LES ÉNUMÉRATIONS ET LES STRUCTURES

Algorithmique et programmation

Partie 1: Les énumérations

Introduction

- Une énumération est un type de données particulier, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs.
- Une énumération se définit à l'aide du mot-clé enum suivi du nom de l'énumération et de ses membres.
- `enum naturel { ZERO, UN, DEUX, TROIS, QUATRE, CINQ };`
- La particularité de cette définition est qu'elle crée en vérité deux choses : un type dit « énuméré » enum naturel et des constantes dites « énumérées » ZERO, UN, DEUX…
- Quant aux constantes énumérées, il s'agit de constantes entières.
- A défaut de préciser leur valeur, chaque constante énumérée se voit attribuer la valeur de celle qui la précède augmentée de un, sachant que la première constante est mise à zéro. Dans notre cas donc, la constante ZERO vaut zéro, la constante UN un et ainsi de suite jusque cinq.

Exemple

```c
#include<stdio.h>
#include<stdlib.h>
enum naturel { ZERO, UN, DEUX, TROIS, QUATRE, CINQ };
main()
{
    enum naturel n = ZERO;
    printf("n = %d.\n", (int)n);
    printf("UN = %d.\n", UN);
    system("pause");
}
```

Valeurs des types énumérés

- Toutefois, il est possible de préciser la valeur de certaines constantes (voire de toutes les constantes) à l'aide d'une affectation.
- ```
  enum naturel { DIX = 10, ONZE, DOUZE, TREIZE,
  QUATORZE, QUINZE };
  ```
- Notez que le code ci-dessous est parfaitement équivalent.
- ```
  enum naturel { DIX = 10, ONZE = 11, DOUZE = 12, TREIZE
  = 13, QUATORZE = 14, QUINZE = 15 };
  ```

Partie 2: Les structures

Introduction

- Une **structure** est une suite finie d'objets de types différents.

Par exemples :

- structure : voiture,
  - champs : marque, couleur, année,...
- structure : Etudiant,
  - champs : nom, prenom, cin, note1, note2, ...
- Chaque élément de la **structure**, appelé **champ**, est désigné par un **identificateur et un type**.

Déclaration

- Déclaration avec le nom du type structure
  - On déclare d'abord le type
  - On déclare après les variables
  - Si on veut déclarer d'autres variable de ce type, il suffit de se référer au type **struct modele**.

Première manière

```
struct modele
{
      Type_1  champ_1;
      Type_2  champ_2;
      …
      Type_n  champ_n;
} ;
struct    modele  var1;
```

Déclaration

- Déclarer en même temps le type et les variables. Si on veut déclarer d'autres variable de ce type, il suffit de se référer au type **struct modele**.

```
2ème manière
struct    modele
{
      Type_1  champ_1;
      Type_2  champ_2;
      …
      Type_n  champ_n;
} var1  , var2 , ….;
```

Déclaration

- Déclaration directe sans le nom du type structure. Le désavantage de cette manière de faire est qu'il n'est pas flexible pour la déclaration de nouvelles variables de même type.

```
3ème manière
struct
{
      Type_1  champ_1;
      Type_2  champ_2;
      …
      Type_n  champ_n;
} var1  , var2 , ….;
```

Accès à un champs de la structure

- On accède aux différents **champ** (membre) d'une structure grâce à l'opérateur membre de structure, point '•'.
- Exemple : Le programme suivant définit une structure complexe composée de deux champs de type **float** ; initialise les champs et les affichent.

```
struct complexe
{
    float R;
    float I;
};

main()
{
    struct complexe Z;
    //lecture
    printf("saisir la valeur imaginaire et la valeur réelle du nombre complexe: \n");
    scanf("%f%f", &Z.R, &Z.I);
    //affichage
    printf("votre nombre complexe = %.2f + i %.2f \n", Z.R, Z.I);
    system("pause");
}
```

Utilisation globale d'une structure

- Il est possible d'affecter à une structure le contenu d'une structure définie à partir du même **modèle**.
- Par exemple, si deux variables **Z1** et **Z2** ont été déclarées suivant le modèle **complexe** défini dans l'exemple précédent.
  - **struct** complexe **Z1**, **Z2** ;
  - Donc, nous pouvons écrire :

    Z1 = Z2;
  - Une telle affectation globale remplace avantageusement :

    Z1.R = Z2.R ;

    Z1.I = Z2.I ;

Utilisation globale d'une structure

```c
#include<stdio.h>
#include<stdlib.h>
struct complexe
{
    float R;
    float I;
};

main()
{
    struct complexe Z;
    struct complexe Z1 = {2.0, 4.5}; //initialisation globale

    Z=Z1;

    //affichage
    printf("votre nombre complexe = %.2f + i %.2f \n", Z.R, Z.I);
    system("pause");
}
```

Définition de synonymes : **typedef**

- **typedef** permet de définir des synonymes.
- **typedef** s'applique à tous les types et pas seulement aux structures.
- Commencerons par l'introduire sur quelques exemples :
- La déclaration : `typedef int entier ;`
  - *Signifie que **entier** est "synonyme" de **int**, de sorte que les déclarations suivantes sont équivalentes :*
    `int n, p ;` et `entier n, p ;`
  - *De même :* *`typedef int *ptr ;`*
  - *Signifie que **ptr** est synonyme de **int ***. Les déclarations suivante sont équivalentes :*
    `int *p1, *p2 ;`
    et `ptr p1, p2 ;`
  - *De même:* `typedef float vecteur[5] ;`
  - *Les déclarations suivante sont équivalentes :*
    `float v[5], w[5] ;`
    et `vecteur v, w ;`

typedef: application aux structures

- En faisant usage de **typedef** les déclarations des enregistrement **Z1** et **Z2** du paragraphe précèdent peuvent être réalisées comme suit :

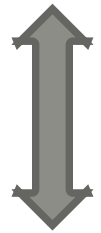
```c
struct complexe
{
    float R;
    float I;
};
typedef struct complexe COMPLEX;
main()
{
    COMPLEX Z1, Z2;
    //lecture
    scanf("%f%f", &Z1.R, &Z1.I);
    //...
    system("pause");
}
```

```c
typedef struct
{
    float R;
    float I;
}comp;

main()
{
    comp Z1, Z2;
    //lecture
    scanf("%f%f", &Z1.R, &Z1.I);
    //...
    system("pause");
}
```

- On vous suggère fortement d'utiliser **typedef.**

Pointeur vers le type structure

```
typedef struct
{
    float R;
    float I;
}comp;

main()
{
    comp Z1= {2.0, 4.5};
    comp *Z2;
    Z2 = &Z1;
    //affichage
    printf("votre nombre complexe = %.2f + i %.2f \n", Z2->R, Z2->I);
    system("pause");
}
```



```
printf("votre nombre complexe = %.2f + i %.2f \n", (*Z2).R, (*Z2).I);
```

Pointeur vers le type structure

- Déclaration d'un pointeur sur une structure

  *comp * Z2 ;*

- Initialisation d'un pointeur sur une structure

  Z2 = &Z1 ;

- Accéder au contenu de la valeur pointée

  *(*Z2).R <==> Z2->R*

  *(*Z2).I <==> Z2->I*

- on préfèrera utiliser **->**

Exemple

```
typedef struct
{
    float R;
    float I;
}complex;
main()
{
    complex *Z;
    Z = (complex*)malloc(sizeof(complex));
    printf("donner votre nombre complexe");
    scanf("%f%f", &Z->R, &Z->I);
    //affichage
    printf("votre nombre complexe = %.2f + i %.2f \n", Z->R, Z->I);
    system("pause");
}
```

Tableau statique de structures

- Supposons qu'on a besoin de manipuler des informations "**structurées** " (des personnes, des étudiants, des employés, etc. ...) dont les traitements prévus sont :
  - *le tri, la recherche d'un élément*
  - *l'affichage*
  - *les statistiques*
  - *la création de nouveaux fichiers*
- Si le nombre de données est raisonnable (exemple : 500 à 600 personnes à traiter, 400 étudiants à calculer leurs notes , etc. ...), on peut utiliser un tableau des structures.

Exemple

- Un tableau de points (par exemple : une courbe). Chaque élément (point(x,y)) du tableau est de type structure nommé "**point**" qui comporte les champs suivants :

  float x et float y

- On a par exemple plus de 100 points ou moins à traiter.

```
#define  NBR_PT  100
typedef struct
    {  float x ;
       float y ;
    } point;
point Droit[NBR_PT] ;
```

Structures comportant autres structures

- Supposons qu'on dispose d'une structure ayant parmi ses champs des champs de type structures. Plus précisément si on a les deux déclarations suivantes:
- DS_Etudiant Etudiant={"Dupont","Jules",{16,18},{13.5,17}};

```
typedef struct{
        float math;
        float info;
   } Note ;

typedef struct {
      char nom[20];
      char prenom[30];
      Note NoteDS1;
      Note NoteDS2;
   } DS_Etudiant ;
```

Exemple

```
#include <stdio.h>
typedef float reel;
int main()
{
 DS_Etudiant Etudiant={"Dupont","Jules",{16,18},{13.5,17}};
 reel moyM, moyI;
 printf("Nom= %s\n",Etudiant.nom);
 printf("Prenom= %s\n",Etudiant.prenom);
 printf("Etudiant.NoteDS1.math= %lg\n",Etudiant.NoteDS1.math);
 printf("Etudiant.NoteDS2.math= %lg\n",Etudiant.NoteDS2.math);
 printf("Etudiant.NoteDS1.info= %lg\n",Etudiant.NoteDS1.info);
 printf("Etudiant.NoteDS2.info= %lg\n",Etudiant.NoteDS2.info);
 moyM=(Etudiant.NoteDS1.math+Etudiant.NoteDS2.math)/2;
 printf("moy math= %lg\n", moyM);
 moyI=(Etudiant.NoteDS1.info+Etudiant.NoteDS2.info)/2;
 printf("moy info= %lg\n", moyI);
}
```

Transmission de la valeur d'une structure

- Aucun problème particulier ne se pose. Il s'agit simplement d'appliquer ce que nous connaissons déjà. Voici un exemple simple:

```
typedef struct
{
    float R;
    float I;
}complex;

void affiche (complex C)
{
    printf("votre nombre complexe = %.2f + i %.2f \n", C.R, C.I);
}
main()
{
    complex C = {2.0, 4.5};
    affiche(C);

    system("pause");
}
```

Transmission de l'adresse d'une structure : l'operateur ->

- Cherchons à modifier notre programme pour que la fonction **fct** reçoive l'adresse d'une structure et non plus sa valeur.
- L'appel de **fct** devra donc se présenter sous la forme :

  y=fct(&EtudiantA) ; (y de type float)

- Cela signifie que son en-tête sera de la forme :

  float fct(struct Note *B)

- Et sa déclaration sera de la forme :

  float fct(struct Note *) ;

- Voici ce que peut devenir notre précèdent exemple en employant l'opérateur **- >**

Exemple

```c
typedef struct
{
    float R;
    float I;
}complex;

void affiche (complex *C)
{
    printf("votre nombre complexe = %.2f + i %.2f \n", C->R, C->I);
}
main()
{
    complex C1 = {2.0, 4.5};
    //affiche(&C1)
    complex *C2 = &C1;
    affiche(C2);

    system("pause");
}
```

Exercice : polynôme comme tableau de structure

- Un polynôme de la variable x est une somme de monômes de la variable x.

```
typedef struct
{
        int D ;
        float C ;
} monome ;
```

1- Ecrire une fonction de prototype : void saisir_monome(monome *pp) qui permet de saisir un monôme.

2- Ecrire une fonction de prototype : monome *allocation_polynome(int nb) qui fait l'allocation d'un polynôme

3- Ecrire une fonction de prototype : void saisir_polynome(monome *P, int nb) qui permet de saisir un polynôme de nb monômes.

4- Ecrire une fonction de prototype : void afficher_polynome(monome *P, int nb) qui permet d'afficher un polynôme.

5- Ecrire une fonction récursive de prototype : float eval_polynomeR(monome *P, int nb, float x) qui permet d'évaluer P au point x.