Chapitre 1 : Programmation Modulaire en C

Les Functions
Les pointeurs

Rafika Brahmi

Niveau : LSIM1
Institut Supérieur d'Informatique et Multimédia de Gabes(ISIMG)

Plan du chapitre

Les Functions

Les Pointeurs

I. Introduction

Un programme en C peut être découpé en plusieurs morceaux (modules), chacun exécutant une tâche précise. Ces modules sont appelés des fonctions dont l'une est principale et dite programme principal (main). Lors de l'exécution, le programme principal est exécuté en premier. Les autres fonctions sont exécutées lorsqu'elles sont appelées. La programmation modulaire est justifiée par :

- La faculté de maintenance (détection facile des erreurs…).

- L'absence de répétition de séquences d'instructions (paramétrage de fonctions).

- Le partage et la réutilisation de modules (fonction).

I. STRUCTURE ET PROTOTYPE D'UNE FONCTION

I. 1 STRUCTURE D'UNE FONCTION

Définition : Une fonction est un ensemble d'instructions réalisant une tâche précise dans un programme

Elle est de la forme

```
type  nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
  [ déclarations de variables locales ]
   liste d'instructions
}
```

}

I. STRUCTURE ET PROTOTYPE D'UNE FONCTION

STRUCTURE D'UNE FONCTION

Une fonction doit être définie sous forme :

```
<Déclaration de la fonction>
{
<Déclaration des variables locales>
<Bloc d'instructions>
}
```

Syntaxe :

Void ou *Type_Résultat Nom_Fonction* **(***Void* ou *Type Param1*,…,*Type Param n*);

```
{
Corps de la fonction (instructions)
}
```

✓ La partie **déclaration de la fonction** indique le **type du résultat** retourné par la fonction **ou void** si elle ne retourne rien. Elle indique également ses **paramètres formels et leurs types** si elle en admet, **sinon** indique **void**.

Fonction

```
/* Routine de calcul du maximum */
int imax(int n, int m)
{
   int max;

   if (n>m)
         max = n;
   else
         max = m;
   return max;
}
```

Déclaration de la fonction

Variable locale

Valeur retournée par la fonction

PROTOTYPE D’UNE FONCTION

La déclaration de **variables locales** sert à déclarer les variables utilisées par la fonction et dont la **portée est uniquement cette fonction**.Le **corps d'une fonction** est constitué de **différentes instructions de C** exécutant la tâche accomplie par la fonction, **en plus** de l'instruction **return** si la fonction retourne un résultat. L'instruction ***Return*** **indique le résultat** retourné par la fonction s'il y en a **et constitue un point de sortie** de la fonction.

PROTOTYPE D'UNE FONCTION

➢La fonction **prototype** donne la règle d'usage de la fonction : nombre et type de paramètres ainsi que le type de la valeur retournée.

➢Les fonctions exigent la déclaration d'un prototype avant son utilisation:

➢***Syntaxe :***

***Void* ou *Type_Résultat Nom_Fonction* (*Void* ou *Type1*,…,*Typen*);**
On peut insérer les noms de paramètres mais ils n'ont aucune signification.

PROTOTYPE D'UNE FONCTION

Exemple :

```
/* Programme principal */
#include <stdio.h>
int imax(int,int);
main()
{   …   }

int imax(int n, int m)
{   …   }
```

Prototype de la fonction

La fonction est définie ici

I. APPEL DE FONCTIONS

Syntaxe :

- ***Si la fonction retourne un résultat et admet des paramètres***

  Variable = Nom_Fonction (*Paramètres effectifs*);

- ***Si la fonction retourne un résultat et n'admet pas de paramètres***

  Variable = Nom_Fonction ( );

- ***Si la fonction ne retourne rien et admet des paramètres***

  Nom_Fonction (*Paramètres effectifs*);

- ***Si la fonction ne retourne rien et n'admet pas de paramètres***

  Nom_Fonction ( );

DOMAINES D'EXISTENCE DE VARIABLES

En fonction de leur localisation (en tête du programme ou à l'intérieur d'une fonction) ou d'un qualificatif (static par exemple), une variable présente différentes caractéristiques :

Domaine d'existence de la variable	Signification
locale	Elle n'est référencée que dans la fonction où elle est déclarée.
globale	Placée en dehors des fonctions (généralement au début du programme avant le main), elle peut être accédée par toutes les fonctions.
Static	C'est une variable locale à une fonction mais qui garde sa valeur d'une invocation de la fonction à l'autre.
Externe	Elle est déclarée dans un module. Néanmoins, la mémoire qui lui est réservée lui sera allouée par un autre module compilé séparément.
Register	C'est une variable que le programmeur souhaiterait placer dans l'un des registres de la machine afin d'améliorer les performances.

Exercice

Écrire une fonction qui renvoie 1 si un nombre entier passé en paramètre est impair, 0 sinon. Son prototype est donc :

```
int estImpair(int nb);
```

Écrire également son programme de test main)

Les pointeurs

Institut Supérieur d'Informatique et Multimédia de Gabes(ISIMG)

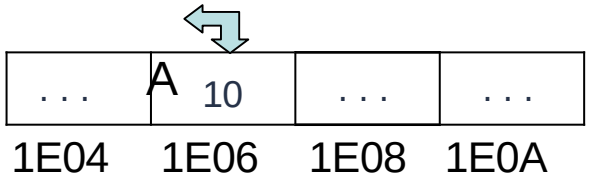
Plan

Modes d’adressage de variables. Définition d’un pointeur. Opérateurs de base. Opérations élémentaires. Pointeurs et tableaux. Pointeurs et chaînes de caractères. Pointeurs et enregistrements. Tableaux de pointeurs. Allocation dynamique de la mémoire. Libération de l’espace mémoire.

Mode d'adressage direct des variables

Adressage direct :

- Jusqu'à maintenant, nous avons surtout utilisé des variables pour stocker des informations.
- La valeur d'une variable se trouve à un endroit spécifique dans la mémoire de l'ordinateur.




```
short A;
A = 10;
```

- Le nom de la variable nous permet alors d'accéder directement à cette valeur.

Dans l'adressage direct, l'accès au contenu d'une variable se fait via le nom de la variable.

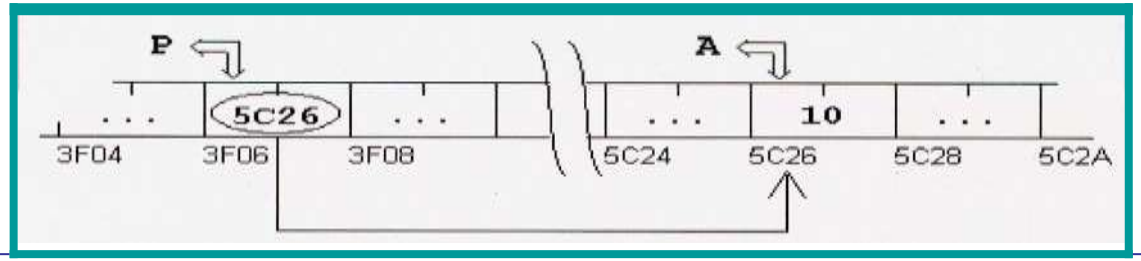
Mode d'adressage indirect des variables

Adressage indirect :

- Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale, disons P, appelée pointeur.
- Nous pouvons alors retrouver l'information de la variable A en passant par le pointeur P.

Dans l'adressage indirect, l'accès au contenu d'une variable se fait via un pointeur qui renferme l'adresse de la variable.

Exemple : Soit A une variable renfermant la valeur 10, et P un pointeur qui contient l'adresse de A.
En mémoire, A et P peuvent se présenter comme suit :



Définition d'un pointeur

Un pointeur est une variable spéciale pouvant contenir l'adresse d'une autre variable.

- En C, chaque pointeur est limité à un type de données. Il ne peut contenir que l'adresse d'une variable de ce type. Cela élimine plusieurs sources d'erreurs.

type de donnée * identificateur de variable pointeur;

Ex. : int * pNombre; pNombre désigne une variable pointeur pouvant contenir uniquement l'adresse d'une variable de type int.

Si pNombre contient l'adresse d'une variable entière A, on dira alors que pNombre pointe vers A.

- Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement en mémoire.
- Par contre, un pointeur peut contenir différentes adresses mais le nom d'une variable (pointeur ou non) reste toujours lié à la même adresse.

Comment obtenir l'adresse d'une variable ?

- Pour obtenir l'adresse d'une variable, on utilise l'opérateur & précédant le nom de la variable.

Syntaxe permettant d'obtenir l'adresse d'une variable :

& nom de la variable

Ex. :
```
int A;
int * pNombre = &A;
```

ou encore,

```
int A;
int * pNombre;
pNombre = &A;
```

pNombre désigne une variable pointeur initialisée à l'adresse de la variable A de type int.

Ex. :
```
int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);
```

scanf a besoin de l'adresse de chaque paramètre pour pouvoir lui attribuer une nouvelle valeur.

Note :

L'opérateur & ne peut pas être appliqué à des constantes ou des expressions.

Comment accéder au contenu d'une adresse ?

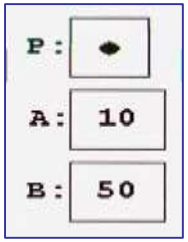
- Pour avoir accès au contenu d'une adresse, on utilise l'opérateur * précédant le nom du pointeur.

Syntaxe permettant d'avoir accès au contenu d'une adresse :

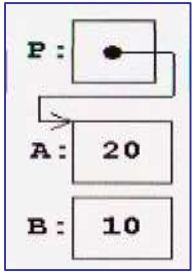
* nom du pointeur

Ex. :

```
int A = 10, B = 50;
int * P;
```




```
P = &A;
B = *P;
*P = 20;
```



*P et A désigne le même emplacement mémoire

Priorité des opérateurs * et &

- Ces 2 opérateurs ont la même priorité que les autres opérateurs unaires (!, ++, --).
- Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

```
Y = *P+1     ⇔ Y = X+1
*P = *P+10   ⇔ X = X+10

*P += 2      ⇔ X += 2
++*P         ⇔ ++X
(*P)++       ⇔ X++
```

Le pointeur NULL

- Pour indiquer qu'un pointeur pointe nulle part, on utilise l'identificateur NULL (On doit inclure stdio.h ou iostream.h).
- On peut aussi utiliser la valeur numérique 0 (zéro).

```
int * P = 0;

if (P == NULL) printf("P pointe nulle part");
```

Pointeurs et tableaux

- Chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

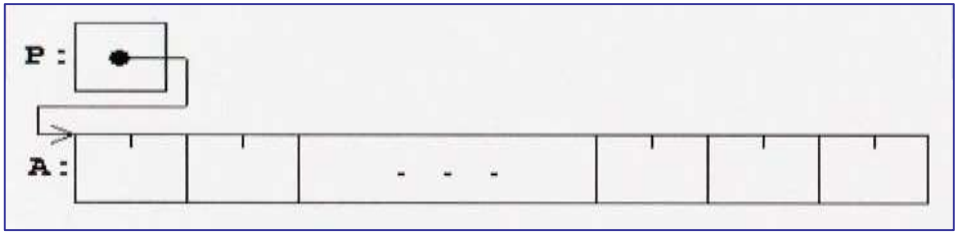
  Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de la première composante.

- &tableau[0] et tableau sont une seule et même adresse.

Le nom d'un tableau est un pointeur **constant** sur le premier élément du tableau.

int A[10];
int * P;

est équivalente à P = &A[0];



Adressage des composantes d'un tableau

- Si P pointe sur une composante quelconque d'un tableau, alors P + 1 pointe sur la composante suivante.

  P + i pointe sur la iième composante à droite de *P

  P - i pointe sur la iième com

  Ainsi, après l'instruction P = A;

`*(P+1)`	désigne le contenu de A[1]
`*(P+2)`	désigne le contenu de A[2]
...	...
`*(P+i)`	désigne le contenu de A[i]

- Incrémentation et décrémentation d'un pointeur

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

`P++;`	P pointe sur A[i+1]
`P+=n;`	P pointe sur A[i+n]
`P--;`	P pointe sur A[i-1]
`P-=n;`	P pointe sur A[i-n]

Ces opérateurs (+, -, ++, --, +=, -=) sont définis seulement à l'intérieur d'un tableau car on en peut pas présumer que 2 variables de même type sont stockées de façon contiguë en mémoire.

Résumons ...

Soit un tableau A de type quelconque et i un indice d'une composante de A,

`A`	désigne l'adresse de	`A[0]`
`A+i`	désigne l'adresse de	`A[i]`
`*(A+i)`	désigne le contenu de	`A[i]`

Si P = A, alors

`P`	pointe sur l'élément	`A[0]`
`P+i`	pointe sur l'élément	`A[i]`
`*(P+i)`	désigne le contenu de	`A[i]`

Différence entre un pointeur et le nom d'un tableau

Comme A représente l'adresse de A[0],

`*(A+1)` désigne le contenu de A[1]

`*(A+2)` désigne le contenu de A[2]

...

`*(A+i)` désigne le contenu de A[i]

- Un *pointeur* est une variable,
donc des opérations comme `P = A` ou `P++` sont permises.

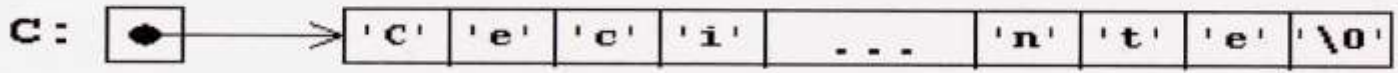
- Le *nom d'un tableau* est une constante,
donc des opérations comme `A = P` ou `A++` sont impossibles.

Pointeurs et chaînes de caractères

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.
- En plus, un pointeur vers une variable de type char peut aussi contenir l'adresse d'une chaîne de caractères constante et peut même être initialisé avec une telle adresse.

Exemple

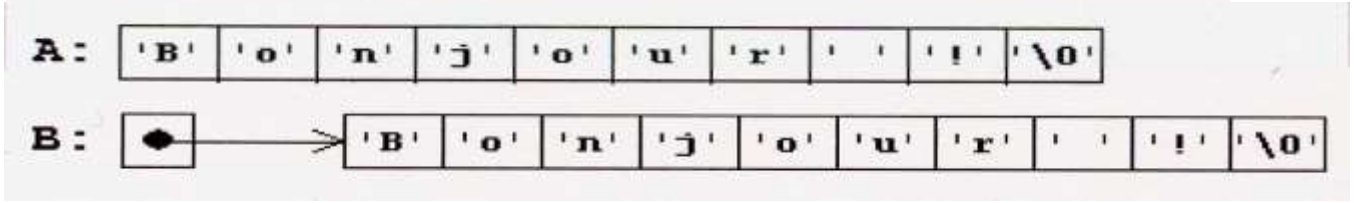
```
char *C;
C = "Ceci est une chaîne de caractères constante";
```




```
char *B = "Bonjour !";
```

Distinction entre un tableau et un pointeur vers une chaîne constante

```
char A[] = "Bonjour !";    /* un tableau  */
char *B  = "Bonjour !";    /* un pointeur */
```



A a exactement la grandeur pour contenir la chaîne de caractères et \0.
Les caractères peuvent être changés mais **A** va toujours pointer sur la même adresse en mémoire (pointeur constant).

Exemple

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;                  /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !";        /* IMPOSSIBLE -> ERREUR !!! */
```

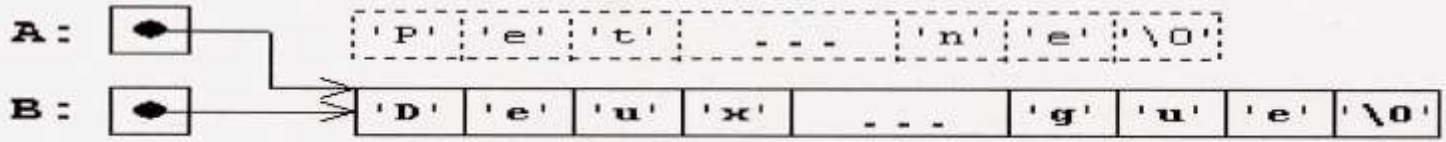
Distinction entre un tableau et un pointeur vers une chaîne constante

B pointe sur une chaîne de caractères constante. Le pointeur peut être modifié et pointer sur autre chose(la chaîne de caractères constante originale sera perdue). La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée (B[1] = 'o'; est illégal).

Exemple

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Un pointeur sur char a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur.

Avantage des pointeurs sur char

- Un pointeur vers char fait en sorte que nous n'avons pas besoin de connaître la longueur des chaînes de caractères grâce au symbole \0.

- Pour illustrer ceci, considérons une portion de code qui copie la chaîne CH2 vers CH1.

```
char * CH1;
char * CH2;
   . . .
```

1ière version :

```
int I;
I=0;
while ((CH1[I]=CH2[I]) != '\0')
      I++;
```

2ième version :

Un simple changement de notation nous donne ceci :

```
int I;
I=0;
while ((*(CH1+I)=*(CH2+I)) != '\0')
      I++;
```

Avantage des pointeurs sur char

Exploitons davantage le concept de pointeur.

```
while ((*CH1=*CH2) != '\0')
      {
       CH1++;
       CH2++;
      }
```

Un professionnel en C obtiendrait finalement :

```
while (*CH1++ = *CH2++)
       ;
```

Allocation statique de la mémoire

- Jusqu'à maintenant, la déclaration d'une variable entraîne automatiquement la réservation de l'espace mémoire nécessaire.
- Le nombre d'octets nécessaires était connu au temps de compilation; le compilateur calcule cette valeur à partir du type de données de la variable.

Exemples d'allocation statique de la mémoire

```
float A, B, C;              /* réservation de 12  octets */
short D[10][20];            /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                            /* réservation de 10  octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                            /* réservation de 40  octets */
```

Allocation dynamique de la mémoire

Problématique :

Souvent, nous devons travailler avec des données dont nous ne pouvons prévoir le nombre et la grandeur lors de l'écriture du programme.La taille des données est connue au temps d'exécution seulement.

But : Nous cherchons un moyen de réserver ou de libérer de l'espace mémoire au fur et à mesure que nous en avons besoin pendant l'exécution du programme.

Il faut éviter le gaspillage qui consiste à réserver l'espace maximal prévisible.

Exemples : La mémoire sera allouée au temps d'exécution.

```
char * P;
                    // P pointera vers une chaîne de caractères
                    // dont la longueur sera connue au temps d'exécution.
int * M[10];        // M permet de représenter une matrice de 10 lignes
                    // où le nombre de colonnes varie pour chaque ligne.
```

La fonction malloc et l'opérateur sizeof

- La fonction malloc de la bibliothèque stdlib nous aide à localiser et à réserver de la mémoire au cours de l'exécution d'un programme.

- La fonction malloc fournit l'adresse d'un bloc en mémoire disponible de N octets.

```
char * T = malloc(4000);
```

Cela fournit l'adresse d'un bloc de 4000 octets disponibles et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

Si nous voulons réserver de l'espace pour des données d'un type dont la grandeur varie d'une machine à l'autre, on peut se servir de sizeof pour connaître la grandeur effective afin de préserver la portabilité du programme.

La fonction malloc et l'opérateur sizeof

Exemple :

```
#include <stdio.h>
void main()
{
        short A[10]; char
        B[5][10];
        printf("%d%d%d%d%d%d", sizeof A, sizeof B,
                                sizeof 4.25,  sizeof
                                "Bonjour !", sizeof(float),
                                sizeof(double));
}
```

205081048

Exemple :

:Réserver de la mémoire pour X valeurs de type int où X est lue au clavier.

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

La fonction malloc et l'opérateur sizeof

Note :

S'il n'y a pas assez de mémoire pour satisfaire une requête, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande exit de stdlib et de renvoyer une valeur non nulle comme code d'erreur.

Exemple :

Lire 10 phrases au clavier et ranger le texte.
La longueur maximale d'une phrase est fixée à 500 caractères.

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

main()
{
 /* Déclarations */
 char INTRO[500];
 char *TEXTE[10];
 int I;
 /* Traitement */
 for (I=0; I<10; I++)
     {
```

La fonction malloc et l'opérateur sizeof

```
        gets(INTRO);
        /* Réservation de la mémoire */
        TEXTE[I] = malloc(strlen(INTRO)+1);
        /* S'il y a assez de mémoire, ... */
        if (TEXTE[I])
              /* copier la phrase à l'adresse */
              /* fournie par malloc, ...      */
              strcpy(TEXTE[I], INTRO);
        else
             {
              /* sinon quitter le programme */
              /* après un message d'erreur. */
              printf("ERREUR: Pas assez de mémoire \n");
              exit(-1);
             }
      }
  return 0;
}
```

Libération de l'espace mémoire

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de malloc, nous pouvons le libérer à l'aide de la fonction free de la librairie stdlib.

```
free(pointeur);
```

Pointe vers le bloc à libérer.

À éviter : Tenter de libérer de la mémoire avec free laquelle n'a pas été allouée par malloc.

Attention : La fonction free ne change pas le contenu du pointeur.

Il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était rattaché.

Note : Si la mémoire n'est pas libérée explicitement à l'aide de free, alors elle l'est automatiquement à la fin de l'exécution du programme.