MODULE 11
TYPE SPECIFIERS
struct, typedef, enum, union

My Training Period: hours

Note: From this Module you can jump to the Object Oriented idea and C++ or proceed to extra C Modules or Microsoft C: implementation specific to experience how C is used in the real implementation.

Abilities

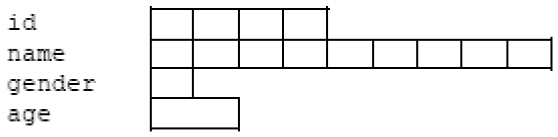
- Able to understand and use structure (`struct`).
- Able to relate structure, functions and array.
- Able to understand and use `typedef`.
- Able to understand and use `union`.
- Able to understand and use Enumeration (`enum`).

11.1 Structure

- We have learned that by using an array, we only can declare **one data type** per array, and it is same for other data types. To use the same data type with different name, we need another declaration.
- `struct` data type overcomes this problem by declaring aggregate data types.
- A structure is a **collection of related data items** stored in one place and can be referenced by more than one names. Usually these data items are **different** basic data types. Therefore, the number of bytes required to store them may also vary.
- It is very useful construct used in data structure, although in C++, many `struct` constructs has been replaced by class construct but you will find it quite a common in the Win32 programming.
- In order to use a structure, we must first declare a structure template. The variables in a structure are called **structure elements** or **members**.
- For example, to store and process a student's record with the elements `id_num` (identification number), `name`, `gender` and `age`, we can declare the following structure.

```
struct student {
     char id_num[5];
     char name[10];
     char gender;
     int age;
     };
```

- Here, `struct` is a **keyword** that tells the compiler that a structure template is being declared and `student` is a tag that identifies its data structure. Tag is not a variable; it is a **label** for the structure's template. Note that there is a semicolon after the closing curly brace.
- A structure tags simply a label for the structure's template but you name the structure tag using the same rules for naming variables. The template for the structure can be illustrated as follow (note the different data size):



- Compiler will not reserve memory for a structure until you declare a structure variable same as you would declare normal variables such as `int` or `float`.
- Declaring structure variables can be done in any of the following ways (by referring to the previous example):

```
1.  struct  student{
        char id_num[5];
        char name[10];
        char gender;
        int age;
        } studno_1, studno_2;

2.  struct{//no tag
```

```
        char id_num[5];
        char name[10];
        char gender;
        int age;
        }studno_1, studno_2;

3.  struct student{
        char id_num[5];
        char name[10];
        char gender;
        int age;
        };
```

- Then in programs we can declare the structure something like this:

```
struct student studno_1, studno_2;
```

- In the above three cases, two structure variables, `studno_1` and `studno_2`, are declared. Each structure variable has 4 elements that is 3 character variables and an integer variable.
- In (1) and (2), the structure variables are declared immediately after the closing brace in the structure declaration whereas in (3) they are declared as `student`. Also in (2) there is no structure tag, this means we cannot declare structure variables of this type elsewhere in the program instead we have to use the structure variables `studno_1` and `studno_2` directly.
- The most widely used may be no (1) and (3) where we put the declaration of the `struct` in header files and use it anywhere in programs as follows:

```
struct student studno_1, studno_2;
```

- Where the `studno_1, studno_2` are variables declared as usual but the type here is **`struct`** `student` instead of integral type such as `int`, `char` and `float`.
- It is also a normal practice to combine the `typedef` (will be explained later on) with `struct`, making the variables declaration even simpler. For example:

```
typedef struct TOKEN_SOURCE {
  CHAR SourceName[8];
  LUID SourceIdentifier;
} TOKEN_SOURCE,
*PTOKEN_SOURCE;
```

- In this example we use `typedef` with `struct`. In our program we just declare variable as follows:

```
TOKEN_SOURCE  myToken;
```

- The `TOKEN_SOURCE` type is used for a normal variable and the `*PTOKEN_SOURCE` type is used for a pointer variable. You will find that these are typical constructs in Win32 programming of Windows. For more information refers to Module A or Tutorial #2.

11.2 Accessing The Structure Element

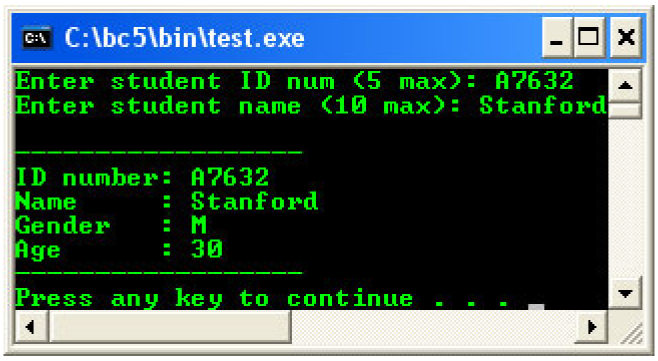
- A structure element can be accessed and assigned a value by using the **structure variable name**, the **dot operator** (`.`) and the **element's name**. For example the following statement:

```
studno_1.name = "jasmine";
```

- Assigns string `"jasmine"` to the element `name` in the structure variable `studno_1`. The **dot operator** simply qualifies that `name` is an element of the structure variable `studno_1`. The other structure elements are referenced in a similar way.
- Unfortunately, we cannot assign string `"jasmine"` (`const char`) directly to an array in the structure (`char []`). For this reason, we have to use other methods such as receiving the string from user input or by using pointers.
- Program example.

```
//A simple structure program example
#include <stdio.h>
#include <stdlib.h>

struct student{
  char id_num[6];
  char name[11];
  char gender;
  int age;
  };

int main(void)
{
struct student studno_1;

//studno_1.id_num = "A3214"; //Illegal, const char to char[]
//studno_1.name = "Smith";   //Illegal, const char to char[]
printf("Enter student ID num (5 max): ");
scanf("%s", studno_1.id_num);
printf("Enter student name (10 max): ");
scanf("%s", studno_1.name);
studno_1.gender = 'M';
studno_1.age = 30;

printf("\n------------------\n");
printf("ID number: %s\n", studno_1.id_num);
printf("Name     : %s\n", studno_1.name);
printf("Gender   : %c\n", studno_1.gender);
printf("Age      : %d\n", studno_1.age);
printf("------------------\n");
system("pause");
return 0;
}
```

Output:

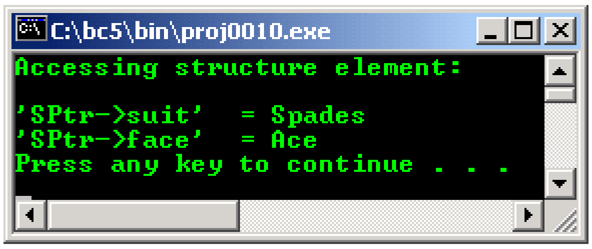


```
Enter student ID num (5 max): A7632
Enter student name (10 max): Stanford

------------------
ID number: A7632
Name     : Stanford
Gender   : M
Age      : 30
------------------
Press any key to continue . . .
```

- The **structure pointer operator** (→), consisting of a minus (`-`) sign and a greater than (`>`) sign with no intervening spaces, accesses a structure member via a **pointer to the structure**.
- By assuming that a pointer `SPtr` has been declared to point to `struct Card`, and the address of structure `p` has been assigned to `SPtr`. To print member `suit` of structure `Card` with pointer `SPtr`, use the statement `Sptr→suit` as shown in the following example.

```
  //accessing structure element
  #include <iostream.h>
  #include <stdlib.h>

  struct Card
  {
     char   *face;  //pointer to char type
     char   *suit;  //pointer to char type
  };

  void main()
  {
       //declare the struct type variables
       struct Card  p;
       struct Card  *SPtr;

    p.face = "Ace";
    p.suit = "Spades";
    SPtr = &p;

     cout<<"Accessing structure element:\n";
     cout<<"\n\'SPtr->suit\'  = "<<SPtr->suit<<endl;
     cout<<"\'SPtr->face\'  = "<<SPtr->face<<endl;
     //for Borland...
    system("pause");
}
```

Output:

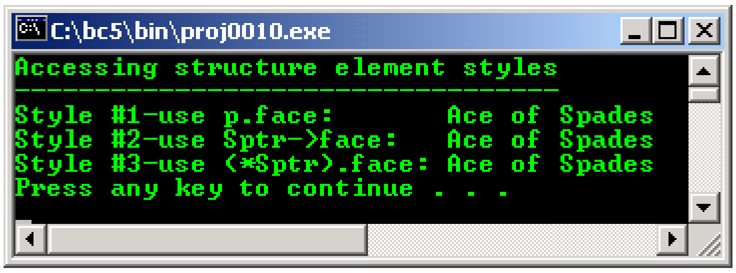


```
Accessing structure element:

'SPtr->suit'   = Spades
'SPtr->face'   = Ace
Press any key to continue . . .
```

- The expression `SPtr->suit` is equivalent to `(*SPtr).suit` which dereferences the pointer and accesses the member `suit` using the structure member operator.
- The parentheses are needed here because the structure member operator, the dot `(.)` has higher precedence than the pointer dereferencing operator, the asterisk `(*)`.
- Program example:

```
//Using the structure member and structure
//pointer operators – accessing structure
//elements styles...
#include <iostream.h>
#include <stdlib.h>

struct Card
{
    char  *face;
    char  *suit;
};

int main()
{
    struct Card  p;
    struct Card  *SPtr;

    p.face = "Ace";
    p.suit = "Spades";
    SPtr = &p;

cout<<"Accessing structure element styles"<<endl;
cout<<"-----------------------------------"<<endl;
cout<<"Style #1-use p.face:       "<<p.face<<" of "<<p.suit<<endl;
cout<<"Style #2-use Sptr->face:   "<<SPtr->face<<" of "<<SPtr->suit<<endl;
cout<<"Style #3-use (*Sptr).face: "<<(*SPtr).face<<" of "<<(*SPtr).suit<<endl;
system("pause");
return 0;
}
```

Output:



```
Accessing structure element styles
-----------------------------------
Style #1-use p.face:       Ace of Spades
Style #2-use Sptr->face:   Ace of Spades
Style #3-use (*Sptr).face: Ace of Spades
Press any key to continue . . .
```

11.3 Arrays Of Structures

- Suppose you would like to store the information of `100` students. It would be tedious and unproductive to create `100` different student array variables and work with them individually. It would be much easier to create an array of student structures.
- Structures of the same type can be grouped together into an array. We can declare an array of structures just like we would declare a normal array of variables.

- For example, to store and manipulate the information contained in `100` student records, we use the following statement:

```
struct student{
      char id[5];
      char name[80];
      char gender; int age;
      }stud[100];
```

- Or something like the following statements:

```
struct student{
      char id[5];
      char name[80];
      char gender;
      };
struct student stud[100];
```

- These statements declare `100` variables of type `student` (a structure). As in arrays, we can use a subscript to reference a particular student structure or record.
- For example, to print the name of the seventh student, we could write the following statement:

```
cout<<stud[6].name;
```

- Example of initializing all the student names to blanks and their ages to `0`, we could do this simply by using `for` loop as shown below:

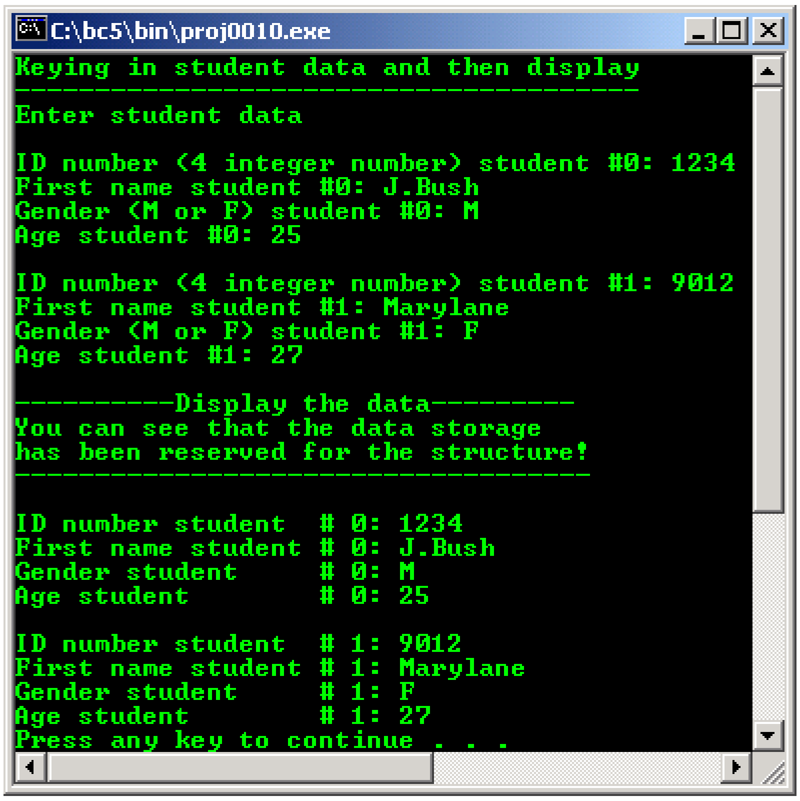
```
for(i=0; i<100; i++)
{
  stud[i].name = " ";
  stud[i].age = 0;
}
```

- Very useful huh! Now let try a program example:

```
//an array structure of student information
#include <iostream.h>
#include <stdlib.h>
//for C++ replace to the following header
//#include <iostream>
//#include <cstdlib>
//using namespace std;

struct student
{
    char id[6];      //student id number, max. 5 integer number
    char name[50];   //student name, max 49 characters
    char gender;     //student gender Male or Female
    int  age;        //student age
};

void main()
{
    //declaring array of 10 element of structure type
    //and some of the element also are arrays
    struct student stud[10];
    int i = 0;

    cout<<"Keying in student data and then display\n";
    cout<<"---------------------------------------\n";
    cout<<"Enter student data\n";

    for(i=0; i<2; i++)
    {
         //Storing the data
         cout<<"\nID number (4 integer number) student #"<<i<<": ";
         cin>>stud[i].id;
         cout<<"First name student #"<<i<<": ";
         cin>>stud[i].name;
         cout<<"Gender (M or F) student #"<<i<<": ";
         cin>>stud[i].gender;
         cout<<"Age student #"<<i<<": ";
         cin>>stud[i].age;
}
```

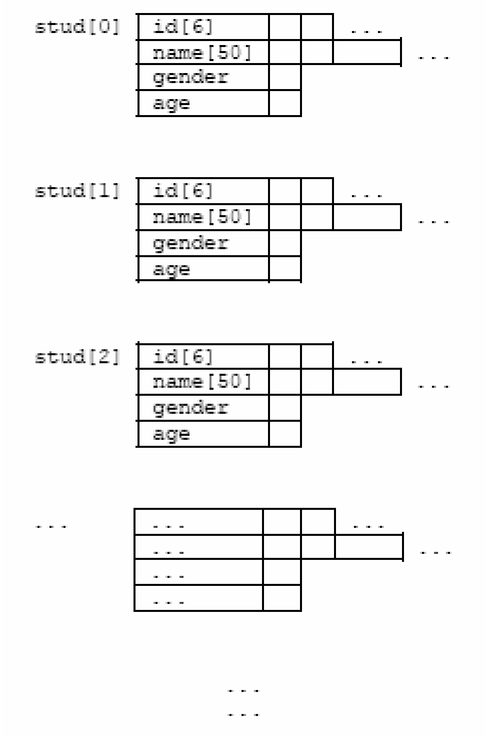
```
    cout<<"\n----------Display the data---------\n";
    cout<<"You can see that the data storage\n";
    cout<<"has been reserved for the structure!\n";
    cout<<"------------------------------------\n";
    for(i=0; i<2; i++)
    {
    //Displaying the stored data
    cout<<"\nID number student  # "<<i<<": "<<stud[i].id;
    cout<<"\nFirst name student # "<<i<<": "<<stud[i].name;
    cout<<"\nGender student     # "<<i<<": "<<stud[i].gender;
    cout<<"\nAge student        # "<<i<<": "<<stud[i].age<<"\n";
    }
    system("pause");
}
```

Output:



```
Keying in student data and then display
---------------------------------------
Enter student data

ID number (4 integer number) student #0: 1234
First name student #0: J.Bush
Gender (M or F) student #0: M
Age student #0: 25

ID number (4 integer number) student #1: 9012
First name student #1: Marylane
Gender (M or F) student #1: F
Age student #1: 27

----------Display the data---------
You can see that the data storage
has been reserved for the structure!
------------------------------------

ID number student  # 0: 1234
First name student # 0: J.Bush
Gender student     # 0: M
Age student        # 0: 25

ID number student  # 1: 9012
First name student # 1: Marylane
Gender student     # 1: F
Age student        # 1: 27
Press any key to continue . . .
```

- The structure template for the program example can be illustrated as follows:



11.4 Structures And Function

- Individual structure elements or even an entire structure can be passed to functions as arguments. For example, to modify the name of the seventh student, let say the function name is `modify()`, we could issue the following function call:

```
modify(stud[6].name);
```

- This statement passes the structure element `name` of the seventh student to the function `modify()`. Only a copy of `name` is passed to the function. This means any change made to `name` in the called function is local; the value of `name` in the calling program remains unchanged.
- An entire structure can also be passed to a function. We can do this either by passing the structure itself as argument or by simply passing the address of the structure.
- For structure element example, to modify the seventh student in the list, we could use any of the following statements.

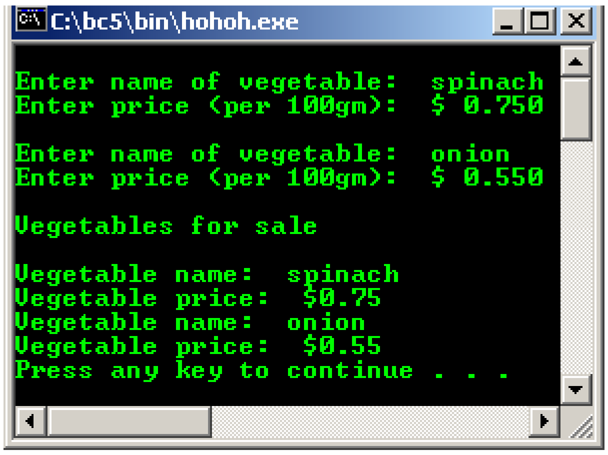
```
modify(stud[6]);
modify(&stud[6]);
```

- In the first statement, **a copy of the structure** is passed to the function while in the second only the **address of the structure** is passed.
- As only a copy of the structure is passed in the first statement, any changes made to the structure within the called function do not affect the structure in the calling function.
- However in the second statement any changes made to the structure within the called function will change the structure values in the calling function since the called function directly accesses the structure and its elements.
- Let take a look at a program example.

```
//passing structures to functions and
//a function returning a structure
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>


//-------------structure part-------------
struct vegetable
{
    char   name[30];
    float  price;
};
```

```
//--------------main program-------------
int main()
{
    //declare 2 structure variables
    struct vegetable veg1, veg2;
    //function prototype of type struct
    struct vegetable  addname();
    //another function prototype
    int list_func(vegetable);

    //functions call for user input...
    veg1  =  addname();
    veg2  =  addname();
    cout<<"\nVegetables for sale\n";

    //function call for data display...
    list_func(veg1);
    list_func(veg2);
    cout<<endl;
    //for Borland
    system("pause");
    return  0;
}
//-------------function---------------
//This functions returns a structure
struct vegetable addname()
{
    char  tmp[20];
    //declare a structure variable
    struct vegetable vege;

    cout<<"\nEnter name of vegetable:  ";
    gets(vege.name);
    cout<<"Enter price (per 100gm):  $ ";
    gets(tmp);
    //converts a string to float
    vege.price = atof(tmp);
    return (vege);
}

//structure passed from main()
int list_func(vegetable list)
{
    cout<<"\nVegetable name:  "<<list.name;
    cout<<"\nVegetable price:  $"<<list.price;
    return 0;
}
```

Output:



```
C:\bc5\bin\hohoh.exe

Enter name of vegetable:  spinach
Enter price (per 100gm):  $ 0.750

Enter name of vegetable:  onion
Enter price (per 100gm):  $ 0.550

Vegetables for sale

Vegetable name:  spinach
Vegetable price:  $0.75
Vegetable name:  onion
Vegetable price:  $0.55
Press any key to continue . . .
```

- It is also possible to create structures in a structure.

11.5 typedef

- In contrast to the **class**, **struct**, **union**, and **enum** declarations, **typedef** declaration do not introduce new type but introduces **new name** or creating **synonym** (or alias) for existing type.
- The syntax is as follows:

```
typedef type-declaration the_synonym;
```

- You cannot use the `typedef` specifier inside a function definition.
- When declaring a local-scope identifier by the same name as a `typedef`, or when declaring a member of a structure or `union` in the same scope or in an inner scope, the type specifier must be specified. For example:

```
typedef float TestType;

int main()
{
}

//function scope...
int MyFunct(int)
{
  //same name with typedef, it is OK
  int TestType;
}
```

- When declaring a local-scope identifier by the same name as a `typedef`, or when declaring a member of a structure or `union` in the same scope or in an inner scope, the type specifier must be specified. For example:

```
//both declarations are different...
typedef float TestType;
const TestType r;
```

- To reuse the `TestType` name for an identifier, a structure member, or a `union` member, the type must be provided, for example:

```
const int TestType;
```

- Typedef names share the name space with ordinary identifiers. Therefore, a program can have a typedef name and a local-scope identifier by the same name.

```
//typedef specifier
typedef char FlagType;

void main()
{
}

void myproc(int)
{
    int FlagType;
}
```

- The following paragraphs illustrate other `typedef` declaration examples that you will find used a lot in Win32 programming:

```
//Character type.
typedef char CHAR;

//Pointer to a string (char *).
typedef CHAR * THESTR;
//Then use it as function parameter...
THESTR strchr(THESTR source, CHAR destination);

typedef unsigned int uint;
//Equivalent to "unsigned long ui;"
uint ui;
```

- To use `typedef` to specify fundamental and derived types in the same declaration, you can separate declarators with comma. For example:

```
typedef char CHAR, *THESTR;
```

- The following example provides the type `Funct()` for a function returning no value and taking two `int` arguments:

```
typedef void Funct(int, int);
```

- After the above `typedef` statement, the following is a valid declaration:

```
Funct test;
```

- And equivalent to the following declaration:

```
void test(int, int);
```

- Names for structure types are often defined with `typedef` to create **shorter** and **simpler** type name. For example, the following statements:

```
typedef  struct Card Card;
typedef unsigned short USHORT;
```

- Defines the new type name `Card` as a synonym for type `struct Card` and `USHORT` as a synonym for type `unsigned short`. Programmers usually use `typedef` to define a structure type so a structure tag is not required. For example, the following definition:

```
typedef struct{
      char  *face;
      char  *suit;
} Card;
```

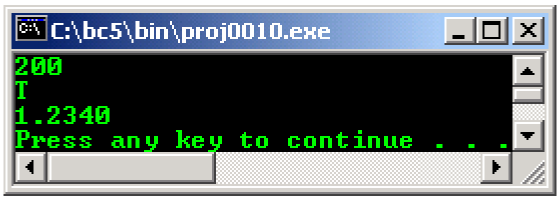
- Creates the structure type `Card` without the need for a separate `typedef` statement. Then `Card` can now be used to declare variables of type `struct Card`. For example, the following declaration:

```
Card  deck[50];
```

- Declares an array of `50` `Card` structures. `typedef` simply creates a **new type name** which may be used as an alias for an existing type name.
- Often, `typedef` is used to create synonyms for the basic data types. For example, a program requiring 4-byte integers may use type `int` on one system and type `long` on another.
- Programs designed for portability often uses `typedef` to create an alias for 4-byte integers such as, let say `Integer`. The alias `Integer` can be changed once in the program to make the program work on both systems.
- Program example:

```
//typedef and struct program example
#include <stdio.h>
#include <stdlib.h>

typedef struct TestStruct
{
   int   p;
   char q;
   double r;
} mine;

void main()
{
  mine testvar; //the declaration becomes simpler
  testvar.p = 200;
  testvar.q = 'T';
  testvar.r = 1.234;
  printf("%d\n%c\n%.4f\n", testvar.p, testvar.q, testvar.r);
  system("pause");
}
```

Output:

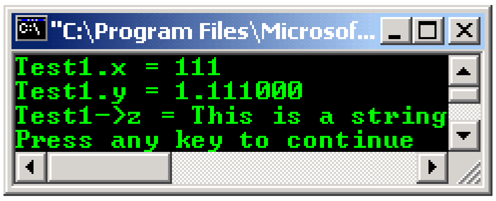


```
200
T
1.2340
Press any key to continue . . .
```

- Another program example.

```
//typedef specifier
#include <stdio.h>

typedef struct mystructtag
{
   int   x;
   double y;
   char* z;
} mystruct;

int main()
{
   mystruct Test1, *Test2;
   Test1.x = 111;
   Test1.y = 1.111;
   printf("Test1.x = %d\nTest1.y = %f\n", Test1.x, Test1.y);

   Test1.z = "This is a string";
   Test2 = &Test1;
   printf("Test1->z = %s\n", Test2->z);
   return 0;
}
```

Output:



```
Test1.x = 111
Test1.y = 1.111000
Test1->z = This is a string
Press any key to continue
```

11.6 enum - Enumeration Constants

- This is another user-defined type consisting of a set of named constants called enumerators.
- Using a keyword `enum`, it is a set of **integer constants** represented by identifiers.
- The syntax is shown below:

```
//for definition of enumerated type
enum [tag]
{
      enum-list
}
[declarator];
```

- And

```
//for declaration of variable of type tag
enum tag declarator;
```

- These enumeration constants are, in effect, **symbolic constants** whose values can be set automatically. The values in an `enum` start with `0`, unless specified otherwise, and are incremented by `1`. For example, the following enumeration:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

- Creates a new data type, `enum days`, in which the identifiers are set automatically to the integers `0` to `6`. To number the `days` 1 to 7, use the following enumeration:

```
enum days {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};
```

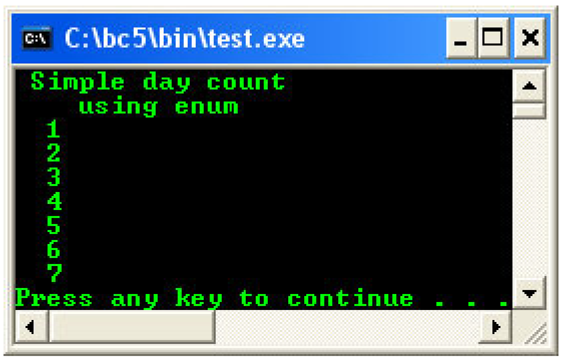
- Or we can re arrange the order:

```
enum days {Mon, Tue, Wed, Thu = 7, Fri, Sat, Sun};
```

- Simple program example:

```
#include <iostream.h>
#include <stdlib.h>

enum days {mon = 1,tue,wed,thu,fri,sat,sun};

void main()
{
    //declaring enum data type
    enum days day_count;

    cout<<" Simple day count\n";
    cout<<"    using enum\n";

    for(day_count=mon; day_count<=sun; day_count++)
    cout<<"  "<<day_count<<"\n";
    system("pause");
}
```

Output:

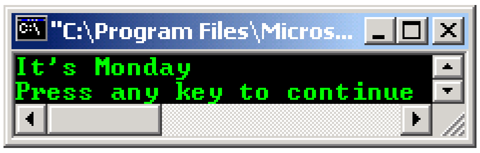


```
 Simple day count
    using enum
  1
  2
  3
  4
  5
  6
  7
Press any key to continue . . .
```

- As said before, by default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator.
- Enumerators needn't have unique values within an enumeration. The name of each enumerator is treated as a constant and must be unique within the scope where the `enum` is defined. An enumerator can be **promoted** to an integer value.
- Converting an integer to an enumerator requires an **explicit cast**, and the results are not defined if the integer value is outside the range of the defined enumeration.
- Enumerated types are valuable when an object can assume a known and reasonably limited set of values.
- Another program example.

```
//enum declarations
#include <iostream>
using namespace std;

//Declare enum data type Days
enum Days
{
    monday,          //monday = 0 by default
    tuesday = 0,     //tuesday = 0 also
    wednesday,       //wednesday = 1
    thursday,        //thursday = 2
    friday,          //an so on.
    saturday,
    sunday
};

int main()
{
    //try changing the tuesday constant,
    //recompile and re run this program
    enum Days WhatDay = tuesday;
    switch (WhatDay)
    {
          case 0:
          cout<<"It's Monday"<<endl;
          break;
          default:
          cout<<"Other day"<<endl;
    }
return 0;
}
```

Output:



```
It's Monday
Press any key to continue
```

- After the `enum` data type has been declared and defined, in C++ it is legal to use the `enum` data type without the keyword `enum`. From the previous example, the following statement is legal in C++:

```
 //is legal in C++
 Days WhatDay = tuesday;
```

- Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators.
- The following example defines an enumerated type that ensures any two enumerators can be combined with the `OR` operator.
- In this example, the preceding enumerator initializes each succeeding enumerator.

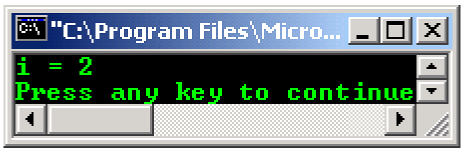
```
//enum definition
#include <iostream>
using namespace std;

enum FileOpenFlags
{
    //defined here...
    OpenReadOnly  = 1,
    //using OpenReadOnly as the next initializer
    //and so on...
    OpenReadWrite = OpenReadOnly,
    OpenBinary = OpenReadWrite,
    OpenText = OpenBinary,
    OpenShareable = OpenText
};

int main()
{
    return 0;
}
//No output
```

- As said and shown in the program example before, enumerated types are integral types, any enumerator can be converted to another integral type by integral promotion. Consider the following example.

```
//enumeration data type
#include <iostream>
using namespace std;

enum Days
{

    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday,
    Monday
};
```

```
int  i;
Days d = Thursday;
int main()
{
  //Converted by integral promotion.
  i = d;
  cout<<"i = "<<i<<"\n";
  return 0;
}
```

Output:



```
i = 2
Press any key to continue
```

- However, there is no implicit conversion from any integral type to an enumerated type. Therefore from the previous example, the following statement is an error:

```
//Erroneous attempt to set d to Saturday.
d = 5;
```

- The assignment d = 5, where no implicit conversion exists, must use a cast to perform the conversion:

```
//Explicit cast-style conversion to type Days.
d = (Days)5;
//Explicit function-style conversion to type Days.
d = Days(4);
```

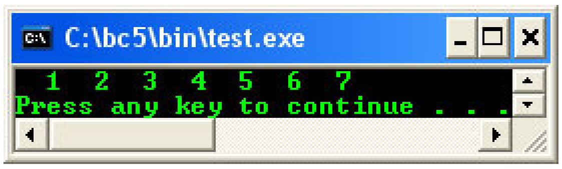
- The preceding example shows conversions of values that coincide with the enumerators. There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example:

```
d = Days(30);
```

- Some such conversions may work but there is no guarantee the resultant value will be one of the enumerators.
- The following program example uses `enum` and `typedef`.

```
//typedef and enum...
#include <stdio.h>
#include <stdlib.h>

typedef enum {
  FailOpenDisk = 1,
  PathNotFound,
  FolderCannotBeFound,
  FileCannotBeFound,
  FailOpenFile,
  FileCannotBeRead,
  DataCorrupted
} ErrorCode;

int main(void)
{
   ErrorCode MyErrorCode;

   for(MyErrorCode=FailOpenDisk; MyErrorCode<=DataCorrupted; MyErrorCode++)
  printf(" %d", MyErrorCode);
    printf("\n");
   system("pause");
   return 0;
}
```

Output:



```
  1  2  3  4  5  6  7
Press any key to continue . . .
```

11.7 union

- A derived data type, whose members **share the same storage space**. The members of a union can be of any type and the **number of bytes** used to store a union must be at least enough to hold the largest member.
- Unions contain two or more data types. **Only** one member, and thus one data type, **can be referenced at a time**. It is the programmer's responsibility to ensure that the data in a union is referenced with the proper data type and this is the weakness of using `union` compared to `struct`.
- A union is declared with the `union` keyword in the same format as a structure. The following is a union declaration:

```
union sample
{
  int   p;
  float q;
};
```

- Indicates that `sample` is a union type with members' `int p` and `float q`. The union definition normally precedes the `main()` in a program so the definition can be used to declare variables in all the program's functions.
- Only a value with same type of the first union member can be used to initialize union in declaration part. For example:

```
union sample
{
  int   p;
  float q;
};
…
union sample content = {234};
```

- But, using the following declaration is invalid:

```
union sample
{
  int   p;
  float q;
};
…
union sample content = {24.67};
```

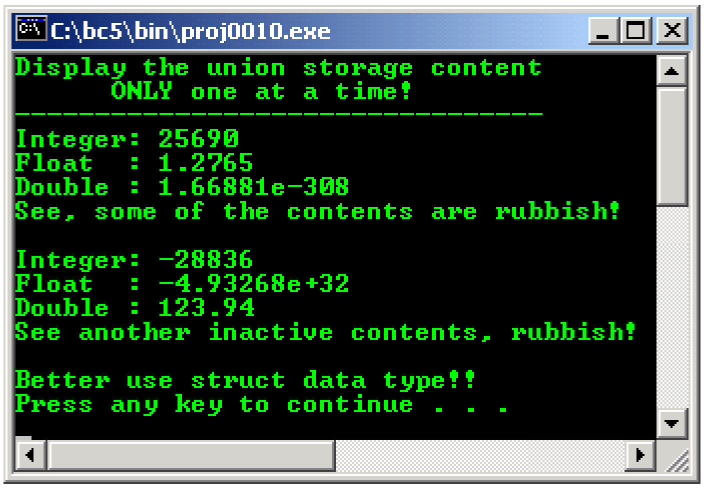
- The operations that can be performed on a union are:

  ▪ Assigning a union to another union of the same type.
  ▪ Taking the address (&) of a union and
  ▪ Accessing union member using the structure member operator and the structure pointer operator.

- Program example:

```
#include <iostream.h>
#include <stdlib.h>

union  sample
{
    int  p;
    float q;
    double r;
};

void main()
{
    //union data type
    union sample content;
    content.p = 37;
    content.q = 1.2765;

    cout<<"Display the union storage content\n";
    cout<<"      ONLY one at a time!\n";
    cout<<"---------------------------------\n";
    cout<<"Integer: "<<content.p<<"\n";
    cout<<"Float  : "<<content.q<<"\n";
    cout<<"Double : "<<content.r<<"\n";
    cout<<"See, some of the contents are rubbish!\n";

    content.q=33.40;
    content.r=123.94;

    cout<<"\nInteger: "<<content.p<<"\n";
    cout<<"Float  : "<<content.q<<"\n";
    cout<<"Double : "<<content.r<<"\n";
    cout<<"See another inactive contents, rubbish!\n";
    cout<<"\nBetter use struct data type!!\n";
    system("pause");
}
```

Output:

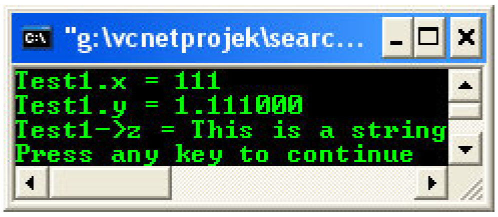


```
Display the union storage content
      ONLY one at a time!
---------------------------------
Integer: 25690
Float  : 1.2765
Double : 1.66881e-308
See, some of the contents are rubbish!

Integer: -28836
Float  : -4.93268e+32
Double : 123.94
See another inactive contents, rubbish!

Better use struct data type!!
Press any key to continue . . .
```

- Program example compiled using VC++/VC++ .Net.

```
//typedef specifier
#include <cstdio>

//typedef oldname newname
typedef struct mystructtag
{
   int   x;
   double y;
   char* z;
} mystruct;

int main()
{
   mystruct Test1, *Test2;
   Test1.x = 111;
   Test1.y = 1.111;
   printf("Test1.x = %d\nTest1.y = %f\n", Test1.x, Test1.y);

   Test1.z = "This is a string";
   Test2 = &Test1;
   printf("Test1->z = %s\n",Test2->z);
   return 0;
}
```

Output:



```
Test1.x = 111
Test1.y = 1.111000
Test1->z = This is a string
Press any key to continue
```

- Previous example compiled using g++.

```
///////typestruct.cpp//////////
//typedef specifier
#include <cstdio>
using namespace std;

//typedef oldname newname
typedef struct mystructtag
{
   int   x;
   double y;
   char* z;
} mystruct;

int main()
{
   mystruct Test1, *Test2;
   Test1.x = 111;
   Test1.y = 1.111;
   printf("Test1.x = %d\nTest1.y = %f\n", Test1.x, Test1.y);

   Test1.z = "This is a string";
   Test2 = &Test1;
   printf("Test1->z = %s\n", Test2->z);
   return 0;
}
```

```
[bodo@bakawali ~]$ g++ typestruct.cpp -o typestruct
[bodo@bakawali ~]$ ./typestruct

Test1.x = 111
Test1.y = 1.111000
Test1->z = This is a string
```

---o0o ---

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.