Chapitre 6 : Les types personnalisés

1. Introduction

En programmation, chaque donnée manipulée doit avoir son propre type. Cela permet au compilateur de valider les valeurs et les opérations à appliquer. Cela aide le programmeur à découvrir et à éviter les erreurs. Il existe plusieurs types prédéfinis dans le langage de programmation. Comme les nombres entiers et les caractères. Le programmeur peut également définir ses propres types, dérivés des types de base. Tels que les tableaux, dont nous avons parlé dans le chapitre précédent, les énumérations, les enregistrements et d'autres types.

2. Enumérations

2.1. Définition

Le type **enum** : un type de données qui est défini par le programmeur. C'est une liste ordonnée de valeurs constantes, définies en leur donnant des noms. Ce qui n'a de sens que pour le programmeur, ce qui lui facilite la mémorisation et la compréhension du programme. Le type **enum** est utilisé pour définir un nouveau type qui ne contient que ces valeurs.

2.2. Déclaration

```
enum type_name {const_name1, ...} var_list ;
```

Le mot enum est un mot réservé utilisé pour définir un ensemble de constantes de type int.

- type_name est le nom que le programmeur donne à ce groupe. Il doit s'agir d'un identifiant valide (facultatif).
- nom_const noms de constantes faisant référence aux éléments de l'ensemble. Cela n'a de sens que pour le programmeur. Ses valeurs peuvent être définies.
- var_list est une liste de variables de ce type (facultatif).

Pour C, ces noms sont des constantes de type int, dont les valeurs sont 0, 1, ... D'autres valeurs peuvent également lui être affectées comme suit :

```
enum type_name {const_name= val1, ...}var_list ;
```

- val est un entier. Si val n'existe pas, sa valeur est la valeur de la constante qui le précède dans la liste +1. Pour la première constante entière de la liste, sa valeur par défaut est 0.

Bien que les noms de variable et le nom d'énumération soient facultatifs, l'un des deux doit apparaître.

Exemple

```
enum Days {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
Days d;
```

2.3. Utilisation

Les constantes fournissent des noms plus faciles à mémoriser lors de la programmation que des nombres, et elles définissent l'ensemble de valeurs acceptable. Cela évite au programmeur de faire une erreur lors de la programmation, car il n'est pas possible d'affecter une valeur en dehors de l'ensemble à une variable **enum** (C++).

Vous pouvez utiliser des opérations de comparaison telles que >, <, =, ≠. Ce type peut également être utilisé avec la directive selon (switch).

2.4. Exemples

```
enum Sexe {Male, Femelle };
enum Mois {Janv, Fevr, Mars, Avr, Mai, Juin, Juil, Aout, Sept, Oct, Nov, Dec};
enum Days {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
Days d;
d=Tuesday ;
switch(d)
{
```

```
   case Friday: printf("Weekend\n") ;   break;
   case Saturday: printf("de 08:00 à 12:00\n") ;   break;
   default : printf("de 08:00 à 16:00\n") ;
}
```

3. Les enregistrements ou structures

3.1. Définition

Une structure, également appelée enregistrement (en anglais record): est un type composé, représentant un ensemble d'éléments nommés, auxquels on peut accéder par leur nom. Chaque élément est appelé un champ. Ces champs peuvent être de n'importe quelle type. La structure est utilisée pour regrouper les variables en un seul enregistrement.

3.2. déclaration

structure_variables : **struct** field_names :type … **finStruct**	**struct** struct_name { type field_names ; … }structure_variables ;

où struct est un mot réservé en C.

- struct_name : un nom pour la structure (facultatif) et doit être un identifiant valide.
- type field_names : Déclare les données (champs) qui composent la structure. Vous pouvez identifier les éléments dans la structure en nommant le type, suivi d'un ou plusieurs noms de champs (séparés par ","). des variables de type Différent sont saisies avec un point-virgule ";".
- variables_structures : noms des variables de type structure, également facultatives.

Bien que les noms de variable et le nom de structure soient facultatifs, l'un des deux doit apparaître.

Exemple

e1, e2 : **struct** nom : chaine de caractères bac : réel **finStruct**	**struct** Etudiant { char nom[20] ; float bac; }e1, e2;

Ici les deux variables e1 et e2 de type struct etudiant sont déclarées. Chaque variable contient deux champs : nom de type chaîne et bac de type nombre réel.

La déclaration des variables peut être retardée comme ceci

type Etudiant = **struct**
nom : chaine de caractères
bac : réel
finStruct
var
e1, e2 : Etudiant
	struct Etudiant {
	char nom[20] ;
	float bac;
	};
	
	struct Etudiant e1, e2 ;

Le ";" est requis après } , et **struct** doit être mentionne avant le nom de la structure en C. mais elle n'est requise en C++.

En C, il est préférable d'utiliser **typedef** pour déclarer un type struct, avant de déclarer des variables. Ainsi l'exemple précédent devient :

type Etudiant = **struct** nom : chaine de caractères bac : réel **finStruct** **var** e1, e2 : Etudiant	**typedef struct** { char nom[20] ; float bac ; } Etudiant ; Etudiant e1, e2 ;

Le nom de la structure n'existe pas, et **Etudiant** est le nouveau nom de la structure.

Les champs peuvent également être de type structure, à condition qu'ils soient définis avant eux.

Étant donné que l'étendue du champ est limitée à l'enregistrement auquel il appartient, vous n'avez pas à vous soucier d'un conflit de noms entre les noms de champ et d'autres variables.

3.3. Représentation

Lors de la définition d'un type de struct, la mémoire n'est pas réservée tant que la variable d'un type struct n'est pas déclarée. La structure en mémoire est représentée par des variables adjacentes. Par exemple, lors de la définition du type Etudiant, aucune mémoire n'est allouée aux champs Nom et BAC, mais de la mémoire leur est allouée lors de la création des enregistrements e1 et e2. La figure suivante représente les enregistrements e1 et e2.

e1		e2	
nom	bac	nom	bac
Ahmed	13.41	Souad	12.50

La taille d'une structure est la somme des tailles de ses champs constitutifs.

3.4. Initialisation

En C, des valeurs initiales peuvent être spécifiées pour tous les éléments de la structure au sein de deux accolades { et } lors de leur déclaration. Les valeurs sont séparées par la virgule « , », et ces valeurs doivent être du même type, ordre et nombre de champs.

Exemple

Etudiant e1= { "Ahmed", 13.41} ;

3.5. Utilisation

Le symbole "." point est utilisé pour accéder aux éléments de la structure. Cela se fait en utilisant le nom de la variable de type Structure, suivi du point, puis du nom d'un de ses champs.

Exemple

e1.nom←"Ahmed" lire(e1.moy) lire(e1.nom) e2.moy←e1.moy+1	strcpy(e1.nom, "Ahmed") ; scanf("%f",&e1.moy) ; gets(e2.nom) ; e2.moy=e1.moy+1 ;

Une variable d'un type de structure peut être affectée à une autre variable du même type, de sorte que tous les champs soient copiés dans la deuxième variable. Mais vous ne pouvez pas les comparer. Une table d'enregistrements peut également être utilisée.

```
e2=e1 ;
Etudiant T[100] ;
```

3.6. Exemple

Écrivez le programme qui définit une structure contenant des informations sur l'étudiant (numéro d'étudiant, nom de l'étudiant, date de naissance, la moyenne du bac). Notez que la date de naissance est une structure qui contient (jour, mois, année). Puis il remplit un tableau de N étudiants, puis demande à l'utilisateur une date, pour afficher tous les étudiants nés à cette date.

Algorithme etudiants
type
Date =**struct**
  Day, Month , Year : entier
finStruct
Student =**struct**
  num : entier
  name : chaine de caractères
  birthday :Date
  bac :réel
finStruct
 	
var
  st[100] : **tableau de** Student
  i, N : entier
  d : Date
Début
écrire("entrer le nbr des étudiants")
	#include <stdio.h>
	#include <string.h>
	typedef struct{
	  int Day, Month , Year;
	} Date ;
	typedef struct{
	  int num;
	  char name [20] ;
	  Date birthday ;
	  float bac ;
	} Student ;
	int main(){
	Student st[100] ;
	int i, N ;
	Date d ;
	printf("entrer le nbr des étudiants\n") ;
	scanf("%d",&N) ;
	//remplir le tableau

```
lire(N) ;
Pour i←0 à N-1 faire
  écrire("étudiants ", i)
  écrire("Num : ")
  lire(st[i].num)
  écrire("Nom : ")
  lire(st[i].name)
  écrire("Date de naissance (j/m/a) : ")
  lire(st[i].birthday. Day,
    st[i].birthday.Month,
    st[i].birthday.Year )
  écrire("Bac : ")
  lire(st[i].bac)
finPour
écrire("entrer une date (j/m/a) : ")
lire(d.Day, d.Month, d.Year )
écrire("Num   Nom   Bac")
Pour i←0 à N-1 faire
  si (st[i].birthday.Day==d.Day) et
    (st[i].birthday.Month==d.Month) et
    (st[i].birthday.Year==d.Year) alors
     écrire(st[i].num, "   ", st[i].name,
       "   ", st[i].bac)
  finSi
finPour
fin
```

```
for(i=0 ;i<N :i++){
  printf("étudiants %d\n",i) ;
  printf("Num : ") ;
  scanf("%d",&st[i].num) ;
  printf("Nom : ") ;
  gets(st[i].name) ;
  printf("Date de naissance (j/m/a) : ") ;
  scanf("%d%d%d", &st[i].birthday.Day,
    &st[i].birthday.Month,
    &st[i].birthday.Year ) ;
  printf("Bac : ") ;
  scanf("%d",&st[i].bac) ;
}
printf("entrer une date (j/m/a) : ") ;
scanf("%d%d%d", &d.Day, &d.Month,
  &d.Year ) ;
//affichage
printf("Num \tNom \tBac\n") ;
for(i=0 ;i<N :i++)
  if( (st[i].birthday.Day==d.Day) &&
    (st[i].birthday.Month==d.Month) &&
    (st[i].birthday.Year==d.Year) )
     printf("%d \t%s \t%.2f\n", st[i].num,
       st[i].name, st[i].bac) ;
return 0 ;
}
```

4. Autres façons de déclarer des données

4.1. union

4.1.1. Définition

Une union, comme une structure, est un groupe d'éléments de types différents. Mais il ne peut contenir, à chaque instant du programme, qu'une seule valeur d'un de ses éléments.

4.1.2. Déclaration

```
union_variables : union
 field_names :type
 …
finUnion
```

```
union union_name {
 type field_names ;
 …
}union_variables ;
```

Où **union** est un mot réservé en C.

- union_name : le nom de l'union (optionnel) et doit être un identifiant valide.
- type field_names : déclare les données (champs) qui composent l'union. Vous pouvez définir des éléments dans l'union en nommant le type, suivi d'une ou plusieurs variables de nom de champ (séparées par ","). des variables de différentes type sont séparées avec un point-virgule ";".
- union_variables : les noms des variables sont de type union, qui sont également optionnels.

Bien que les noms de variable et le nom de l'union soient facultatifs, l'un des deux doit apparaître.

Exemple

```
r1, r2 : union
  grade: caractère
  moy : réel
finUnion
```

```
union Result{
  char grade;
  float moy;
}e1, e2;
```

Ici, les variables de type union Result r1 et r2 sont déclarées, où chaque variable contient deux champs : grade de type caractère et moy de type nombre réel.

Il est préférable d'utiliser typedef en C pour déclarer le type union avant de déclarer les variables. Ainsi l'exemple précédent devient :

```
type Result = union
  grade: caractère
  moy: réel
finStruct
var
```

```
typedef union {
  char grade;
  float moy;
} Result;
```

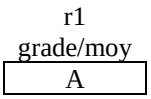
`r1, r2 : Result`	`Result r1, r2 ;`

4.1.3. Représentation

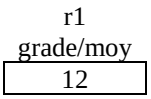
Lors de la définition du type d'union, la mémoire n'est pas réservée tant que la variable de type d'union n'est pas déclarée. L'union est représentée en mémoire comme une variable unique, qui prend la taille du plus grand élément de l'union. Par exemple, lors de la déclaration de la variable R1 du type Result, si nous supposons que la taille du grade char est de 1 octet et que la taille de float moy est de 4 octets, la taille de la variable r1 est de 4 octets, et non de 5 octets. Si le champ grade est utilisé, le premier octet sera utilisé et les trois autres seront ignorés, et si le champ moy est utilisé, ils seront utilisés tous les 4.

Si nous changeons un champ, l'autre change, car en fait il n'y a qu'un seul endroit, qui est traduit par le type de champ utilisé.

r1.grade='A';



r1.moy=12;



4.2. Autres types

- **Référence** (C++ uniquement) Les références permettent de manipuler une variable avec un nom différent de celui qui a été déclaré.

  ```
  type &référence = identificateur;
  int &ref = i;
  ```

  Ici, i et ref deviennent des noms pour la même variable.
- Pointeur : pour déclarer une variable pouvant contenir des adresses en mémoire (le deuxième semestre).
- Listes chaînées, files et piles (deuxième semestre)
- Arbres (deuxième année).
- Class (C++ uniquement) : est un ensemble de données (comme une structure), et un ensemble de fonctions sur ces données (deuxième année).