Faculté des Sciences

Département d'Informatique

Informatique appliquée

Filière : IA - Semestre (3)

Strctures de données

Année Universitaire : 2024/2025

Partie I : Rappels et compléments du langage C

1 Les types composés

1.1 Introduction

A partir des types prédéfinis du langage **C** (caractères, entiers, flottants), on peut créer de nouveaux types, appelés types composés (tableau, structure, union, énumération, ...). Ils permettent de représenter des ensembles de données organisées.

1.2 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration en **C** d'un tableau à une dimension se fait de la façon suivante :

```
Type nomTableau[nombreElements];
```

Exemple :

```
float tabR[5];              /* tableau de 5 flottants (réels). */
int tabE[8];                /* tableau de 8 entiers.   */
```

Pour plus de clarté, il est recommandé de donner un nom à la constante nombre-éléments par une directive au préprocesseur, par exemple :

```
#define nombreElements 10
```

On accède à un élément du tableau en lui appliquant l'opérateur **[]**.

Les éléments d'un tableau sont toujours numérotés de **0** à **nombreElements -1**.

Exemple :

```
#define N 10
main(){
  int tab[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
}
```

Un **tableau** correspond en fait à un **pointeur constant** (*voir les pointeurs*) vers le premier élément du tableau et aucune opération globale n'est autorisée sur un tableau. Par exemple, on ne peut pas écrire :

```
tab1 = tab2;
```

Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main(){
  int tab1[N], tab2[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste d'éléments :

Exemple :

```
#define N 4
int tab[N] = {11, 22, 33, 44};
main(){
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
}
```

Le programme affichera :

```
tab[0] = 11
tab[1] = 22
tab[2] = 33
tab[3] = 44
```

Cas d'un tableau de caractères : Un tableau de caractères est en fait une chaîne de caractères. Son initialisation peut se faire de plusieurs façons :

```
char p1[10] = 'B','o','n','j','o','u','r';
char p2[10] = "Bonjour";            /* init. par une chaîne litterale */
char p3[] = "Bonjour";              /* p3 aura alors 8 éléments       */
```

ATTENTION ! Le compilateur rajoute toujours un caractère '**\0**' à la fin d'une chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus.

1.3 Les tableau à n-dimensions

De manière similaire, on peut déclarer un tableau à plusieurs dimensions.

Par exemple, pour un tableau à deux dimensions :

```
Type nomTableau [nombreLignes][nombreColonnes];
```

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau.

On accède à un élément du tableau par l'expression "**nomTableau[i][j]**".

Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main(){
  int i, j;
  for (i = 0 ; i < M; i++){
      for (j = 0; j < N; j++)
        printf("%d\t", tab[i][j]);
    printf("\n");
  }
}
```

Le programme affichera :

```
1       2       3
4       5       6
```

1.4 Les types énumérés

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef **enum** et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

```
enum modele {constante1, constante2, ..., constanten};
```

En réalité, les objets de type **enum** sont représentés comme des **int**.

Les valeurs possibles **constante1**, **constante2**, **...**, **constanten** sont codées par des entiers de **0** à **n-1**.

Par exemple, le type **enum booleen** défini dans le programme suivant associe l'entier **0** à la valeur **faux** et l'entier **1** à la valeur **vrai**.

```
enum booleen {faux, vrai};
main(){
  enum booleen b;
  b = vrai;
  printf("b = %d", b);
}
```

Le programme affichera :

```
b = 1
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

```
enum booleen {faux = 12, vrai = 23};
```

Exemple :

```
/* type énuméré couleurs */
enum couleur {
    noir,                           /* 0 = absence de couleur */
    rouge = 1, vert = 2, bleu = 4,  /* couleurs fondamentales */
    jaune = rouge + vert,           /* valeur 3*/
    cyan = vert + bleu,             /* valeur 6 */
    magenta = rouge + bleu,         /* valeur 5 */
    blanc = rouge + vert + bleu     /* valeur 7 */
};
main (){
    enum couleur col;
    col = noir;
    printf ("%d ", col);
    switch(col){
        case 0: printf("noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default: printf("Ce n'est pas une couleur\n");
    }
    col += rouge;
    printf ("%d ", col);
    switch(col){
        case 0: printf( "noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default : printf("Ce n'est pas une couleur\n");
    }
    col += cyan;
    printf ("%d ", col);
    switch(col){
        case 0 : printf( "noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default : printf("Ce n'est pas une couleur\n");
    }
}
```

Le programme affichera :

```
0 noir
1 rouge
7 blanc
```

1.5 Les structures

Une structure est un agrégat de plusieurs objets de types différents regroupés dans une même variable. Chacune des données composant une structure est appelée un champ et peuvent être de types quelconques (type simple, tableaux, autres structures, …). Chacun des champs possède un identificateur qui permet d'accéder directement à l'information qu'il contient.

Exemple :

Pour un étudiant, on peut regrouper dans une seule variable :

- son nom (chaîne de caractères),
- son prénom (chaîne de caractères),
- son âge (entier),
- son sexe (masculin ou féminin),
- son numéro CNE (tableau de 10 chiffres), ...

La déclaration d'un modèle de structure dont l'identificateur est **`modele`** est la suivante :

```
struct modele{
  type1 membre1;
  type2 membre2;
    ...
  typen membren;
};
```

Pour déclarer une variable de type structure correspondante au modèle précédent, on utilise la syntaxe :

```
struct modele objet;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele{
  type1 membre1;
  type2 membre2;
    ...
  typen membren;
}objet;
```

Exemples :

- Une structure de nom **`Etudiant`** correspondant à un étudiant :

```
enum sexes {Feminin, Masculin};

struct Etudiant {
    char nom[20], prenom[20];
    int age;
    enum sexes sexe;
    int numero[5];
} element = {"Aissaoui", "Ali", 22, masculin, {02,49,11,39,42}};
```

- Une structure **`Point`** correspondant à un point de coordonnées **`x`**, **`y`** dans un plan :

```
struct Point {
      int x;
      int y;
};
```

- Une structure de nom **`Adresse`** possible pour coder une adresse postale :

```
struct Adresse {
      int num;
      char rue[40];
      long code;
      char ville[20], pays[20];
};
```

- Une structure de nom **`Individu`** dont l'un des champs a la structure **`Adresse`** précédente (dont la définition est supposée connue) et contenant la définition d'une structure interne anonyme pour le champ de nom **`Identite`** :

```
struct Individu{
       struct {
         char nom[20];
         char prenom[20];
       } Identite;
       int age;
       struct Adresse domicile;
};
```

Il est évidemment possible de créer des alias sur une structure en utilisant **`typedef`** comme dans :

```
typedef struct {
          int a, b;                 // a et b dans la même déclaration
} couple;
```

On accède aux différents membres (champs) d'une structure grâce à l'opérateur membre de structure, noté "**.**".

```
nomObjet.nomChamp
```

Remarque :

- L'affectation globale au moyen de l'opérateur **=** entre deux objets de même structure est maintenant autorisée par la plupart des compilateurs.
- La comparaison (**==**) n'est par contre pas admise généralement. Il est nécessaire de tester chacun des champs l'un après l'autre.

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct Complexe z = {2. , 2.};
```

En **C**, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux).

Dans le contexte précédent, on peut écrire :

```
struct Complexe z1, z2;
...
z2 = z1;
```

Exemple 1 :

```
#include <math.h>
struct Complexe{
    double reelle;
    double imaginaire;
};
main(){
    struct Complexe z = {3.5, 2.9};
    double norme;
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);

    printf("La norme de (%.2f + i %.2f) = %.2f\n",
                                        z.reelle, z.imaginaire, norme);
}
```

Le programme affichera :

```
La norme de (3.50 + i 2.90) = 4.55
```

Exemple 2 :

```
enum propulsion {pedales, moteur, reacteur};
struct vehicule {
    char nom[20];
    int longueur, poids;
    enum propulsion mode;
} velo = {"Euroteam", 2, 5, pedales};
```

```
main () {
    struct vehicule voiture ={"Toyota", 5, 1500, moteur };
    struct vehicule avion;
    printf ("Nom: %s\n", velo.nom);
    printf ("Longueur: %d, poids: %d\n", velo.longueur, velo.poids);
    printf("Mode de propulsion: %d\n ", velo.mode);
    strcpy(avion.nom, "Jumbo");
    avion.longueur = 60;
    avion.poids = 450000;
    avion.mode = reacteur;
    printf ("Nom: %s\n", avion.nom);
    printf ("Longueur: %d, poids: %d\n", avion.longueur, avion.poids);
    printf("Mode de propulsion:%d\n ", avion.mode);
}
```

Le programme affichera :

```
Nom: Euroteam
Longueur: 2, poids: 5
Mode de propulsion: 0
Nom: Jumbo
Longueur: 60, poids: 450000
Mode de propulsion:2
```

1.6 Les champs de bits

Il est possible en **C** de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (**`int`** ou **`unsigned int`**). Cela se fait en précisant le nombre de bits du champ avant le "**`;`**" qui suit sa déclaration.

Par exemple, la structure suivante possède deux membres, **`actif`** qui est codé sur un seul bit, et **`valeur`** qui est codé sur 31 bits:

```
struct Registre{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

Tout objet de type **`struct Registre`** est donc codé sur 32 bits.

1.7 Les unions

Une **`union`** désigne un ensemble de variables de types différents susceptibles d'occuper **alternativement** une même zone mémoire. Si les membres d'une **`union`** sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type **`union`** sont les mêmes que celles sur les objets de type **`struct`**.

Dans l'exemple suivant, la variable **`hier`** de type **`union Jour`** peut être soit un entier, soit un caractère.

```c
union Jour{
    char lettre;
    int numero;
};

main(){
    union Jour hier, demain;
    hier.lettre = 'J';
    printf("Hier = %c\n", hier.lettre);
    hier.numero = 4;
    demain.numero = (hier.numero + 2) % 7;
    printf("Demain = %d\n", demain.numero);
}
```

Le programme affichera :

```
Hier = J
Demain = 6
```

1.8 Définition de types composés avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **`typedef`** :

Exemple 1 :

```c
struct Eleve{
    char nom[20];
    char prenom[20];
    float note;
};
typedef struct Eleve Eleve;
 main(){
  Eleve a = {"Alaoui", "Hamza", 17};
  Eleve b;
  strcpy(b.nom, "Hatimi");
  strcpy(b.prenom, "Bouchra");
  b.note = 13;
  printf("Eleve 1 : %s %s %.2f\n", a.nom, a.prenom, a.note);
  printf("Eleve 2 : %s %s %.2f\n", b.nom, b.prenom, b.note);
}
```

Le programme affichera :

```
Eleve 1 : Alaoui Hamza 17.00
Eleve 2 : Hatimi Bouchra 13.00
```

Exemple 2 :

```c
struct Eleve{
    char nom[20];
    char prenom[20];
    float note;
};
typedef struct Eleve Eleve;
```

```c
typedef Eleve TabEleve[100];
main(){
    TabEleve T;
    int i;
    printf("------ Saisie ------\n");
    for(i=0;i<3;i++){
        printf("Eleve %d : ", i+1);
        scanf("%s %s %f", T[i].nom, T[i].prenom, &T[i].note);
    }
    printf("------ Affichage ------\n");
    for(i=0;i<3;i++)
        printf("Eleve %d : %s %s %.2f\n",
                                    i+1, T[i].nom, T[i].prenom, T[i].note);
}
```

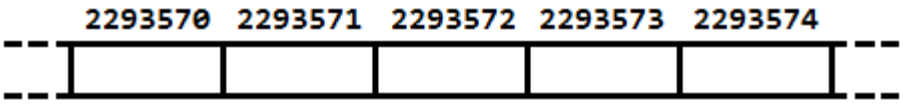
Le programme affichera :

```
------ Saisie ------
Eleve 1 : Hachimi Souad 16
Eleve 2 : Alami Brahim 13
Eleve 3 : Hamdi Merieme 14
------ Affichage ------
Eleve 1 : Hachimi Souad 16.00
Eleve 2 : Alami Brahim 13.00
Eleve 3 : Hamdi Merieme 14.00
```

2 Les pointeurs

2.1 Introduction

Les variables utilisées dans un programme sont stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets adjacents identifiés par un numéro unique appelé **adresse**.



Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (s'il s'agit d'une variable qui recouvre plusieurs octets adjacents, l'adresse du premier de ces octets).

Rappel : En **C**, l'opérateur d'adresse **&** appliqué à une variable retourne l'adresse-mémoire de cette variable : **`&variable.`**

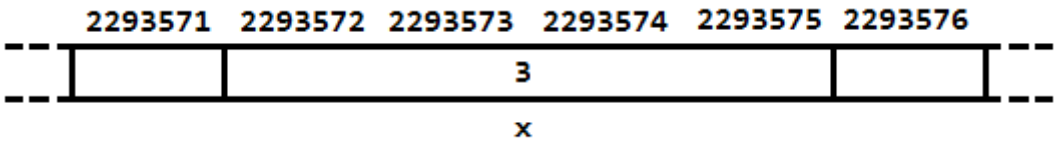
Quand on écrit :

```
int x = 3;
printf("x = %d et se trouve dans l'adresse : %d\n", x, &x);
```

Cela affichera :

```
x = 3 et se trouve dans l'adresse : 2293572
```

Ce fragment de code réserve un emplacement de **4 octets** pour la variable **x** dans la mémoire (les entiers sont codés dans ce cas sur 4 octets), à partir de la case numéro **2293572** dans le cas du schéma et l'initialise avec la valeur **3**.



Rappel : En **C**, l'opérateur fonctionnel **`sizeof(Type)`** retourne le nombre d'octets occupé par le type **`Type.`**

2.2 Adresse et valeur d'un objet

On appelle ***Lvalue*** (*left value*) tout objet pouvant être placé à gauche de l'opérateur d'affectation (**=**).

Une ***Lvalue*** est caractérisée par :

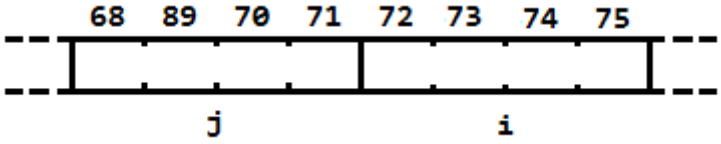
- **son adresse :** l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- **sa valeur :** l'objet stocké à cette adresse.

Dans l'exemple :

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable **i** à l'adresse **2293572** en mémoire, et la variable **j** à l'adresse **2293568**, on a :

objet	adresse	valeur
i	2293572	3
j	2293568	3



2.3 Notion de pointeur

Un pointeur est un objet (***Lvalue***) dont **la valeur** est égale à **l'adresse** d'un autre objet.

Un pointeur est déclaré par l'instruction :

```
Type * nomPointeur;
```

où **Type** est le type de l'objet pointé.

Cette déclaration déclare un identificateur, **nomPointeur**, associé à un objet dont la valeur est l'adresse d'un autre objet de type **Type**.

L'identificateur **nomPointeur** est donc en quelque sorte un **identificateur d'adresse**. Comme pour n'importe quelle ***Lvalue***, sa valeur est modifiable.

Exemples :

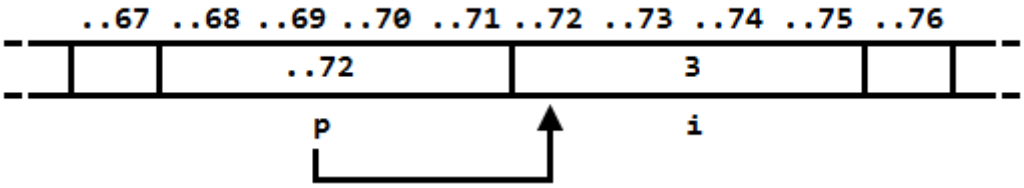
```
int *p1;       /* pointeur sur une variable de type entier */
float *p2;     /* pointeur sur une variable de type float  */
Eleve *p3;     /* pointeur sur une variable de type Eleve  */
double **p5;   /* pointeur sur un pointeur sur un double!  */
```

Dans l'exemple suivant, on définit un pointeur **p** qui pointe vers un entier **i** :

```
int i = 3;
int *p;
p = &i;
```

On se trouve dans la configuration :

objet	Adresse	valeur
i	2293572	3
p	2293568	2293572



L'opérateur unaire d'indirection ***** permet d'accéder directement à la valeur de l'objet pointé.

Si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**.

Par exemple, le programme :

```
main(){
  int i = 3;
  int *p;
  p = &i;
  printf("*p = %d \n", *p);
}
```

affichera :

```
*p = 3
```

Dans le programme, les objets **i** et ***p** sont identiques : ils ont mêmes adresse et valeur.

Nous sommes dans la configuration :

objet	adresse	valeur
i	**2293572**	**3**
p	**2293568**	**2293572**
***p**	**2293572**	**3**

Cela signifie en particulier que toute modification de ***p** modifie **i**.

Exemple :

```
main(){
    int i = 3;
    int *p;
    p = &i;
    printf("Avant : i = %d et *p = %d \n", i, *p);
    *p = 7;
    printf("Apres : i = %d et *p = %d \n", i, *p);
}
```

Le programme affichera :

```
Avant : i = 3 et *p = 3
Apres : i = 7 et *p = 7
```

On peut donc dans un programme manipuler à la fois les objets **p** et ***p**.

Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

(Programme 1)

```
main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

(Programme 2)

```
main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    p1 = p2;
}
```

Avant la dernière affectation des deux programmes, on est dans une configuration du type :

Objet	adresse	valeur
i	2293572	3
j	2293568	6
p1	2293564	2293572
p2	2293560	2293568

Après l'affectation `*p1 = *p2` du premier programme, on a :

Objet	adresse	valeur
i	2293572	6
j	2293568	6
p1	2293564	2293572
p2	2293560	2293568

Par contre, l'affectation `p1 = p2` du second programme, conduit à la situation :

Objet	adresse	valeur
i	2293572	3
j	2293568	6
p1	2293564	2293568
p2	2293560	2293568

2.4 Arithmétiques des pointeurs

La valeur d'un pointeur étant un entier :

⇨ on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

Les seules opérations arithmétiques valides sur les pointeurs sont :

- **l'addition d'un entier à un pointeur :** Le résultat est un pointeur de même type que le pointeur de départ ;
- **la soustraction d'un entier à un pointeur :** Le résultat est un pointeur de même type que le pointeur de départ ;
- **la différence de deux pointeurs pointant tous deux vers des objets de même type :** Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

Si **i** est un entier et **p** est un pointeur sur un objet de type **Type** :

- **p + i** désigne un pointeur sur un objet de type **Type** dont la valeur est égale à la valeur de **p** incrémentée de **i * sizeof(type)**.

Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation **++** et **--**.

Exemple :

```
main(){
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

Le programme affichera :

```
p1 = 2293572      p2 = 2293576
```

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
main(){
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

Le programme affichera :

```
p1 = 2293568      p2 = 2293576
```

Les opérateurs de comparaison sont également applicables aux pointeurs.

⇨ à condition de comparer des pointeurs qui pointent vers des objets de même type.

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.

Exemple :

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int *p;
    printf("Ordre croissant:\t");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf("%d \t",*p);
    printf("\nOrdre decroissant:\t");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf("%d \t",*p);
}
```

Le programme affichera :

```
Ordre croissant:      1     2     6     0     7
Ordre decroissant:    7     0     6     2     1
```

Remarque : Si **p** et **q** sont deux pointeurs sur des objets de type **Type**, l'expression **p - q** désigne un entier dont la valeur est égale à **(p - q)/sizeof(type)**.

2.5 Allocation dynamique

Avant de manipuler un pointeur, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée **NULL** définie dans **stdio.h**. En général, cette constante vaut **0**.

Le test **p == NULL** permet de savoir si le pointeur **p** pointe vers un objet.

On peut initialiser un pointeur **p** par une affectation sur **p**.

```
p = &a;
```

Il est également possible d'affecter directement une valeur à ***p**, mais pour cela, il faut d'abord réserver à ***p** un espace-mémoire de taille adéquate.

L'allocation de la mémoire en **C** se fait par la fonction **malloc** de la librairie standard **stdlib.h**. dont le prototype est :

```
void *malloc(size_t size);
```

Le seul paramètre à passer à **malloc** est le nombre d'octets à allouer. La valeur retournée est l'adresse du premier octet de la zone mémoire alloué. Si l'allocation n'a pu se réaliser (par manque de mémoire libre), la valeur de retour est la constante **NULL**.

Pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
main(){
  int *p;
  p = (int*)malloc(sizeof(int));
}
```

On aurait pu écrire également

```
p = (int*)malloc(4);
```

puisqu'un objet de type **int** est stocké sur **4** octets. Mais on préférera la première écriture qui a l'avantage d'être portable.

Le programme suivant définit un pointeur **p** sur un objet ***p** de type **int**, et affecte à ***p** la valeur de la variable **i** :

```c
#include <stdio.h>
#include <stdlib.h>
main(){
    int i = 3;
    int *p;
    printf("valeur de p avant initialisation = %d\n", p);
    p = (int*)malloc(sizeof(int));
    printf("valeur de p apres initialisation = %d\n", p);
    *p = i;
    printf("valeur de *p = %d\n",*p);
}
```

Le programme affichera :

```
valeur de p avant initialisation = 2293576
valeur de p apres initialisation = 5508960
valeur de *p = 3
```

Lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement, il faut libérer cette place en mémoire. Ceci se fait à l'aide de la fonction **`free`** dont le prototype est :

```c
void free(void *ptr);
```

2.6 Pointeurs et tableaux

L'usage des pointeurs en **C** est, en grande partie, orienté vers la manipulation des tableaux. Tout tableau en **C** est en fait un pointeur constant. Dans la déclaration :

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

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, **`tab`** a pour valeur **`&tab[0]`**.

On peut donc utiliser un pointeur initialisé à **`tab`** pour parcourir les éléments du tableau.

```c
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++){
        printf("%d\t",*p);
        p++;
    }
}
```

Le programme affichera :

```
1       2       6       0       7
```

On accède à l'élément d'indice **i** du tableau **tab** grâce à l'opérateur d'indexation **[]**, par l'expression **tab[i]**. Cet opérateur d'indexation peut en fait s'appliquer à tout objet **p** de type pointeur. Il est lié à l'opérateur d'indirection * par la formule :

```
p[i] == *(p + i)
```

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire :

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
        printf("%d\t", p[i]);
}
```

Le programme affichera :

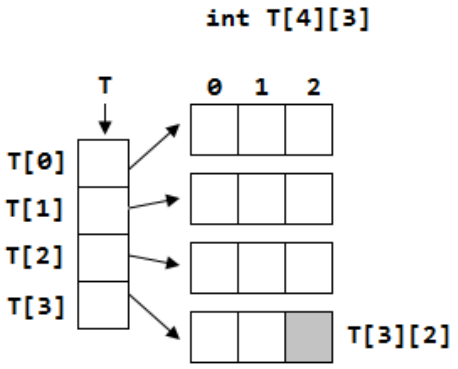
```
1       2       6       0       7
```

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

tab est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier et a une valeur constante égale à l'adresse du premier élément du tableau, **&tab[0][0]**.

De même **tab[i]** (pour **i** entre **0** et **M-1**) est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice **i**. **tab[i]** a donc une valeur constante qui est égale à **&tab[i][0]**.



On déclare un pointeur qui pointe sur un objet de type **Type *** (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire :

```
Type ** nomPointeur;
```

De même un pointeur qui pointe sur un objet de type **Type** ** (équivalent à un tableau à 3 dimensions) se déclare par :

```
Type *** nomPointeur;
```

Par exemple, pour créer avec un pointeur de pointeur une matrice à **k** lignes et **n** colonnes à coefficients entiers, on écrit :

```
main(){
  int k, n;
  int **tab;
  tab = (int**)malloc(k * sizeof(int*));

  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int));
      ...

  for (i = 0; i < k; i++)
    free(tab[i]);
  free(tab);
}
```

2.7 Pointeurs et structures

Contrairement aux tableaux, les objets de type structure en **C** sont des ***Lvalues***. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures.

Exemple :

```
#include <stdlib.h>
#include <stdio.h>
struct Eleve {
    char nom[20];
    float note;
};
typedef struct Eleve Eleve;
typedef Eleve * Classe;
main(){
    int n, i;
    Classe TE;
    printf("Nombre d'eleves de la classe : ");
    scanf("%d",&n);
    TE = (Classe)malloc(n * sizeof(Eleve));
    for (i =0 ; i < n; i++){
        printf("\nSaisie de l'eleve numero : %d\n",i);
        printf("\tNom : ");    scanf("%s", TE[i].nom);
        printf("\tNote : ");    scanf("%f",&TE[i].note);
    }
    printf("\nEntrez un numero : ");   scanf("%d",&i);
    printf("\nEleve numero %d est : ", i);
    printf("\nNom ===> %s",TE[i].nom);
    printf("\nNote ===> %.2f\n",TE[i].note);
    free(TE);
}
```

Si **p** est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression :

```
p->membre
```

ou

```
(*p).membre
```

Exemple :

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

struct Eleve {
    char nom[20];
    float note;
};

typedef struct Eleve Eleve;

main(){
    Eleve * pE;
    pE = (Eleve*)malloc(sizeof(Eleve));
    strcpy(pE->nom, "Alami");
    pE->note = 13;
    printf("L'eleve %s a %.2f/20\n", (*pE).nom, (*pE).note);
    free(pE);
    getch();
}
```

Le programme affichera :

```
L'eleve Alami a 13.00/20
```

3 Les fonctions et la récursivité

3.1 Introduction

La structuration de programmes en sous-programmes se fait en **C** à l'aide de fonctions. Les fonctions en **C** correspondent aux fonctions et procédures en langage algorithmique. Créer une fonction est utile quand on a à faire le même type de traitement plusieurs fois dans le programme, mais avec des valeurs différentes.

3.2 Définition d'une fonction

On définit une fonction comme suit :

```
Type nomFonction(Type1 param1 , Type2 param 2, ... , Typen paramn){
   déclaration variables locales ;
   instructions ;
   return (expression) ;
}
```

Remarque : Quand le programme rencontre l'instruction **`return`**, l'appel de la fonction est terminé. Toute instruction située après lui sera ignorée.

Exemples :

```
int produit (int a, int b) {
  return (a * b);
}
float affine(float x ) {
    int a, b;
    a = 3;
    b = 5;
    return (a * x + b) ;
}
float distance(int x, int y){
    return (sqrt(x * x + y * y)) ;
}
float valAbsolue(float x ) {
    return (x < 0) ? (-x) : (x) ;
}
double pi() {
    return (3.14159) ;
}
void messageErreur() {
    printf("Vous n'avez fait aucune erreur\n") ;
}
```

3.3 Appel d'une fonction

Une fonction **f()** peut être appelée depuis le programme principal **main()** ou bien depuis une autre fonction **g()**.

Exemple 1 :

```c
#include <stdio.h>
main(){
    int x, y, r;
    int plus( int x, int y ) ;   /* déclaration de la fonction */
    x = 5;
    y = 235;
    r = plus(x, y);              /* appel d'une fonction avec arguments */
    printf("%d + %d = %d", x, y, r);
}
int plus(int x, int y){
    void mess();                 /* déclaration de la fonction */
    mess() ;                     /* appel d'une fonction sans arguments */
    return (x+y) ;
}
void mess() {
    printf("Vous n'avez fait aucune erreur\n");
    return ;
}
```

Le programme affichera :

```
Vous n'avez fait aucune erreur
5 + 235 = 240
```

Exemple 2 :

```c
double conversion(double euros){
    double dhs;
    dhs = 11 * euros;
    return dhs;
}
main(){
    printf("10 euros = %.2f dhs\n", conversion(10));
    printf("50 euros = %.2f dhs\n", conversion(50));
    printf("100 euros = %.2f dhs\n", conversion(100));
    printf("200 euros = %.2f dhs\n", conversion(200));
}
```

Le programme affichera :

```
10 euros = 110.00 dhs
50 euros = 550.00 dhs
100 euros = 1100.00 dhs
200 euros = 2200.00 dhs
```

Exemple 3 :

```c
#include <stdio.h>
main(){
    /* Prototypes des fonctions appelées */
    int ENTREE(void);
    int MAX(int N1, int N2);
    /* Déclaration des variables */
    int A, B;
    /* Traitement avec appel des fonctions */
    A = ENTREE();
    B = ENTREE();
    printf("Le maximum est %d\n", MAX(A,B));
}
/* Définition de la fonction ENTREE */
int ENTREE(void){
    int NOMBRE;
    printf("Entrez un nombre entier : ");
    scanf("%d", &NOMBRE);
    return NOMBRE;
}
/* Définition de la fonction MAX */
int MAX(int N1, int N2){
    return (N1>N2) ? N1 : N2;
}
```

3.4 Durée de vie des variables

Les variables manipulées dans un programme **C** n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

- **Les variables permanentes (statiques) :** occupe le même emplacement en mémoire (*segment de données*) durant toute l'exécution du programme. Elles sont initialisées à zéro par le compilateur par défaut et se caractérisent par le mot-clef **`static`**.
- **Les variables temporaires (automatiques) :** se voient allouer un emplacement en mémoire (*segment de pile*) de façon dynamique lors de l'exécution du programme et ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

3.4.1 Variables globales

On appelle variable globale une variable déclarée en dehors de toute fonction et est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, **`n`** est une variable globale :

```c
int n;
void fonction(){
  n++;      printf("appel numero %d\n",n);
}
main() {
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affichera :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

3.4.2 Variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues. Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom.

Exemple :

```c
int n = 10;
void fonction(){
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}
main(){
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affiche :

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef **`static`** :

```c
static Type nomVariable;
```

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, **`n`** est une variable locale à la fonction secondaire fonction, mais de classe **statique**.

Exemple :

```
int n = 10;
void fonction(){
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}
main(){
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affichera :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale **n** est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

3.5 Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique. La fonction travaille alors uniquement sur cette copie qui disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée (La variable du programme appelant, elle, ne sera pas modifiée). On dit que les paramètres d'une fonction sont transmis par valeurs.

Exemple :

```
void echange (int x, int y){
    int t;
    printf("Debut fonction :\n x = %d \t y = %d\n", x, y);
    t = x;      x = y;     y = t;
    printf("Fin fonction :\n x = %d \t y = %d\n", x, y);
}
main(){
    int a = 2, b = 5;
    printf("Debut programme principal :\n a = %d \t b = %d\n", a, b);
    echange(a, b);
    printf("Fin programme principal :\n a = %d \t b = %d\n", a, b);
}
```

Le programme affichera :

```
Debut programme principal :
 a = 2   b = 5
Debut fonction :
 x = 2   y = 5
Fin fonction :
 x = 5   y = 2
Fin programme principal :
 a = 2   b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur (sa copie). Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```c
void echange (int * adrA, int * adrB){
    int t = *adrA;
    *adrA = *adrB;
    *adrB = t;
}
main(){
    int a = 2, b = 5;
    printf("Debut programme principal :\n a = %d \t b = %d\n", a, b);
    echange(&a, &b);
    printf("Fin programme principal :\n a = %d \t b = %d\n", a, b);
}
```

Le programme affichera :

```
Debut programme principal :
 a = 2   b = 5
Fin programme principal :
 a = 5   b = 2
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction.

Exemple :

```c
void init (int tab[], int n){
    int i;
    for (i = 0; i < n; i++)
        tab[i] = i;
}
main(){
    int i;
    int T[5];
    init(T, 5);
    for (i = 0; i < 5; i++){
        printf("%d\t",T[i]);
    }
}
```

Exemple de fonctions utilisant une structure :

```
struct Eleve{
    int code;
    char nom[20];
    float note;
};
typedef struct Eleve Eleve;

void initEleve(Eleve * adrE, int c, char nm[20], float nt){
    adrE->code = c;      strcpy(adrE->nom, nm);      adrE->note = nt;
}
void modifierNote(Eleve * adrE, float nt){
    adrE->note = nt;
}
void afficher(Eleve E){
    printf("Eleve : %d %s %.2f\n", E.code, E.nom, E.note);
}
main(){
    Eleve a;
    initEleve(&a, 111, "Khalid", 13);
    afficher(a);
    modifierNote(&a, 16);
    afficher(a);
}
```

Le programme affichera :

```
Eleve : 111 Khalid 13.00
Eleve : 111 Khalid 16.00
```

3.6 Pointeurs de fonction

Comme une fonction n'est pas un objet ou une variable, il n'est pas possible de passer une fonction directement en argument à une fonction. Par contre, les fonctions ont une adresse. Il est donc possible de déclarer un pointeur vers cette adresse et de passer ce pointeur à une fonction.

Voici un exemple de la déclaration d'un pointeur qui pointe successivement vers les fonctions **`fonction1()`** et **`fonction2()`** :

```
void fonction1() {
  printf("affiche fonction 1\n");
}
void fonction2() {
   printf("affiche fonction 2\n");
}
main() {
   void (*fonction)();
   fonction = fonction1;
   fonction();
   fonction = fonction2;
   fonction();
}
```

Le programme affichera :

```
affiche fonction 1
affiche fonction 2
```

Exemple de fonctions utilisant une structure :

```c
#include <math.h>

// typedef pour simplifier la notation
typedef double(*Fonction)(double );

// Liste des fonctions "calculables"
double carre(double x) { return x*x;}
double inverse(double x) { return 1/x;}
double racine(double x) { return sqrt(x);}
double exponentielle(double x) { return exp(x);}

double minimum(double a, double b, Fonction f){
    double x;
    double min = 100000;
    for(x=a; x<b ; x+= 0.01)
        min = min< f(x)? min : f(x);
    return min;
}

main(){
    printf("De quelle fonction voulez-vous chercher le minimum ?\n");
    printf("1  --  x^2\n");
    printf("2  --  1/x\n");
    printf("3  --  racine de x\n");
    printf("4  --  exponentielle de x\n");
    printf("5  --  sinus de x\n");

    int reponse;
    scanf("%d", &reponse);
    //On declare un pointeur sur fonction
    Fonction monPointeur;

    //Et on déplace le pointeur sur la fonction choisie
    switch(reponse){
        case 1:  monPointeur = carre; break;
        case 2:  monPointeur = inverse;  break;
        case 3:  monPointeur = racine;  break;
        case 4:  monPointeur = exponentielle;  break;
        // On peut même utiliser les fonctions de math.h !
        case 5:  monPointeur = sin; break;
    }
    //Finalement on affiche le résultat de l'appel de la fonction via le pointeur
    printf("Le minimum de la fonction entre 3 et 4 est : %.2f\n",
                                                    minimum(3, 4, monPointeur));
}
```

Le programme affichera :

```
De quelle fonction voulez-vous chercher le minimum ?
1  --  x^2
2  --  1/x
3  --  racine de x
4  --  exponentielle de x
5  --  sinus de x
4
Le minimum de la fonction entre 3 et 4 est : 20.09
```

3.7 Les fonctions récursives

Une fonction est dite récursive lorsqu'elle est définie en fonction d'elle-même. La programmation récursive est une technique de programmation qui remplace les instructions de boucle (**`while`**, **`for`**, etc.) par des appels de fonction (**N.B. :** ne pas confondre avec la notion de récursivité en mathématiques).

3.7.1 Premier exemple

Le mécanisme le plus simple pour faire boucler une fonction : elle se rappelle elle-même.

Exemple :

```
void boucle() {
  boucle();
}
```

On peut en profiter pour faire quelque chose :

```
void boucle() {
  printf ("Je tourne \n");
  boucle();
}
```

C'est ce qu'on appelle une récursivité simple. Il faut encore ajouter un mécanisme de test d'arrêt.

Exemple :

Ecrire 100 fois "**`Je tourne`**" : on a besoin d'un compteur. On choisit ici de le passer d'un appel de fonction à l'autre comme un paramètre et il faut appeler **`boucle(0)`**.

```
void boucle(int i) {
  if (i < 100) {
    printf ("Je tourne \n") ;
    boucle(i+1) ;
  }
  //sinon on ne relance pas => fin du programme
}
```

3.7.2 Apprendre à programmer récursivement avec des variables

Calculer la somme des **`n`** premiers nombres avec une boucle **`while`** :

```
int somme(int n) {
  int s = 0;
  int i = 1;
  while (i <= n) {
    s += i;
    i++;
  }
  return s ;
}
```

Par une procédure récursive, **méthode très naïve** :

```
int n = 100 ;
int s = 0 ;
void sommeRec(int i) {
  if (i <= n) {
    s += i ;
    sommeRec(i+1) ;
  }
  //si i = n, fin du programme
}
```

On lance **`sommeRec(0);`**

C'est extrêmement maladroit parce que :

1. on ne contrôle pas la valeur de **`s`** au début
2. on ne contrôle pas la valeur terminale **`N`**.

Pour éviter cela, il faut accéder à **`s`** et **`n`** sans qu'elles soient en variables globales. Une solution pour construire une procédure récursive sérieuse est d'utiliser pour passer **`s`** et **`n`** les paramètres et la valeur de retour de la fonction.

```
int sommeRec(int i, int s, int n) {
  if (i <= n)
    return sommeRec(i+1, s+i, n) ;
  else
    return (s) ;                    // sommeRec(n,s,n), on a fini
}
```

On lance **`sommeRec(0, 0, 100);`**

On aurait pu éviter de passer à la fois **`i`** et **`n`**, en comptant à l'envers de **`n`** à **`0`** :

```
int sommeRec(int s, int n) {
  if (n > 0)
    return sommeRec(s + n, n - 1 );
  else
    return (s) ;                  // Pour sommeRec(0, s), le calcul est immédiat
}
```

On lance **`x = sommeRec(0, 100);`**

Et on pouvait même éviter de passer s, en gérant plus efficacement la valeur de retour :

```
int sommeRec(int n) {
  if (n > 0)
    return sommeRec(n - 1 ) + n;
  else
    return (0) ;            // pour sommeRec(0), le calcul est immédiat
}
```

On lance **`x = sommeRec(100);`**

Enfin, on peut s'apercevoir qu'il est plus astucieux de programmer une fonction plus générale :

`sommeRec(debut, fin)` : c'est "faire la somme des nombres de **`debut`** jusqu'à **`fin`** ".

La programmation récursive, c'est : on appelle la même fonction avec un nombre de moins dans la liste, puis on ajoute ce nombre au résultat.

D'où deux versions :

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)
    return sommeRec(deb, fin - 1 ) + fin;
  else
    return (0) ;           // pour sommeRec(x,x), le calcul est immédiat
}
```

ou

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)
    return deb + sommeRec(deb + 1, fin );
  else
    return (0) ;          // pour sommeRec(x,x), le calcul est immédiat
}
```

Dans les deux cas, on lance **`sommeRec(0, 100)`**, **`sommeRec(10, 50)`**, etc.

3.7.3 Différents types de récursivité

a. Récursivité simple

Une fonction simplement récursive, c'est une fonction qui s'appelle elle-même une seule fois, comme c'était le cas pour **`sommeRec()`** ci dessus.

Prenons à la fonction puissance $x \to x^n$ x. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & si \;\; n = 0; \\ x \times x^{n-1} & si \;\; n \geq 1. \end{cases}$$

La fonction correspondante s'écrit :

```
int puissance(int x, int n){
    if(n == 0)
        return 1;
    else
        return x * puissance(x, n-1);
}
```

b. Récursivité multiple

Une fonction peut exécuter plusieurs appels récursifs.

Par exemple le calcule des combinaisons C_n^p en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & si \quad p = 0 \quad ou \quad p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon. \end{cases}$$

La fonction correspondante s'écrit :

```
int combinaison(int n, int p){
    if(p == 0 || p == n)
        return 1;
    else
        return combinaison(n-1, p) + combinaison(n-1, p-1);
}
```

c. Récursivité mutuelle

Des fonctions sont dites mutuellement récursives si elles dépendent les unes des autres. Par exemple, deux fonctions **`A(x)`** and **`B(x)`** définies comme suit :

$$A(x) = \begin{cases} 1 & si \quad x \leq 1; \\ B(x+2) & si \quad x > 1. \end{cases} \qquad B(x) = \{A(x-3) + 4$$

Les fonctions correspondantes s'écrivent :

```
int A(int x){
    if(x <= 1)
        return 1;
    else
        return B(x+2);
}
int B(int x){
    return A(x-3) + 4;
}
```

d. Récursivité imbriquée

Une fonction contient une récursivité imbriquée s'il contient comme paramètre un appel à lui-même.

C'est le cas de la fonction d'Ackermann définie comme suit :

$$A(m,n) = \begin{cases} n+1 & si \;\; m = 0; \\ A(m-1,1) & si \; m > 0 \quad et \;\; n = 0; \\ A\big(m-1, A(m,n-1)\big) & sinon. \end{cases}$$

La fonction correspondante s'écrit :

```
int ackermann(int m, int n) {
    if (m == 0)
        return n + 1;
    else
        if (n == 0)
            return ackermann (m - 1, 1);
        else
            return ackermann (m - 1, ackermann (m, n - 1));
}
```

3.7.4 Principe et dangers de la récursivité

Une fonction récursive est dite bien définie si elle possède les deux propriétés suivantes:

- Il doit exister certains critères, appelés critères d'arrêt ou conditions d'arrêt, pour lesquels la fonction ne s'appelle pas elle-même.
- Chaque fois que la procédure s'appelle elle-même (directement ou indirectement), elle doit converger vers ses conditions d'arrêt.