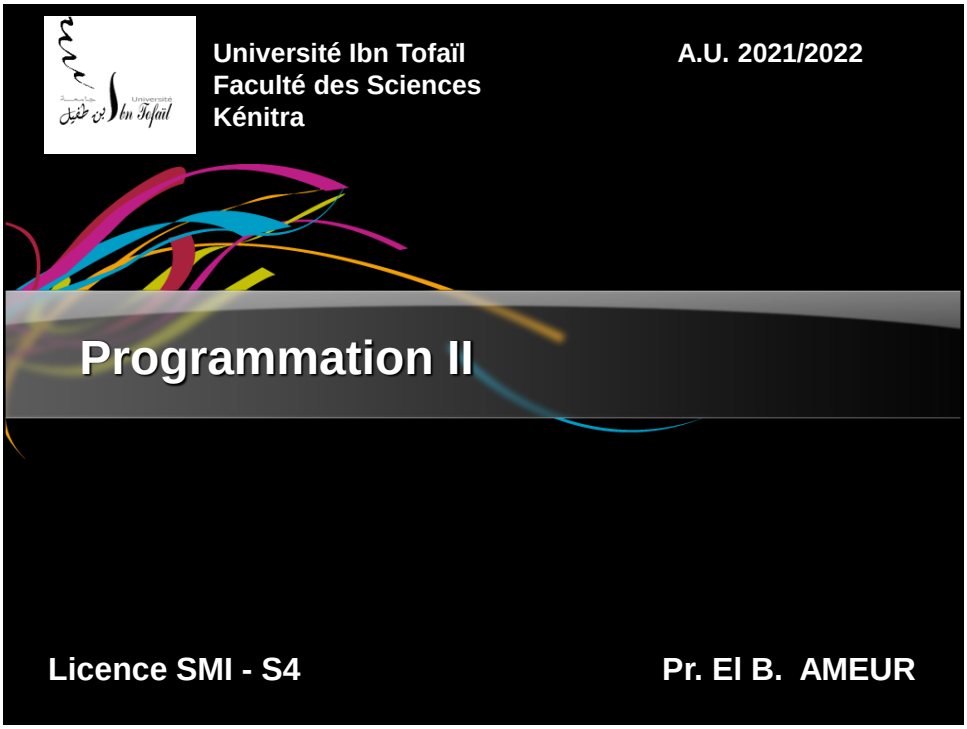


Plan du cours

Partie 1: Rappels et compléments du langage C

1. Les types composés
2. Les pointeurs
3. Les fonctions et la récursivité
4. Les fichiers

Partie 2: Implémentation des Types de Données Abstraits en C

5. Les listes chainées
6. Les piles
7. Les files
8. Les arbres

2

Support du cours en ligne

Google Classroom:

Programmation II

Utilisez votre émail institutionnel

Code d'accès:

s6yy2mw

3

Partie 1 : Rappels et compléments du langage C

3 LES FONCTIONS ET LA RÉCURSIVITÉ

5

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.

6

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) ;
}
```

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

7

3.2 Définition d'une fonction

- **Exemples :**

```
int produit (int a, int b) {
  return (a * b);
}
float affine(float x ) {
    int 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:**

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

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

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:**

```c
#include <stdio.h>
//Prototypes des fonctions
void mess();          /* déclaration de la fonction */
int plus( int x, int y ) ;  /* déclaration de la fonction */

main(){
    int x, y, r;
    x = 5;          y = 235;
    r = plus(x, y);             /* appel d'une fonction avec arguments */
    printf("%d + %d = %d", x, y, r);
}
void mess() {
    printf("Vous n'avez fait aucune erreur\n");
    return ;
}
int plus(int x, int y){
```

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

3.3 Appel d'une fonction

- **Exemple 2 :**

```c
double conversion(double euros){
    double dhs;
    dhs = 10 * 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));
}
```

```
10 euros = 100.00 dhs
50 euros = 500.00 dhs
100 euros = 1000.00 dhs
200 euros = 2000.00 dhs
```


3.3 Appel d'une fonction

- **Exemple 3 :**

```c
/* Prototypes des fonctions appelées */
int ENTREE(void);
int MAX(int N1, int N2);
main(){
/* 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(){
    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.

13

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 :

14

3.4.1 Variables globales

- **Exemple:**

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

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

15

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.
- 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.

16

3.4.2 Variables locales

- **Exemple:**

```
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();
}
```

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

3.4.2 Variables locales

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

```
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.

18

3.4.2 Variables locales

- **Exemple:**

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

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

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 ne sera pas modifiée).
- On dit que les paramètres d'une fonction **sont transmis par valeurs.**

20

3.5 Transmission des paramètres d'une fonction

- **Exemple :**

```
void echange (int x, int y){
    int temp;
    printf("Debut fonction :\n x = %d \t y = %d\n", x, y);
    temp = x;       x = y;      y = temp;
    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);
}
```

```
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
```

3.5 Transmission des paramètres d'une fonction

- 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 pas sa valeur (sa copie).
- **Exemple:**

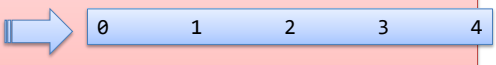
```
void echange (int * adrA, int * adrB){
    int temp = *adrA;    *adrA = *adrB;      *adrB = temp;
}
main(){
    int a = 2, b = 5;
    printf("Debut programme principal :\n a = %d \t b = %d\n", a, b);
    echange(&a, &b);
    printf("Fin programme principal :\n a = %d \t b = %d\n", a, b);
}
```

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

3.5 Transmission des paramètres d'une fonction

- 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:**

```
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);//Appel
    for (i = 0; i < 5; i++){
        printf("%d\t",T[i]);
    }
}
```



23

3.5 Transmission des paramètres d'une fonction

- **Exemple:** fonctions utilisant une structure

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

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

3.6 Les fonctions récursives

- Une fonction est dite récursive lorsqu'elle est **définie en fonction d'elle-même.**
  (Fait appel à elle-même un certain nombre de fois **fini**).

- La programmation récursive est une technique de programmation qui remplace les instructions de boucle (**while**, **for**, etc.) par des appels de fonction.

25

3.6.2 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 ;
}
```

26

3.6.2 Apprendre à programmer récursivement avec des variables

- Version récursive:

```
int sommeRec(int n) {
  if (n > 0)
    return sommeRec(n - 1 ) + n;
  else
    return (0) ;

}
```

- On lance **s = sommeRec(100);**

27

3.6.2 Apprendre à programmer récursivement avec des variables

- 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** ".
- D'où deux versions :

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)    return sommeRec(deb, fin - 1 ) + fin;
  else  return (0) ;
 }
```

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)    return deb + sommeRec(deb + 1, fin );
  else  return (0) ;
}
```

Dans les deux cas, on lance **sommeRec(0, 100)**, **sommeRec(10, 50)**, etc. 28

3.6.3 Différents types de récursivité

- **Récursivité simple**
- **Récursivité multiple**
- **Récursivité mutuelle**
- **Récursivité imbriquée**

29

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()**.
- Prenons à la fonction puissance $x \rightarrow x^n$. Cette fonction peut être définie récursivement :

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

- La fonction correspondante s'écrit :

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

30

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));
}
```

33

3.6.4 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**.

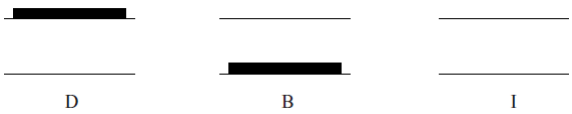
34

Exercice: **Tours de Hanoi**

- Les « Tours de Hanoi » est un jeu où il s'agit de déplacer un par un des disques superposés de diamètre décroissant d'un socle de départ **D** sur un socle de but **B**, en utilisant éventuellement un socle intermédiaire **I**. Un disque ne peut se trouver au dessus d'un disque plus petit que lui. Ecrire la fonction récursive deplacer (n, D, B, I) qui déplace n disques su d vers B.

- **Le cas d'un disque:** Pour déplacer un disque de **D** vers **B**, le socle **I** (intermédiaire) est inutile.

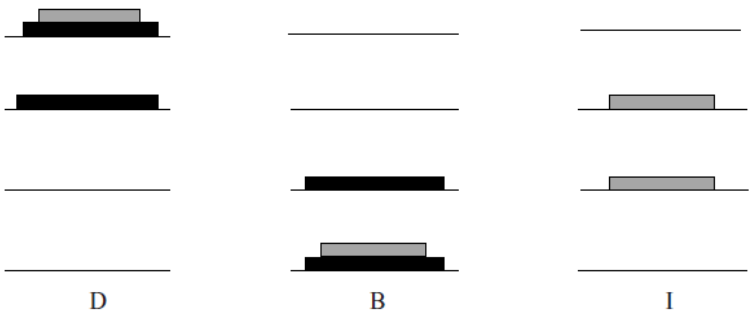
```
deplacer(1, D ,B ,I);
```



Exercice: **Tours de Hanoi**

- **Le cas de 2 disques:** Pour déplacer **2** disques, il faut transférer celui qui est au sommet sur le socle **I**, déplacer le disque reposant sur le socle **D** vers **B**, et ramener le disque du socle **I** au sommet de **B**.

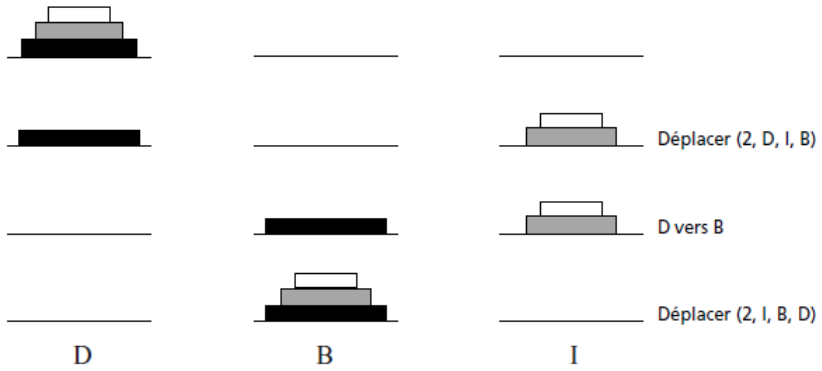
```
deplacer(2, D ,B ,I);
```



Exercice: **Tours de Hanoi**

- **Le cas de 3 disques:** Pour déplacer **3** disques, il faut déplacer les 2 disques en sommet de **D** vers **I** (en utilisant **B** comme intermédiaire), ensuite déplacer le disque reposant sur le socle de **D** vers **B**, et ramener les 2 disques mis de côté sur **I**, en sommet de **B**.

```
deplacer(3, D ,B ,I);
```



4 LES FICHIERS