Cours de C - ANSI
2ème partie (3 cours de 2 heures)
Bertrand LIAUDET

Qu'est-ce que la science ? La science c'est ce que le père enseigne à son fils. Qu'est-ce que la technologie? C'est ce que le fils enseigne à son père.

Michel Serres

SOMMAIRE DE LA DEUXIEME PARTIE

CHAPITRE 9 : LES POINTEURS

La notion de pointeur est une notion fondamentale de l'informatique. Elle n'est pas très compliquée à comprendre : les quelques explications ci-après doivent permettre de la clarifier. Par contre, il faut reconnaître qu'elle peut être difficile à mettre en œuvre. Quand on a des difficultés à mettre en œuvre cette notion, il faut toujours revenir aux définitions.

La notion de pointeur va intervenir dans les fonctions, les tableaux et les chaînes de caractères. On a déjà vu intervenir les pointeurs avec le scanf.

Dans ce chapitre, on va présenter les deux opérateurs qui permettent de manipuler les pointeurs : & et *.

Dans les chapitres suivants, on abordera les relations entre pointeurs et fonctions, pointeurs et tableaux. Les relations entre pointeurs et chaînes de caractères seront abordées dans le chapitre sur les chaînes de caractères.

9.1 Rappels sur la variable

On a déjà vu qu'une variable est caractérisée par :

- un nom : n
- une valeur (ou contenu): 3
- un contenant : le carré dessiné
- une adresse : ad_n
- un type : int
- un sens (ou signification) : par exemple, le nombre d'éléments d'un tableau.

Le nom c'est "n". Le nom fait référence à (on peut dire c'est) soit la valeur, soit le contenant.

Si le **nom** "n" est utilisé en entrée (c'est-à-dire à droite de l'affectation, en entrée d'une fonction ou plus généralement en entrée de toute expression), "n" c'est la **valeur** de "n".

Si le **nom** "n" est utilisé en sortie (c'est-à-dire à gauche de l'affectation, en sortie d'une fonction ou plus généralement en sortie de toute expression), "n" c'est le **contenant** de n. On parle aussi de "lvalue" ("left value", "valeur à gauche" de l'affectation, ou "valeur_g"[1]).

1 K&R91 p. 196, §A5.

9.2 Déclaration et usage

- Quand on déclare une variable, on crée un contenant qui a un nom, un type et une adresse.
- Quand on affecte une valeur à une variable, on donne une valeur au contenant.
- Initialiser une variable, c'est donner pour la première fois une valeur au contenant de la variable. A noter qu'avant l'initialisation, la variable a quand même une valeur. En effet les bits constituant la variable sont dans un certain état (0 ou 1). Cet état, interprété selon le type, aboutit forcément à une valeur. Donc la valeur d'initialisation c'est la deuxième valeur que prend la variable.
- Quand on utilise une variable dans des opérations autre que l'affectation ou la récupération de l'adresse, on utilise sa valeur.

9.3 Les deux significations de l'adresse

La notion d'adresse est un peu difficile car elle a deux significations différentes :

- l'adresse en tant que caractéristique d'une variable
- l'adresse en tant que valeur d'une variable

1 L'adresse en tant que caractéristique d'une variable

L'**adresse** en tant que caractéristique d'une variable, c'est l'adresse du contenant, c'est l'emplacement du contenant dans la mémoire.

Une fois la variable déclarée (int n), son adresse est donnée par l'opérateur & :

```
&n c'est l'adresse de n.
```

& est un opérateur appliqué à la variable n. L'évaluation de l'expression &n donne l'adresse de n dans la mémoire.

2 L'adresse en tant que valeur : le type pointeur : int *

L'adresse d'une variable peut aussi être considérée comme une valeur comme une autre (entier, caractère) mais d'un type particulier : le **type adresse**.

Une variable pourra donc avoir comme valeur une adresse.

Une variable dont la valeur est une adresse est appelée un pointeur.

Un pointeur est une variable dont la valeur est une adresse.

Un pointeur est de type adresse. **Mais : attention : on ne parle pas de type adresse, mais toujours de type pointeur.**

Il existe plusieurs types pointeurs différents : un par type de base, soit : des pointeurs d'entier, des pointeurs de flottant, des pointeurs de caractère; mais aussi des pointeurs de structures.

En langage C, le type pointeur s'écrit : type *

```
int *
float *
double *
char *
typEleve *
```

Comme pour les types de base, on peut ajouter des attributs : short, long, signed, unsigned ou const :

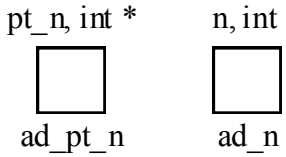
```
short int *
```

Exemple :

Soit n un entier et pt_n un pointeur d'entier. On écrit ça :

```
int n;
int * pt_n;
```

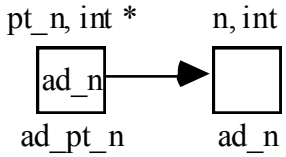
pt_n est une variable avec ses caractéristiques : elle a une adresse, comme toute les variables et son type est "pointeur d'entier" : int *.



Pour affecter à pt_n la valeur de l'adresse de n, on écrira :

```
pt_n = &n
```

Schématiquement cela donne :



On dit que pt_n est un pointeur d'entier, comme on dit que n est un entier.

9.4 Du pointeur à la variable pointée : l'opérateur d'indirection (* pt_n) [2]

Remarquons d'abord qu'on peut aussi écrire la déclaration ainsi :

```
int n, * pt_n;
```

C'est équivalent à ce qu'on a écrit précédemment, mais cette écriture nous montre que *pt_n, comme n, est un int.

1 Qu'est-ce que *pt_n ?

C'est une variable, une lvalue, au même titre que pt_n.

Définition

```
   * pt_n   c'est la variable dont l'adresse est donnée par la
valeur de "pt_n" (dans notre exemple c'est n).
```

On dit aussi :

```
* pt_n  c'est la variable pointée par pt_n.
```

ou encore :

```
* pt_n  c'est la variable référencée par pt_n.
```

Dans notre exemple, *pt_n c'est n. *pt_n peut donc jouer le même rôle que n, peut s'utiliser comme n.

Toute la difficulté de l'usage des pointeurs réside dans cette définition.

Donc

- **de même que :** si "n" est utilisé en entrée, "n" c'est la valeur de "n",
- **et que** : si "n" est utilisé en sortie, "n" c'est le contenant de n (exemple : n=n+1)
- **de même : si "*pt_n" est utilisé en entrée** (c'est-à-dire à droite de l'affectation, en entrée d'une fonction ou plus généralement en entrée de toute expression), **alors "*pt_n" c'est la valeur de la variable** dont l'adresse est donnée par la valeur de "pt_n", c'est-à-dire la valeur de la variable pointée par pt_n, c'est-à-dire, ni plus ni moins, la valeur de "n",
- **et : si "*pt_n" est utilisé en sortie** (c'est-à-dire à gauche de l'affectation, en sortie d'une fonction ou plus généralement en sortie de toute expression), **alors "*pt_n" c'est le contenant de la variable** dont l'adresse est donnée par la valeur de "pt_n", c'est-à-dire le contenant de la variable pointée par pt_n, c'est-à-dire le contenant de "n".

Plus simplement, comme on l'a déjà dit :

```
*pt_n  s'utilise exactement comme n
```

2 **Début du cinquième cours.**

```
int x, n, *pt_n;
pt_n = &n;
*pt_n = 5;     /* n prend la valeur 5 */
n = 7;
x = *pt_n;     /* x prend la valeur 7 */
```

Attention :

La séquence d'instructions suivante n'est pas bonne :

```
main() {
    int n, * ad1;
    * ad1 = 3; /* erreur !! */
    …
}
```

C'est faux car ad1 n'a pas été initialisée. Il n'y a pas de valeur dans ad1. Donc il n'y a pas de variable pointée par ad1. Donc on ne peut pas affecter son contenu.

Il faudrait écrire, par exemple :

```
main() {
    int n, * ad1;
    ad1 = &n;
    * ad1 = 3; /* c'est équivalent à "n = 3;" */
    …
}
```

&* et *&

```
#include <stdio.h>
void main(void) {
    int *ad1;
    int n=10;
    printf("ad1=%p\n", ad1);                    // ad1=CCCCCCCC
    printf("&*ad1=%p\n", &*ad1);               // &*ad1=CCCCCCCC
    printf("*&ad1=%p\n", *&ad1);               // *&ad1=CCCCCCCC
    printf("n=%d\n", n);//                                  n=10

    printf("*&n=%d\n", *&n);                          // *&n=10
//  printf("&*n=%d\n", &*n);                        erreur ! ! !
}
```

9.5 Allocation dynamique de la mémoire : malloc

L'exemple précédent vient de nous montrer que pour affecter une valeur à une variable pointée par un pointeur, encore faut-il que cette variable (pointée) existe.

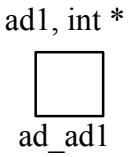
Il y a deux façons de faire "exister" cette variable. Par une déclaration de variable et ensuite en affectant l'adresse de cette variable au pointeur. C'est ce qu'on a fait jusqu'à présent.

Il est aussi possible de créer une variable pointée dynamiquement.

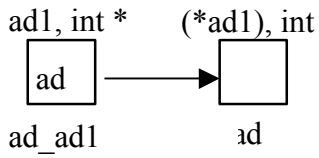
1 Exemple

```
#include <stdlib.h>
main() {
    int * ad1;
    ad1 = malloc (2);
    * ad1 = 3;                  /* cette fois ça marche ! ! ! */
    …
}
```

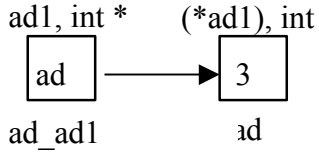
int * ad1;



ad1 = malloc(2);



***ad1 = 3 ;**



Ici on utilise la fonction malloc pour allouer de la mémoire. En général, l'allocation de la mémoire est automatique ou statique. Cependant, on peut allouer dynamiquement de la mémoire pendant le programme.

Le malloc permet d'allouer de la mémoire, ici pour un entier, et que l'adresse de cette mémoire soit rangée dans ad1.

Quand on alloue dynamiquement une variable, on crée une variable qui n'a pas de nom. On crée donc un contenant qui a un type et une adresse mais qui n'a pas de nom.

Une variable créée dynamiquement ne sera donc pas accessible par son nom puisqu'elle n'en a pas, mais uniquement par le pointeur qui pointe sur elle.

2 Syntaxe du malloc

```
#include <stdlib.h>
void * malloc (int taille)
```

malloc alloue au moins taille octets contigus et renvoie l'adresse du premier octet alloué. Si l'allocation n'a pas pu se passer, malloc renvoie NULL.

On évitera d'écrire, comme on l'a fait dans l'exemple de présentation :

```
        ad1 = malloc (2);
```

On écrira :

```
ad1 = (int *) malloc (sizeof(int));
```

3 l'opérateur sizeof[3]

Cet opérateur fournit la taille d'un objet.

On peut écrire sizeof(int) ou sizeof(i), i étant un int, ou sizeof i, ou sizeof 3.

Exemples :

```
sizeof(char) vaut 1 ;
sizeof(int)  vaut 2

int i ;
sizeof(i) vaut 2
ou sizeof i

int tab[5] ;
sizeof(tab) vaut 5 ;
```

Syntaxe

La syntaxe c'est sizeof expression ou sizeof(type)[4].

3 Del97 p 49.
4 Bra98 pp. 125-128

4 le type void *

C'est le type pointeur générique[5], c'est-à-dire pointeur sur n'importe quel type. C'est nécessaire d'avoir ce type car le malloc renvoie l'adresse d'une variable de n'importe quel type.

Du point de vue de la taille de la variable et de son interprétation, ce n'est pas gênant d'avoir un type pointeur générique puisque la taille d'un pointeur c'est la taille nécessaire pour coder une adresse (généralement 4 octets[6]). C'est la même taille et le même codage quelque soit le type.

Par contre, un pointeur doit être typé, car quand on fait une indirection (opérateur étoile), il faut connaître le type de la variable sur laquelle on arrive.

Mais du coup, pour bien faire (pour que l'expression soit syntaxiquement juste), on caste le résultat avant de l'affecter, d'où le (int *) : je caste le résultat en pointeur d'entier : du coup j'affecte un pointeur d'entier dans un pointeur d'entier et non pas un pointeur générique dans un pointeur d'entier.

5 Bra98 p 69, 110-111,

6 Bra98 p. 127.

9.6 Pointeur de pointeur

Sur le même modèle, on peut définir des pointeurs de pointeur (on parle parfois de "super pointeur" : ce n'est pas très explicite ! ! !).

Un pointeur de pointeur c'est une variable dont la valeur est une adresse (comme un pointeur simple).

A la différence d'un pointeur simple, le pointeur de pointeur pointe non pas sur un entier, un caractère ou tout autre valeur classique, mais sur un pointeur.

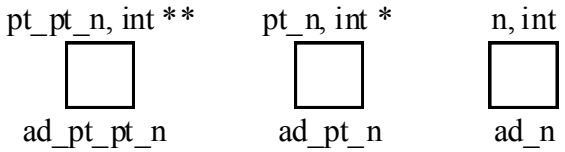
Au lieu de pointer sur une variable de type int, par exemple, il pointe sur une variable de type int *, c'est-à-dire sur un pointeur d'entier.

1 Exemple :

Soit n un entier, pt_n un pointeur d'entier et pt_pt_n un pointeur de pointeur d'entier. On écrit ça :

```
int n;
int * pt_n;
int ** pt_pt_n;
```

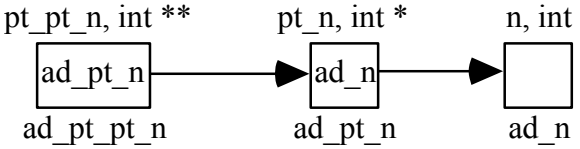
pt_pt_n est une variable avec ses caractéristiques. Elle est de type "pointeur de pointeur d'entier" : int **. Elle doit donc contenir l'adresse d'un pointeur d'entier : * pt_pt_n c'est un int *.



Pour affecter à pt_n la valeur de l'adresse de n, on écrira :

```
pt_n = &n
pt_pt_n = &pt_n
```

Schématiquement cela donne :



On dit que pt_pt_n est un pointeur de pointeur d'entier, comme on dit que pt_n est un pointeur d'entier et que n est un entier.

Attention

```
pt_pt_n = &&n; /* ne veut rien dire */
```

On ne peut pas utiliser deux & à la suite. L'adresse d'une adresse ça ne veut rien dire.

2 Autres exemples :

```
int x, n, *pt_n, *pt_x, ** pt_pt_n;
pt_n = &n;     /* pt_n prend la valeur ad_n */
pt_pt_n = &pt_n;                        /* pt_pt_n vaut ad_pt_n */
*pt_n = 5;     /* n prend la valeur 5 */
** pt_pt_n = 10;                        /* n prend la valeur 10 */
x = 3;                                              /* x vaut 3 */
* pt_pt_n = &x;                  /* pt_n prend la valeur ad_x */
n = **pt_pt_n;/ n prend la valeur 3 */
pt_x = *pt_pt_n;                /* pt_x prend la valeur ad_x */
```

Les pointeurs de pointeurs sont difficiles à utiliser. On verra d'une part quand ils sont nécessaires : pour les tableaux à deux dimensions, les tableaux de chaînes de caractères, les tableaux de pointeurs, les pointeurs en sortie dans les fonctions. D'autre part, on verra comment on peut cacher leur utilisation par la définition de nouveaux types.

CHAPITRE 10 : POINTEURS ET PARAMETRES DES FONCTIONS : LES VARIABLES EN SORTIE

10.1 Paramètres en entrée : passage par valeur

1 Exemple

Voyons une fonction simple : celle qui permet de calculer la surface d'un rectangle.

Analyse :

Entrée : deux entiers : longueur et largeur

Sortie : la surface

Etapes : surface = longueur * largeur ;

en C :

```
int surface (int longueur, int largeur)
{
    return longueur * largeur;
}
```

et l'usage suivant :

```
#include
int surface (int longueur, int largeur);
void main (void){
    int lon=2, surf;
    surf = surface(lon, 5) ;
    printf(« surface = %d\n »,surf) ;
}                                                    essai.c
```

2 Commentaires :

A l'appel de la fonction, on a deux paramètres d'appel, lon et 5. Ici c'est la valeur de lon qui est passée en paramètre (2) et la valeur 5. Ces deux valeurs sont en quelque sorte affectées aux paramètres formels qui leur correspondent : longueur en tant que paramètre formel prend la valeur 2 et largeur la valeur 5.

On dit que les paramètres sont passés par valeur.

3 Pour bien comprendre :

Imaginons qu'on ait écrit la fonction surface comme suit :

```
int surface (int longueur, int largeur)
{
    longueur = longueur * largeur
    return longueur;
}
```

Avec le même *main* que précédemment, bien que longueur soit modifié dans la fonction, lon, passé en paramètre d'appel ne sera pas modifié.

10.2 Paramètres en sortie : passage par adresse

1 Comment pouvoir modifier un paramètre d'appel ?

Dans la fonction précédente, plutôt que de renvoyer le résultat sur la sortie standard, on pourrait le renvoyer dans une variable passée en paramètre.

Du coup, comme il n'y a qu'une seule sortie, on ne ressort plus rien sur la sortie standard.

```
void surface (int lon, int larg, int (*ptSurf) )
{
    *ptSurf = lon * larg;
}
```

Dans le cas du C, on a toujours affaire à une fonction.

Mais cette fois ci :

- elle ne renvoie plus de valeur : d'où le "void surface".
- il y a un troisième paramètre formel : int (*ptSurf).

Les parenthèses sont facultatives (on pourrait ne pas les mettre).

2 Explications

On peut lire la déclaration de deux façons :

- soit en considérant la variable (*ptSurf) de type int
- soit en considérant la variable ptSurf de type (int*)

*(int *) ptSurf*

Ici ptSurf est un pointeur d'entier. Il est en entrée comme lon et larg. Lue comme cela, la fonction est une fonction C classique. Ce qu'on passe en paramètre c'est un pointeur d'entier, donc l'adresse d'un entier. Ce pointeur est passé par valeur. Il n'est pas modifiable.

Attention : syntaxiquement, il est faux de mettre des parenthèses autour de int*.

*int (*ptSurf)*

Par contre *ptSurf est un entier qui lui est en sortie : il est modifiable. Cet entier n'est pas passé par valeur, mais par adresse (c'est l'adresse de l'entier qui est passée en paramètre).

En général, on ne met pas de parenthèses : int * ptSurf

**ptSurf = lon * larg;*

*ptSurf (à gauche de l'affectation) c'est la variable (le contenu, la lvalue) dont l'adresse est donnée par la valeur de ptSurf, valeur qui est fournie en entrée de la fonction et qui n'est pas modifiable.

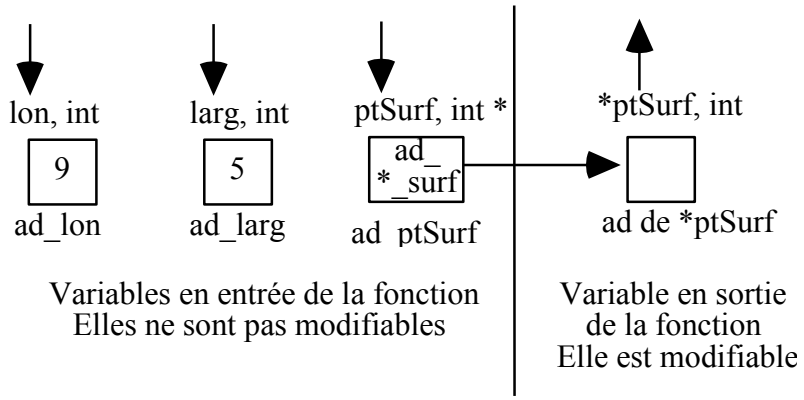
par contre la variable pointée par ptSurf est modifiable.

Si * ptSurf était à droite de l'affectation, ce serait la valeur de la variable dont l'adresse est donnée par la valeur de ptSurf. *ptSurf serait alors en entrée et en sortie.

(Notez que le premier "*" est un pointeur, le deuxième une multiplication!)

schéma :

En réalité, on peut considérer qu'on a deux variables : 1 : (ptSurf) le pointeur et 2 : (* ptSurf) l'entier.



Il faut bien noter que ça n'est pas "ptSurf" qui contiendra le résultat, mais la variable pointée par "ptSurf" : *ptSurf. Cette variable pointée par "ptSurf" n'est pas passée en paramètre de la fonction. Mais on peut y accéder grâce à son adresse qui elle est passée en paramètre.

3 Utilisation C :

```
void main(void){
    int a, surf;
    …
    surface (a, 4, &surf);
    …
}
```

On passe en paramètre l'adresse d'un entier.

4 Attention:

Si on écrivait :

```
void main(void){
    int a, * surf;
    surface (a, 4, surf);/ * erreur !! */
    …
}
```

Le programme ne marcherait pas car surf ne pointe sur rien.

5 Remarque terminologique

On parle plutôt de fonction quand une valeur est renvoyée sur la sortie standard. Ainsi, la fonction peut s'utiliser dans une expression : 4 * surface(5, larg).

On parle plutôt de procédure dès qu'un paramètre formel est en sortie (passé par adresse).

10.3 Les procédures avec au moins deux paramètres en sortie

On vient de voir qu'on peut toujours passer la sortie standard d'une fonction comme un paramètre formel en sortie (passé par adresse) : donc transformer la fonction en procédure.

Les fonctions qui ont deux paramètres ou plus en sortie auront forcément des paramètres formels en sortie (passés par adresse).

Voyons l'exemple de la procédure de résolution d'une équation du second degré.

1 Analyse :

E: a, b, c : réel; (paramètres de l'équation)

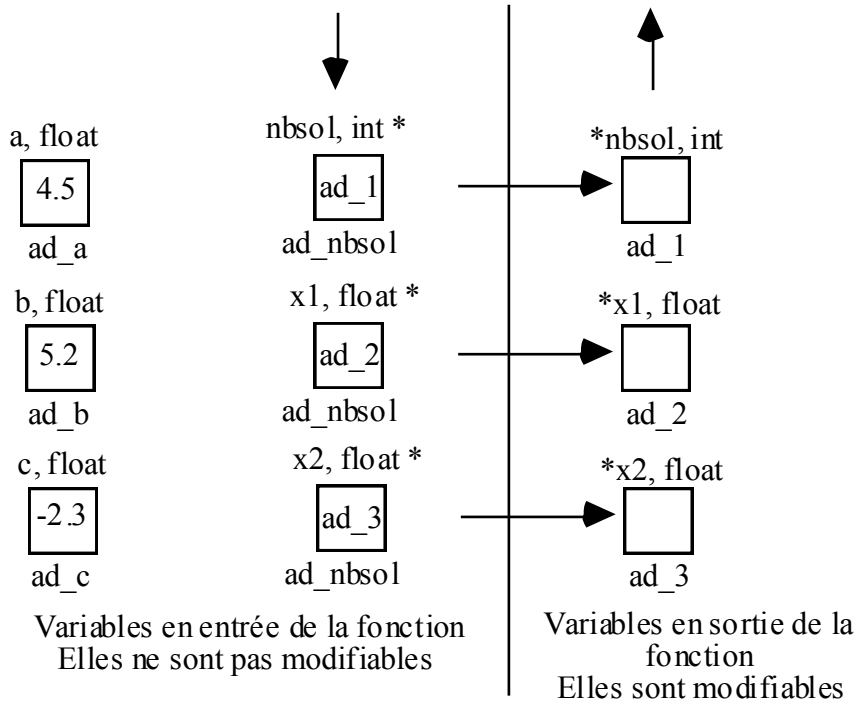
S: nbsol : entier; (nombre de solutions de l'équation, -1 pour l'infini)

x1, x2 : réels; (solutions de l'équation. x1 = x2 si il n'y a qu'une solution)

2 C :

```
  /*  fonction equa2
      E:  a, b, c: réels (paramètres de l'équation)
      S:  *nbsol : entier (nombre de solutions de l'équation, -1
pour l'infini)
          *x1, *x2 : réels (solutions de l'équation. x1 = x2 s'il
n'y a qu'une solution)
  */

  void equa2 (float a, float b, float c, int *nbsol, float *x1,
float *x2)
  {
      float delta;
      if (a == 0) {
          equa1 (b, c, nbsol, x1) /* pas de & ! */
          *x2 = * x1; /* pas x2 = x1 ! */
          return;
      }
      delta = b*b - 4*a*c;
      if (delta > 0) {
          *nbsol = 2;
          *x1 = (-b - sqrt(delta)) / (2*a);
          *x2 = (-b + sqrt(delta)) / (2*a);
      }
      else if (delta == 0 ) {
          *nbsol = 1;
          *x1 = (-b) / (2*a);
          *x2 = *x1;
      }
      else {
          *nbsol = 0;
      }
  }
```



3 à noter que :

- dans cette fonction, a, b, c, x1, x2 et nbsol sont passés par valeur, *x1, *x2 et *nbsol sont passés par adresse.
- on n'a pas repris le formalisme "ptX1, ptNbsol, parce que c'est beaucoup trop lourd. Dans la pratique, personne ne fait cela. Du coup cela engendre des polysémies parfois difficiles à dénouer ! C'est comme ça !
- dans cette fonction, quand on fait appel à "equa1", on ne met pas de "&" car les variables nbsol, x1 et x2 sont déjà des pointeurs.
- pour mettre x2 dans x1, on écrit :

```
*x2 = *x1;
```

et pas

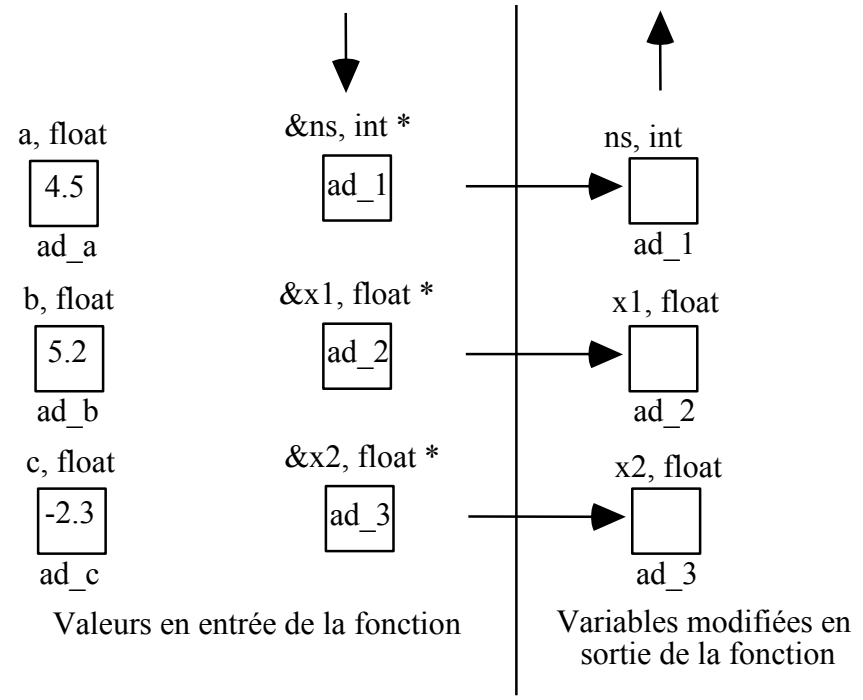
```
x2 = x1;
```

Si on écrit cela, certes désormais x2 pointe sur la même variable que x1 et donc à ce moment *x2 vaut la même chose que *x1. Mais, le pointeur x2 n'est pas une variable en sortie. Donc, à la sortie de la fonction, les valeurs de x2 et de *x2 n'auront pas été modifiées!

- On précise les entrées et les sorties de la fonction en commentaires, avec la signification des variables et les valeurs particulières qu'elles peuvent prendre.

4 Utilisation :

```
main ()
{
    float x1, x2, a, b, c;
    int ns;
    equa1 (b, c, &ns, &x1);
    ..
    equa2 (a, b, c, &ns, &x1, &x2);
    ..
    printf ("x1=%f, x2=%f, ns=%d \n",x1, x2, ns);
    ..
    equa2 (1, 2, 3, &ns, &x1, &x2);
    ..
    printf ("x1=%f, x2=%f, ns=%d \n",x1, x2, ns);
}
```



5 à noter que :

- pour chaque fonction, on précise en commentaire la liste des paramètres en entrée et en sortie. On redonne leur type, on explique leur sens, on explicite les cas particuliers.
- dans l'appel de equa2, a, b et c sont des valeurs : on aurait pu les remplacer par des expressions (par exemple 3 ou 5*4). &ns, &x1, &x2 sont aussi des valeurs : ces valeurs sont des adresses. Ces trois paramètres sont des expressions : expressions construites avec l'opérateur & et les opérandes ns, x1 et x2. On pourrait, dans l'absolu, les remplacer par n'importe quelle expression, à condition qu'elle fournisse l'adresse d'une variable.
- l'appel d'équa1 dans le main est différent de l'appel d'équa1 dans équa2. Ceci vient du fait que les variables x1 et nbsol de equa2 ne sont pas du même type que les variables x1 et ns dans le main. Dans le main, x1 est un float, ns est un entier. Dans equa2, x1 est un pointeur de float, nbsol est un pointeur d'entier.

10.4 Procédure et fonction en même temps

En C, une fonction peut à la fois renvoyer une valeur comme une fonction et avoir des variables en sortie.

Dans le cas de la procédure "equa2", on pourrait choisir de renvoyer le nombre de solutions comme valeur de la fonction et de garder les solutions comme paramètres en sortie de la procédure.

Dans ce cas on écrirait :

1 C :

```
  /*  fonction equa2
      E:  a, b, c: réels (paramètres de l'équation)
      S standard :entier (nombre de solutions de l'équation, -1
pour l'infini)
      S:  *x1, *x2 : réels (solutions de l'équation. x1 = x2 s'il
n'y a qu'une solution)
  */

  int equa2(float a, float b, float c, float *x1, float *x2)
  {
      float delta;
      int nbsol;
      if (a == 0) {
          nbsol = equa1 (b, c, x1) /* pas de & ! */
          *x2 = * x1; /* pas x2 = x1 ! */
          return nbsol;
      }
      delta = b*b - 4*a*c;
      if (delta > 0) {
          nbsol = 2;
          *x1 = (-b - sqrt(delta)) / (2*a);
          *x2 = (-b - sqrt(delta)) / (2*a);
      }
      else if (delta == 0 ) {
          nbsol = 1;
          *x1 = (-b) / (2*a);
          *x2 = x1;
      }
      else {
          nbsol = 0;
      }
      return nbsol;
  }
```

2 Utilisation :

```
main ()
{
    float x1, x2, a, b, c;
    int ns;
    …
    ns = equa1 (b, c, &x1);
    …
    ns = equa2 (a, b, c, &x1, &x2);
    …
    printf ("x1=%f, x2=%f, ns=%d \n",x1, x2, nbsol);

}
```

On sait maintenant tout ce qu'il faut savoir sur les modes de passage des paramètres dans les fonctions C. On mettra tout cela en application dans les cas des tableaux, des chaînes de caractères, des listes chaînées et autres arbres.

10.5 Procédure avec variables en entrée-sortie

Considérons un exemple simple : celui d'une procédure qui permet d'inverser deux valeurs.

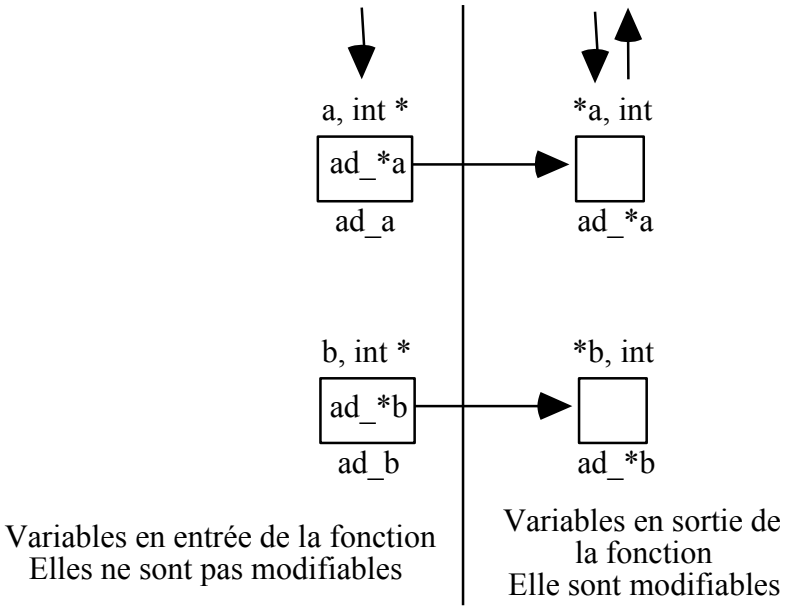
1 Analyse :

E : a, b : entier;

S : a, b : entier;

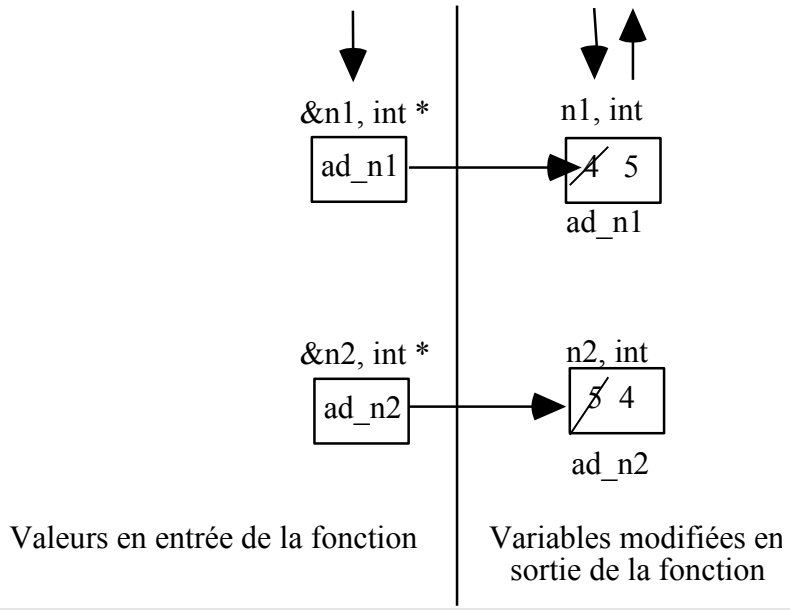
2 C :

```
void inverser (int *a, int *b )
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```



3 Utilisation :

```
main ()
{
    int n1, n2;
    n1 = 4;
    n2 = 5;
    inverser (&n1, &n2);
}
```



10.6 Utilisation de globales dans une fonction : ce qu'il ne faut pas faire ! ! !

```
#include
int surface ();
int largeur, longueur ;
void main (void){
    int surf;
    largeur=2 ;
    longueur=5 ;
    surf = surface( ) ;
    printf(« surface = %d\n », surf) ;
}

int surface ()
{
    int surf;
    surf = longueur * largeur ;
    return surf;
}                                                    essai.c
```

Dans cet exemple, la fonction n'a pas de paramètres. On a deux variables globales : largeur et longueur qui sont utilisées dans le main comme dans la fonction.

Il faut comprendre et mettre en œuvre les notions de paramètres formels et de paramètres d'appel.

10.7 Utilisation de structures "fourre-tout" : ce qu'il ne faut pas faire ! ! !

```
typedef struct fourreTout{
    int nbsol;
    float x1;
    float x2;
} typFourreTout;

typFourreTout equa2(float a, float b, float c)
{
    typFourreTout ft;
    float delta;
    if (a == 0) {
        ft = equa1 (b, c);
        ft.x2 = ft.x1;
        return ft;
    }
    delta = b*b - 4*a*c;
    if (delta > 0) {
        ft.nbsol = 2;
        ft.x1 = (-b - sqrt(delta)) / (2*a);
        ft.x2 = (-b - sqrt(delta)) / (2*a);
    }
    else if (delta == 0 ) {
        ft.nbsol = 1;
        ft.x1 = (-b) / (2*a);
        ft.x2 = x1;
    }
    else {
        ft.nbsol = 0;
    }
    return ft;
}
```

Il ne faut jamais créer une structure uniquement pour passer des paramètres en sortie. Une structure est un type abstrait de donnée. Elle doit toujours correspondre à une certaine forme de réalité. Dans cette exemple, on pourrait par contre définir un type ensemble et renvoyer en sortie l'ensemble des solutions, en ajoutant à ce type des fonctions permettant de calculer le cardinal de l'ensemble et de connaître la valeur de chaque élément de l'ensemble.

CHAPITRE 11 : TABLEAUX ET POINTEURS[7]

11.1 Généralités

1 Présentation

On a vu que dans une fonction, on peut écrire tab[] ou *tab. Expliquons maintenant les raisons.

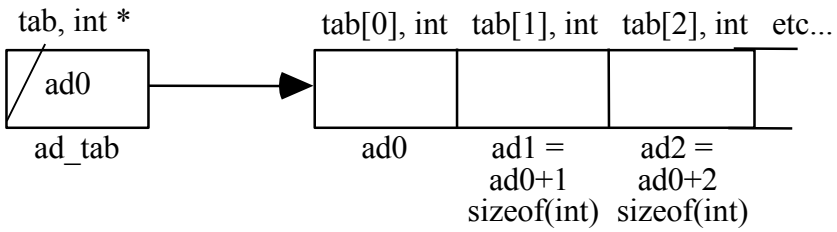
Quand on déclare un tableau :

```
int tab[10];
```

la notation **tab** est équivalente à **&tab[0] :** c'est l'adresse de tab[0].

tab est un pointeur sur le premier élément du tableau.

tab est de type int *



Notez que **tab est une constante** : on ne peut pas modifier sa valeur. Ceci pour éviter de perdre, alors même qu'on a fait une déclaration statique, l'adresse des éléments du tableau.

Les notations suivantes sont équivalentes :

tab	&tab[0]
tab+ 0	&tab[0]
tab+ 1	&tab[1]
tab+ i	&tab[i]
*(tab+i)	*&tab[i] soit tab[i] (puisque *& c'est équivalent à Ø)

On ajoute à tab la taille du type : dans notre exemple ce sont des entiers (sur deux octets). On pourrait avoir un tableau de double (sur huit octets), l'écriture serait la même.

On pourra donc écrire (et lire) :

```
int i, tab[20];
for (i=0; i<20; i++) {
    printf ("entrer tab [%2d] : ", i);
    scanf ("%d", tab+i);
}
for (i=0; i<20; i++) {
    printf ("entrer tab [%2d] = %d", i, *(tab+i));
}
```

On a ici "tab + i" à la place de "&tab[i]" et "*(tab+i)" à la place de tab[i].

---

7 **Début du sixième cours.**

2 Exemple : la fonction maxTab

La fonction maxTab peut s'écrire :

```
int maxTab (int * tab, int taille)
{
    int i, max;
    max = *tab;
    for (i=0; i< taille; i++){
        if (*(tab+i) > max){
            max = *(tab+i) ;
        }
    }
}
```

ou encore, en déplaçant le pointeur :

```
int maxTab (int * tab, int taille)
{
    int i, max;
    for (i=0, max= *tab; i< taille; i++, tab++){
        if (*tab > max){
            max = *tab ;
        }
    }
}
```

tab ++ permet de déplacer le pointeur d'une case (d'un entier parce que tab est un pointeur d'entier).

On peut faire ça parce qu'on est dans une fonction et que tab est un pointeur qui contient la copie de l'adresse du premier élément du tableau. Si cet algorithme était dans un main, ce serait impossible car tab serait alors une constante : la valeur de tab, c'est-à-dire l'adresse de tab[0], ne pourrait pas être modifiée. On ne pourrait donc pas écrire "tab++".

3 Le calcul d'adresse[8]

On a vu qu'on pouvait additionner un entier et un pointeur. Voyons maintenant toutes les opérations possibles sur les pointeurs.

- On peut affecter à un pointeur un pointeur de même type.
- On peut additionner ou soustraire un entier à un pointeur.
- On peut comparer deux pointeurs faisant référence à des éléments d'un même tableau.
- On peut soustraire deux pointeurs faisant référence à des éléments d'un même tableau.
- On peut affecter un pointeur à 0 (NULL).
- On peut comparer un pointeur à NULL.

Tout autre calcul sur les pointeurs est interdit. En particulier, il n'est pas permis d'additionner deux pointeurs, *a fortiori* de les multiplier.

8 K&R91 p. 101.

4 Quelle écriture adopter ?

*int * tab ou int tab [] ?*

Revenons sur le passage de paramètres à une fonction.

Quelle écriture choisir parmi les trois présentées?

- Entre int * t et t [] : il vaut mieux choisir t[] qui a le mérite de signifier que "t" est un tableau. Quand on écrit "int * t", on peut avoir un entier en sortie.
- Entre int t[] et int t[10] : il vaut mieux choisie t[] car
  - le valeur n'a aucune conséquence
  - sa signification est extérieure à la fonction : elle correspond à une variable globale et viole donc les principes de la modularité. Il vaut donc mieux l'éviter.
- Pour cette même raison, on évitera d'autant plus "int t [NMAX]" qui fait référence à une déclaration de constante symbolique.

Enfin, rappelons qu'il faut distinguer entre la **taille déclarée** du tableau et sa **taille utilisée**. On peut avoir un tableau déclaré avec 100 éléments et n'en utiliser que 10. Pour respecter les principes de la modularité, la taille utilisée du tableau doit **toujours** être passée en paramètre.

Précepte n° 12

Dans l'en-tête d'une fonction, associez toujours à un tableau la taille utilisée de ce tableau.

tab+i ou &tab[i] ?

Comme pour le passage de paramètres, il vaut mieux utiliser les crochets qui rendent l'écriture plus lisible.

11.2 Tableaux en sortie d'une fonction

Comment faire pour pouvoir ressortir d'une fonction un tableau dont les valeurs auraient été modifiées?

On peut le faire sans aucune différence avec le cas où le tableau n'est qu'en entrée puisque dans ce cas c'est déjà l'adresse du tableau, c'est-à-dire l'adresse du premier élément du tableau qu'on fournit.

Donc quand on passe un tableau en paramètre d'une fonction, il est toujours potentiellement en sortie.

1 Exemple : fonction qui met à 100 les valeurs d'un tableau d'entiers

```
void initTab (int tab [], int taille)
{
    int i;
    for (i=0; i < taille; i++) {
        tab[i] = 100;
    }
}
```

ou encore :

```
void initTab (int * tab, int taille)
{
    int i;
    for (i=0; i< taille; i++, tab++){
            *tab = 100;
    }
}
```

2 Remarques

L'entête est la même que pour maxTab

C'est pourquoi on prendra soin de commenter les entêtes des fonctions afin de préciser la signification de l'interface :

```
/* initTab
fonction d'initialisation d'un tableau de "taille" entier
E : taille : entier (taille utilisée du tableau)
S : tab : tableau d'entiers
*/
void initTab (int tab[], int taille)
{ … }
```

Appel

L'utilisation de la fonction donnera :

```
void main (void)
{
    int t[50];
    initTab (t, 20);
    …
}
```

On initialise que les 20 premiers éléments du tableau.

Deux valeurs sont passées en paramètres à initTab : 20 et l'adresse de t[0].

3 Usage du mot clé const dans les paramètres formels:[9]

Les différents cas

On vient de voir qu'il n'y a pas de différences syntaxiques entre un tableau en entrée et un tableau en sortie.

En fait, on peut, et il est très conseillé, de marquer cette différence.

Ainsi plutôt que :

```
int max(int tab [], int n)
```

on préférera :

```
int max(const int tab[], int n)
```

La déclaration **const int tab[]** définit une variable de type "pointeur d'entier" référençant (pointant sur) une zone mémoire non modifiable. La variable tab n'est pas modifiable non plus.

Cette formule est équivalente à :

```
int max(const int * const tab, int n)
```

Attention, avec une telle déclaration, on ne pourra plus écrire tab ++. tab est alors une constante non modifiable. Mais de toute façons, il vaut mieux choisir l'écriture avec les crochets.

Les pointeurs constants :

On peut aussi définir dans le programme des pointeurs constants comme on avait défini des entiers constants.

Pour cela, on place l'attribut const juste après l'opérateur de déclaration

```
int * const deb=table;"[10]
```

9 Bra98 p.85.

10 Bra98, p. 85.

11.3 Tableaux à deux dimensions et tableaux de pointeurs[11]

1 Définition

En C, les crochets : " [] " forment un opérateur de déclaration. Cet opérateur nous a permis de déclarer des tableaux à une dimension. On peut appliquer cette opérateur de déclaration plusieurs fois dans une même construction. On déclarera ainsi des tableaux à plusieurs dimensions.

Par exemple:

```
int tab[2][3];
```

Comme pour les tableaux à une dimension, l'identificateur d'un tableau à plusieurs dimensions est une adresse. Mais quel est son type? Quel est le type de "tab" dans l'exemple ci-dessus? Ce n'est plus, comme dans le cas des tableaux d'entiers à une dimension, un pointeur d'entier (int *). C'est un pointeur de bloc de 3 entiers qu'on pourrait noter théoriquement : int[3] * (en pratique on n'utilise jamais cette écriture[12]).

Qu'est-ce que cela change?

tab contient toujours l'adresse de tab[0][0], tab c'est toujours &tab[0][0]. Mais, dans ces conditions, tab+1 correspond à l'adresse de tab augmenté de 3 entiers, c'est-à-dire l'adresse de tab[1][0], tandis que &tab[0][0] + 1 c'est l'adresse de tab augmenté de 1 entier, c'est-à-dire l'adresse de tab[0][1].

Il y a donc les équivalences suivantes[13] :

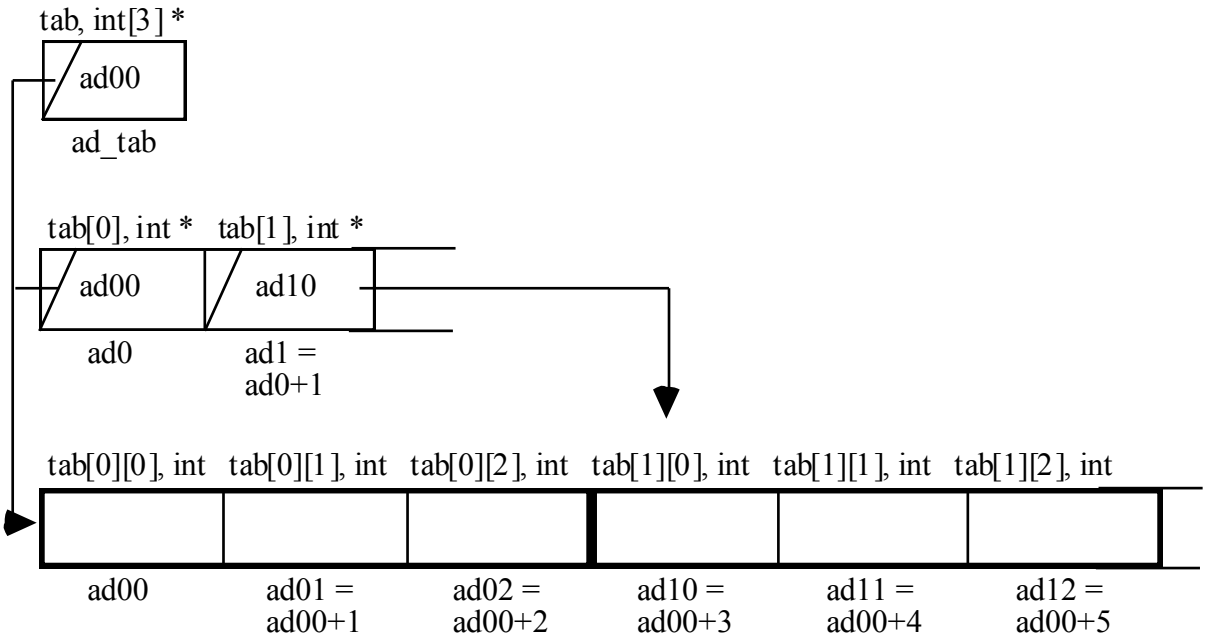
tab	tab[0]	&tab[0][0]
tab+1	tab[1]	&tab[1][0]
tab+2	tab[2]	&tab[2][0]
tab+i	tab[i]	&tab[i][0]

Notons que dans ce cas, de même que tab est une constante, tab[i] est aussi une constante. On ne peut pas modifier sa valeur.

tab[i] + 1 <-> &tab[i][0] + 1 <-> &tab[i][1]

Un tableau à deux dimensions est en fait un tableau à une dimension dont chaque élément est un tableau[14]. On peut donc se représenter un tableau à deux dimensions ainsi :

11 Del97 p. 139, K&R91 p. 108.
12 Del97 p. 139.
13 Del97 p. 140.
14 K&R91 p. 109.



tab[i] donne l'adresse du premier élément de la ligne i.

On peut aussi noter l'équivalence entre : tab[0]+1, &tab[0][0]+1, &tab[0][1]

2 Tableaux à 2 dimensions en argument d'une fonction

Avec le nombre de colonnes en paramètre

Présentation

Soit le tableau : int mat[NL_MAX][NC_MAX]; NL_MAX et NC_MAX étant des constantes symboliques représentant des constantes entières.

"Si l'on doit passer un tableau à deux dimensions en argument à une fonction, il faut préciser le nombre de colonnes au niveau de la déclaration du paramètre; le nombre de lignes est sans importance puisque ce que l'on passe en argument est […], un pointeur sur un tableau de lignes dans lequel chaque ligne est un tableau de [NL_MAX] int."[15]

L'entête de la fonction permettant de trouver le maximum d'une matrice pourra s'écrire :

```
int maxMatrice (int mat[][NC_MAX], int nl, int nc)
```

Avec :

NC_MAX : nombre de colonnes déclarées de la matrice

nl : nombre de lignes utilisées.

nc : nombre de colonnes utilisées.

Soit la fonction :

```
int maxMatrice (int mat[][NC_MAX], int nl, int nc)
{
    int i, max;
    max = mat[0][0];
    for (i=0; i< nl; i++){
        for (j=0; j< nc; j++){
        if (mat[i][j] > max){
            max = mat[i][j] ;
        }
    }
}
```

Avec ce corps, on peut avoir comme entête :

```
int maxMatrice (int * mat[NC_MAX], int nl, int nc)
```

Cette écriture signifie que mat est un pointeur sur un tableau de NC_MAX entiers.

Ou encore :

```
int maxMatrice (int mat[NL_MAX][NC_MAX], int nl, int nc)
```

Comme dans le cas d'un tableau à une seule dimension, on préférera la première écriture.

15 K&R91 pp. 109-110.

Par contre les écritures suivantes sont fausses :

```
int maxMatrice (int mat[][], int nl, int nc)
int maxMatrice (int * mat[], int nl, int nc)
int maxMatrice (int ** mat, int nl, int nc)
```

Elles sont fausses, car du coup, mat[i][j] n'est plus déterminé. En effet, mat[i][j] correspond au (i * NC_MAX + j) ième élément de la matrice. Or on ne connaît plus NC_MAX.

Explications:

- mat est de type int[NCMAX]*
- donc quand j'écris mat[i][j] , j'écris mat +i +j, mais attention, i compte des blocs de NCMAX entiers, j compte des entiers.
- quand j'écris mat[i][j] , j'écris donc mat + i*NCMAX*sizeof(int) + j*sizeof(int).
- j'ai donc besoin de connaître NCMAX pour déterminer mat[i][j]
- Si on ne connaît pas NC_MAX on ne peut pas déterminer mat[i][j].

Typedef

Avec des typedef, on peut ensuite écrire cette entête :

```
#define NCMAX 10
#define NLMAX 20
typedef int typMat [NLMAX][NCMAX];
void maxMat (typMat mat, nl, nc);
```

Présentation

Passer en paramètre nombre de colonnes déclarées de la matrice contredit les principes de la modularité. Comment éviter cela?

En n'utilisant plus les crochets et travaillant directement au niveau des pointeurs.

```
int maxMatrice (int * mat, int nl, int nc)
{
    int i, j, max;
    max = *mat;
    for (i=0; i< nl; i++){
        for (j=0; j< nc; j++){
            if (*(mat+i*nc+j) > max){
                max = *(mat+i*nc+j) ;
            }
        }
    }
}
```

Du coup on peut n'utiliser qu'une seule boucle :

```
    for (i=0; i< nl*nc; i++){
        if (*(mat+i) > max){
            max = *(mat+i) ;
        }
    }
```

Du coup la fonction maxMatrice est identique à la fonction maxTab :

Appel

L'utilisation de la fonction donnera :

```
void main (void)
{
    int t[3][10], res;
    res = maxMatrice ((int*)t, 3, 10);
    …
}
```

Remarque 1

On constate que le type de t (int **) n'est pas le même que celui de la fonction (int *). C'est pour cela qu'on caste. Cependant si on ne le fait pas c'est fait automatiquement.

Cependant, on pourrait aussi écrire l'entête ainsi :

```
int maxMatrice (int ** mat, int nl, int nc)
```

ou encore :

```
int maxMatrice (int *mat[], int nl, int nc)
```

Cela ne changerait pas le corps de la fonction.

Attention :

```
int maxMatrice (int mat[][], int nl, int nc)
```

ne marche pas.

```
int maxMatrice (int *mat[], int nl, int nc)
```

a l'avantage de montrer qu'on a affaire à un tableau à deux dimensions, mais il génèrera des warnings.

Remarque 2

Le tableau passé en paramètre est de type int*. Il faut noter que ce n'est plus un int[NC]*. Donc mat+x c'est tab+x*sizeof(int).

D'où la formule pour accéder aux éléments du tableau : mat+i*ncmax+j.

Remarque 3

Tel qu'on a écrit les fonctions, quand on utilise les pointeurs, le **nombre de colonnes déclarées** doit être le même que le **nombre de colonnes utilisées**.

Dans les deux instructions :

```
        for (j=0; j< nc; i++){
            if (*(mat+i*nc+j) > max){
```

le premier nc, c'est le nombre de colonnes utilisées, le second c'est le nombre de colonnes déclarées.

Quand on écrit :

```
    for (i=0; i< nl*nc; i++){
```

nc, c'est le nombre de colonnes déclarées et utilisées qui doivent donc être identiques.

Si on veut distinguer le nombre de colonnes déclarées et le nombre de colonnes utilisées, on écrira par exemple :

```
int maxMatrice (int * mat, int ncmax, int nl, int nc)
{
    int i, j, max;
    max = mat[0][0];
    for (i=0; i< nl; i++){
        for (j=0; j< nc; j++){
            if (*(mat+i*ncmax+j) > max){
                max = *(mat+i*ncmax+j) ;
            }
        }
    }
}
```

Sans le nombre de colonnes en paramètre, version tableau de pointeurs

Présentation

On voit que la solution précédente est assez lourde puisqu'elle oblige à passer un paramètre supplémentaire et à écrire des formules un peu lourdes ! ! !

On a une autre solution possible et qui est préférable : les tableaux de pointeurs.

On peut écrire :

```
int maxMatrice (int * mat[], int nl, int nc)
{
    int i, j, max;
    max = mat[0][0];
    for (i=0; i< nl; i++){
        for (j=0; j< nc; j++){
            if (mat[i][j] > max){
               max = mat[i][j] ;
            }
        }
    }
    return max;
}
```

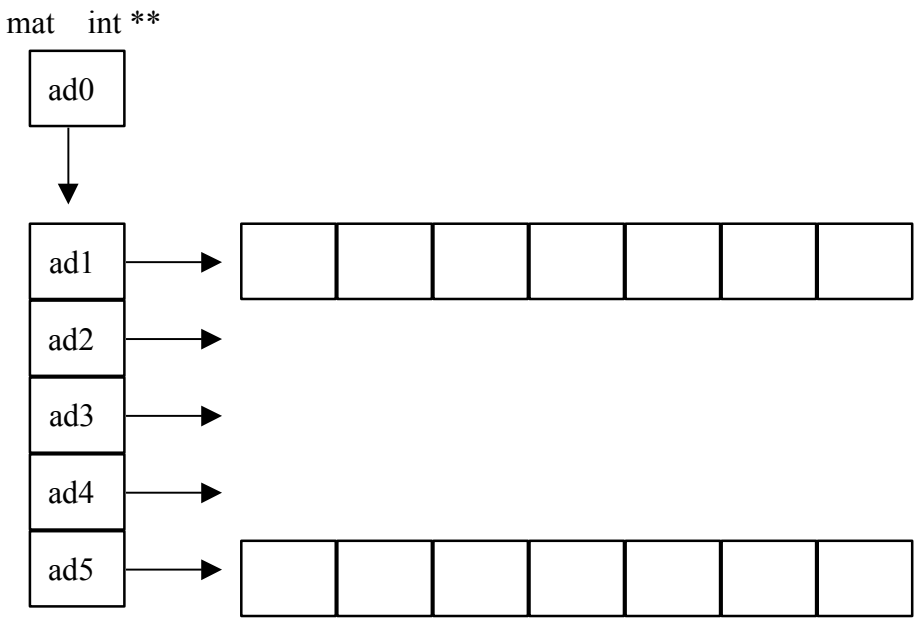
A condition que

mat ne soit pas un tableau à deux dimensions classique, mais un tableau de pointeurs, déclaré dynamiquement comme dans le main suivant :

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

int maxMatrice (int * mat[], int nl, int nc);

void main(void) {
    int **mat ;
    int i, j, nl, nc, max;
    nl=3 ;
    nc=4 ;
    mat = (int**) malloc(sizeof(int*)*nl);
    for(i=0;i<nl;i++){
        mat[i]=(int*)malloc(sizeof(int)*nc);
    }
    for(i=0; i<nl; i++){
        for(j=0; j<nc; j++){
            printf("mat[%2d][%2d]=", i,j);
            scanf("%d", &mat[i][j]);
        }
    }
    for(i=0; i<nl; i++){
        for(j=0; j<nc; j++){
            printf("mat[%2d][%2d]=%3d  ", i,j,mat[i][j]);
        }
        printf("\n");
    }
    max=maxMatrice(mat, nl, nc);
    printf("max=%3d  ", max);
}
```



Notez que :

Dans la fonction, on aurait pu écrire :

```
int maxMatrice (int * mat[], int nl, int nc)
```

Par contre dans le main, on ne peut pas écrire :

```
void main(void) {
    int *mat[] ;
```

11.4 Précisions sur l'allocation dynamique de mémoire

1 malloc

```
#include <stdlib.h>
void * malloc (size_t taille_elt);
```

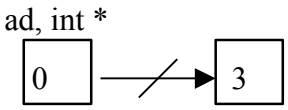
On a vu qu'avec le malloc, on peut créer des variables sans nom auxquelles on accèdent avec un pointeur.

Quand on fait ainsi, il faut faire attention à ne pas « perdre » sa variable !

Par exemple :

```
ad=malloc(2) ;
*ad=3;
ad=NULL ;
```

Avec ces instructions, la mémoire est allouée : elle ne pourra pas être allouée une nouvelle fois. Elle est aussi perdue : on ne peut plus y accéder.



La durée de vie des variables créées avec un malloc est controlée par le programmeur : tant que la mémoire de la variable dynamique n'a pas été libérée, la variable dynamique existe.

Pour libérer la mémoire allouée dynamiquement, on utilise la fonction free.

affichage de l'adresse : %p

```
printf(« %p », ad) // donne l'adresse en hexa
```

l'opérateur sizeof et les pointeurs

```
ad=malloc(2);
printf(« %d », sizeof(ad) ) ; // la taille d'un poointeur
                              // vaut en général 4
```

Attention à bien distinguer les deux cas suivants :

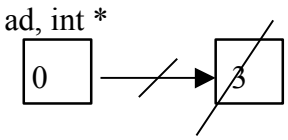
```
int tab[10];
printf(« %d »,sizeof(tab));// vaut 40 : 10 * sizeof(int)
```

```
int maxTab(int * tab, int n) {
printf(« %d »,sizeof(tab)); // vaut 4, la taille d'un pt
```

2 free

exemple

```
ad=malloc(2) ;
*ad=3;
free (ad);
```



On constate qu'il n'est pas nécessaire de préciser la taille qu'on va libérer. Celle-ci correspond au type de la variable pointée, c'est-à-dire au type de *ad. Cette taille est donnée par sizeof(*ad).

syntaxe

```
#include <stdlib.h>
void free (void * ad)
```

free libère une zone mémoire précédemment allouée par une fonction d'allocation. Elle reçoit l'adresse de la zone à libérer en paramètre.

3 calloc et realloc[16]

Voyons maintenant les deux autres fonctions d'allocation, moins utilisées que les précédentes

calloc

```
#include <stdlib.h>
void * calloc (size_t nb_elt, size_t taille_elt);
```

- calloc alloue une zone de : **nb_elt * taille_elt** octets.
- **Tous les octets sont initialisés à zéro.**
- On peut toujours utiliser malloc à la place de calloc. calloc est intéressant dans le cas de tableau de structures, de tableau de chaînes ou de tableau de tableaux (tableau à 2 dimensions par exemple) car il rend la formule d'allocation plus claire. Il est aussi intéressant pour son initialisation.

16 **Début du septième cours.**

```
#include <stdlib.h>
void * realloc (void  * ad, size_t nouvelle_taille)
```

- realloc modifie la taille de la zone pointée par **ad**.
- ad doit pointer initialement sur une zone précédement allouée dynamiquement, ou valoir NULL auquel cas realloc équivaut à malloc.
- realloc renvoie l'adresse du nouveau bloc. L'adresse de l'ancien bloc est perdue. En cas de problème, realloc renvoit NULL. Si la nouvelle taille est supérieure à la précédente, les nouveaux octets ne sont pas initialisés (leur contenu est indéterminé).
- realloc est pratique pour gérer des tableaux dynamiques, mais ce n'est pas une technique rapide.

11.5 Tableaux à n dimensions

Les principes sont les mêmes pour les tableaux à plus de deux dimensions.

CHAPITRE 12 : LES CHAINES DES CARACTERES

12.1 Généralités

1 Les constantes chaînes de caractères

On a déjà utilisé des constantes chaînes de caractères, dans les fonctions printf et scanf.

Les constantes chaînes de caractères sont définies entre guillemets (doubles quotes).

```
"voici une chaîne"
"" /* chaîne vide */
```

On peut utiliser les caractéristiques de l'antislash : \n, \t, etc.

A la compilation, deux chaînes juxtaposées sont concaténées. "bonjour" "monsieur" donne "bonjour monsieur". C'est utile pour diviser des longues chaînes en plusieurs lignes.

2 Convention de fin de chaîne

Le dernier caractère d'une chaîne est invisible : c'est le caractère '\0'.

Donc la chaîne "a" est différente du caractère 'a'. La chaîne "a" est constituée des caractères 'a' et '\0'.

3 Déclaration d'une chaîne de caractères

Tableau de caractères

En C, il n'existe pas de type chaîne de caractères, comme int est un type entier (dans d'autres langage, il existe un type chaîne de caractères). Cependant, les chaînes de caractères sont manipulables bien qu'elles ne soient pas typées (c'est la même chose pour les booléens). On a déjà vu qu'il existait un type caractère : char.

Pour travailler sur des chaînes de caractères C, on utilise des tableaux de caractères.

```
char ch[20];
```

ch est un tableau de 20 caractères.

Ce tableau de caractères pourra être considéré comme une chaîne si au moins un des caractères est un '\0'. Dans ce cas les caractères suivants ne seront plus pris en compte.

En tant que chaîne, ch aura 19 caractères maximum.

ch est en fait un pointeur de caractère : char * ch contient l'adresse du premier élément du tableau.

Donc une variable permettant d'accéder à une chaîne de caractères sera de type " char * ".

Pointeur de caractères

Il y a donc une deuxième façon de déclarer une chaîne de caractères :

```
char * ch;
```

ch est ici un pointeur sur un caractère ou sur une chaîne de caractères. Bien sur, dans ce cas, il n'y a eu que l'allocation de la mémoire nécessaire pour un pointeur, pas pour une chaîne de caractères.

4 Initialisation à la déclaration

Tableau de caractères

```
char ch[20] = "bonjour";
char ch1[20] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
char ch2[] = "bonjour";
```

- Les deux premières instructions sont équivalentes : dans un cas on affecte une chaîne, dans l'autre tous les caractères (il ne faut pas oublier l' \0).
- La dernière instruction réserve automatiquement un tableau de 8 caractères : les 7 lettres du mot et le "\0".
- Dans les trois cas : ch, ch1 et ch2 sont des tableaux. On peut donc modifier les éléments du tableau mais on ne peut pas modifier l'adresse contenu dans ch, ch1 ou ch2, c'est-à-dire l'adresse du premier élément du tableau.

Pointeur de caractères

```
char * ptch = "bonjour";
```

- Cette instruction réserve aussi automatiquement un tableau de 8 caractères : les 7 lettres du mot et le "\0".
- Par contre, ptch est un pointeur initialisé de telle sorte qu'il pointe sur une constante de type chaîne. On peut modifier ptch de telle sorte qu'il pointe sur autre chose. La mémoire allouée est alors libérée.

5 lecture et écriture : %s

```
#include <stdio.h>
void main (void) {
    char s[10] = "bonjour";
    char nom[20];
    scanf("%s", &nom);
    printf ("%s monsieur %s\n", s, nom);
}
```

```
c :>test
Dupond
bonjour monsieur Dupond
c :>
```

- Le "nom" du scanf pourrait ne pas être écrit avec un "&". En effet, "nom" est déjà une adresse.

- Si on entre « Dupond Jean », le scanf ne prend que « Dupond ».

6 Exemple

```
#include <stdio.h>
void main (void) {
    char s[5] = "bla";
    int i;
    for (i=0; i<7; i++) {
        printf ("%d=<%c><%d> \n", i, s[i], *(s+i));
    }
}
```

```
0=<b><98>
1=<l><108>
2=<a><97>
3=< ><0>
4=< ><0>
5=<&><-12>
6=<&><-12>
```

- Comme s[5] a été initialisé à "bla", les 10 caractères de la chaîne sont initialisés : à "\0" s'il n'y a pas de caractère.
- Si on n'itialise pas s[5], les caractères contiennent n'importe quoi.
- Si on affiche au delà du 5ème, on obtient n'importe quoi.

12.2 Algorithmique du traitement de chaînes

Avant de présenter les fonctions C de traitement de chaînes, on va présenter une sorte d'axiomatique du traitement des chaînes, c'est-à-dire le jeu minimal de fonctions qui permet de faire tous les traitements possibles sur les chaînes de caractères.

1 Affectation

Il faut pouvoir affecter une valeur dans une variable chaine :

```
ch2 <- ch
```

ou bien :

```
ch <- « bonjour »
```

2 Longueur

La longueur d'un chaîne c'est nombre de caractères utilisés (et non pas déclarés).

```
int lengtn(ch)
```

vaut 7 avec l'affectation précédente.

3 Concaténation

Il faut pouvoir coller des chaînes les unes derrière les autres : c'est la concaténation.

```
ch3 <- concat (ch1, «  » , ch2)
```

Avec ch2 valant « Monsieur » cela donne : « bonjour Monsieur » dans ch3.

Length de ch3 vaut 16.

4 Extraction

Il faut pour pouvoir extraire un morceau de chaine d'une autre chaine

```
ch2 <- Extract (ch1, 3, 2) ;
```

ch2 vaut « nj »

5 Comparaison

On peut comparer deux chaînes entre elles :

```
si ch1 = ch2
```

Ou bien

```
si ch1 < ch2
```

Avec l'inégalité, on compare, la valeur ASCII de chaque caractère.

6 Application

Avec ces cinq fonctionnalités, je peux écrire tous les algorithmes de traitement de chaînes.

Par exemple, si je veux compter le nombre d'occurrences d'un mot dans une chaîne, j'écrirai :

```
Fonction nbOccur :
    E : texte, mot : chaines
    S : nbOcc : entier
Début de fonction
    NbOcc=0 ;
    Pour i allant de 1 à length(texte)
        Si extract(texte, i, lentgth(mot) = = mot)
            NbOcc++ ;
        Fin de si
    Fin de pour
Fin de fonction
```

12.3 Premières fonctions C de traitement de chaînes

Il existe un certain nombre de fonctions de traitement de chaînes de caractères définies dans la bibliothèque standard du C et utilisables grâce aux fichiers d'entêtes : <string.h>.

1 Affectation : strcpy

On ne peut pas écrire :

```
char ch[10], *ptch ;
ch = "bonjour"; /* erreur */
```

pas plus que :

```
ch = ptch; / * erreur : ch n'est pas une lvalue */
```

En effet, on a vu avec les tableaux que le nom du tableau n'est pas une lvalue, est une constante qu'on ne peut pas modifier[17].

On peut par contre écrire :

```
ptch = "monsieur";
```

Dans ce cas, ptch pointe sur la chaîne de caractères constante "monsieur".

ou

```
ptch = ch;
```

Les deux instructions sont équivalentes : elles consistent modifier l'adresse contenue par ptch. Dans le premier cas, ptch pointe sur la chaîne constante, dans le second, ptch pointe sur la même chose que ch.

La fonction d'affectation : strcpy

Pour pouvoir affecter une valeur à une variable chaîne de caractères, on utilise la fonction :

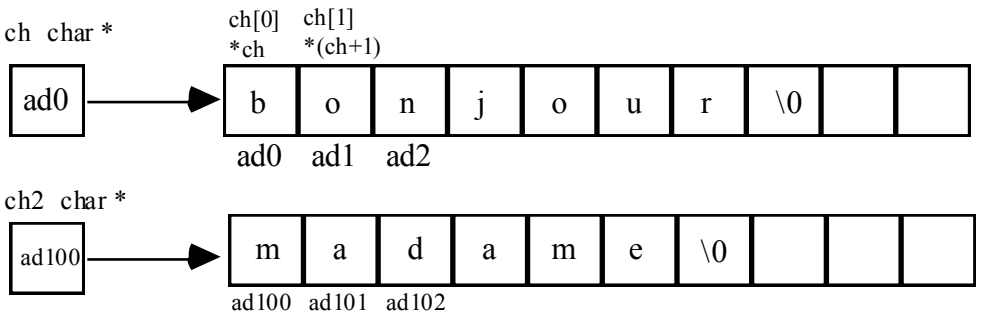
```
#include <string.h>
/* fonction strcpy : copie la chaîne che dans la chaîne chs.
retourne le pointeur chs.
        E : che : char * (chaîne de caractères)
        S : chs : char * (chaîne de caractères)
        Sstd :    char * (même valeur que chs)
*/
char * strcpy (char *chs, const char *che)
```

17 Del97 pp. 155-156.

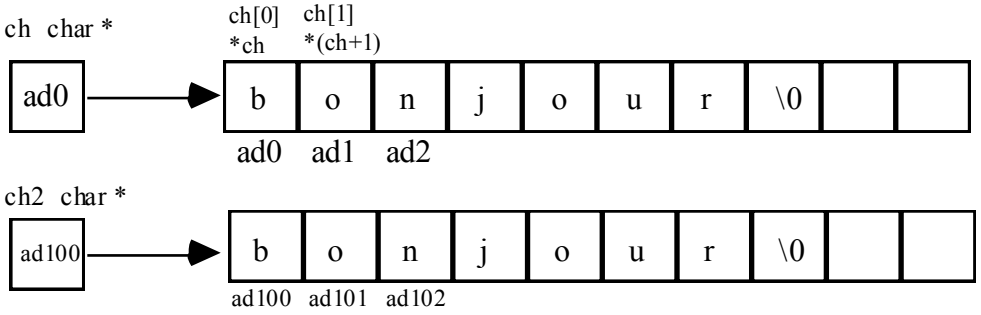
Exemple :

```
char ch[10] = "bonjour";
char ch2[10];
char * ch3;
strcpy (ch2, ch);                          /* ch2 vaut bonjour */
strcpy (ch2, "monsieur");                 /* ch2 vaut monsieur */
ch3 = strcpy (ch2, "madame");              /* ch2 vaut madame */
```

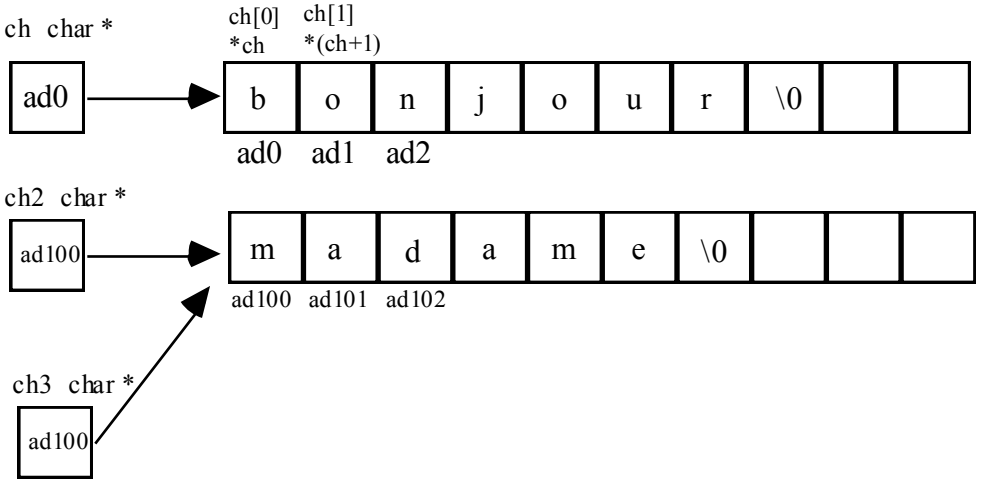
Situation de départ :



Situation avant strcpy (ch2, ch) :



Situation après ch3 = strcpy (ch2, "madame"):



on ne peut pas écrire :

```
  ch2 = strcpy (ch2, "madame");
```

En effet, ch2 est une lvalue[18], qui ne peut pas être modifiée.
On ne peut pas non plus écrire :

```
  ch3 = strcpy (ch3, "madame");
```

Car ch3 ne pointe sur rien, il n'y a donc pas de mémoire dans laquelle faire la copie.

2 Comparaison : strcmp

On ne peut pas comparer directement deux chaînes, car ce qu'on comparerait alors ce serait des adresses. Il y a donc une fonction pour faire cela.

```
  #include <string.h>
  /*  fonction strcmp : compare la chaîne ch1 à la chaîne ch2.
Retourne *ch1 - *ch2 : 0 si les deux chaînes sont identiques, une
valeur positive si ch1 > ch2, une valeur négative si ch1 < ch2.
          E : ch1, ch2 : char * (chaîne de caractères)
          Sstd :    int
  */
  int strcmp (const char * ch1, const char * ch2)
```

3 Longueur : strlen

```
  #include <string.h>
  /*  fonction strlen: renvoie la longueur de la chaîne che.
          E : che : char * (chaîne de caractères)
          Sstd :    int
  */
  int strlen (const char * che)
```

4 Concaténation : strcat

```
  #include <string.h>
  /*  fonction strcat: concatène la chaîne che à la suite de la
chaîne ches et renvoie le pointeur ches.
          E : che : char * (chaîne de caractères)
          ES : ches : char * (chaîne de caractères)
          Sstd :    char * (chaîne de caractères)
  */
  char * strcat (char * ches, const char * che)
```

18 Bra98 p. 66.

Notez que ches est en entrée-sortie : en effet, si c'est un tableau de caractères, en tant que pointeur ce n'est pas une l_value, donc on ne peut pas modifier sa valeur. Par contre on peut modifier les valeurs du tableau.

Ce pointeur est aussi renvoyé par la fonction, mais il ne pourra être affecté qu'à une lvalue.

On peut écrire :

```
char ch[20] = "ab";
char * ptch;
ptch = strcat (ch, "cd");
```

On ne peut pas écrire :

```
char ch[20] = "ab";
ch = strcat (ch, "cd");
```

5 Extraction

Il n'existe pas de fonction d'extraction d'un morceau de chaîne dans la librairie standard. On peut tout de même extraire un caractère en considérant la chaîne comme un tableau :

Extract (ch, i, 1) c'est ch[i]

On verra comment écrire la fonction d'extraction.

6 Exemples de codage des fonctions de traitement de chaînes :

strcpy[19] :

version sans pointeur :

```
char * strcpy (char *s, const char *e)
{
    int i;
    i=0;
    s[0]=e[0];
    while ( e[i] != '\0' ) {
        i++;
        s[i] = e[i];
    }
    return s ;
}
```

version avec pointeurs :

```
{
    char * pt=s;
    *s = *e;
    while ( *e != '\0' ) {
        e++;
        s++;
        *s = *e ;
    }
    return pt;
}
```

version compacte :

```
{
    char * pt=s;
    while ( (*s = *e) != '\0'){
        s++;
        e++;
    }
    return pt;
}
```

version plus compacte :

```
{
    char * pt=s;
    while ( (*s++ = *e++) != '\0');
    return pt;
}
```

19 K&R91 pp. 102-103.

version ultra compacte :

On sait qu'un entier peut être considéré comme un booléen. On sait qu'un caractère peut être considéré comme un entier. Donc un caractère peut être considéré comme un booléen. Si un caractère vaut '\0' alors en tant que booléen il est faux. Si un caractère est différent de '\0' alors il est vrai en tant que booléen. Donc écrire != '\0' c'est comme écrire différent de faux, donc écrire = vrai, donc on peux ne rien écrire :

```
{
    char * pt=s;
    while (*s++ = *e++);
    return pt;
}
```

Il faut éviter de coder comme cela dans les projets! Cependant, il faut aussi savoir lire ce genre d'écriture.

Quand utiliser l'écriture compacte ?

- JAMAIS !!!

La vitesse n'a aucune importance dans un premier temps. L'important c'est la clarté, c'est-à-dire la simplicité de l'algorithme de résolution du problème.

Précepte n°13

Il faut toujours privilégier la clarté du code sur sa prétendue rapidité

Bref : évitez les astuces !

Ce type d'écriture est tolérable si elle s'applique à des fonctions :

- de bas niveau, très courtes et dont le sens est facile à comprendre.
- faciles à tester et à valider.
- non évolutives.

Ces conditions n'étant presque jamais réalisées, on en revient au précepte précédent !

```
int strlen (const char *che)
{
    int n;
    for (n=0;;n++){
        if (che[n] == '\0') {
            return n;
        }
    }
}
```

On peut aussi écrire :

```
{
    int n;
    for (n=0; *che; che++, n++);
    return n;
}
```

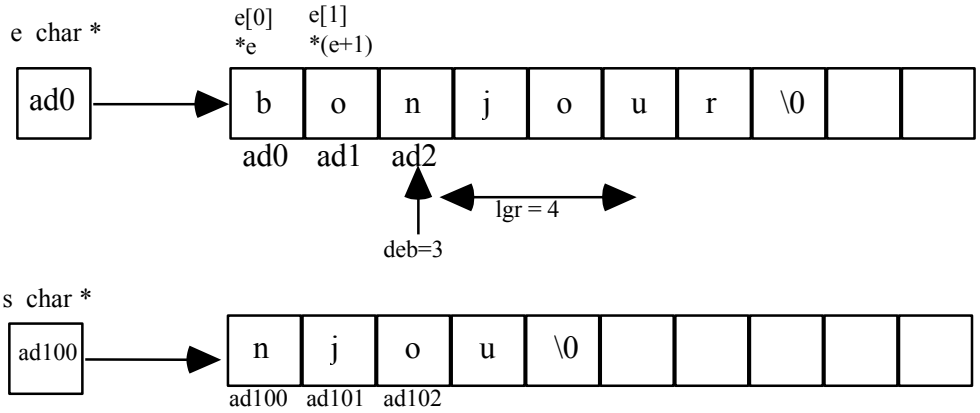
ou encore

```
{
    char *p=che;
    while (*p++);
    return p - che-1;
}
```

Dans cet exemple :

- on incrémente p jusqu'à ce qu'on tombe sur '\0', puis renvoi l'adresse de '\0' - l'adresse du premier caractère -1 : ça nous donne la longueur.
- char *p=che génère un warning : en effet, on va pouvoir modifier la chaîne che via p, malgré le const char de l'en-tête.

```
  char * extract (const char *e, char *s, int deb, int lgr)
  /*   copie dans s la sous chaîne de e démarrant en déb et de
longueur lgr
      renvoie s
```




```
  */
  {
      int i;
      /* 1 : si deb est négatif, on le met à 0*/
      if (deb <0) deb = 1;
      /* 2 : si deb dépasse la taille de e, on revoit une
             chaine vide */
      for (i=0; i<deb; i++) {
          if (e[i] == '\0') {
              s[0] = '\0';
              return s ;
          }
      }
      /* 3 : cas général : on copie lgr caractères, sauf si
             on tombe sur la fin de chaine de l'entrée
             De plus, si lgr est < 1 on sort de suite de la
             boucle */
      for (i=0; i < lgr && e[deb-1+i]!='\0'; i++) {
           printf("%c", e[deb-1+i]);
           s[i] = e[deb-1+i];
      }
      /* après avoir copié les caractères, on met le \0 */
      s[i] = '\0';
      return s;
  }
```

12.4 Autres fonctions C de traitement de chaînes : tout <string.h> ! ! !

1 strncat

```
char * strncat (char * ches, const char *che, int nb)
```

concatène nb caractères de che derrière ches. Renvoit l'adresse de ches.

2 strncmp

```
int strncmp(const char* che1, const char * che2, int n)
```

compare les n premiers caractères de che1 et che2.

3 strncpy

```
char * strncpy (char * ches, const char * che, int nb)
```

copie les nb premiers caractères de che dans ches. Renvoit l'adresse de ches.

4 strchr

```
char * strchr (const char * che, char c)
```

renvoit l'adresse de la première occurrence de c dans che, NULL si pas trouvé.

Il existe encore d'autres fonctions de traitement de chaînes. Il faut les chercher dans les livres ou dans l'aide du compilateur.

12.5 Fonctions C du traitement de caractères : tout <ctype.h> ! ! !

Toutes les fonctions de traitements de caractères ont le même prototype :

```
int is--- (char)
```

Les fonctions is--- (isalpha par exemple) teste le caractère en entrée. Elles renvoient un entier booléen, 1 ou 0, vrai ou faux, ou un caractère (donc dans tous les cas, un entier).

A connaître, principalement :

isalpha :	lettres	A-Z, a-z
isupper :	majuscules	A-Z
islower :	minuscules	a-z
isdigit :	chiffres	0-9
isalnum :	lettres et chiffres	isalpha et isdigit
isspace :	espaces	
toupper :	retourne une majuscule	
tolower :	retourne une minuscule	
ispunct :	caractère imprimable différent de alphanum et espace	

Il existe encore d'autres fonctions de traitement de caractères. Il faut les chercher dans les livres ou dans l'aide du compilateur.

12.6 Lecture et écriture formatée d'une chaîne de caractères

```
#include <stdio.h>
int sprintf (char * chs, const char * format, …)
```

Permet d'écrire, non plus à l'écran, mais dans une chaîne de caractères. La fonction renvoie la longueur de la chaîne construite.

```
#include <stdio.h>
int sscanf (char * che, const char * format, …)
```

Permet de lire, non plus au clavier, mais à partir d'une chaîne de caractères.

On verra un usage de ces fonctions dans les § 15 et 16.

12.7 Les tableaux de chaînes de caractères

Abordons maintenant les tableaux de chaînes de caractères.

Imaginons un tableau contenant des chaînes de caractères. Ce tableau peut être vu comme un tableau classique à deux dimensions :

```
char tabmot[3][8];
```

tabmot est un tableau de 3 lignes contenant chacune un tableau de 8 caractères. C'est un tableau de 24 caractères. Ce tableau sera utilisé selon les mêmes règles que dans le cas d'un tableau d'entiers.

Si par contre nous écrivons :

```
char *tabmot[3]
```

tabmot est un tableau de 3 pointeurs de caractère.

Voyons l'initialisation et la représentation dans les deux cas :

```
char tabmot[3][8] = {"Pascal", "C", "ADA"};
```

tabmot est ici un tableau de 3 fois 8 caractères. On pourrait aussi écrire :

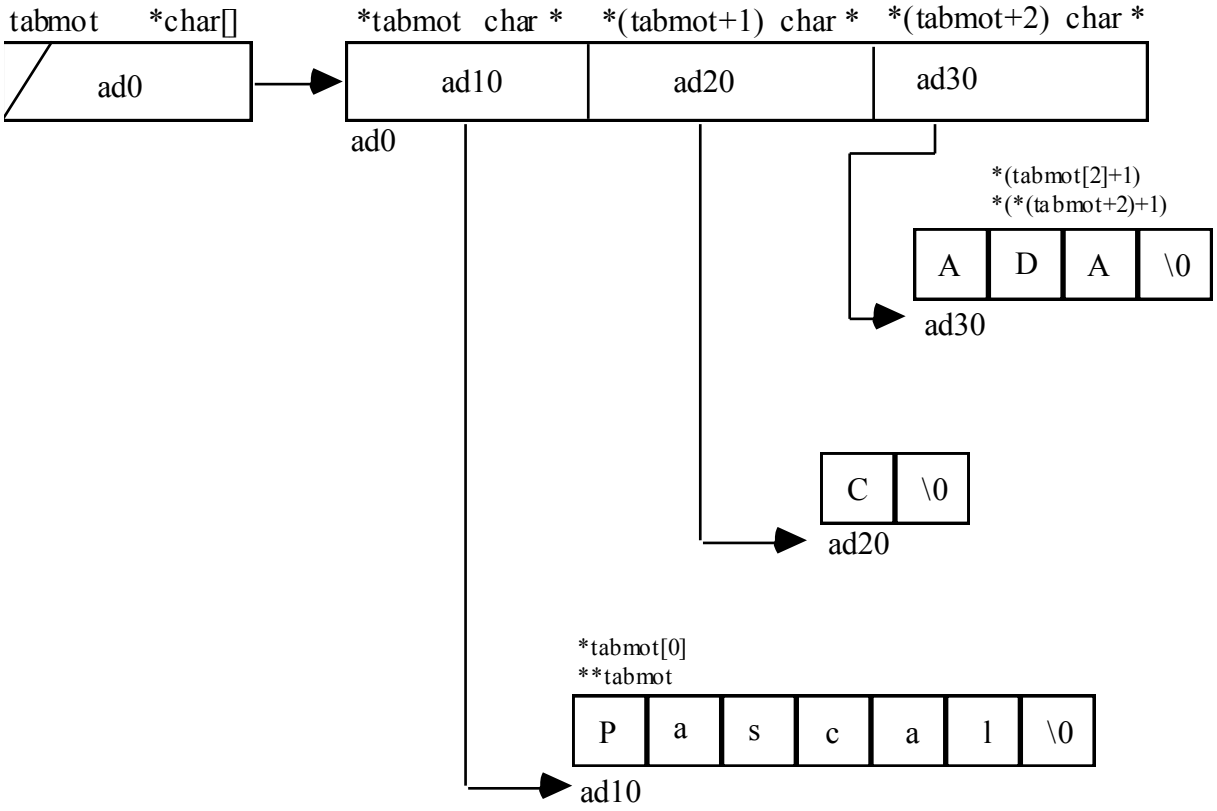
```
char tabmot[][8] = {"Pascal", "C", "ADA"};
```

Les deux écritures sont équivalentes.

Voyons maintenant le cas suivant :

```
char * tabmot[3] = {"Pascal", "C", "ADA"};
```

Cette fois ci, tabmot est un tableau de char *, c'est-à-dire un tableau de pointeurs, un tableau de 3 pointeurs de caractères. Les pointeurs sont initialisés à la valeur des adresses des constantes proposées.



La différence entre un tableau de pointeurs et un tableau de tableau, c'est que le tableau de pointeurs contient des valeurs modifiables : ce sont des lvalues.

L'intérêt d'un tableau de pointeurs par rapport à un tableau de tableaux, c'est d'abord que la place mémoire réservée est plus petite, c'est ensuite que toutes les modifications des cases du tableau seront beaucoup plus rapides, par exemple pour un tri. En effet, il n'est pas nécessaire de déplacer les chaînes de caractères, il suffit de mettre à jour les pointeurs. Enfin cela permet, comme on l'a déjà vu, un usage plus aisé dans les fonctions.

Remarque :

Avec le tableau de pointeurs, je peux écrire :

```
char * tabMot[3] ;
strcpy(tabMot[0], "ADA") ;
strcpy(tabMot[0], "Pascal") ;
```

Il n'est pas nécessaire de faire d'allocation dynamique. La première fois, tabMot[0] pointe sur « ADA », la seconde fois sur « pascal ».

12.8 Argc, argv

1 Premier exemple

Considérons le fichier "argument.c" contenant le programme suivant :

```
# include <stdio.h>

void main(int argc, char *argv[] )
{
    int i;
    printf ("vous avez passé %d argument%s au main \n",
           argc-1, (argc-1) > 1 ? "s" :"");
    printf ("vous avez écrit : ");
    for (i=0; i < argc; i++) {
        printf ("%s ", argv[i]);
    }
    printf (" \n");
}                                                    argument.c
```

Si on exécute ce programme (dans notre exemple sous UNIX) :

```
$ argument
vous avez passé 0 argument
vous avez écrit : argument
$ argument a b c
vous avez passé 3 arguments
vous avez écrit : argument a b c
$ argument a
vous avez passé 1 argument
vous avez écrit : argument a
$                                                    UNIX
```

A noter dans ce programme:

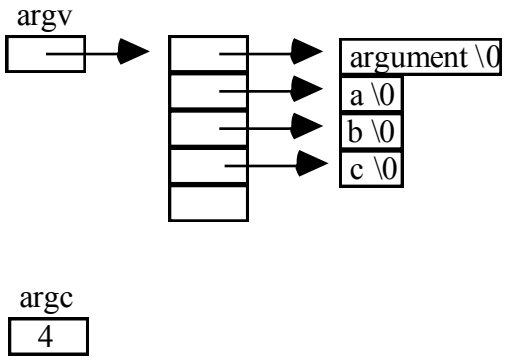
int argc :

- C'est un paramètre du main : c'est le nombre d'arguments du main.

char * argv []:

- C'est un paramètre du main : c'est un tableau contenant les arguments du main et, en premier, le nom du main (dans notre exemple : argument).

  Ce tableau est déclaré comme un pointeur sur un tableau de pointeur.



2 Deuxième exemple

Considérons le fichier "equa1.c" contenant le programme suivant :

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

int equa1 (float, float, float *);

void main(int argc, char *argv[] )
{
    float a, b, res;
    int nbsol;
    if (argc != 3) {
        printf ("Usage : %s <a> <b> \n", argv[0]);
        return;
    }
    a = (float) atof (argv[1]);
    b = (float) atof (argv[2]);
    nbsol = equa1 (a, b, &res);
    switch (nbsol) {
        case -1 :
            printf ("il y a une infinité de solutions \n");
            break;
        case 0 :
            printf ("il n'y a pas de solution \n");
            break;
        case 1 :
            printf ("la solution = %f \n", res);
            break;
    }
}

int equa1 (float a, float b, float * x) {
    if(a==0 && b==0) return -1;
    if(a==0) return 0;
    *x=-b/a;
    return 1;
}
```

equa1.c

Si on exécute ce programme (dans notre exemple sous UNIX) :

```
$ equa1
Usage : equa1 a b
$ equa1 2 -4
la solution est 2.000000
$
```

UNIX

CHAPITRE 13 : STRUCTURE ET FONCTION : POINTEUR DE STRUCTURE

13.1 structure en entrée d'une fonction : passage par valeur

```
typedef struct eleve {
    char    nom[20]
    char    groupe;
    int     note;
} typEleve;
```

fonction d'affichage d'un élève :

```
void afficheEleve (typEleve e);
```

Le corps de la fonction ne pose pas de problème particulier.

13.2 structure en sortie d'une fonction : passage par adresse

Reprennons l'exemple du type eleve, avec deux fonctions : une fonction de saisie et une fonction d'affichage.

```
#include <stdio.h>
typedef struct eleve {
    char    nom[20];
    char    groupe;
    int     note;
} typEleve ;
void saisieEleve (typEleve *e) ;
void afficheEleve (typEleve e) ;

void main(void){
    typEleve e1 ;
    saisieEleve(&e1) ;
    afficheEleve(e1) ;
}
void saisieEleve (typEleve * e){
    printf("nom :");
    scanf("%s", &e->nom);
    printf("goupe :");
    fflush(stdin);
    scanf("%c", &e->groupe);
    printf("note :");
    scanf("%d", &e->note);
}
void afficheEleve (typEleve  e){
    printf("<%s> ", e.nom);
    printf("<%c> ", e.groupe);
    printf("<%d>\n", e.note);
}
```

13.3 opérateur ->

L'opérateur -> permet d'accéder aux champs d'une structure à partir de son adresse.

On a l'équivalence suivante :

```
(*pt).champ <-> pt->champ
```

L'opérateur "." est prioritaire sur l'opérateur "*" : il faut donc mettre des parenthèses.

*pte donne le contenu de la variable pointée par pte, c'est-à-dire la structure. De là, on accède aux champs avec le point.

Si on écrivait *pte.nom : ça ne voudrait rien dire car pte est un pointeur d'eleve.

Si on écrivait *e1.nom : c'est le contenu de la variable pointé par nom : c'est donc la première lettre du nom.

13.4 Typedef et Pointeurs

```
typedef int * PtInt;
PtInt pt1, pt2;
```

13.5 Usages

Pour le moment, on n'utilisera peu les structures et pointeurs de structures. Par contre leur usage sera généralisée à l'occasion du cours de structures de données (liste, pile, file, arbre, etc.).