To:
Tenouk

MODULE 11a
THE C/C++ TYPE SPECIFIERS 2
struct, typedef, enum, union

My Training Period: xx hours

The struct lab worksheets are: C/C++ struct part 1 and C/C++ struct part 2. Also the combination of the struct, arrays, pointers and function C worksheet 1, C lab worksheet part 2 and C lab worksheet part 3.

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.

```
// a 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:

```
// a character type.
typedef   char CHAR;
```

```
// a 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 int 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 and look more natural as normal variable. For example, the following declaration:

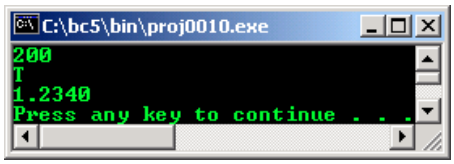
```
Card  deck[50];
```

- 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 system.
- 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. Microsoft uses this extensively in defining new type name for Win32 and Windows programming.
- A program example:

```
// typedef and struct program example
#include <stdio.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);
}
```

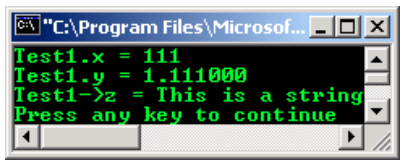
Output:



- 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:



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};
```

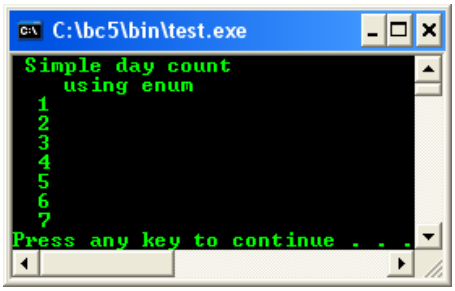
- Or we can re arrange the order:

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

- Simple program example:

```
#include <iostream>
using namespace std;

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";
}
```

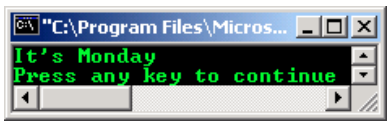
Output:



- 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:



- 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;
```

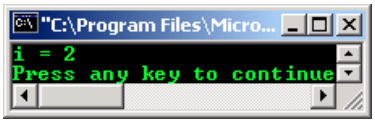
- Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators.
- The following example defines an enumerated type that ensures any two enumerators can be combined with the OR operator.
- In this example, the preceding enumerator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



- 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);
```

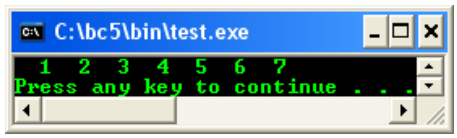
- The preceding example shows conversions of values that coincide with the enumerators. There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example:

```
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>

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");
return 0;
}
```

Output:



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:

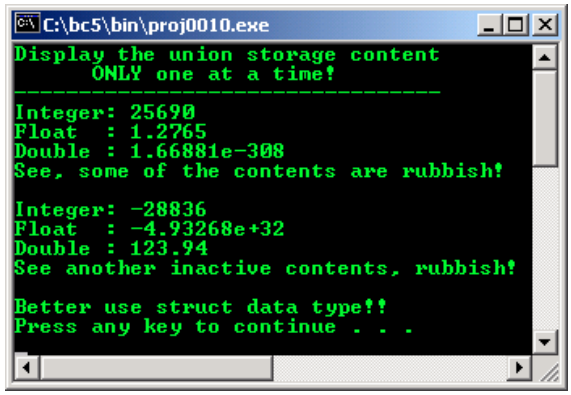
```
union sample
{
   int  p;
   float q;
};
...
...
union sample content = {24.67};
```

- 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>
using namespace std;

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";
}
```

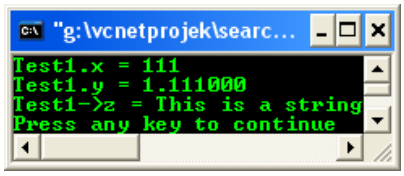
Output:



- 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:



- 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
```

C & C++ programming tutorials

Further C & C++ related reading and digging:

1. The struct lab worksheets for your practice are: C/C++ struct part 1 and C/C++ struct part 2.
2. Also the combination of the struct, arrays, pointers and function C worksheet 1, C lab worksheet part 2 and C lab worksheet part 3.
3. Check the best selling C/C++ books at Amazon.com.

|< C & C++ Preprocessor Directives | Main | C & C++ Type Specifiers 2 >| C-Extra | Microsoft C/Win32 | Site Index | Download |

C and C++ Type Specifiers: Part 1 | Part 2 |