


Université Ibn Tofail
Faculté des Sciences
Kénitra

A.U. 2021/2022



Programmation II

Licence SMI - S4

Pr. El B. AMEUR

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 chaînées
6. Les piles
7. Les files
8. Les arbres

Support du cours en ligne

Google Classroom:

Programmation II

Utilisez votre email institutionnel

Code d'accès:

s6yy2mw

3

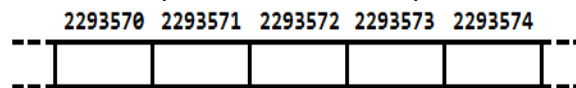
Partie 1 : Rappels et compléments du langage C

2 LES POINTEURS

5

2.1 Introduction

- Les variables utilisées dans un programme sont stockées quelque part en mémoire centrale constituée d'octets adjacents identifiés par un numéro unique : **adresse**.



- Retrouver variable = connaître l'adresse du 1^{er} octet.
- **Rappel :**
 - **&** appliqué à une variable retourne l'adresse-mémoire de cette variable : **&variable**.

6

2.1 Introduction

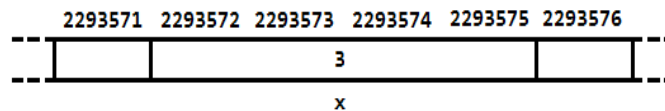
- Quand on écrit :

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



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

- Ce code réserve **4 octets** pour la variable **x** dans la mémoire (int sont codés sur 4 octets) à partir de **2293572** 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**.

7

2.2 Adresse et valeur d'un objet

- On appelle **Lvalue** (*left value*) tout objet placé à gauche de l'opérateur d'affectation (=).
- **Lvalue** est caractérisée par :
 - **son adresse** : à partir de laquelle l'objet est stocké ;
 - **sa valeur** : donnée stockée à cette adresse.

8

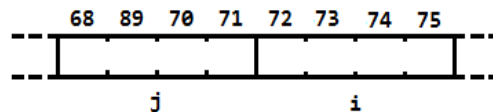
2.2 Adresse et valeur d'un objet

- 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



9

2.3 Notion de pointeur

- Pointeur** : objet (*Lvalue*) dont la **valeur** = une **adresse**.
- Déclaration**:

```
Type * nomPointeur;
```

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

Comme pour n'importe quelle *Lvalue*, la valeur d'un pointeur est modifiable.



10

2.3 Notion de pointeur

- 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! */
```

11

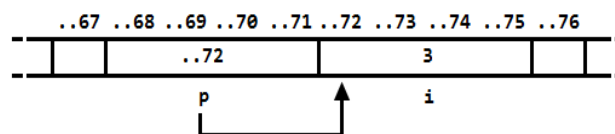
2.3 Notion de pointeur

- L'exemple suivant, définit un pointeur **p** sur 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



12

2.3 Notion de pointeur

- L'opérateur `*` permet d'accéder à la valeur de l'objet pointé.
- Si `p` est un pointeur vers un entier `i`, `*p` désigne la valeur de `i`.
- **Exemple:**

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



`*p = 3`

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

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



13

2.3 Notion de pointeur

- **Exemple:**

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



Avant : i = 3 et *p = 3
Après : i = 7 et *p = 7

14

2.3 Notion de pointeur

- On peut donc dans un programme manipuler à la fois les objets **p** et ***p**.
 → Ces deux manipulations sont très différentes.
- Comparons 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;
}
```

15

2.3 Notion de pointeur

- Configuration des deux programmes avant la dernière affectation :

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

Après ***p1 = *p2**Après **p1 = p2**

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

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

16

2.4 Arithmétiques des pointeurs

- La valeur d'un pointeur étant un entier :
 - on peut lui appliquer un certain nombre d'opérateurs arithmétiques sur un pointeur.
- Les seules opérations valides sont :
 - **addition d'un entier à un pointeur** :
 - Résultat = pointeur de même type que le pointeur de départ ;
 - **soustraction d'un entier à un pointeur** :
 - Résultat = pointeur de même type que le pointeur de départ ;
 - **différence de deux pointeurs pointant tous deux vers des objets de même type** :
 - Résultat est un entier.

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



17

2.4 Arithmétiques des pointeurs

- 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 $i * \text{sizeof}(\text{Type})$.
- Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrément et de décrémentation $++$ et $--$.

18

2.4 Arithmétiques des pointeurs

- Exemple :

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



p1 = 2293572

p2 = 2293576

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



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.



2.4 Arithmétiques des pointeurs

- 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: ");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf("%d ", *p);
    printf("\nOrdre décroissant: ");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf("%d ", *p);
}
```



Ordre croissant: 1 2 6 0 7

Ordre décroissant: 7 0 6 2 1

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.
- On peut initialiser un pointeur **p** par la constante symbolique notée **NULL** définie dans **stdio.h** :

```
p = NULL; // p ne pointe sur aucun objet
```

- Le test **p == NULL** permet de savoir si le pointeur **p** pointe vers un objet ou non.
- On peut initialiser un pointeur **p** par une affectation sur **p**.

```
p = &a;
```

21

2.5 Allocation dynamique

- 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**.

22

2.5 Allocation dynamique

- 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);
```

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

23

2.5 Allocation dynamique

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

```
#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);
}
```



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

24

2.5 Allocation dynamique

- 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 :

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

25

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 :

```
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]**.

26

2.6 Pointeurs et tableaux

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

```
#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));

    }
}
```



1 2 6 0 7

27

2.6 Pointeurs et tableaux

- 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)

- Résultat:** Pointeurs et tableaux se manipulent donc exactement de même manière.

28

2.6 Pointeurs et tableaux

- **Exemple:**

```
#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]); // *(p+i)
}
```



1	2	6	0	7
---	---	---	---	---

29

2.6 Pointeurs et tableaux

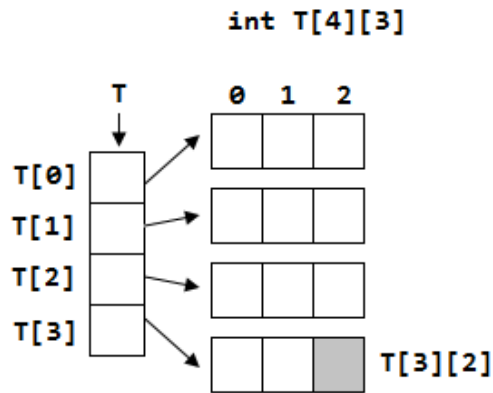
- Un tableau à 2D est, par définition, un tableau de tableaux.
→ Il s'agit donc en fait d'un pointeur vers un pointeur.
- Considérons le tableau à deux dimensions défini par :

```
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]**.

30

2.6 Pointeurs et tableaux



31

2.6 Pointeurs et tableaux

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

```
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;
```

32

2.6 Pointeurs et tableaux

- Exemple: tableau de deux dimensions

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

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

    for (i = 0; i < k; i++)
        free(A[i]);

    free(A);
}
```

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.

2.7 Pointeurs et structures

- 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

35

2.7 Pointeurs et structures

- Exemple:

```
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);
}
```



L'eleve Alami a 13.00/20

36

2.7 Pointeurs et structures

- 9 On considère la structure suivante:
- ```
typedef struct Personne {
 char CIN[8];
 char nom[10];
 int age;
}Personne;
Personne *P1 , P2={"B123" , "Farid", 26};
//Allocation dynamique de *P1:
P1=(Personne*)malloc(sizeof(Personne));
strcpy (P1->nom, "Hamid");
strcpy (P1->CIN, "A123");
P1->age = 21;
printf ("%s %s %d\n", P1->CIN, P1->nom, P1->age);
```
- 9 L'accès aux champs d'un pointeur d'enregistrement se fait via ->  
P1->age est équivalente à (\*P1).age

P1->CIN  
8 Octets

P1->nom  
10 Octets

P1->age  
4 Octets

37

## 2.7 Pointeurs et structures

- 9 On considère la structure suivante où le champ nom est un pointeur:

```
typedef struct Personne {
 char CIN[8];
 char *nom;
 int age;
}Personne;
```

```
Personne P1;
strcpy (P1.CIN, "A123");
P1.age = 21;
```

```
//Allocation dynamique de P1.nom:
```

```
P1.nom=(char *)malloc(10*sizeof(char));
P1.nom ="Hamid";
```

```
printf ("%s %s %d\n", P1.CIN, P1.nom, P1.age);
```

P1.CIN  
8 Octets

P1.nom

P1.age  
4 Octets

10 Octets

38

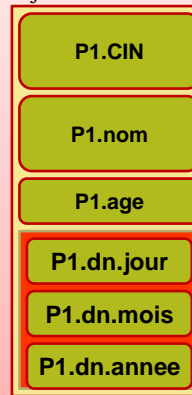
## 2.7 Pointeurs et structures

- 9 On considère la structure suivante où on a ajouter à la structure Personne le champ date de naissance:

```
typedef struct date{
 int jour, mois, annee;
}date;

typedef struct Personne {
 char CIN[8];
 char nom[10];
 int age;
 date dn;
}Personne;

Personne P1;
strcpy (P1.CIN, "B123");
strcpy(P1.nom , "Hamid");
P1.age = 21;
P1.dn.jour=15; P1.dn.mois=11; P1.dn.annee=1998;
```



39

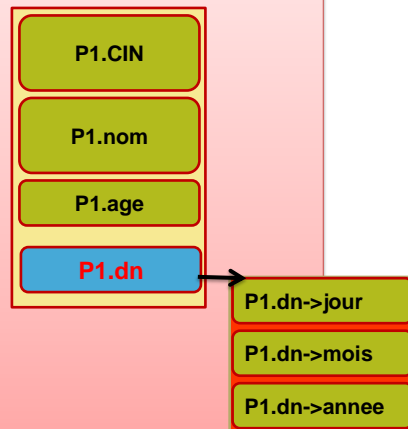
## 2.7 Pointeurs et structures

- 9 On considère la structure suivante où on a ajouter à la structure Personne le champ date de naissance:

```
typedef struct date{
 int jour, mois, annee;
}date;

typedef struct Personne {
 char CIN[8];
 char nom[10];
 int age;
 date * dn;
}Personne;

Personne P1;
strcpy (P1.CIN, "B123");
strcpy(P1.nom , "Hamid");
P1.age = 21;
```



```
P1.dn=(date*)malloc(sizeof(date));
P1.dn->jour=15;P1.dn->mois=11; P1.dn->annee=1998;
```

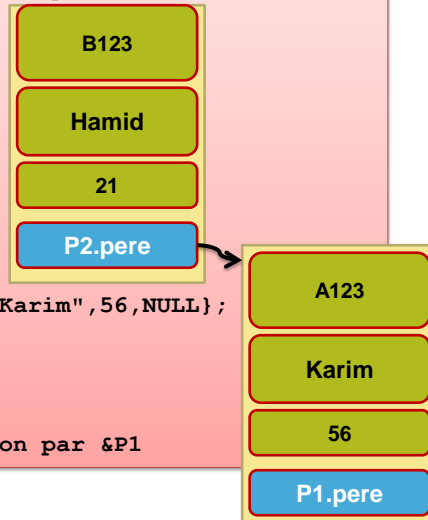
40

## 2.7 Pointeurs et structures

- 9 Afin de relier entre elles les différentes enregistrements qui correspondent aux différentes personnes, on décrit un pointeur sur un champ de type Personne. Ce pointeur fournit l'adresse de la personne pointée.

```
typedef struct Personne {
 char CIN[8];
 char nom[10];
 int age;
 struct Personne *pere;
} Personne;
```

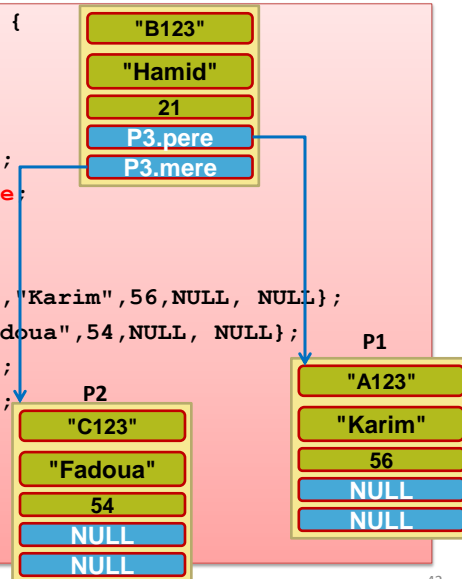
```
Personne P2, P1={"A123", "Karim", 56, NULL};
strcpy (P2.CIN, "B123");
strcpy(P2.nom, "Hamid");
P2.age = 21;
P2.pere=&P1; //initialisation par &P1
```



## 2.7 Pointeurs et structures

```
typedef struct Personne {
 char CIN[8];
 char nom[10];
 int age;
 struct Personne *pere;
 struct Personne *mere;
} Personne;
```

```
Personne P3, P1={"A123", "Karim", 56, NULL, NULL};
Personne P2={"C123", "Fadoua", 54, NULL, NULL};
strcpy (P3.CIN, "B123");
strcpy(P3.nom, "Hamid");
P3.age = 21;
P3.pere=&P1;
P3.mere=&P2;
```



42

## **3 LES FONCTIONS ET LA RÉCURSIVITÉ**

43