Chapter 6: Special Types

1. Introduction

In programming, each manipulated data must have its own type. This allows the compiler to validate values and operations to be applied. It helps the programmer discover and avoid errors. There are several predefined types in the programming language, such as integers and characters. The programmer can also define their own types, derived from basic types, such as arrays (as discussed in the previous chapter), enumerations, records, and other types.

2. Enumerations

2.1. Definition

The **enum** type: a data type defined by the programmer. It's an ordered list of constant values, defined by giving them names. This only makes sense to the programmer, making program understanding and memorization easier. The enum type is used to define a new type that only contains these values.

2.2. Declaration

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

The "**enum**" keyword is a reserved word used to define a set of integer constants.

- "type_name" is the name given by the programmer to this group. It must be a valid identifier (optional).
- "const_name" are constant names referring to the elements of the set. These names are meaningful only to the programmer. Their values can be defined.
- "var_list" is a list of variables of this type (optional).

In C, these names are integer constants with values 0, 1, ... Other values can also be assigned as follows:

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

- "val" is an integer. If "val" doesn't exist, its value is the value of the constant preceding it in the list +1. For the first integer constant in the list, its default value is 0.

Although variable names and enumeration names are optional, one of them must appear.

Example

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

2.3. Usage

Constants provide easier-to-remember names during programming than numbers, and they define the set of acceptable values. This prevents programmers from making errors in programming since it's not possible to assign a value outside the set to an enum variable (in C).

You can use comparison operations such as >, <, =, ?. This type can also be used with the "switch" directive.

2.4. Examples

```
enum Gender {Male, Female };
enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, 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("from 08:00 to 12:00\n") ;   break;
   default : printf("from 08:00 to 16:00\n") ;
}
```

3. Records or Structures

3.1. Definition

A structure, also called a record, is a composite type representing a set of named elements that can be accessed by their names. Each element is called a field. These fields can be of any type. The structure is used to group variables into a single record.

3.2. Declaration

Algorithm:	**C:**
structure_variables : **struct** field_names :type … **endStruct**	**struct** struct_name { type field_names ; … }structure_variables ;

Where "struct" is a reserved word in C.

- struct_name: a name for the structure (optional) and must be a valid identifier.
- type field_names: Declares the data (fields) that compose the structure. You can identify elements in the structure by naming the type, followed by one or more field names (separated by ","). Variables of different types are separated with a semicolon ";".
- structure_variables: names of structure-type variables, also optional.

Although variable names and structure names are optional, one of them must appear.

Example

e1, e2 : **struct** name: string bac : real **endStruct**	**struct** Student { char name [20] ; float bac; }e1, e2;

Here, two variables e1 and e2 of type struct Student are declared. Each variable contains two fields: "name" of type string and " bac " of type floating-point number.

Variable declaration can be deferred as follows:

type Student = **struct**
name: string
bac : real
endStruct
var
e1, e2 : Student
	struct Student {
	char name [20] ;
	float bac;
	};
	
	struct Student e1, e2 ;

The semicolon is required after "}", and "struct" must be mentioned before the structure name in C, but it's not required in C++.

In C, it's preferable to use "typedef" to declare a struct type before declaring variables. Thus, the previous example becomes:

type Student = **struct** name: string bac : real **endStruct** **var** e1, e2 : Student	**typedef struct** { char name [20] ; float bac ; } Student; Student e1, e2 ;

- The structure name doesn't exist, and "Student" is the new name for the structure.
- Fields can also be of structure type, as long as they are defined before being used.
- Since the scope of the field is limited to the record it belongs to, you don't need to worry about name conflicts between field names and other variables.

3.3. Representation

When defining a struct type, memory isn't reserved until a variable of that struct type is declared. The structure in memory is represented by adjacent variables. For example, when defining the "Student" type, no memory is allocated for the " name " and "bac" fields, but memory is allocated for them when creating the "e1" and "e2" records. The following figure represents the "e1" and "e2" records.

e1 e2

name	bac
Ahmed	13.41

nom	bac
Souad	12.50

The size of a structure is the sum of the sizes of its constituent fields.

3.4. Initialization

In C, initial values can be specified for all structure elements within two braces { and } during their declaration. The values are separated by a comma ",", and these values must be of the same type, order, and number of fields.

Example

```
Student e1= { "Ahmed", 13.41} ;
```

3.5. Usage

The "." symbol is used to access structure elements. This is done by using the variable name of the Structure type, followed by the dot, then the name of one of its fields.

Example

```
e1. name ←"Ahmed"
read (e1.moy)
read (e1. name)
e2.moy←e1.moy+1
```

```
strcpy(e1. name, "Ahmed") ;
scanf("%f",&e1.moy) ;
gets(e2. name) ;
e2.moy=e1.moy+1 ;
```

A variable of a structure type can be assigned to another variable of the same type, so that all fields are copied to the second variable. But you cannot compare them. An array of records can also be used.

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

3.6. Example

Write a program that defines a structure containing information about a student (student number, student name, birthdate, high school average). Note that the birthdate is a structure containing (day, month, year). Then, fill an array of N students, and ask the user for a date, to display all students born on that date.

```
Algorithm students
type
Date =struct
  Day, Month , Year : integer
endStruct
Student =struct
  num : integer
  name : string
  birthday :Date
  bac : real
endStruct

var
  st[100] : array of Student
  i, N : integer
  d : Date
Begin
write("enter number of students")
read(N) ;
for i←0 to N-1 do
  write("students ", i)
  write("Num : ")
  read(st[i].num)
  write("Name : ")
  read (st[i].name)
  write("Date of birth (d/m/y) : ")
  read(st[i].birthday. Day,
    st[i].birthday.Month,
    st[i].birthday.Year )
  write ("Bac : ")
```

```
#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("enter number of students") ;
scanf("%d",&N) ;
//fill table
for(i=0 ;i<N ;i++){
  printf("students %d\n",i) ;
  printf("Num : ") ;
  scanf("%d",&st[i].num) ;
  getch() ;
  printf("Name: ") ;
  gets(st[i].name) ;
  printf("Date of birth (j/m/a) : ") ;
  scanf("%d%d%d", &st[i].birthday.Day,
    &st[i].birthday.Month,
    &st[i].birthday.Year ) ;
  printf("Bac : ") ;
```

```
  lire(st[i].bac)
end For
write("enter a date (d/m/y) : ")
read (d.Day, d.Month, d.Year )
write("Num Name   Bac")
For i←0 to N-1 do
 if (st[i].birthday.Day==d.Day) et
   (st[i].birthday.Month==d.Month) et
   (st[i].birthday.Year==d.Year) alors
    write (st[i].num, "   ", st[i].name,
     "   ", st[i].bac)
 end if
end For
end
```

```
  scanf("%f",&st[i].bac) ;
}
printf("enter a date (d/m/y) : ") ;
scanf("%d%d%d", &d.Day, &d.Month,
  &d.Year ) ;
// display
printf("Num \tName \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. Other Ways to Declare Data

4.1. Union

4.1.1. Definition

A union, like a structure, is a group of elements of different types. But at any given time in the program, it can hold only one value of one of its elements.

4.1.2. Declaration

```
union_variables : union
 field_names :type
 …
EndUnion
```

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

Where "union" is a reserved word in C.

- union_name: the name of the union (optional) and must be a valid identifier.
- type field_names: declares the data (fields) that compose the union. You can define elements in the union by naming the type, followed by one or more field names (separated by ","). Variables of different types are separated by a semicolon ";".
- union_variables: the names of union-type variables, also optional.

Although variable names and union names are optional, one of them must appear.

Example

```
r1, r2 : union
  grade: character
  moy : real
endUnion
```

```
union Result{
  char grade;
  float moy;
}r1, r2;
```

Here, variables of type union Result r1 and r2 are declared, where each variable contains two fields: grade of type char and avg of type floating-point number.

It's preferable to use "typedef" in C to declare the union type before declaring the variables. Thus, the previous example becomes:

```
type Result = union
  grade: character
  moy: real
endunion
var
r1, r2 : Result
```

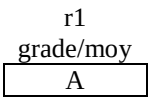
```
typedef union {
  char grade;
  float moy;
} Result;

Result r1, r2 ;
```

4.1.3. Representation

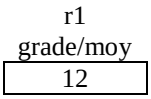
When defining the union type, memory isn't reserved until a variable of that union type is declared. The union is represented in memory as a single variable, which takes the size of the largest element in the union. For example, when declaring the variable "r1" of type Result, assuming that the size of char grade is 1 byte and the size of float avg is 4 bytes, the size of the variable r1 is 4 bytes, not 5 bytes. If the grade field is used, the first byte will be used, and the other three will be ignored. If the avg field is used, all four bytes will be used.

If we change one field, the other changes as well, because in fact there is only one location, translated by the type of field used.

```
r1.grade='A';
```




```
r1.moy=12;
```



4.2. Other Types

- Reference (C++ only) References allow manipulating a variable with a different name than the one declared.

  ```
  type & reference = identifier;
  int &ref = i;
  ```

  Here, "i" and "ref" become names for the same variable.
- Pointer: To declare a variable that can hold memory addresses (second semester).
- Linked lists, queues, and stacks (second semester).
- Trees (second year).
- Class (C++ only): A set of data (like a structure), and a set of functions on this data (second year).