void Passe_Derriere()
{     char caract ;  /* locale allouee dynamiquement */

      caract = 'Z' ;
}
```

Une exécution de ce programme risque en effet d'afficher les lignes suivantes :

```
Avant Passe_derriere, *pc vaut A
Apres Passe_derriere, *pc vaut Z
```

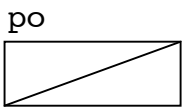
On pourrait corriger cet effet malenconteux en déclarant `static`[5] la variable locale `c` de la procédure `initialise_mal`.

3ème méthode : en le faisant pointer sur rien, noté ici par la constante NULL (adresse vide par définition). Si on cherche ensuite à accéder à sa valeur, on a une erreur à l'exécution.

**Exemple**

```
int *po ;

po = NULL ;          /* po ne pointe sur rien  */
```

`po` ne pointe alors sur rien. La valeur nulle est représentée par la constante `NULL` (définie dans `stdlib.h` par `0`). Sur des schémas, on la représente parfois par une barre oblique :




Une fonction qui imprime la valeur pointée par un pointeur d'entier s'écrira alors dans ce cadre :

```
void printPtVal ( int * pt)
{     if (pt != NULL)
           printf("%d", *pt) ;
      /* sinon, ne rien faire ! */
}
```

[5] Comme on l'a vu, dans ce cas la variable reste locale à la fonction, mais son adresse n'est plus allouée dynamiquement, et elle bénéficie d'un emplacement fixe (statique) pendant toute la durée d'exécution du programme (comme une variable globale). La seule différence avec une variable globale est qu'elle n'est accessible que dans le bloc de définition de la fonction. Sa « portée » est donc restreinte à la fonction, et la fonction `Passe_derriere` serait alors sans effet.

## 9.3 Opérateur * (opérateur de déréférence)

L'opérateur unaire de **déréférence** (ou d'**indirection**), noté *****, dont l'opérande doit être de type pointeur, donne accès à la valeur rangée à l'adresse vers laquelle pointe le pointeur. On parle *d'indirection*, car on accède à cette valeur indirectement (via son adresse), et de *dé-référence*, car on supprime l'intermédiaire qu'est la référence (c'est-à-dire l'adresse) de la variable pointée.

**Exemple :**

```
int *po;      /* po pointe sur n'importe où */
int c=1;      /* une adresse est allouée à c */
            /* et la valeur  1 y est rangée */

po = &c;    /* po pointe sur c et *po == c */
y = *po;    /* la valeur de *po est affectée à y */
            /* donc y vaut 1, car *po == c == 1 */
```

Tant que la valeur de `po` n'est pas modifiée, on peut alors indifféremment utiliser `*po` et `c` pour désigner une valeur dans une expression :

`y = *po + 10;` *est exactement équivalent à* `y = c + 10;`

Les opérateurs de déréférence ('`*`') et d'adresse ('`&`') s'appliquent à des identificateurs de variables. Ils sont associatifs de droite à gauche et ont même niveau de priorité. Cette priorité est élevée, et, en particulier, supérieure à celles de tous les opérateurs numériques et logiques.

Elle est néanmoins inférieure à celle des quatre opérateurs de priorité la plus haute : l'opérateur d'accès aux champs de structures ou d'unions (noté '.'), l'opérateur d'indexation des cellules d'un tableau (noté '`[...]`'), les parenthèses ou appels de fonctions (notés '`(...)`'), ainsi que, comme nous le verrons plus loin, l'opérateur d'accès aux champs de structures ou d'unions à partir d'adresses (noté '`->`'). (Rappelons que l'ordre de priorité de tous les opérateurs du langage C est donné Figure 5 du polycopié de cours n°1, p. 48).

## 9.4 Passage de paramètre par référence

Nous savons que, à l'appel d'une fonction, la valeur du paramètre effectif est transmise au paramètre formel correspondant. Si le paramètre formel est modifié dans le corps de la fonction, cette modification n'affectera pas les variables du programme appelant. C'est le principe de la transmission de paramètres par valeur. Par exemple, si l'on souhaite écrire une fonction qui échange les valeurs des deux variables, mais que l'on écrit naïvement cette fonction `echange` de la façon suivante :

```
#include <stdio.h>
void echange(int x,int y); /* declaration de la fonction */

int main()
{     int a=10, b=5;

      printf("\t a=%d, b=%d\n",a,b);
      echange(a,b);
      printf("\t a=%d, b=%d\n",a,b);
      return(0);
}
```

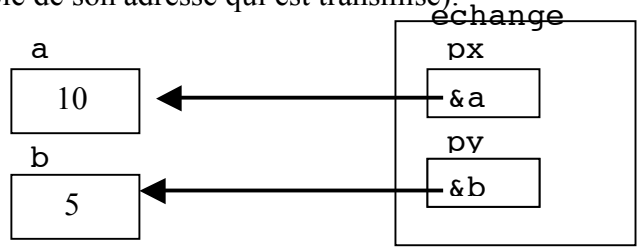
```
/* mauvaise définition d'une fonction qui echangerait */
/* les valeurs de ses deux paramètres d'appels */
void echange(int x,int y)
{     int temp;

      printf("x=%d, y=%d\n",x,y) ;
      temp = x;
      x    = y;
      y    = temp;
      printf("x=%d, y=%d\n",x,y) ;
}
```

Une exécution de ce programme imprimera :

```
a=10, b=5
      x=10, y=5
      x=5, y=10
a=10, b=5
```

On constate que la fonction `echange` n'a pas permis d'échanger les valeurs des variables `a` et `b` qui figurait dans l'appel de fonction à cause du mode de transmission des paramètres, restreint aux valeurs.

Grâce aux pointeurs, un autre mode de transmission de paramètres est possible : c'est le mode de transmission **par référence** (ou **par adresse**). Dans ce nouveau mode, ce qui est passé, ce n'est plus la valeur du paramètre effectif (i.e. une copie de cette valeur), c'est son adresse, c'est-à-dire l'adresse de la zone mémoire où la valeur de cette variable est stockée. Un traitement effectué sur cette adresse donne donc accès à la valeur stockée dans l'originale (c'est la copie de son adresse qui est transmise).




Par défaut en C, un paramètre est passé par valeur. Lorsqu'on veut passer un paramètre par référence (ou par adresse), il faut utiliser des pointeurs. Voici une nouvelle version de la fonction `echange` qui utilise le mode de passage par référence (i.e. par adresse) :

```
void echange(int *px,int *py)
{
      int temp;

      temp = *px;
      *px  = *py;
      *py  = temp;
}
```

La déclaration en tête du programme doit donc être modifiée

```
void echange(int *x,int *y);
```

et dans le main, il faut également modifier l'appel à la fonction car on ne transmet plus les valeurs des variables `a` et `b`, mais leurs adresses :

```
int main()
{
      int a=10, b=5;

      echange(&a,&b);
      printf("a=%d, b=%d\n",a,b);
      return(0);
}
```

Cette fois, une exécution du programme imprimera bien

```
a=5, b=10
```

**Remarque :** Lors d'un appel à la fonction `scanf`, les paramètres effectifs sont les adresses des variables (on utilise donc l'opérateur d'adresse `&`). Nous pouvons maintenant comprendre pourquoi : il faut en effet utiliser un mode de transmission par adresse pour que leur contenu puisse être modifié.

A l'inverse, lorsqu'un tableau est passé en paramètre d'appel d'une fonction, il n'est pas nécessaire d'en passer l'adresse, parce que le passage des variables de type tableau s'effectue toujours par référence. Mais on va voir maintenant qu'un identificateur de tableau est en réalité équivalent à un pointeur (constant). Cette façon de voir rend ce choix du langage (i.e. passer les tableaux par référence) assez cohérent.

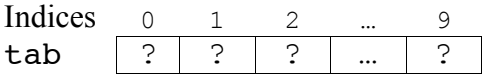
# 10 Pointeurs et tableaux (à une dimension)

## 10.1 Opérations arithmétiques sur les pointeurs

Soit une variable de type tableau `tab` de taille 10 :

```
int tab[10]; /*      les 10 éléments de tab indexés de 0 à
9
                  sont tab[0] tab[1] ... tab[9] */
```

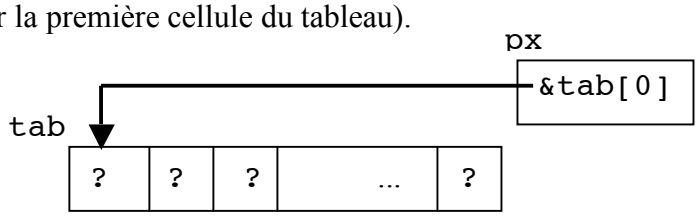
Cette variable occupe en mémoire un ensemble de mots mémoire adjacents qu'on peut se représentés comme suit :




Soit la variable `px` de type pointeur sur un entier, définie comme suit :

```
int *px;

px=&tab[0];
```

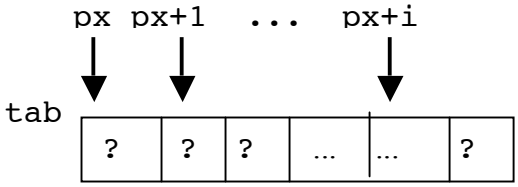
on fait pointer `px` sur la variable de type `int` indicée par `0` dans le tableau `tab` (c'est-à-dire sur la première cellule du tableau).




Il est alors possible d'accéder aux différentes cellules du tableau en incrémentant la variable pointeur `px`. Lorsque l'on écrit `px = px + 1`, ou `px++` la nouvelle valeur de `px` n'est pas l'adresse précédente (comme entier) augmentée d'une unité (ni même d'un octet), mais désigne l'adresse suivante en mémoire permettant de stocker un `int`, c'est-à-dire la valeur numérique précédente augmentée de la taille en mot-mémoire permettant de stocker un `int`.

Ainsi, si `px` pointe sur `tab[0]`, alors
- `px + 1` pointe sur la cellule `tab[1]`
- `px + i` pointe sur la cellule `tab[i]`

et `*(px+i)` a pour valeur le contenu de `tab[i]`




et ceci, indépendamment de la taille du type des cellules du tableau. C'est en partie pour cette raison qu'une variable pointeur n'est pas seulement une adresse mémoire, mais une **adresse mémoire typée**. La valeur d'un pointeur est en quelque sorte, une adresse + un type.

Une variable pointeur `px` sur un réel `x` de type `float` ne se comporte pas comme une variable pointeur `pc` sur un caractère `char c`. En effet `px+1` désigne l'adresse suivante permettant de stocker un flottant après l'emplacement occupé par `x`, et la taille mémoire nécessaire au stockage d'un flottant est toujours plus grande que celle nécessaire au stockage d'un caractère. Ajouter `1` à l'adresse rangée dans `px`, et ajouter `1` à celle rangée dans `pc` n'est pas augmenter les adresses en question d'une même unité (par exemple la taille d'un

mot mémoire) car `px+1` et `pc+1` seront différents, même si `px` et `pc` ont été initialisés par la même valeur d'adresse. On pourrait le vérifier en exécutant le programme suivant :

```
int main()
{
      float *px ;
      char *pc ;
      float x;

      printf("sizeof px = %d,", sizeof(px));
      printf("sizeof pc = %d,", sizeof(pc));
      printf("sizeof x = %d, ", sizeof(x));
      printf("sizeof char = %d\n", sizeof(char));
     printf("adresses: &px=%lu, &pc=%lu, &x=%lu\n",
               &px, &pc, &x);
      px = &x ;
      pc = (char *) &x ;

      printf("valeurs: px=%lu, pc=%lu\n", px++, pc++);
      printf("nouvelles valeurs: px=%lu, pc=%lu\n", px,
pc);
      printf("adresses: &px=%lu, &pc=%lu, &x=%lu\n",
               &px, &pc, &x);
      return 0 ;
}
```

Une exécution (sous MacOS X) :

```
% a.out
sizeof px = 4,sizeof pc = 4,sizeof x = 4, sizeof char = 1
adresses: &px=3221224640, &pc=3221224644, &x=3221224648
valeurs: px=3221224648, pc=3221224648
nouvelles valeurs: px=3221224652, pc=3221224649
adresses: &px=3221224640, &pc=3221224644, &x=3221224648
```

**Mais manipuler les valeurs des adresses comme on l'a fait ici, n'a aucun intérêt.** Au contraire, l'intérêt de cette « pseudo » arithmétique sur les pointeurs, est qu'on peut faire usage des pointeurs dans les tableaux **sans se préoccuper des adresses effectives des objets, ni de la taille mémoire occupée par ces objets**. Les pointeurs permettent en effet un code très concis, car on pourra à la fois manipuler les valeurs et avancer, sans se soucier des tailles des objets, dans des tableaux, grâce aux opérateurs * (de déréfence, à ne pas confondre bien entendu avec la multiplication !), et ++.

Par exemple, la procédure `strcpy` que nous avions écrite sur les tableaux de caractères comporte différentes versions avec pointeurs (ici, de plus en plus simplifiées) :

```
void strcpy (char *s, char *t)      /* 1ere version */
{
      *s = *t ;
      while ( (*t) != '\0') {
           s++ ;          /* on avance d'un cran */
           t++ ;
           *s = *t ;  /* on recopie *t dans *s   */
```

```
        }
}
void strcpy (char *s, char *t)      /* 2eme version */
{
        while ( (*s = *t) != '\0') {
                s++ ;
                t++ ;
        }
}
void strcpy (char *s, char *t)      /* 3eme version */
{
        while ( (*s++ = *t++) != '\0')
                ;
}
void strcpy (char *s, char *t)      /* 4eme version */
{
        while (*s++ = *t++)
                ;
}
```

On peut de la même manière retrancher `1` à un pointeur, ou le décrémenter d'un avec l'opérateur `--`, et le pointeur reculera en quelque sorte d'une cellule dans le tableau. Cela permet de parcourir un tableau dans l'autre sens, en partant de la fin (pour les chaînes de caractères, en partant de l'endroit où est stocké le caractère `'\0'`), mais il faudra prendre garde à ne pas reculer trop et se retrouver dans la mémoire en deçà de l'adresse de début attribuée à ce tableau.

En général, l'opérateur d'incrémentation et l'opérateur * suffisent amplement pour implanter un calcul. Mais la table qui suit récapitule les types des différentes opérations que l'on peut finalement effectuer avec des opérandes de type pointeur (notés `pt` ou `pt'`) ou entier (noté `n`). Il faudra toujours manipuler ces opérations avec précaution.

| Opérateurs | opération | résultat de type |
|---|---|---|
| + | pt + n | pointeur |
| + | n + pt | pointeur |
| - | pt - n | pointeur |
| - | pt – pt' | entier |
| += | pt += n | pointeur |
| -= | pt -= n | pointeur |
| ++ | pt++ ou ++pt | pointeur |
| -- | pt-- ou --pt | pointeur |

Figure 2 : Pointeurs et opérations arithmétiques

*(Le paragraphe qui suit et la remarque suivante pourront être sautés en première lecture).*

Si l'on considère comme on l'a dit, que la valeur d'un pointeur est en réalité une adresse et un type, notée `<a, type>`, on a ainsi :

```
<a, char> + 1 = <a + 1, char>
<a, long> + 1 = <a + 4, long>
```

et plus généralement pour un pointeur sur un type `t` de valeur `a` :

```
<a, t> + n = <a + n*sizeof(t), t>
```

On peut en outre effectuer la différence entre deux pointeurs sur un même type de donnée, mais cette opération n'a réellement de sens que si le premier pointe sur une adresse supérieure à celle pointée par le second. On a en effet :

```
<a, t> - <a', t>  = (a – a')/sizeof(t)
```

ce qui signifie que si les deux pointeurs sont des pointeurs sur les cellules d'un même tableau de type `t`, la différence entre les deux pointeurs indique (à la place prise par un objet de type `t` près) le nombre de cellules du tableau qui les séparent. Un cas intéressant est celui des chaînes de caractères, car pour un tableau de `char`, ou de `unsigned char`, la différence entre deux pointeurs indiquent exactement le nombre de caractères qui les séparent, puisque `sizeof(char)` = `1`.

**Remarque** :

Attention cependant à ne pas confondre les opérations sur les pointeurs et les opérations sur les valeurs pointées. Supposons que l'on ait déclaré :

```
int x=0, y, *px, tab[10];
```

alors, si `px` a été initialisé par l'adresse d'une cellule du tableau, par exemple `&tab[1]`

`y = *px + 1;`  est différent de  `y = *(px + 1) ;`

Dans le premier cas, on affecte à `y` la valeur pointée par `px` augmentée de `1`(donc la valeur de `tab[1]` augmentée de 1). Dans le second cas, on affecte à `y` la valeur située dans la cellule suivante (c'est-à-dire `tab[2]`). De même, il faudra prendre garde à ne pas confondre

`(*px)++;`  et  `*px++ ;`

Le premier cas étant parenthésé, il est clair qu'on retourne la valeur pointée par `px` (c'est-à-dire celle de la 2ème cellule : `tab[1]`) puis on augmente de `1` la valeur rangée dans cette cellule. Dans le deuxième cas, on accède à la valeur pointée par `px` et on la retourne, (on retourne donc la même valeur `tab[1]`), mais on ne modifie pas la cellule correspondante du tableau ; par contre, on avance le pointeur d'un cran, de sorte qu'il pointe maintenant sur la cellule suivante (c'est-à-dire sur `tab[2]`).

### 10.2 Identificateur de tableau et constante de type pointeur

Le lien entre variable pointeur et variable tableau ne se restreint pas aux opérations ci-dessus. En effet, en C, **l'identificateur d'une variable de type tableau est en réalité une constante de type pointeur** (dont la valeur est l'adresse de la première cellule du tableau).

Si `tab` est un tableau, on a les équivalences suivantes entre les expressions :

```
tab[0]    ⇔   *tab
tab[i]    ⇔   *(tab+i)
&tab[0]   ⇔   tab
&tab[i]   ⇔   tab+i
```

et ceci, pour tout `i` compris entre `0` et la taille du tableau, qui, rappelons-le, peut être définie à l'aide de l'opérateur `sizeof`, par :

```
#define TAILLE   sizeof(tab)/sizeof(tab[0])
```

Néanmoins, cette équivalence entre pointeur et tableau a ses limites, car l'identificateur d'une variable de type tableau est une constante**,** ce qui signifie que **sa valeur ne peut pas être modifiée**. Des instructions comme `tab = pt;` avec `pt` pointeur, ou `tab = tab+1;` ou encore `tab++` sont incorrectes, car `tab` ne peut pas être modifié. Mais cette équivalence permet de passer des tableaux à des fonctions prenant en paramètre un type pointeur.

### 10.3 Les variables pointeurs comme paramètres de fonction

Le fait que l'identificateur d'un tableau soit une constante de type pointeur dont la valeur est l'adresse de sa première cellule a une conséquence très importante. Lors d'un appel de fonction, quand une variable de type tableau est paramètre effectif, la valeur qui est copiée dans le paramètre formel (de même type tableau) est le contenu de cette variable, c'est-à-dire l'adresse de la première cellule du tableau. En conséquence, le paramètre de type tableau est passé par référence et non par valeur, ce qui signifie que le paramètre formel et effectif de type tableau pointent sur la même zone mémoire. Ainsi, si la fonction appelée modifie les valeurs d'un paramètre formel de type tableau, alors la modification de ses cellules est effective même à l'extérieur de la fonction.

**Exemple**

Fonction permettant d'initialiser un tableau de réels.

```
#include <stdio.h>
#define TAILLEMAX 100

void init(double*, int*);

int main() {
    double tab[TAILLEMAX];
    int n,i;

/* appel de la fonction init avec le tableau tab
 * et l'adresse de la variable n (qui sera modifiee)
 */
    init(tab,&n);
/* le tableau est maintenant initialise par n valeurs */
/* impression de ces n valeurs */
    for(i=0; i<n; i=i+1)
        printf("%d\t",*(tab+i)) ;
    printf("\n") ;
    return(0);
}

void init(double *pt,int *pn) {
    int i;

    printf("Nombre d'éléments du tableau :");
    scanf("%d",pn);       /*   la valeur de pn est une
adresse,
                          l'opérateur & est donc inutile */
    printf("\nEntrez %d réels: ", *pn);
    for(i=0; i<*pn ; i=i+1)
        scanf("%lf", pt+i);
}
```

## 10.4 Tableau (ou chaînes) de caractères et pointeur sur des caractères

### 10.4.1 Les chaînes de caractères

Rappelons qu'en C une chaîne de caractères est un tableau de caractères dont le dernier caractère significatif est le caractère '\0'. Ainsi, la constante de type chaîne de caractères "bonjour", est représentée en machine par le tableau de caractères ci-dessous :

"bonjour" ⇔

| 'b' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | '\0' |
|---|---|---|---|---|---|---|---|

Une variable de type chaîne de caractères peut être déclarée et initialisée comme suit :

```
char chaine[10] = "bonjour" ;
```

ce qui signifie que la variable `chaine` est un tableau contenant 10 cellules de type `char`. En mémoire centrale, on a alors :

chaine

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 'b' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | '\0' | ? | ? |

Comme `chaine` est un identificateur de tableau, il s'agit d'une constante de type pointeur. La valeur de chaine `(*chaine)` est le contenu de la première cellule du tableau : `chaine[0]`. Ainsi pour accéder à la valeur d'une cellule indicée par i dans le tableau (avec i<10) on peut indifféremment écrire : `chaine[i]` ou `*(chaine+i)`.

On peut aussi utiliser une variable de type pointeur sur un caractère pour déclarer et initialiser une variable de type chaîne de caractère :

```
char *Pchaine = "bonjour" ;
```

Le pointeur `Pchaine` pointe alors vers le caractère `b` d'une chaîne constante `"bonjour"`, et on peut l'utiliser comme s'il s'agissait d'un identificateur de tableau de caractères, par exemple, pour imprimer cette chaîne avec :

```
printf("chaine Pchaine : %s\n", Pchaine) ;
```

La procédure `printf` , du fait de la spécification de format `%s`, imprimera tous les caractères qui suivent jusqu'à rencontrer un caractère '\0'. En mémoire centrale, on a quelque chose comme :

Pchaine

| 1230 → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 'b' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | '\0' |
| *Adresses* | 1230 | 1234 | 1238 | 1242 | 1246 | 1250 | 1254 | 1258 |

`chaine` et `*Pchaine` ont donc toutes deux à ce moment pour valeur une chaîne de caractères contenant les caractères de « bonjour ». Mais, `chaine` est un tableau implanté à une adresse fixe ; pour changer sa valeur, il faut changer la valeur de ses cellules une à une, alors que `Pchaine` est une variable pointeur initialisée de façon à pointer initialement sur les mêmes caractères, mais sa valeur peut changer. Par exemple, on peut la modifier en écrivant :

```
Pchaine = "salut" ;
```

`Pchaine` pointe maintenant vers une nouvelle constante (allouée quelque part ailleurs en mémoire) sans qu'aucune recopie de caractères n'ait été effectuée.

Par contre, bien qu'on puisse écrire `char chaine[10] = "bonjour"` pour initialiser un tableau de taille 10, il ne faut pas considérer cette initialisation comme une affectation. On

ne peut pas en effet affecter un tableau directement avec une autre chaîne ou un autre tableau et écrire par exemple :

```
char chaine[10] ;
chaine = "bonjour" ;
```

Alors que l'on peut écrire sans aucun problème :

```
char *Pchaine ;
Pchaine = "bonjour" ;
```

En effet, lors d'une affectation, on distingue clairement la partie gauche du signe de l'affectation (souvent notée en anglais *lvalue*, pour *left value*) et sa partie droite. La partie droite doit contenir une expression évaluable, et la partie gauche un emplacement situé à une adresse modifiable (par cette expression). La valeur de gauche indique donc une adresse de variable, généralement via un identicateur. Mais cette valeur de gauche ne peut pas être de type tableau ou fonction, car ces derniers ont un comportement semblable à celui de pointeurs constants. (Il ne peut d'ailleurs pas non plus s'agir d'un autre type s'il a été qualifié de `const`, et lorsqu'il s'agira d'une structure, aucun des membres ou sous-membres ne pourra avoir été lui même qualifié de `const` pour que l'affectation généralisée soit possible).

De la même façon, il n'est pas possible à une fonction d'avoir pour valeur de retour un objet de type tableau ou fonction, ces derniers ne pouvant pas être créés dynamiquement. Par contre, on pourra utiliser des pointeurs pour simuler en quelque sorte, une valeur de retour de type tableau (ou fonction) : on utilisera alors un pointeur dont la valeur (une adresse) est celle précisément d'un tableau (ou d'une fonction), ce dernier pouvant alors être utilisé pour accéder aux éléments du tableau (ou pour appeler la fonction avec des arguments effectifs).

### 10.4.2 La bibliothèque `<string.h>`

La valeur de retour d'une fonction peut être de type simple (`int`, `float`, ...), de type pointeur, ou de type enregistrement (car l'affectation généralisée est possible), mais elle ne peut pas être de type tableau. Pour lever cette contrainte, on peut utiliser des pointeurs, car une fonction peut renvoyer une valeur de type pointeur. La bibliothèque standard sur les chaînes de caractères fournit beaucoup d'exemples de telles fonctions, ramenant des « tableaux » de caractères (en réalité, la taille de ces tableaux n'est pas fixée par avance, et il s'agit de simple pointeurs sur des `char`, indiquant le début de la chaîne).

Pour utiliser une fonction prédéfinie de cette librairie, il faut inclure le fichier d'entête `<string.h>`. Dans ce fichier on trouve de nombreuses fonctions que nous avons déjà introduites dans le polycopié n°2, mais souvent sans réellement en préciser la déclaration dans la librairie standard, car la plupart de ces fonctions retournent en réalité un type pointeur. Mais, comme nous l'avions indiqué, un type de paramètre `char *str` étant équivalent à un type `char str[]` et nous avions pu réécrire certaines de ces fonctions en n'utilisant que des tableaux de caractères.

La fonction permettant de calculer la longueur d'une chaîne de caractères (à savoir le nombre de caractères significatifs – i.e. ceux qui précèdent le caractère '`\0`') s'appelle `strlen` et a pour prototype :

```
int strlen(const char *str) ;
```

Le qualificatif `const` indique que la chaîne de caractère argument ne sera pas modifiée par la fonction.

Ainsi, `strlen("bonjour")` renvoie `7`.

On peut l'écrire, avec des pointeurs, sous la forme :

```
int strlen (const char str[])
{     char * s = str ;  /* le pointeur s est au début
                           du tableau de caractères str[] */
      int nb=0 ;

      /* tant que ce sur quoi pointe s n'est pas '\0'*/
      while (*s != '\0') {
           nb++; /* rajouter un caractere au compteur */
           s++ ; /* et avancer le pointeur s sur
                     le caractère suivant */
      }
      /* ici, on est sorti, parce que s pointe sur '\0' */
      return nb;
}
```

ou encore, sous la forme plus condensée :

```
int strlen (const char *s)
{     int nb=0 ;

      while (*s++) /* ce sur quoi pointe s != 0 i.e. '\0'*/
           nb++;
      return nb;
}
```

De manière similaire, la fonction permettant de copier une chaîne de caractères (appelée source, ici `src`) dans une autre (destination) est `strcpy` :

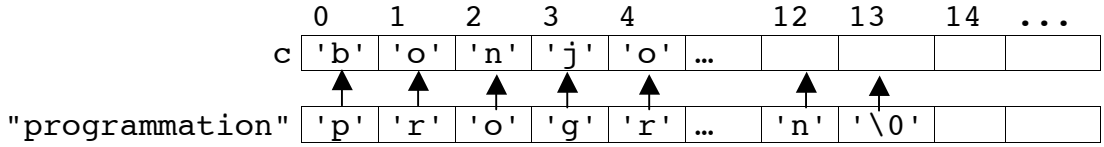
```
char *strcpy(char *dest, const char *src) ;
```

Cette fonction renvoie un pointeur sur le début de la chaîne modifiée (c'est-à-dire la valeur du premier paramètre formel). On a vu qu'on pouvait l'écrire avec des pointeurs sous la forme très condensée :

```
char * strcpy (char *dest, const char *src)
{     char *s = dest;
      while (*s++ = *src++)
           ;
      return dest;
}
```

**Exemple**

```
char c[100]= "bonjour" ;
/* pour afficher la chaîne «programmation» et la stocker
dans la variable c */
printf(«%s», strcpy(c, "programmation") );
```




Remarque : Attention, cette fonction recopie tous les caractères de la chaine source dans la chaîne résultat, mais elle suppose que la chaîne résultat contient suffisamment de place pour que la recopie soit possible. Tout se passera donc bien si on a déclaré par exemple

```
char tab[80];
char *ret ;                    /*  ou char ret[] ;  */
ret = strcpy(tab, "programmation") ;
```

Mais, le programme sortira en erreur avec :

```
ret = strcpy(ret, "programmation") ;
```

La fonction `strcat` permet de **concaténer** (c'est-à-dire de mettre bout à bout) deux chaînes de caractères :

```
char *strcat(char *dest, const char *src) ;
```

Cette fonction concatène la chaine `dest` et la chaine source `src`, et retourne un pointeur sur le début de la chaine initiale, laquelle contient maintenant le résultat de cette mise bout à bout. (Elle recherche la fin de la chaîne du premier paramètre formel, puis effectue la recopie de la chaine source à partir de la fin). Il faut que la chaîne `dest` soit suffisamment grande (pas en longueur, mais en place mémoire allouée) pour que la recopie soit possible.

**Exemple**

```
char c[128]= "" ; /* chaine c initialisée à la chaîne
vide*/
char *pc = " impérative";
  /*  pour concaténer la chaîne de caractères stockée dans
c       avec celle pointée par la variable pc
      et afficher le contenu de c */
printf("%s", strcat(c, pc)) ;
```

Remarque : Attention, ici aussi, la fonction suppose qu'il y a assez de place disponible dans la chaîne résultat (ici `c`) pour y recopier à la fin le contenu de la chaîne source.

De la même manière, le prototype de la fonction prédéfinie `strcmp` permettant de comparer deux chaînes de caractères est en réalité :

```
int strcmp(char *s1, char *s2) ;
```

Rappelons que le résultat est :

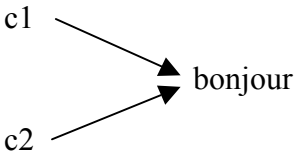
- inférieur à 0 si `s1[]` est inférieure à `s2[]` selon l'ordre lexicographique,
- égal à 0 si `s1[]`et `s2[]` ont les mêmes caractères,
- supérieur à 0 si `s1[]` est supérieure à `s2[]` selon l'ordre lexicographique.

**Exemple**

```
char chaine[10]= "bonjour";
```

```
char *c1,*c2 ;

c1  = chaine ;
c2  = chaine ;
```

Dans ce cas `c1==c2` vaut vrai et `strcmp(c1,c2)` vaut 0.




**Exercices** : Ecrire ces nouvelles versions de `strcat` et `strcmp` avec des pointeurs.

# 11 Pointeurs et tableaux à plusieurs dimensions

## 11.1 tableaux à deux dimensions

Un tableau à deux dimensions est en fait un ensemble de deux tableaux à une dimension consécutifs et la déclaration

*type* `tableau [N1][N2];`

peut être vue comme la déclaration d'un tableau à une dimension (de taille N1) dont les éléments sont des tableaux (à une dimension), de taille N2. Si on fait abstraction de la taille du premier tableau à une dimension, on peut ne considérer qu'un pointeur sur un tableau ayant ce type d'éléments, et simplifier la première déclaration en :

*type* `(*tableau)[N2];`

c'est-à-dire, comme la déclaration d'un pointeur sur : un tableau de N2 objets de type *type*. Si on avance ce pointeur d'un cran, on avancera au « tableau suivant » de N2 objets de ce type.

Les deux expressions ne sont pas équivalentes, car dans le premier cas, on a bien défini un tableau de taille N1xN2 et un emplacement mémoire bien défini a été attribué aux cellules de type *type*, alors que dans le deuxième cas, il s'agit d'une déclaration de pointeur sur un tableau de N2 éléments de type *type*, dont l'emplacement mémoire et le nombre de cellules n'ont pas du tout été alloué. Le compilateur n'a en effet à ce stade réservé de place mémoire que pour mettre l'adresse de début de ce « tableau ». Aucune place mémoire n'a été attribuée pour les cellules, pas même celles d'un tableau.

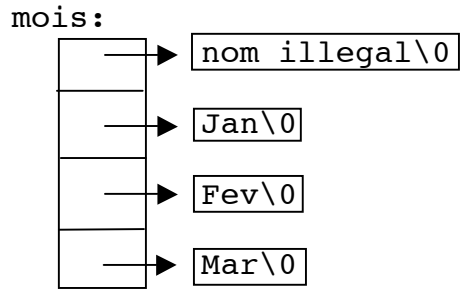
**Exemples :**

```
int x[3][5];
int (*px)[5] ;
```

Ces deux déclarations n'allouent pas les mêmes emplacements mémoire mais elles permettent d'accéder aux mêmes types d'objets.

Notez également que les parenthèses sont nécessaires. Les opérateurs [] ont en effet une plus grande priorité que *. Comparez par exemple la définition et l'image en mémoire d'un tableau de pointeurs initialisé avec :

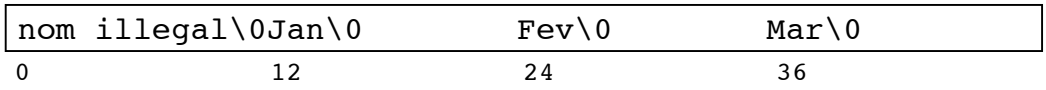
```
char *mois[] ={ "nom illegal", "Jan", "Fev","Mar" } ;
```




Avec celles d'un tableau de caractères à deux dimensions :

```
char unmois[][12] ={ "nom illegal", "Jan", "Fev","Mar" } ;
```

unmois:




Sans les parenthèses, la variable `mois` est un tableau de pointeurs sur des char*, c'est-à-dire des chaînes de caractères. Avec des parenthèses, on aurait eu

```
char (*Pmois)[12] ;     /* Pmois pointe sur un tableau dont
          les éléments sont des chaines de 12 caractères */
```

qui aurait pu être initialisé avec `unmois.`

Ainsi, la variable tableau définie par

```
int tableau[][12] ;
```

et le pointeur ptableau sur des chaines de 12 entiers

```
int (*ptableau)[12] ;
```

pourraient tous deux être passés en argument à une fonction qui prendrait en argument un paramètre de type tableau à deux dimensions

```
int doubletab[2][12] ;
```

**Remarque** : De façon générale, pour le passage d'un tableau multidimensionnel à une fonction, seule la première dimension du tableau sera « libre » et substituable par un pointeur, les autres devront être entièrement spécifiées.

De la même façon, un identificateur de tableau ayant le même type qu'un pointeur sur son premier élément, la déclaration

*type* `(*identificateur) [TAILLE];`

introduit une variable de même type que

*type* `**identificateur ;`

Pour accéder à une variable particulière d'un tableau à plusieurs dimensions, on pourra donc appliquer plusieurs fois l'opérateur d'indirection.

**Exemple :**

`x[2][4]` ⇔`*(*(x+2)+4)`

Ce type de remarque peut se généraliser à un tableau de dimension N. Ainsi

*type* `identificateur[P1][P2]...[PN];`

ou

*type* `(*identificateur)[P1][P2]...[PN] ;`

ou, sans parenthèses :

*type* `**`...(`N` fois l'étoile)...`*  identificateur;`

sont des déclarations de variables de même type. **Mais elles ne sont pas équivalentes en terme d'espace mémoire alloué à ces variables.**

Une politique générale pourra donc être, de déclarer des variables de type tableau, correctement allouées (avec des tailles fixées, introduites avec des constantes) au niveau global ou comme variables de la fonction `main`, et d'utiliser les types pointeurs plus simples précédents comme paramètres des fonctions.

## 11.2 Tableau de pointeurs

Il est bien entendu aussi possible de définir un tableau dont les cellules sont des variables de type pointeurs comme suit :

```
type * tablePt[TAILLEMAX];
/* On définit ainsi un tableau de TAILLEMAX variables
  de type pointeurs sur des cellules de type type */
```

Cette fois, **la déclaration ne contient pas de parenthèses**, et `tablePt` est un tableau (à une dimension) dont les cellules sont de type pointeur vers des valeurs de type *type* :

**Exemple**

Ecrire une procédure qui prend en arguments deux tableaux de pointeurs sur des caractères, le nombre de cellules initialisées dans le premier tableau et réalise la copie du premier tableau dans le second.

```
#include <stdio.h>
#include <string.h>
#define TAILLEMAX 10

void copierTabChaine( char *tabSource[],
                      char *tabDest[], int taille)
{   int i;

    if (TAILLEMAX < taille) {
        printf("recopie impossible\nPlace insuffisante\n");
        exit(0);      /* quitte le programme */
    }
    for(i=0; i<taille; i++)
        tabDest[i]= tabSource[i];
}

int main()
{   int i;
    char *ocean[] = {"pacifique", "atlantique", "indien"} ;
    char *oceanBis[TAILLEMAX] ;

    copierTabChaine(ocean, oceanBis, sizeof(ocean));
    for(i=0; i< sizeof(ocean); i++)
        printf("%s\n",oceanBis[i]);
    return(0);
}
```

Remarque : les caractères des chaînes ne sont pas eux-mêmes recopiés ici. Les deux tableaux de chaînes pointent vers les trois mêmes chaînes constantes.

| Définition | Signification |
|---|---|
| `int *p ;` | `p` est un pointeur sur un entier de type `int` |
| `float *p ;` | `p` est un pointeur sur un réel de type `float` |
| `int (*p)[3] ;` | `p` est un pointeur sur un tableau contenant 3 cellules de type `int` |
| `int *tab[3] ;` | `tab` est un tableau contenant 3 cellules de type pointeur sur un entier de type `int` |
| `struct etiq {`<br>`  char *p ;`<br>`  int x ;`<br>`} var ;` | `p` est un membre de la variable `var` de type `struct etiq`. Ce membre `p` est un pointeur sur une valeur de type `char`. |
| `int *f() ;` | `f` est une fonction, ne prenant aucun argument et renvoyant une valeur de type pointeur sur un `int`. |
| `char *f(double *p) ;` | `f` est une fonction, prenant un argument de type pointeur sur un `double` et, renvoyant une valeur de type pointeur sur un `char`. |

Figure . Récapitulatif sur les pointeurs

## Bibliographie


[1] *Le langage C, norme ANSI*, Kernighan & Ritchie, 2e ed., Dunod.

[2] *Belle programmation et langage C*, Yves Noyelle, cours de l'école Supérieure d'Electricité dans la collection TECHNOSUP, Ellipses Editions, 2001.

[3] *Initiation à la programmation*, C. Delannoy, Editions Eyrolles.