C++ Quick Guide

Previous Page
Next Page

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

C++ Compiler:

This is actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use .cpp by default

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have respective Operating Systems.

C++ Program Structure:

Let us look at a simple code that would print the words *Hello World*.

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

// main() is where program execution begins.

int main()
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

Comments in C++

C++ supports single line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */. For example:

```
/* This is a comment */

/* C++ comments can  also
 * span multiple lines
 */
```

A comment can also start with //, extending to the end of the line. For example:

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

main()
{
   cout << "Hello World"; // prints Hello World

   return 0;
}
```

C++ Primitive Built-in Types:

C++ offer the programmer a rich assortment of built-in as well as user-defined data types. Following table list down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Variable Definition & Initialization in C++:

Some examples are:

```
extern int d, f       // declaration of d and f
int d = 3, f = 5;     // definition and initializing d and f.
byte z = 22;          // definition and initializes z.
char x = 'x';         // the variable x has the value 'x'.
```

C++ Variable Scope:

A scope is a region of the program and broadly speaking there are three places where variables can be declared:

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

C++ Constants/Literals:

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

C++ Modifier Types:

C++ allows the **char, int,** and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here:

- signed
- unsigned
- long
- short

The modifiers **signed, unsigned, long,** and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.

The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned, short,** or **long** integers. You can simply use the word **unsigned, short,** or **long**, without the int. The int is implied. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
unsigned int y;
```

Storage Classes in C++:

A storage class defines the scope (visibility) and life time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes which can be used in a C++ Program

- auto

- register
- static
- extern
- mutable

C++ Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides following type of operators:

- Arithmetic Operators (+, -, \, *, ++, --)
- Relational Operators (==, !=, >. <, >=, <=)
- Logical Operators (&&, ||, !)
- Bitwise Operators (& |, ^, ~, <<, >>)
- Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=)
- Misc Operators (sizeof, & cast, comma, conditional etc.)

C++ Loop Types:

C++ programming language provides the following types of loops to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

C++ Decision Making:

C++ programming language provides following types of decision making statements. Click the following links to check their detail.

Statement	Description

if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
nested switch statements	You can use one swicth statement inside another switch statement(s).

C++ Functions:

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Numbers in C++:

Following a simple example to show few of the mathematical operations on C++ numbers:

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

int main ()
{
   // number definition:
   short  s = 10;
   int    i = -1000;
   long   l = 100000;
```

```
   float  f = 230.47;
   double d = 200.374;

   // mathematical operations;
   cout << "sin(d) :" << sin(d) << endl;
   cout << "abs(i)  :" << abs(i) << endl;
   cout << "floor(d) :" << floor(d) << endl;
   cout << "sqrt(f) :" << sqrt(f) << endl;
   cout << "pow( d, 2) :" << pow(d, 2) << endl;

   return 0;
}
```

C++ Arrays:

Following is an example, which will show array declaration, assignment and accessing arrays in C++:

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

#include <iomanip>
using std::setw;

int main ()
{
   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; // set element at location i to i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
   for ( int j = 0; j < 10; j++ )
   {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

C++ Strings:

C++ provides following two types of string representations:

The C-style character string as follows:

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality. Following is the example:

```
#include <iostream>
#include <string>

using namespace std;

int main ()
{
   string str1 = "Hello";
   string str2 = "World";
   string str3;

   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;

   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;

   return 0;
}
```

C++ Classes & Objects

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example we defined the Box data type using the keyword **class** as follows:

```
class Box
{
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try following example to make the things clear:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};

int main( )
{
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

C++ Inheritance:

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer

can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);
```

```
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

C++ Overloading

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

Following is the example where same function **print()** is being used to print different data types:

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

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;

   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Polymorphism in C++

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes and area() method has been implemented by the two sub-classes with different implementation.

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

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
```

```
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

Data Abstraction in C++:

Data abstraction refers to, providing only essential information to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

For example, in C++ we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( )
{
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need only know the public interface and the underlying implementation of cout is free to change.

Data Encapsulation in C++:

All C++ programs are composed of following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:

```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

C++ Files and Streams:

The **iostream** standard library **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

To read and write from a file requires another standard C++ library called **fstream** which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

Following is the C++ program, which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

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

int main ()
{

   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;
```

```
   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

Previous Page
Print
PDF
Next Page
