Structures de Données en langage C

Abdellatif HAIR

Université Sultan Moulay Slimane

Faculté des Sciences et Techniques

B.P. 523, Béni-Mellal, MAROC

Chap1. GENERALITES ET RAPPELS

- LES STRUCTURES
- LES POINTEURS
- LES FONCTIONS
- EXEMPLE COMPLET

LES STRUCTURES

Définition d'une structure

- Un tableau permet de regrouper des éléments de même type, c'est-à-dire codés sur le même nombre de bit et de la même façon

- Il est généralement utile de pouvoir rassembler des éléments de type différents tels que des entiers et des chaînes de caractères

- Les structures permettent de remédier à cette lacune des tableaux, en regroupant des objets (des variables) au sein d'une entité repérée par un seul nom de variable

- Les objets contenus dans la structure sont appelés champs de la structure

LES STRUCTURES

Définition d'une structure

- Une structure possède un nom et est composée de plusieurs champs
- Chaque champ à son propre type et son propre nom
- Pour déclarer une structure on utilise le mot-clé struct

```
struct nomStructure {
        type1 champ1;
        ...
        typeN champN;
};
```

mot-clé struct

Le nom de la structure

champs de la structure

LES STRUCTURES

Définition d'une structure

- un exemple qui déclare une structure permettant de stocker un nombre complexe :

```
struct complexe {
    float reel; /* partie réelle */
    float imag; /* partie imaginaire */
};
```

- Le nom des champs répond aux critères des noms de variable
- Deux champs ne peuvent avoir le même nom
- Les données peuvent être de n'importe quel type sauf le type de la structure dans laquelle elles se trouvent

LES STRUCTURES

Définition d'une structure

- **Exemple :**

```
struct MaStructure {
    int Age;
    char Sexe;
    char Nom[12];
    float MoyenneScolaire;
    struct AutreStructure StructBis;
}; // en considérant que la structure AutreStructure est definie
```

LES STRUCTURES

Déclaration d'une variable structurée

- La définition d'une structure ne fait que donner l'allure de la structure et ne réserve donc pas d'espace mémoire pour une variable structurée

- Pour déclarer une (ou plusieurs) variable(s) structurée(s) après avoir définie la structure

```
struct Nom_Structure Nom_Variable_Structuree;
```

Nom_Structure représente le nom d'une structure (déjà déclarée)

Nom_Variable_Structuree est le nom que l'on donne à la variable structurée

LES STRUCTURES

Déclaration d'une variable structurée

- On peut déclarer plusieurs variables structurées en les séparant avec des virgules :

```
struct Nom_Structure Nom1, Nom2, Nom3...;
```

- Soit la structure *Personne* :

```
struct Personne{
        int Age;
        char Sexe;
};

struct Personne P1, P2, P3;
```

LES STRUCTURES

Accès aux champs d'une variable structurée

- Chaque variable de type structure possède des champs repérés avec des noms uniques

- Pour accéder aux champs d'une structure on utilise l'opérateur de champ (un simple point .) placé entre le nom de la variable structurée que l'on a définit et le nom du champ:

```
Nom_Variable.Nom_Champ;
```

- Soit P1 une variable de type struct Personne définie précédemment, on pourra écrire:

```
P1.Age = 18;
P1.Sexe = 'M';
```

LES STRUCTURES

Utilisation de typedef

- Le mot-clé typedef permet d'associer un nom à un type donné

- On l'utilise suivi de la déclaration d'un type puis du nom qui remplacera ce type

```
typedef struct {
                float reel; /* partie reelle */
                float imag; /* partie imaginaire */
                 } complexe;

complexe a={0,1}; /*déclaration d 'une variable a de type complexe*/
complexe b,c;
```

- Ceci permet de s'affranchir de l'emploi de struct à chaque utilisation d'une structure (il n'est pas alors nécessaire de donner un nom à la structure)

LES STRUCTURES

Tableaux de structures

- il est possible de créer un tableau ne contenant que des éléments du type d'une structure donnée
- créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable :

```
struct Nom_Structure Nom_Tableau[Nb_Elements];
```

- Chaque élément du tableau représente alors une structure du type que l'on a défini
- Le tableau suivant (Repertoire) pourra par exemple contenir 8 variables structurées de type struct Personne:

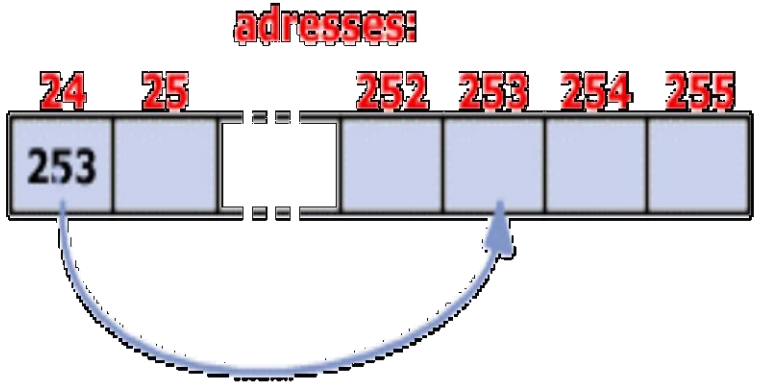
```
struct Personne Repertoire[8];
```

LES POINTEURS

Définition d'un pointeur

- Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné

le pointeur stocké à l'adresse 24 pointe vers une variable stockée à l'adresse 253 (les valeurs sont bien évidemment arbitraires).



- On parle de pointeur sur int ou de pointeur sur double…
- Les pointeurs permettent de définir des structures dynamiques, c'est-à-dire qui évolue au cours du temps

LES POINTEURS

Déclaration d'un pointeur

- Pour déclarer un pointeur :
  - on doit écrire le type d'objet sur lequel il pointera
  - suivi du caractère * pour préciser que c'est un pointeur
  - puis enfin son nom

- Exemple :

```
double *p;
char *c1; *c2;
float *p1, p2;
int **q;
```

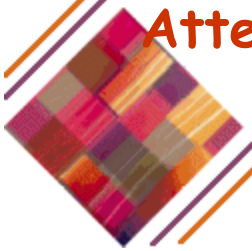
p est un pointeur sur un double

c1 et c2 sont deux pointeurs sur un char

p1 est pointeur sur un float et p2 est une variable de type float

q est un pointeur sur un pointeur sur un int

Attention : dans la définition d'un pointeur, le caractère * est rattaché au nom qui le suit et non pas au type



LES POINTEURS

Initialisation d'un pointeur

- Après avoir déclaré un pointeur, il faut l'initialiser : sinon le pointeur contient ce que la case contenait avant, c'est-à-dire n'importe quel nombre

- Pour initialiser un pointeur, il faut utiliser l'opérateur d'affectation '=' suivi de l'opérateur d'adresse '&' auquel est accollé un nom de variable :

```
Nom_du_pointeur = &nom_de_la_variable_pointee;
```

- Exemple

```
int *p1, a = 2;
char *p2, c='A';
p1=&a; p2=&c;
```

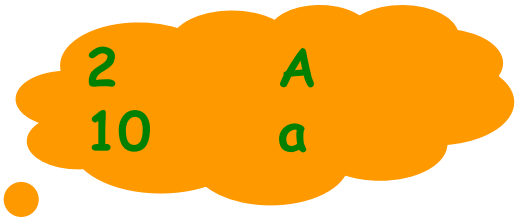
LES POINTEURS

Accès à une variable pointée

◆ Après avoir déclaré et initialisé un pointeur, il est possible d'accéder au contenu de l'adresse mémoire pointée par le pointeur grâce à l'opérateur '*'.

Syntaxe : *pointeur

```
#include<stdio.h>
 main() {
        int *p1, a = 2;
        char *p2, c='A';
        p1=&a; p2=&c;
        printf("p1    = %d      ", *p1);
        printf("p2    = %c \n", *p2);
        *p1 = 10; *p2 = 'a';
        printf("p1    = %d      ", *p1);
        printf("p2    = %c \n", *p2);
        system("pause");
}
```



LES POINTEURS

calculs sur les pointeurs

- On peut récupérer l'adresse de n'importe quel objet
  - il est possible d'obtenir l'adresse d'un élément d'un tableau

```
double a[20];
double *p;
p = &(a[10]);
```

Adresse du onzième élément

  - Par convention, le nom d'un tableau est une constante égale à l'adresse du premier élément du tableau

```
p = &a[0];
p = a;
```

Les deux instructions sont équivalentes

LES POINTEURS

calculs sur les pointeurs

- Tous les opérateurs classiques d'addition et de soustraction sont utilisables en particulier les opérateurs d'incrémentation

```
double a[20];
double * p;
p = &(a[10]);
p = p - 8;
```

p pointe à la fin sur le troisième élément du tableau a (a[2])

Exercice 1 : Compter le nbre de caractères d'une chaîne de caractères. Une chaîne de caractères toujours par le caractère ('\0')

```
char *p = str;
int NbCar = 0;
while ( *p != '\0')  { p++; NbCar++; }
```

LES POINTEURS

➔ calculs sur les pointeurs

- les calculs sur pointeurs et l'utilisation de l'opérateur [ ] d'accès à un élément d'un tableau peuvent être considérés comme équivalent

```
double Tab[100]
Tab[45] = 123.456;
*(Tab + 45) = 123.456;
```

Les deux instructions sont équivalentes

- Ceci est tellement vrai qu'on peut même utiliser un pointeur directement comme un tableau

p[i] ⟷ *(p + i)

LES POINTEURS

calculs sur les pointeurs

```
#include<stdio.h>
main() {
int Toto[10];
int *p;
int Indice;
  for (Indice = 0; Indice <= 9; Indice++) Toto[Indice] = 2*Indice+1;
  p=Toto;
  for (Indice = 0; Indice <= 9; Indice++)
  printf("adresse %d élément = %d contenu = %d \n", Indice+1, p+Indice, p[Indice]) ;
  system("pause");
}
```

```
adresse 1 élément = 229368 contenu = 1
adresse 2 élément = 229372 contenu = 3
adresse 3 élément = 229376 contenu = 5
adresse 4 élément = 229380 contenu = 7
…..
```

LES FONCTIONS

Définition

- Les fonctions sont les éléments de base d'un programme C

- Un programme C bien conçu contient en général de nombreuses petites fonctions plutôt que peu de grosses fonctions

- On appelle *fonction* un sous-programme (une série d'instructions) qui permet d'effectuer un ensemble d'instructions par simple appel de la fonction dans le corps du programme principal

- Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, cela permet une simplicité du code et donc une taille de programme minimale

LES FONCTIONS

Déclaration d'une fonction

- Avant d'être utilisée, une fonction doit être définie car pour l'appeler dans le corps du programme il faut que le compilateur la connaisse

- La définition d'une fonction s'appelle "*déclaration*"

- Pour définir une fonction, il faut donner le type de la valeur retourné, son nom, la liste de ses paramètres(arguments) et enfin le corps de la fonction (instructions qu'elle contient) :

```
type_valeur_retourne Nom_Fonction (type1 argument1,
                                   type2 argument2, ...)
        {
        liste d'instructions
        }
```

LES FONCTIONS

Type de résultat retourné

- Le mot-clé **return** est suivi d'une expression. La valeur de cette expression sera le résultat de la fonction

- Il est possible d'utiliser **return** n'importe où et plusieurs fois dans une fonction, et ce sera toutefois la première instruction **return** rencontrée qui provoquera la fin de la fonction

- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé **void**

LES FONCTIONS

Paramètres d'une fonction

- Le passage d'arguments à une fonction se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la fonction

- Il n'y a pas théoriquement de limite au nombre de paramètres (ou arguments) que peut prendre une fonction

- S'il n'y a pas d'arguments, les parenthèses doivent rester présentes

LES FONCTIONS

Appel d'une fonction

- La fonction ne s'exécutera jamais tant que l'on ne fait pas appel à elle quelque part dans le programme

- Pour appeler une fonction, il suffit de mettre son nom suivi de la liste des valeurs à donner aux paramètres entre parenthèses

- Un appel de fonction est considéré comme une expression du type du résultat de la fonction et la valeur de cette expression est le résultat de l'exécution de la fonction

LES FONCTIONS

Appel d'une fonction

```
#include <stdio.h>
float moyenne (float a, float b) { return (a + b)/2;}
float c, d, m;
 main() {
      c=30; d=50;
      m=moyenne(c,d);
            /* m contiendra la valeur moyenne de 30 et 50*/
      printf(" la moyenne de %f et %f = %f \n ", c, d, m);
      system (" PAUSE ");
 }
```

LES FONCTIONS

Corps de la fonction

- On appelle *corps de la fonction* le bloc d'instructions qui commence après l'accolade ouvrante qui suit la liste des paramètres et qui se termine par l'accolade fermante correspondante

- le corps de la fonction peut débuter par une série de définitions de variables. Ces variables sont appelées variables locales

- Les fonctions C sont réentrantes. Cela signifie que les fonctions peuvent faire appels à elles-mêmes, on parle alors des fonctions récursives

LES FONCTIONS

Corps de la fonction

La suite de Fibonacci se définit récursivement de la manière suivante :

$$Fib(0)=1 \quad Fib(1)=1$$

$$Fib(n)= Fib(n-1)+Fib(n-2)$$

```
unsigned long Fib(int n) {
unsigned long resultat;
    if (n<2) resultat = 1L; /* (n<2) équivalent (n=0)||(n=1)*/
    else resultat = Fib(n-1) + Fib(n-2);
    return resultat;
}  /*le nombre 1L est représenté comme nombre de type long*/
```

LES FONCTIONS

Exercice: Ecrire un programme C qui permet d'afficher par bloc (10 lignes) les 40 premiers nombres de Fibonacci.

```
#include <stdio.h>
#define ligne 10
unsigned long int Fib(int n)  {
    unsigned long int resultat;
    if (n<2) resultat=1L;  else resultat =Fib(n-1)+Fib(n-2);
    return resultat; }
int i; unsigned int long nombre;
main()    {
  i=0;
  while (i<=39)  {   nombre=Fib(i);
                     printf("Fib(%d) = %ld \n",i, nombre);
                     i++;
                     if (i%ligne==0) getchar(); };
   system (" PAUSE ");
}
```

LES FONCTIONS

Passage par valeur (paramètre)

- Les arguments (paramètres) peuvent être considérés comme des variables locales à la fonction
- Lors de l'appel de la fonction, la valeur de chaque paramètre est recopiée dans un nouvel espace mémoire réservé pour ce paramètre
- Une fonction peut donc localement modifier le contenu de ses paramètres
- Les valeurs utilisées lors de l'appel ne seront absolument pas modifiées

LES FONCTIONS

passage par valeur

```
#include<stdio.h>
Echange(int p1, int p2) {
int t ;
t=p1; p1=p2; p2=t ;
}
main() {
int i=100, j=2
printf("valeurs avant l'appel de la fonction i=%d  j=%d \n", i,j);
Echange(i,j);
printf(" valeurs après l'appel de la fonction i=%d  j=%d \n", i,j);
system("PAUSE");
}
```

valeurs avant l'appel de la fonction i=100 j=200

valeurs après l'appel de la fonction i=100 j=200

LES FONCTIONS

➔ passage par adresse (référence)

- Le passage des paramètres par valeur ne permet donc pas de modifier une variable de la fonction appelante
- Grâce aux pointeurs, il est possible d'agir à distance sur les variables
  - Il suffit pour cela d'envoyer non pas la valeur d'une variable mais son adresse en mémoire (c'est à dire un pointeur sur cette variable)
  - La fonction appelée peut alors modifier le contenu de cet emplacement mémoire (et donc le contenu de la variable elle-même)

LES FONCTIONS

Passage par adresse (référence)

```
#include<stdio.h>
Echange(int *p1, int *p2) {
int t;
t= *p1; *p1=*p2; *p2=t;
}
main() {
int i, j;
printf("Donner le premier entier i : "); scanf("%d", &i);
printf("Donner le deuxième entier j : "); scanf("%d", &j);
printf("valeurs avant échange i=%d  j=%d \n", i,j);
Echange(&i,&j);
printf("valeurs après échange i=%d  j=%d \n", i,j);
system("PAUSE");
}
```

LES FONCTIONS

Pointeur vers une fonction

Comme pour les variables, un pointeur vers une fonction doit être déclaré

```
Type (*ptr_vers_fonction)(liste_d_arguments) ;
```

Cela signifie qu'on déclare un pointeur appelé ptr_vers_fonction qui renvoie un résultat type et admet une liste d' arguments

```
int (*fonc1)(int x);

void (*fonc2)(double y, double z);

char (*fonc3)(char *p[]);
```

LES FONCTIONS

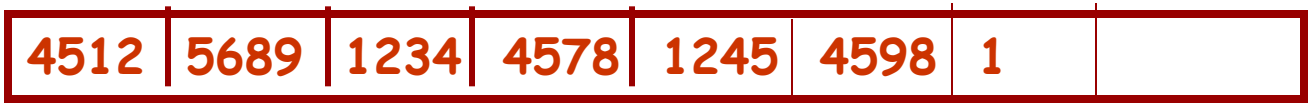
Pointeur vers une fonction

```
#include<stdio.h>

float carre(int n){ return n*n;}
float racine(int n){ return sqrt(n);}
float teste(float (*f)(int),int n) { return f(n);}

main(){
float (*fonc)(int);
fonc=carre;
x=teste(fonc,2);
printf("%f carre \n",x);
x=teste(racine,2);
printf("%f racine \n",x);
system("pause");
}
```

EXEMPLE COMPLET

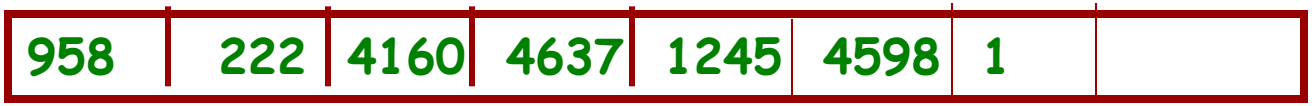
Ecrire un programme qui permet de faire la somme de deux grands nombres entiers. On doit lire une C.C. représentant un grand nombre entier et le charger par paquets de 4 chiffres dans un tableau

1 4598 1245 4578 1234 5689 4512



59 2925 4532 6446

la somme est :



1 4598 1245 4637 4160 0222 0958

EXEMPLE COMPLET

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

typedef int tableau[30];

int N1, N2, i, taille;
char ch1[30], ch2[30];
tableau tab1, tab2, tab_som;
```

EXEMPLE COMPLET

```
Convertir(tableau tab, char *chaine, int *NL) {

  int Long, reste, i, val, N;
  char ch4[4], ch_reste[4],*ch;

  Long= strlen(chaine);
  N=Long/4;
  for (i=0; i<N; i++)  {   val=Long - 4*(i+1);
                           ch=&chaine[val];
                           strncpy(ch4, ch, 4);
                           tab[i]=atoi(ch4); };
  reste=Long%4;
  if (reste!=0)    {  N++;
                      ch=&chaine[0];
                     strncpy(ch_reste, ch, reste);
                      tab[N-1]=atoi(ch_reste); };
  (*NL)=N;
};
```

EXEMPLE COMPLET

```
somme(int N, int M, int *taille, tableau t1, tableau t2, tableau t3) {
int s,ret, i;

(*taille)=M; ret=0;

for (i=0; i<=N-1;i++) {  s=t1[i] + t2[i] + ret;
                         t3[i]= s%10000;
                         ret= s/10000; };

for (i=N; i<=M-1; i++) {  s=t2[i] + ret;
                          t3[i]= s%10000;
                          ret= s/10000; };

if (ret !=0)  { t3[i]=ret;
                (*taille) = (*taille)+1; };
}
```

EXEMPLE COMPLET

```
afficher(int taille, tableau t) {
  char ch[4];
  int i;

 sprintf(ch,"%d", t[taille-1]);
 printf("%s",ch);
 for (i=taille-2; i>=0; i--)
    {  if (t[i]<10) sprintf(ch,"000%d", t[i]);
       else if (t[i]<=99) sprintf(ch,"00%d", t[i]);
            else if (t[i]<=999) sprintf(ch,"0%d", t[i]);
                 else sprintf(ch,"%d", t[i]);
     printf("%s",ch);
    };
};
```

EXEMPLE COMPLET

```
main() {
printf("Donner un très grand nombre : "); gets(ch1);
printf("Donner un très grand nombre : "); gets(ch2);

Convertir(tab1, ch1, &N1);
Convertir(tab2, ch2, &N2);

if (N1<=N2) somme(N1, N2, &taille, tab1, tab2, tab_som);
else somme(N2, N1, &taille, tab2, tab1, tab_som);

printf(" la somme   =\n");

afficher(taille, tab_som);

system("pause");
}
```