Université Paris 13
Institut Galilée
Deug Mias 1ère année
2002-2003

Programmation Impérative

Troisième Partie

(Chapitre 4)

Enseignante : Adeline Nazarenko

Table des matières

4 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 :

- Les pointeurs sont des outils qui permettent la gestion des suites d'objets en mémoire.
- Le mode de passage des paramètres de fonction se fait par valeur uniquement : d'où la nécessité de passer par un pointeur pour pouvoir passer des adresses de variables en paramètres. On parle alors de *passage de paramètre par variable*, (par opposition au passage de paramètre par valeur que nous avons vu jusqu'à présent). 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.
- L'identificateur d'un tableau est en fait l'équivalent d'un pointeur. (Ce pointeur indique l'adresse en mémoire où sont stockés les éléments du tableau). Les pointeurs et les tableaux sont intimement reliés et sont souvent utilisés simultanément.
- La valeur de retour d'une fonction ne peut être de type tableau, mais elle peut être de type pointeur. On pourra donc retourner un pointeur sur un tableau, c'est-à-dire donner son adresse, pour contourner cette règle.
- De la même façon, la valeur de retour d'une fonction ne peut pas être une fonction, mais l'identificateur d'une fonction est aussi l'équivalent d'un pointeur. (Ce pointeur indique l'adresse en mémoire de l'espace nécessaire à l'exécution de la fonction). On pourra donc ici encore contourner la règle et renvoyer un identificateur de fonction.

4.1 Définition et initialisation

Un pointeur est une **variable** dont la valeur est l'adresse d'une variable.

4.1.1 Définition d'une variable pointeur

La syntaxe est la suivante :

```
<id_type> *<id_pointeur>,...;
```

<id_pointeur> est l'identificateur d'une variable pointeur pointant sur une zone mémoire pouvant contenir des valeurs du type <id_type>. Le contenu de cette variable est donc une adresse mémoire où peut être stockée une variable de type <id_type>. Le type <id_type> peut être indifféremment défini ou prédéfini.

Exemple

```
int *po;      /*    po est une variable de type pointeur
sur une                  valeur de type entier, ou plus
simplement
                po est un pointeur sur un entier */
```

4.1.2 Définition d'un type pointeur

Pour définir un nouveau type pointeur, on écrira :

```
typedef <id_type> *<id_type_pointeur>;
```

<id_type_pointeur> est un nouveau type pointeur vers une zone mémoire pouvant contenir des valeurs de type <id_type>.

Exemple

```
typedef int *Pint;        /*   définition d'un type
                              pointeur sur int */
Pint po;        /* po est un pointeur sur un entier */
```

4.1.3 Initialisation d'une variable pointeur

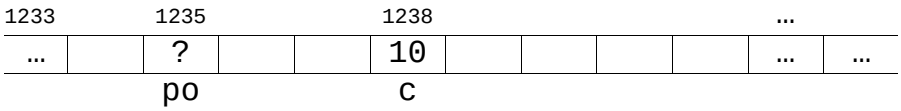
Il existe plusieurs façons d'initialiser une variable pointeur :

- En lui donnant l'adresse d'une variable déjà définie grâce à l'opérateur unaire **&** 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 le contenu de cette variable ou pointe vers cette variable.

  Exemple

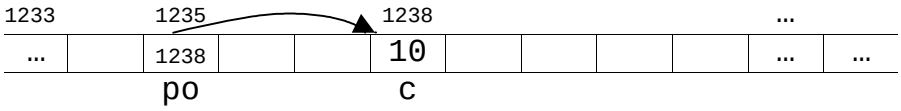
  ```
  int *po;
  int c=10;
  ```

  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:




  ```
  po = &c;  /* affecte l'adresse mémoire de c (ici 1238)
               au pointeur po. po pointe sur c */
  ```

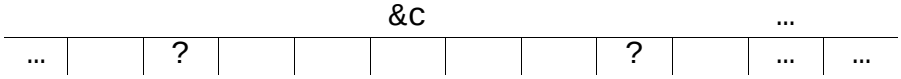
  Le fait d'affecter l'adresse de c à po peut se représenter comme suit :

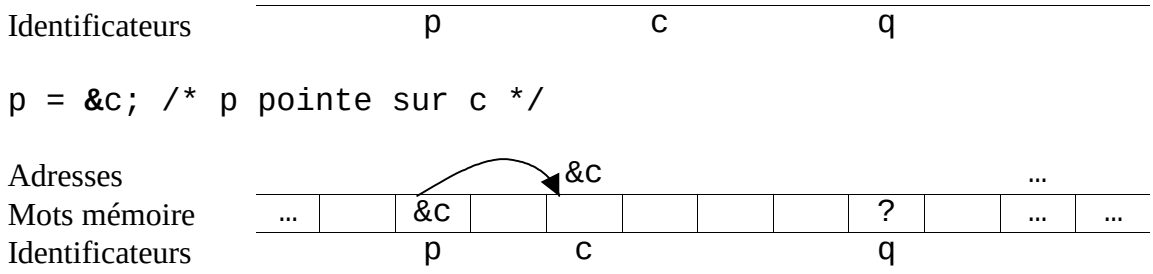


- En lui affectant la valeur d'une autre variable pointeur de même type.

  Exemple

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

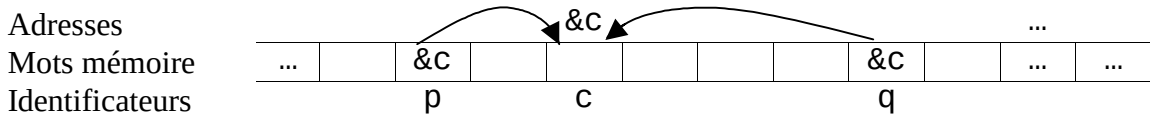





```
p = &c; /* p pointe sur c */
```



```
q = p; /* q pointe sur la même adresse que p */
```



- En le faisant pointer sur rien.

Exemple

```
int *po;
po = NULL;  /*  po ne pointe sur rien :
                représenté par la constante NULL
                (définie dans stdlib.h) */
```

4.1.4 Les opérateurs et leur priorité

L'opérateur unaire d'**indirection** (ou de déréférence), noté *****, dont l'opérande doit être de type pointeur, donne accès à la valeur de la variable sur laquelle pointe le pointeur. On parle d'indirection, car on accède à cette valeur indirectement (via le pointeur).

Exemple :

```
int *po;
int c=1;

po = &c;    /* po pointe sur c */
y = *po;    /* la valeur de c est affectée à y : y vaut 1
*/
```

On peut alors indifféremment utiliser `*po` et `c` dans une expression :
`y = *po + 10;` ⇔ `y = c + 10;`

Les ordres de priorité des opérateurs sont :

Priorité	**Opérateur**	**Associativité**
1	() [] -> .	→
2	* (opérateur d'indirection) & !	←
3	* (multiplication) / %	→
4	+ -	→
5	< <= > >=	→
6	== !=	→
7	&&	→
8	||	→

4.2 Le mode de passage de paramètre par variable

Nous savons que, à l'appel d'une fonction, la valeur du paramètre effectif est recopiée dans le paramètre formel correspondant. Si le paramètre formel est modifié dans le corps de la fonction, c'est la copie qui est modifiée et cette nouvelle valeur n'est pas récupérée par le programme appelant. C'est le mode de passage de paramètre par valeur.

Exemple

On souhaite écrire une procédure qui échange les valeurs des deux variables qui lui sont passées en argument. Si l'on écrit naïvement :

```
#include <stdio.h>
void echange(int x,int y);

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

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

void echange(int x,int y) {
  int temp;

  temp    = x;
  x = y;
  y = temp;
}
```

La simulation de l'exécution de ce programme donnera :

{a=10 ;b=5}
Appel de la fonction echange({a=10 ;b=5}){x=10 ;y=5}
{temp=10}
{x=5}

{y=10}

A ce point de l'exécution du programme les valeurs des paramètres formels ont bien été échangées…

Fin de la fonction echange{x=5 ;y=10}{a=10 ;b=5}

… mais pas les valeurs des paramètres effectifs !

Ecriture de a=10 et b=5.

On constate que la fonction `echange` ne permet pas d'échanger les valeurs des variables a et b du fait du mode de passage des paramètres par valeur.

Un autre mode de passage des paramètres est possible : c'est le mode de passage par variable (ou par adresse). Dans le cas du passage d'un argument par adresse, ce qui est passé, ce n'est plus la valeur du paramètre effectif ou sa copie, c'est son adresse, c'est-à-dire l'adresse de la zone mémoire où la valeur de ce paramètre est stockée. Un traitement effectué sur cette adresse a donc accès à la donnée originale (et non seulement à sa copie).

Par défaut en C, un paramètre est passé par valeur. Lorsqu'on veut passer un paramètre par adresse (ou par variable), il faut passer par des variables pointeurs.

Exemple

Voici une nouvelle version de la fonction `echange` qui utilise le mode de passage par adresse :

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

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

Dans le main, on appelle la fonction en lui-donnant comme paramètres effectifs les adresses des variables `a` et `b` :

```
echange(&a, &b);
```

Simulation de l'exécution de ce nouveau programme :

{a=10 ;b=5}

Appel de la fonction

echange({&a=adresse(a) ;&b=adresse(b)}){px=adresse(a) ;py=adresse(b)}

{temp=10}

{*px=5, a=5}

{*py=10, b=10}

Fin de la fonction echange{*px=5 ;*py=10}{a=5 ;b=10}

Ecriture de a=5 et b=10.

Remarque : Lors d'un appel à la fonction `scanf`, les paramètres effectifs sont les adresses des variables (on utilise &) pour que leur contenu soit bien modifié !!

4.3 Relation entre les pointeurs et les tableaux

4.3.1 Les tableaux unidimensionnels

a. Prémisses d'arithmétique sur les pointeurs

Soit la variable `x` de type tableau suivante :

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

On rappelle que cette variable occupe en mémoire un ensemble de mots mémoire adjacents comme suit :

Indices	0	1	2	…	9
x	?	?	?	…	?

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

```
int *px;
```

Avec l'affectation

```
px=&x[0];
```

on fait pointer `px` sur la variable de type `int` indicée par `0` dans le tableau unidimensionnel `x`.

Il est alors possible d'accéder aux différentes cellules du tableau en incrémentant la variable pointeur `px`. Lorsqu'on écrit px = px + 1, la nouvelle valeur de px n'est pas l'adresse précédente augmentée d'un bit mais désigne l'adresse suivante, c'est-à-dire la valeur précédente augmentée de la taille d'un mot-mémoire permettant de stocker un entier. Ainsi par exemple :

- `px+1` pointe sur la cellule `x[1]`.
- `px+i` pointe sur la cellule `x[i]`.
- `*(px+i)` vaut le contenu de `x[i]`.

C'est pour cela qu'une variable pointeur n'est pas seulement une adresse mémoire mais une adresse **typée**. Par exemple, une variable pointeur `px` sur un réel `r` n'est pas une variable pointeur sur un caractère, car dans ce cas `px+1` désigne l'adresse suivante après `&r`, et la taille mémoire nécessaire au stockage d'un réel est plus grande que celle nécessaire au stockage d'un caractère.

b. Identificateur de tableau et constante de type pointeur

Le lien entre variable pointeur et variable tableau ne se borne pas à ce qui est expliqué ci-dessus, il est beaucoup plus fondamental en C. Il se trouve en C que **l'identificateur d'une variable de type tableau est une constante de type pointeur** (dont la valeur est l'adresse de la première cellule du tableau).

Si `x` est un tableau on a :

```
x[0]        ⇔   *x
x[i]        ⇔   *(x+i)
```

```
&x[0]     ⇔    x
&x[i]     ⇔    x+i
```

Mais, l'identificateur d'**une variable de type tableau est une constante ce qui signifie que sa valeur ne peut pas être modifiée**. Les instructions `x = px;` et `x = x+1;` sont alors incorrectes.

c. Les variables tableau 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 la première cellule a une conséquence très importante sur le mode de passage des paramètres de type tableau. 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 variable et non par valeur, ce qui signifie que les paramètres formel et effectif de type tableau pointent sur la même zone mémoire. Ainsi, si la fonction appelée modifie la valeur d'un paramètre formel de type tableau, alors cette modification 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 entree(double*, int*);

int main() {
    double tab[TAILLEMAX];
    int n,i;
/* Appel de la fonction entree avec comme paramètres
  effectifs la variable tab et l'adresse de la variable n
*/
    entree(tab,&n);
    for(i=0; i<n; i=i+1)
        printf("%d\t",*(tab+i));
    return(0);
}

void entree(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);
}
```

d. Les variables tableau comme valeur de retour d'une fonction

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, il est nécessaire de passer par les pointeurs, car une fonction pouvant renvoyer une valeur de type pointeur, peut renvoyer une valeur de type pointeur sur un tableau.

4.3.2 Manipulation de chaînes de caractères et utilisation de la bibliothèque string.h

a. Tableau de caractères et pointeur

Rappelons qu'en C une chaîne de caractères est un tableau unidimensionnel 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 unidimensionnel 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 l'adresse de la première cellule du tableau : `chaine[0]`. Ainsi pour accéder à une cellule indicée par i (i<10) dans le tableau 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 vers le caractère `b` de la chaîne constante `"bonjour"`, mais on peut l'utiliser comme s'il s'agissait d'un identificateur de tableau de caractères, par exemple, pour imprimer la chaîne avec :

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

La procédure `printf` imprimera tous les caractères qui suivent, jusqu'à rencontrer le caractère '\0' ;

En mémoire centrale, on aurait par exemple:

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

`chaine` et `*Pchaine` ont toutes deux à ce moment pour valeur la chaîne de caractères bonjour. Mais, `chaine` est un tableau implanté à une adresse fixe (ici 1230), et, 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 la même valeur, mais sa valeur peut changer. Par exemple, on peut écrire :

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

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

Ainsi on ne peut pas écrire :

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

Alors que l'on peut écrire :

```
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 une valeur modifiable par cette expression. Une valeur modifiable est une variable dont le type doit être défini (et l'expression figurant en partie droite devra avoir un type compatible) et ne peut être de type tableau ou fonction. (Il ne peut pas s'agir non plus d'une constante, et lorsqu'il s'agira d'une structure, aucun des membres ou sous-membres ne pourra être constant.)

b. La bibliothèque string

Pour utiliser une fonction prédéfinie permettant de manipuler des chaînes de caractères, il faut inclure le fichier d'entête <`string.h`>. Dans ce fichier on trouve de nombreuses fonctions intéressantes.

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(<chaine>) ;
```

où `<chaine>` est : soit une constante de type chaîne de caractères, soit un identificateur de tableau de caractères, soit une variable de type pointeur sur un caractère.

Exemple

```
int longueur ;
longueur = strlen("bonjour") ;
/* longueur vaut alors 7 */
```

La fonction prédéfinie permettant de **copier** une chaîne de caractères (appelée `<chaine source>`) dans une autre (ou à une adresse donnée) est `strcpy` :

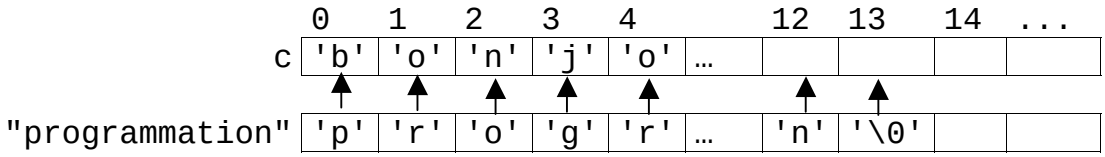
```
char *strcpy(<chaine résultat>, <chaine source>) ;
```

`<chaine résultat>` est soit l'identificateur d'un tableau de caractères, soit l'identificateur d'une variable de type pointeur sur un caractère. `<chaine source>` est soit une constante de type chaîne de caractères, soit l'identificateur d'une variable de type tableau de caractères, soit une variable de type pointeur sur un caractère à laquelle est appliqué l'opérateur d'indirection.

Cette fonction renvoie une valeur de type pointeur sur un caractère. Cette valeur est l'adresse du début de la chaîne recopiée (c'est-à-dire la valeur du premier paramètre formel).

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 prédéfinie `strcat` permettant de **concaténer** deux chaînes de caractères a pour prototype :

```
char *strcat(<chaine resultat>, <chaine source>) ;
```

Cette fonction concatène la chaine résultat et la chaine source et retourne la chaine source : elle recherche la fin de la chaîne ayant pour adresse la valeur du premier paramètre formel, puis effectue la recopie de la chaine source à cette adresse et retourne le premier paramètre formel.

Exemple

```
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.

La fonction prédéfinie permettant de **comparer** deux chaînes de caractères a pour prototype :

```
int strcmp(char *chaine1, char *chaine2) ;
```

Le résultat de l'exécution de

```
strcmp (<chaine1>,<chaine2>)
```

est :

- inférieur à 0 si `<chaine1>` est inférieure à `<chaine2>` selon l'ordre lexicographique,

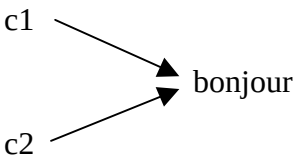
- égal à 0 si <chaine1> et <chaine2> sont identiques,
- supérieur à 0 si <chaine1> est supérieure à <chaine2> selon l'ordre lexicographique.

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.



4.3.3 Les tableaux multidimensionnels

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

```
<id_type> <id_variable>[<val1>][<val2>];
```

peut être vue comme

```
<id_type> (*<id_variable_pointeur>)[<val2>];
```

c'est-à-dire, comme la déclaration d'un pointeur sur un tableau de <val2> objets de type <id_type>. Les deux expressions ne sont cependant pas équivalentes, car dans le premier cas, on a bien défini un tableau de taille <val1>x<val2> et un emplacement mémoire défini a été attribué aux cellules de <id_type>, alors que dans le deuxième cas, il s'agit d'une déclaration de pointeur sur un tableau de <val2> éléments de type <id_type>, dont l'emplacement mémoire et le nombre de cellules n'ont pas encore été alloué.

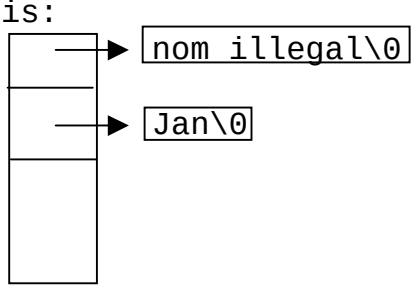
Exemples :

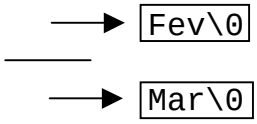
```
int x[3][5];
int (*px)[5] ;
```

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:

nom illegal\0	Jan\0	Fev\0	Mar\0
0	12	24	36

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] ;
```

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

```
<id_type> (*<id_pointeur>)[<val2>];
```

introduit une variable de même type que

```
<id_type> **<id_pointeur>;
```

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 déclarations peut se généraliser à un tableau de dimension n :

```
<id_type> <id_variable>[<val1>][<val2>]...[<valn>];
```

ou

```
<id_type> (*<id_pointeur>)[<val2>][<val3>]...[<valn>] ;
```

ou

```
<id_type> *...*<id_pointeur>;
```

sont des déclarations de même type. (Mais elles ne sont pas équivalentes du point de vue de l'espace mémoire alloué aux variables).

4.4 Les pointeurs dans les tableaux et les enregistrements

4.4.1 Les pointeurs comme cellules de tableau

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

```
<id_type> *<id_tab_pointeur>[TAILLEMAX];
/* On définit ainsi un tableau de TAILLEMAX variables
  de type pointeurs sur des id_type */
```

<id_tab_pointeur> est un tableau (à une dimension) de cellules de type pointeur vers des valeurs de type <id_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.

4.4.2 Les variables pointeurs et les enregistrements

a. Pointeur sur des enregistrements

Il est possible de définir une variable pointeur sur un enregistrement.

Considérons le type enregistrement suivant :

```
typedef struct {
      <id_type1> <id_membre1>;
                …
      <id_typeK> <id_membreK>;
} <id_type_enregistrement> ;
```

`<id_var>` est une variable de ce type enregistrement :

```
<id_type_enregistrement> <id_var> ;
```

Il est possible de définir une variable pointeur de nom `<id_var_pointeur>` comme suit :

```
<id_type_enregistrement> *<id_var_pointeur> ;
```

Et pour initialiser cette variable pointeur, on peut écrire l'affectation suivante :

```
<id_var_pointeur> = &<id_var>;
```

On remarquera que l'opérateur d'adresse & peut s'appliquer à une variable de type enregistrement. Dans ce contexte `<id_var_pointeur>` pointe sur l'enregistrement `<id_var>`. On peut donc accéder à chacun des membres de `<id_var>` grâce à la variable pointeur `<id_var_pointeur>` en appliquant l'opérateur * d'indirection :

```
(*<id_var_pointeur>).<id_membrei> ⇔ <id_var>.<id_membrei>
```

Comme l'opérateur d'accès aux membres (l'opérateur .) a une priorité plus grande que celle de l'opérateur * d'indirection, les parenthèses sont nécessaires : on doit dans un premier temps pointer sur la valeur de type `<id_type_enregistrement>` pour accéder ensuite, au sein de cet enregistrement, au ième membre. Pour éviter la lourdeur des parenthèses, il existe en C un opérateur d'accès aux membres, noté →, qui permet d'accéder aux membres d'un enregistrement à partir d'une variable de type pointeur sur un enregistrement. On a alors :

```
   <id_var_pointeur> -> <id_membrei>
⇔ (*<id_var_pointeur>).<id_membrei>
```

Ce nouvel opérateur fait partie des opérateurs ayant la priorité la plus élevée en C.

Exemple

```
typedef struct {
      char intitule[50];
      int annee;
      char matiere[50];
      int nbInscrits;
      char formation[50];
} Cours ;
Cours c, *pc ;   /* pc est un poiteur sur un Cours */

pc = &c ;         /* pc pointe maintenant sur c */
c.annee = 2001 ;
    /* Pour initialiser la variable c on peut
```

```
      aussi passer par le pointeur pc */
strcpy((*pc).matiere,"informatique");
pc->nbInscrits = 250 ;
strcpy(pc->formation,"MIAS 1") ;
```

b. Pointeur dans les enregistrements

Il est également possible qu'un membre d'une variable de type enregistrement soit lui-même une variable de type pointeur. Il sera alors introduit par une étoile, comme suit :

```
typedef struct {
    <id_type1> <id_membre1>;
            …
    <id_typeI> *<id_membreI> ;
            …
    <id_typeK> <id_membreK>;
} <id_type_enregistrement> ;
```

Soit `var` une variable de ce type enregistrement :

```
<id_type_enregistrement> var ;
```

Pour accéder à la valeur du ième membre on peut écrire :

```
*(var.<id_membreI>) ;
```

ou

```
*var.<id_membreI> ;
```

Et si on initialise une variable pointeur de nom `var_pointeur` comme suit :

```
<id_type_enregistrement> *var_pointeur  = &var ;
```

On peut écrire :

```
*var_pointeur -> <id_membreI>
```

⇔

```
*(*var_pointeur).<id_membreI>
```

4.5 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 decaractères) à la console, lors du lancement de la commande exécutable compilée à partir du programme.

Le prototype de la fonction main doit être :

```
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*)

Ainsi, si la commande de lancement du programme est

```
prog.exe  arg1 arg2 10 arg4
```

la fonction `main` sera exécutée avec

```
argc == 5, argv[0]== ″prog.exe″ ,
argv[1]== ″arg1″,argv[2]== ″arg2″ ,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) ou la fonction `sscanf` qui permet de lire une chaîne de caractères de manière formattée.

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

A l'exécution si on lance l'exécutable `affich.exe` suivi des arguments : `numero 222`

```
affich.exe numero 222
```

On aura l'affichage suivant :

```
Paramètre effectif 0 : affich.exe
Paramètre effectif 1 : numero
Paramètre effectif 2 : 222
```

4.6 Récapitulatif sur les pointeurs

Définition	Signification
`int *p ;`	p est un pointeur sur un entier
`float *p ;`	p est un pointeur sur un réel
`int (*p)[3] ;`	p est un pointeur sur un tableau contenant 3 cellules de type entier
`int *tab[3] ;`	tab est un tableau contenant 3 cellules de type pointeur sur un entier
`struct etiq *p ;`	p est un pointeur sur une valeur de type `struct etiq`

`struct etiq {`
`  char *p ;`
`  int x ;`
`} var ;`
	p est un membre de la variable var qui est de type `struct etiq`. le 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 entier

`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 caractère