Chapitre 6

Les Tableaux

6.1 Les tableaux à une dimension

Un tableau est un objet structuré. Il contient un certain nombre d'éléments de type identique.

Déclaration :

$<$ **type** $>$ $<$ **identificateur** $>$ [$<$ **expression − constante** $>$];

$<$ **type** $>$ $<$ **identificateur** $>$ [$<$ **expression − constante − opt** $>$] =

{$<$ **expression$_1$** $>$, ..., $<$ **expression$_n$** $>$};

A la déclaration d'un tableau, le programme procède à l'allocation d'un espace mémoire suffisant pour contenir le tableau. Les éléments du tableau sont donc dans des cases mémoires contiguës.

Le premier élément est à la position d'indice 0. Pour un tableau *tab* déclaré de taille N, les éléments ont pour indices : 0, 1,..., $N-1$. La syntaxe pour accéder à ces éléments est $tab[i]$, $i = 0, 1, ..., N-1$. Chaque élément d'un tableau a un comportement similaire à celui d'une variable. Ce qui n'est pas le cas du tableau dans sa globalité. En effet, il n'est pas possible d'affecter un tableau à un autre tableau. Pour cela, il faut procéder à l'affectation élément par élément.

Exemple : Écrire les fonctions permettant la saisie dans un tableau et l'affichage de notes.

Algorithme 16 : Programme C : Tableaux et fonctions

```
1  #include < stdio.h >
2  int saisie_notes(int tab[100])
3  {
4  int note, i = 0;
5  do
6  {
7  printf("Donner la note suivante ou taper -1 si terminé : ");
8  scanf("%d", &note);
9  if(note >= 0)
10 {
11 tab[i] = note;
12 i++;
13 }
14 }
15 while(note!=-1);
16 return i;
17 }
```

Algorithme 17 : Programme C : Tableaux et fonctions

```
1 #include < stdio.h >
2 void affichage_notes(int tab[100], int n)
3 {
4 int i ;
5 for(i=0 ; i<n ; i++) printf("%d \n", tab[i]) ;
6 }
```

6.2 Les tableaux multidimensionnels

Il n'existe pas en C un type dédié pour les tableaux à plusieurs dimensions. Cependant, les éléments d'un tableau unidimensionnel peuvent être à leur tour des tableaux. Cela rend possible la définition de tableaux multidimensionnels.

Déclaration :
< **type** >< **identificateur** > [< $\mathbf{expression_1}$ − **constante** >]...[< $\mathbf{expression_p}$ − **constante** >];

Exemple 6.2.1 *int matrice[10][10] ;*
matrice[i] est un tableau de 10 entiers et matrice[i][j] est un élément (entier) du tableau matrice[i] (si $0 <= i, j < 10$*).*

Chapitre 7

Les Pointeurs

7.1 Définition

Un pointeur est une variable dont la valeur est l'adresse d'une cellule de la mémoire. Les pointeurs sont utilisés énormément en C. En effet, ils permettent de construire des programmes où le nombre et la taille des informations à gérer ne sont pas fixés à la compilation mais ajustés dynamiquement en cours d'exécution.

Cependant, il faut bien noter qu'un pointeur demeure une variable comme une autre et qu'une adresse n'est qu'un nombre entier qui donne l'indice d'un élément du tableau qu'est la mémoire de l'ordinateur. De ce fait, un pointeur doit pouvoir supporter toutes les opérations réalisables sur un nombre si tant est qu'elles aient un sens quand ce nombre exprime un rang. L'apport d'un pointeur réside dans la possibilité de manipuler la variable dont la valeur du pointeur est l'adresse.

7.2 Déclaration et Initialisation

Déclaration : $<$ **type** $>*<$ **identificateur** $><=$ **initialisation** $-$ **opt** $>$;
Cette déclaration entraîne la création d'une variable pointeur qui ne peut contenir qu'une adresse contenant ou susceptible de contenir une donnée dont le type est $<$ **type** $>$. Cette variable peut être initialisée à la déclaration : cela est optionnel.

Si le pointeur est typé c'est-à-dire qu'il connaît le type de l'objet pointé, il permet la manipulation de ce dernier. Sinon, il est atypique (il contient une simple adresse) et permet de repérer en mémoire n'importe quel objet mais pas de le manipuler.

Déclaration pointeur atypique : void$*<$ **identificateur** $><=$ **initialisation** $-$ **opt** $>$;

Algorithme 18 : Programme C : Déclaration de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt = NULL ;
6 int ** pt_pt = NULL ;
7 int **** pt4 ;
8 }
```

Par convention, on attribue la valeur **NULL** à un pointeur qui ne pointe sur aucun objet. A la déclaration d'une variable pointeur, il est possible qu'elle contienne une valeur différente de **NULL**. Cette valeur sera interprétée comme une adresse si cette variable est utilisée sans initialisation ou modification, et cela peut détruire des données et causer des incohérences dans le programme.

La taille d'une variable pointeur ne dépend pas de l'objet pointé, mais de la machine utilisée : c'est le nombre d'octets nécessaire pour coder une adresse sur cette machine.

7.3 Les Opérateurs & et *

L'opérateur **&** permet de connaître l'adresse d'une variable. Notons qu'appliquer cet opérateur à une constante n'a pas de sens.

Syntaxe : & < **variable** >

Algorithme 19 : Programme C : Initialisation de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt_variable = &variable ; /* Commentaire : pt_variable pointe vers variable */
6 int ** pt_pt = &pt_variable ; /* Commentaire : pt_pt pointe vers pt_variable */
7 }
```

L'opérateur d'indirection ***** est l'inverse de **&** : il signifie "objet pointé par". Il permet de manipuler l'objet repéré par un pointeur typé.

Syntaxe : ∗ < **expression** >
où < **expression** > est de type pointeur typé.

Algorithme 20 : Programme C : Initialisation de Pointeurs

```
1 #include < stdio.h >
2 int main( )
3 {
4 int variable = 1 ;
5 int * pt_variable = &variable ;
6 /* Commentaire : pt_variable est un pointeur, ∗pt_variable est l'objet pointé et donc
  ∗pt_variable vaut 1 */
7 }
```

7.4 Arithmétique des adresses

Certaines opérations définies sur les nombres gardent un sens lorsque les opérandes sont des adresses. Si **exp** est l'adresse d'un objet $\mathbf{O_1}$, $\mathbf{exp + 1}$ exprime l'adresse de l'objet de même type que $\mathbf{O_1}$ et qui se trouverait dans la mémoire immédiatement après $\mathbf{O_1}$ et $\mathbf{exp - 1}$ l'adresse de celui qui se trouverait juste avant.

Ainsi $\mathbf{exp} + \mathbf{n}$ correspond à $(\mathbf{Type}*)((\mathbf{unsignedlong})\mathbf{exp} + \mathbf{n} * \mathbf{sizeof}(\mathbf{Type}))$. De même $\mathbf{exp_2} - \mathbf{exp_1}$ correspond à $((\mathbf{unsignedlong})\mathbf{exp_2} - (\mathbf{unsignedlong})\mathbf{exp_1})/\mathbf{sizeof}(\mathbf{Type})$ dans le cas où $\mathbf{exp_2}$ et $\mathbf{exp_1}$ sont de types tout à fait identiques.

L'indexation permet également d'avoir une notation différente pour certaines de ces opérations. En effet, $*(\mathbf{exp} + \mathbf{1})$ et $\mathbf{exp}[\mathbf{1}]$ sont équivalentes, de même que $*(\mathbf{exp} + \mathbf{n})$ et $\mathbf{exp}[\mathbf{n}]$.

7.5 Passage de paramètres et Pointeurs

Le passage de paramètres en C se fait uniquement par valeur. Cependant, il est parfois nécessaire (voire souvent) qu'une fonction modifie la valeur d'une variable définie dans le programme appelant, celle-ci étant passée en argument. Prenons l'exemple de la fonction **echange** qui doit échanger la valeur de deux variables passées en argument. Nous savons que donner directement en paramètre ces deux variables à notre fonction ne permet pas de résoudre le problème étant donné que le passage de paramètres se fait par valeurs (la fonction travaille sur des copies). La solution consiste à donner comme arguments à **echange** des pointeurs sur les variables en question.

Algorithme 21 : Programme C : Échange des valeurs de deux variables

```
1  #include < stdio.h >
2  void echange (int * p, int * q)
3  {
4  int x = *p;
5  *p = *q;
6  *q = x;
7  }
8  int main( )
9  {
10 int a = 2, b = 3;
11 echange(&a, &b);
12 }
```

7.6 Tableaux et Pointeurs

L'identificateur d'un tableau en C est un pointeur vers le premier élément du tableau.

Considérons la déclaration **int tab**[**10**];, **tab** est équivalent à &**tab**[**0**]. Cela explique pourquoi le passage d'un tableau en paramètre d'une fonction en permet la modification. En outre, on peut en déduire l'équivalence de $*\mathbf{tab}$ et $\mathbf{tab}[\mathbf{0}]$, de même que celle de $*(\mathbf{tab} + \mathbf{1})$ et $\mathbf{tab}[\mathbf{1}]$, $*(\mathbf{tab} + \mathbf{2})$ et $\mathbf{tab}[\mathbf{2}]$... Ce constat est valable aussi pour une variable pointeur quelconque : il est possible d'utiliser l'opérateur d'indexation.

On observe qu'au final un tableau et un pointeur sont des concepts très voisins. Cependant, une différence notable réside dans le fait que l'identificateur d'un tableau est une constante contrairement à une variable de type pointeur. Il est donc interdit de modifier l'adresse contenue dans l'identificateur d'un tableau.

7.6.1 Tableaux multidimensionnels et Pointeurs

L'identificateur d'un tableau multidimensionnel demeure un pointeur vers le premier élément du tableau. Si, on prend l'exemple d'un tableau à deux dimensions **int matrice**[**10**][**10**], les éléments sont rangés de manière linéaire dans des cases contigües de la mémoire (les éléments de

la première ligne étant suivis par ceux de la deuxième...). **matrice** est un pointeur de type **int**$*$ qui contient l'adresse du premier élément du tableau. Ainsi, l'utilisation de l'indirection pour répérer les éléments de notre tableau est beaucoup plus complexe que dans le cas des tableaux unidimensionnels. En effet, l'expression **matrice**[**i**][**j**] est équivalente à
$*(\mathbf{matrice} + (\mathbf{i} * \mathbf{10} + \mathbf{j}) * \mathbf{sizeof}(\mathbf{int}))$.

7.7 Conversion de type : opérateur de "cast"

La définition d'un pointeur générique permet d'avoir une variable compatible avec tout type d'adresse. Par contre l'utilisation et la modification de l'objet pointé nécessitent de savoir son type. De ce fait, il peut être nécessaire de repasser à un pointeur typé pour intervenir sur l'objet. La conversion de type (cast) permet de modifier le type d'un objet : il suffit de rajouter le nouveau type entre parenthèses juste devant l'expression.

Syntaxe : (**nouveau_type**)**expression**
Nous listons ci-dessous les conversions dites légitimes :
- entier vers entier plus long : le codage de **expression** est étendu de sorte que sa valeur soit inchangée.
- entier vers entier plus court : si la valeur de **expression** est assez petite pour tenir dans le nouveau type, la valeur est inchangée. Sinon, elle est tronquée (sans intérêt).
- entier signé vers non signé et vice versa : l'interprétation du compilateur change alors que le codage reste inchangé.
- flottant vers entier : la partie fractionnaire est supprimée.
- entier vers flottant : le flottant obtenu est celui qui approche le mieux l'entier sauf en cas de débordement (imprévisible).
- adresse d'un objet de type $\mathbf{Type_1}$ vers adresse d'un objet de type $\mathbf{Type_2}$: l'interprétation change alors que le codage demeure inchangé.
- entier vers adresse d'un objet de type **Type** : l'interprétation change alors que le codage reste inchangé.

De nombreux dangers sont présents dans ces conversions de type. L'opération de "cast" est rarement nécessaire car l'objectif principal est de faire taire le compilateur.

Chapitre 8

Chaînes de caractères

Une chaîne de caractères est un tableau de caractères qui se termine par le caractère **NULL** ('\0').
Une constante chaîne de caractères est constituée d'une suite de caractères délimitée par **" "**. Le compilateur ajoute automatiquement le caractère **NULL** et range la suite dans la zone des données statiques et la considère comme un tableau de caractères sans nom. Dans un programme, les chaînes identiques ne sont pas nécessairement stockées à des endroits distincts ; aussi, il est fortement déconseillé de modifier le contenu d'une chaîne constante.

8.1 Les variables chaînes de caractères

On peut manipuler les chaînes de caractères à l'aide de tableaux ou de pointeurs.

Algorithme 22 : Programme C : Déclaration et Initialisation de chaînes de caractères

```
1  #include < stdio.h >
2  int main( )
3  {
4  char * message ="coucou le monde" ;
5  /* Commentaire : "coucou le monde" est une constante dont l'adresse du premier
   caractère est affectée à la variable message */
6  char tab[ ] ="coucou le monde" ;
7  /* Commentaire : l'identificateur tab est une constante mais les éléments du tableau sont
   modifiables */
8  char tab2[ ] = {'c','o','u','c','o','u',' ','l','e',' ','m','o','n','d','e','\0'} ;
9  /* Commentaire : les déclarations et initialisations des deux tableaux sont équivalentes */
10 }
```

8.2 Manipulation

La fonction **scanf** permet la saisie de chaînes de caractères grâce au spécificateur de type **%s**.

La fonction **scanf** ne permet pas de gérer les espaces dans les chaînes de caractères : ils sont traités comme des symboles de fin de chaînes.

La bibliothèque **string.h** contient des fonctions permettant une manipulation simple des chaînes de caractères.

Algorithme 23 : Programme C : Saisie de chaînes de caractères

```
1 #include < stdio.h >
2 int main( )
3 {
4 char chaine[30] ;
5 scanf("%s", chaine) ;
6 }
```

Ainsi, pour la saisie d'une chaîne, elle contient les fonctions **gets** et **fgets**. Malheureusement, **gets** contient des bugs, donc il est préférable d'utiliser **fgets**.

Syntaxe : fgets(machaine, expression − entiere, stdin)

où **stdin** est l'entrée standard (clavier) et **expression − entiere** le nombre maximum de caractères à lire.

La bibliothèque contient également d'autres fonctions telles que : **strlen** (calcule la longueur d'une chaîne), **strcpy** (copie une chaîne dans une autre), **strcat** (concatène deux chaînes), **strcmp** (compare deux chaînes), **strchr** (recherche d'un caractère dans une chaîne)....

Chapitre 9

Gestion dynamique de la mémoire

L'attribution d'une adresse valide à un pointeur peut être faite de manière simple en lui affectant l'adresse d'une variable existante (déclarée au préalable). Une autre manière de procéder consiste à demander au système l'allocation d'un nouvel espace mémoire, et cela grâce à des fonctions de la bibliothèque **stdlib.h**. Cette approche permet une gestion dynamique de la mémoire en réservant des zones, non plus durant toute l'exécution du programme, mais seulement pendant le laps de temps où cela est nécessaire.

9.1 Principales fonctions de gestion de la mémoire

Voici les prototypes de ces fonctions (**size_t** est un entier non signé déclaré dans **stdlib.h**) :
- **void * calloc (size_t nb, size_t taille) ;**
- **void free (void * pointeur) ;**
- **void * malloc (size_t nb_octets) ;**
- **void * realloc (void * pointeur, size_t nb_octets) ;**

9.1.1 Allocation

- **malloc (nb_octets) ;**
  La fonction **malloc** fournit un pointeur de type **void*** sur une zone de la mémoire dont la taille est **nb_octets** ou **NULL** en cas d'échec.

Algorithme 24 : Programme C : malloc

```
1 #include < stdio.h >
2 #include < stdlib.h >
3 int main( )
4 {
5 int * pt = NULL;
6 pt = malloc(sizeof(int)) ;
7 if(pt == NULL) printf("Echec allocation dans malloc \n") ;
8 }
```

- **calloc(nb, taille) ;**

La fonction **calloc** fournit un pointeur de type **void *** sur une zone de la mémoire permettant de ranger **nb** objets de grandeur **taille** ou **NULL** en cas d'échec.

Algorithme 25 : Programme C : calloc

```
1 #include < stdio.h >
2 #include < stdlib.h >
3 int main( )
4 {
5 int * pt = NULL;
6 pt = calloc(5, sizeof(int)) ;
7 if(pt == NULL) printf("Echec allocation dans calloc \n") ;
8 }
```

– **realloc(pointeur, nb_octets) ;**
La fonction **realloc** permet d'agrandir une zone précédemment allouée et repérée par le pointeur **pointeur** à la taille **nb_octets**, sans perte d'informations. En cas d'échec, **realloc** retourne le pointeur **NULL** et le bloc pointé par **pointeur** peut être détruit.

Algorithme 26 : Programme C : realloc

```
1 #include < stdio.h >
2 #include < stdlib.h >
3 int main( )
4 {
5 int * pt1 = NULL, * pt2 = NULL ;
6 pt1 = calloc(5, sizeof(int)) ;
7 if(pt1 == NULL) printf("Echec allocation dans calloc \n") ;
8 else
9   {
10   pt2 = realloc(pt1, 10∗sizeof(int)) ;
11   if(pt2 == NULL) printf("Echec allocation dans realloc \n") ;
12  }
13 }
```

9.1.2 Libération

– **free(pointeur) ;**
La fonction **free** libère l'emplacement mémoire précédemment alloué par l'une des fonctions précédentes.

9.1.3 Les tableaux dynamiques

L'utilisation de tableaux dynamiques permet de ne pas encombrer la mémoire avec des cases inutiles dans le cas où, par exemple, c'est à l'utilisateur de fixer le nombre d'éléments. En effet, à la déclaration d'un tableau fixe, la taille est forcément fournie par une expression constante.

On peut procéder de la même manière pour définir des tableaux multidimensionnels dynamiques.

Algorithme 27 : Programme C : Tableaux dynamiques

```
#include < stdio.h >
#include < stdlib.h >
int main( )
{
int n, *tab;
printf("Donner la taille du tableau \n") ;
scanf("%d", &n) ;
tab =calloc(n,sizeof(int)) ;
if(tab == NULL) printf("Echec allocation dans calloc \n") ;
else
  {
   initialiser(tab,n) ;
   afficher(tab,n) ;
   free(tab) ;
  }
}
```

Algorithme 28 : Programme C : Tableaux multidimensionnels dynamiques

```
#include < stdio.h >
#include < stdlib.h >
int main( )
{
int n, m, pb = 0, *matrice;
printf("Donner le nombre de lignes : ") ;
scanf("%d", &n) ;
printf("Donner le nombre de colonnes : ") ;
scanf("%d", &m) ;
matrice =calloc(n,sizeof(int*)) ;
if(matrice == NULL) printf("Echec allocation dans premier calloc \n") ;
else
  {
   for(i = 0; i < n; i + +)
     {
      matrice[i] =calloc(m, sizeof(int)) ;
      if(matrice[i] == NULL) pb = 1 ;
     }
   if(pb == 1) printf("Echec allocation dans deuxieme calloc \n") ;
   else
     {
      initialiser2(matrice, n, m) ;
      afficher2(matrice, n, m) ;
      for(i = 0; i < n; i + +) free(matrice[i]) ;
     }
   free(matrice) ;
  }
}
```

9.2 Organisation du programme en mémoire

En mémoire, le stockage des informations associées à un programme est fait dans l'ordre suivant :

- Le code du programme (les instructions)
- La zone des données statiques (variables globales, les variables locales statiques, les constantes)
- Le tas qui contient les données gérées de manière dynamique (allocation/libération de mémoire)
- La pile qui contient les variables automatiques (variables locales et paramètres)

Chapitre 10

Les structures

Les structures sont des variables composées de champs de types différents, chaque champ étant repéré par un nom.

Syntaxe :
< **struct** >< **identificateur** − **opt** >
{
< **declaration** >;
...
< **declaration** >;
} < **identificateur** − **opt** ><= {**initialisation**, ..., **initialisation**} − **opt** >, ...,
< **identificateur** − **opt** ><= {**initialisation**, ..., **initialisation**} − **opt** >;

Ceci déclare une structure dont les différents champs sont déclarés entre accolades, mais aussi des variables qui sont des structures de ce type. L'utilisation de l'identificateur permet par la suite de déclarer d'autres variables de ce type en faisant précéder uniquement struct identificateur comme indicateur de type.

L'accès à un champ d'une structure est effectué grâce à l'opérateur "**.**". Cependant si la variable est un pointeur vers une structure, on accède à un champ par l'opérateur "− >".

Algorithme 29 : Programme C : Déclaration de Structure

```
1  #include < stdio.h >
2  struct etudiant
3  {
4  char nom[30], prenom[30] ;
5  int groupe ;
6  float moyenne ;
7  } ;
8  int main( )
9  {
10 struct etudiant x = {"ndiaye", "samba", 1} ;
11 /* Le champ moyenne va être initialisé à 0 */
12 struct etudiant *y ;
13 x.moyenne = 10. ;
14 y− > moyenne = 10. ;
15 }
```

Manipulations des structures :

- On peut affecter une expression d'un type structure à une variable de type structure pourvu que ces deux types soient identiques.
- Les structures sont passées par valeur lors de l'appel des fonctions.
- Le résultat d'une fonction peut être une structure.

10.1 Typedef : définition de nouveaux types

Cet opérateur permet de définir de nouveaux types. Cela permet d'alléger par la suite les expressions.

Syntaxe : **typedef** < **declaration** >

l'identificateur de la déclaration est le nom du nouveau type.

Algorithme 30 : Programme C : Définition de nouveaux types

```
1  #include < stdio.h >
2  typedef struct etudiant
3  {
4  char nom[30], prenom[30] ;
5  int groupe ;
6  float moyenne ;
7  } ETUDIANT ;
8  int main( )
9  {
10 ETUDIANT x = {"ndiaye", "samba", 1} ;
11 /* Le champ moyenne va être initialisé à 0 */
12 ETUDIANT *y;
13 x.moyenne = 10. ;
14 y− > moyenne = 15.5 ;
15 }
```

Chapitre 11

Quelques éléments supplémentaires

11.1 Variables, fonctions et compilation séparée

11.1.1 Identificateurs publics et privés

Cette partie donne les règles qui régissent la visibilité inter-fichiers des identificateurs. La question concerne uniquement les noms des variables et des fonctions. Le problème ne se pose pas pour les variables locales dont la visibilité est restreinte à l'étendue du bloc ou de la fonction contenant leur définition. Il sera donc uniquement question des noms des variables globales et des fonctions.

Un nom de variable ou de fonction définit dans un fichier source et pouvant être utilisé dans d'autres fichiers sources est dit public. Un identificateur qui n'est pas public est dit privé.

- sauf indication contraire, tout identificateur global est public ;
- le qualifieur **static**, précédent la déclaration d'un identificateur global, rend celui-ci privé

11.1.2 Déclaration d'objets externes

Nous ne considérons désormais que les noms publics. Un identificateur référencé dans un fichier alors qu'il est défini dans un autre fichier est appelé externe. En général, les noms externes doivent faire l'objet d'une déclaration car le compilateur ne traite qu'un fichier à la fois.

- Toute variable doit être définie (déclaration normale) ou déclarée externe avant son utilisation
- Une fonction peut être référence sans aucune définition ou déclaration externe au préalable ; elle est alors supposée externe, de type int et sans prototype. Il est donc préférable de procéder à sa définition ou déclaration avant toute utilisation.

Syntaxe declaration externe variable : **extern** < **declaration** >

Syntaxe declaration externe fonction : < **extern** − **opt** >< **prototype** >

11.1.3 Variables locales statiques

Le qualifieur **static**, placé devant la déclaration d'une variable locale, produit une variable locale pour sa visibilité et statique pour sa durée de vie (permanente).

11.1.4 Variables critiques

Le qualifieur **register** précédant une déclaration de variable informe le compilateur que la variable en question est très fréquemment accédée pendant l'exécution du programme et qu'il y a donc lieu de prendre toutes les dispositions pour en accélérer l'accès.

Les variables ainsi déclarées doivent être locales et d'un type simple (nombre, pointeur).

11.1.5 Variables constantes et volatiles

Le qualifieur **const** placé devant une variable ou un argument formel informe le compilateur que la variable ou l'argument ne changera pas de valeur tout au long de l'exécution du programme ou de l'activation de la fonction. Ce renseignement permet au compilateur d'en optimiser la gestion.

Le qualifieur **volatile** informe le compilateur que la valeur de la variable en question peut changer mystérieusement, y compris dans une section du programme qui ne comporte aucune référence à cette variable.