Unit 2: Basics of C++ Programming [5hrs]

C++ Program Structure

Typical c++ program would contain four sections as shown in figure below. These sections may be placed in separate code files and then compiled independently or jointly.

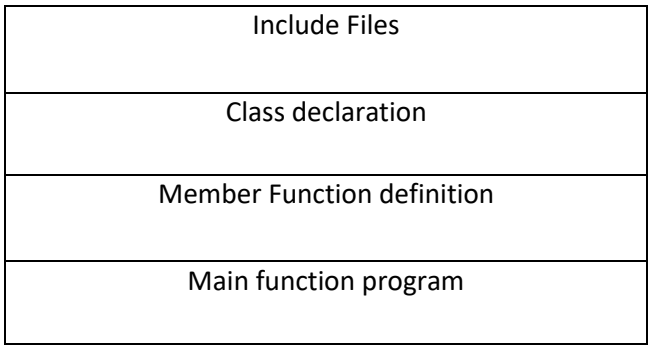


Fig: Structure of a C++ program

It is a common practice to organize a program into three separate files. The class declarations are placed in header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third files as well as any other files required.

This approach is based on the concept of client-server model. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Character Set and Tokens

The smallest individual units in a program are known as tokens. C++ has following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

Data Type

Data types in C++ can be classified under various categories as shown in figure below:

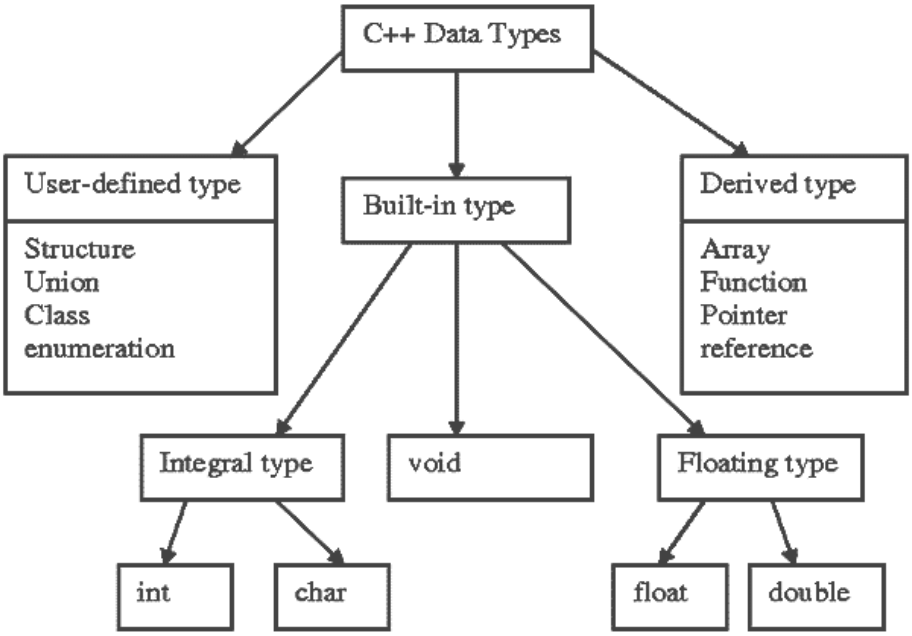


Fig: Hierarchy of C++ data types

Type Conversion

a) Explicit Type Conversion

C++ permits explicit type conversion of variables or expression using the type cast operator. Traditional C casts are augmented in C++ by a function-call notation as syntactic alternative. The following two versions are equivalent:

```
( type-name ) expression        // C notation
type-name ( expression )        // C++ notation
```

Examples:

```
average = sum/ (float) ;        // C notation
average = sum/float (i);        // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type:

Eg: p = (int *) q;

Alternatively, we can use " typedef " to create an identifier of the required type and use it in the functional notation.

typedef int * int_pt;

P = int_pt(q);

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

b) Implicit Type Conversion

We mix data types in expressions. For example,

m = 5 + 2.75;

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion.*

Preprocessor Directives

Preprocessors are programs that process our source code before compilation. Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that, whatever statement starts with #, is going to the preprocessor program, and preprocessor program will execute this statement. Examples of some preprocessor directives are: #include, #define etc. Remember that # symbol only provides a path that it will go to the preprocessor, and command such as include is processed by preprocessor program. For example, include will include extra code to your program.

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directives, like

Using namespace std;

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **using** and **namespace** are the new keywords of c++.

Input/Output Streams and Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are ***endl*** and ***setw.***

The endl manipulators, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n".

The manipulator setw() specifies a field width to be right-shifted. For eg :

```
cout << setw(5) << sum << endl;
```

setw(5) specifies width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

		3	4	5

Function Overloading

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. It means, we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Example:

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

void print(int i)
        {
        cout << " \nThis is int " << i << endl;
        }
        void print(double f)
        {
        cout << " \nThis is float " << f << endl;
        }
        void print(char c)
```

```
{
cout << " \nThis is character  " << c << endl;
}


int main()
{
print(10);
print(10.10);
print('t');
return 0;
}
```

Inline Function

- To eliminate the cost of call of small function, C++ proposes a new feature called inline function
- An inline function is expanded in a line when the function is called or invoked. i.e. the compiler replaces the function call with the corresponding function code.
- All inline function must be defined before they are called.

Syntax:

```
inline return_type function function_name (argument_list)

    {

    Function body

    }
```

- Some of the situation where inlined espressin may not work are;
  - For function returning value, if a loop 'switch' or 'goto' statement.
  - For the function not returning the value if a return statement exist
  - If the function consiss static variable
  - If the inline function are recursive

Example program of inline function:

Default Argument

- Argument is the data passed in the function. Default argument is useful in situation where some argument have same value. For e.g. the bank interest for all customers is same.

- Unlike C, C++ allows us to call a function without specifying all its arguments. IN such cases, the function assigns a default value to the parameter which doesn't has the matching argment in the function call.
- Default values are specified when the function is declared.

Example:

float amount(float principal, int period, float r=0.15)

float value = amount(5000,7); //one argument missing. Explicit value of r =0.15

Example Program of default argument

```
#include<iostream>
using namespace std;
class weight
{
        float w;
        public:
        void process(float m, float g=9.8)          //here g = 9.8 is used as default argument
        {
                w= m*g;
                cout<<"weight="<<w;

        }
};
int main()
{
        weight ob;
        ob.process(10,1.6);
        return 0;
}
```

Call by Reference

Reference Variable

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. Provision of reference variables in C++ permits us to pass parameters to the functions by reference. We can pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

```
void swap ( int &a, int &b)
{       Int t = a;
        a = b;
        b = t;
}
```

Here the function call swap(m, n) will exchange the values of m and n using their aliases a and b. In traditional C, this is accomplished using pointers and indirection as follows:

```
void swap ( int *a, int *b)
{       Int t;
        t = *a;
        *a = *b;
        *b = t;
}
```

This function can be called as:

```
Swap ( &x, &y);
```

Example: Swapping by call by reference

```
#include<iostream>
using namespace std;
void swap ( int &a, int &b)
        {       int t = a;
                a = b;
                b = t;
        }

int main()
{       int m = 10;
        int n = 20;
        cout<<"\nBefore swap  "<<"m= "<<m<<"n= "<<n;
        swap(m, n);
        cout<<"\nAfter swap   "<<"m= "<<m<<"n= "<<n;
        return 0;
}
```

Example of reference Variable

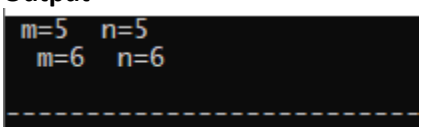
```
#include<iostream>
using namespace std;

int main()
{
int m=5;
int &n=m;

cout<<"m="<<m<<"  n="<<n<<endl;
n++;
//n is reference of m so m and n have now same value
cout<<"m="<<m<<"  n="<<n<<endl;

}
```

Output



Example of Pass by reference

```
#include<iostream>

using namespace std;

        void get(int &m) //m is reference variable

        {

                m++;

        }

int main()

{

        int a=100;

        get(a);

        cout<<"a ="<<a; //prints 101

}
```

Output:

Return by reference

A function can also return a reference. Consider the following function:

```
int & max (int &x, int &y)
{
        If ( x > y)
                return x;
        else
                return y;
}
```

Since the return value max( ) is int &, the function returns reference to x or y (and not the values). Then a function call such as max (a, b) will yield a reference to either a or b depending on their values.

Pointer variable declaration & initialization

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is

```
type *variable_name;
```

There are few important operations, which we will do with the pointers very frequently. **a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

```
#include <iostream>
using namespace std;
int main () {
  int  var = 20;   // actual variable declaration.
  int  *ptr_var;       // pointer variable

  ptr_var = &var;      // store address of var in pointer variable

  cout << "Value of var variable: ";
  cout << var << endl;

  cout << "Address stored in ptr_var variable: ";
  cout << ptr_var << endl;
```

```
   cout << "Value of *ptr_var variable: ";
   cout << *ptr_var << endl;

   return 0;
}
```