Array :

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

**Declaring Arrays:**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

**Initializing Arrays:**

You can initialize C++ array elements either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

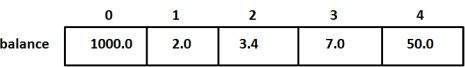
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above:



**Accessing Array Elements:**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

#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;

}

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result:

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

**C++ Arrays in Detail:**

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| [Multi-dimensional arrays](http://www.tutorialspoint.com/cplusplus/cpp_multi_dimensional_arrays.htm) | C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| [Pointer to an array](http://www.tutorialspoint.com/cplusplus/cpp_pointer_to_an_array.htm) | You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |
| [Passing arrays to functions](http://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm) | You can pass to the function a pointer to an array by specifying the array's name without an index. |
| [Return array from functions](http://www.tutorialspoint.com/cplusplus/cpp_return_arrays_from_functions.htm) | C++ allows a function to return an array. |

Function :

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

**Defining a Function:**

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.

**Example:**

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

// function returning the max between two numbers

int max(int num1, int num2)

{

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

**Function Declarations:**

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return\_type function\_name( parameter list );

For the above defined function max(), following is the function declaration:

int max(int num1, int num2);

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration:

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

**Calling a Function:**

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

#include <iostream>

using namespace std;

// function declaration

int max(int num1, int num2);

int main ()

{

// local variable declaration:

int a = 100;

int b = 200;

int ret;

// calling a function to get max value.

ret = max(a, b);

cout << "Max value is : " << ret << endl;

return 0;

}

// function returning the max between two numbers

int max(int num1, int num2)

{

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

Max value is : 200

**Function Arguments:**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

|  |  |
| --- | --- |
| **Call Type** | **Description** |
| [Call by value](http://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm) | This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| [Call by pointer](http://www.tutorialspoint.com/cplusplus/cpp_function_call_by_pointer.htm) | This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |
| [Call by reference](http://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm) | This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

**Default Values for Parameters:**

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

#include <iostream>

using namespace std;

int sum(int a, int b=20)

{

int result;

result = a + b;

return (result);

}

int main ()

{

// local variable declaration:

int a = 100;

int b = 200;

int result;

// calling a function to add the values.

result = sum(a, b);

cout << "Total value is :" << result << endl;

// calling a function again as follows.

result = sum(a);

cout << "Total value is :" << result << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Total value is :300

Total value is :120

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.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

**Base & Derived Classes:**

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;

}

When the above code is compiled and executed, it produces the following result:

Total area: 35

**Access Control and Inheritance:**

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

A derived class inherits all base class methods with the following exceptions:

* Constructors, destructors and copy constructors of the base class.
* Overloaded operators of the base class.
* The friend functions of the base class.

**Type of Inheritance:**

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

* **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
* **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
* **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

**Multiple Inheritances:**

A C++ class can inherit members from more than one class and here is the extended syntax:

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

#include <iostream>

using namespace std;

// Base class Shape

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost

{

public:

int getCost(int area)

{

return area \* 70;

}

};

// Derived class

class Rectangle: public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Total area: 35

Total paint cost: $2450

# Polymorphism in C++

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

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:

#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;

}

When the above code is compiled and executed, it produces the following result:

Parent class area

Parent class area

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

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

**Virtual Function:**

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

**Pure Virtual Functions:**

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

# Data Encapsulation in C++

All C++ programs are composed of the 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**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

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

};

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

**Data Encapsulation Example:**

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

#include <iostream>

using namespace std;

class Adder{

public:

// constructor

Adder(int i = 0)

{

total = i;

}

// interface to outside world

void addNum(int number)

{

total += number;

}

// interface to outside world

int getTotal()

{

return total;

};

private:

// hidden data from outside world

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

**Designing Strategy:**

Most of us have learned through bitter experience to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This wisdom is applied most frequently to data members, but it applies equally to all members, including virtual functions.

# Data Abstraction in C++

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., 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.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

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 to only know the public interface and the underlying implementation of cout is free to change.

**Access Labels Enforce Abstraction:**

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

* Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
* Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

**Benefits of Data Abstraction:**

Data abstraction provides two important advantages:

* Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
* The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

**Data Abstraction Example:**

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

#include <iostream>

using namespace std;

class Adder{

public:

// constructor

Adder(int i = 0)

{

total = i;

}

// interface to outside world

void addNum(int number)

{

total += number;

}

// interface to outside world

int getTotal()

{

return total;

};

private:

// hidden data from outside world

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

**Designing Strategy:**

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

# C++ Overloading (Operator and Function)

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.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

**Function overloading in C++:**

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

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;

}

When the above code is compiled and executed, it produces the following result:

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

**Operators overloading in C++:**

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

#include <iostream>

using namespace std;

class Box

{

public:

double getVolume(void)

{

return length \* breadth \* height;

}

void setLength( double len )

{

length = len;

}

void setBreadth( double bre )

{

breadth = bre;

}

void setHeight( double hei )

{

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b)

{

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main( )

{

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

**Overloadable/Non-overloadableOperators:**

Following is the list of operators which can be overloaded:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | delete | delete [] |

Following is the list of operators, which can not be overloaded:

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

# C++ Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



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

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [while loop](http://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [for loop](http://www.tutorialspoint.com/cplusplus/cpp_for_loop.htm) | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [do...while loop](http://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm) | Like a while statement, except that it tests the condition at the end of the loop body |
| [nested loops](http://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm) | You can use one or more loop inside any another while, for or do..while loop. |

**Loop Control Statements:**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [break statement](http://www.tutorialspoint.com/cplusplus/cpp_break_statement.htm) | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| [continue statement](http://www.tutorialspoint.com/cplusplus/cpp_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| [goto statement](http://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm) | Transfers control to the labeled statement. Though it is not advised to use goto statement in your program. |

**The Infinite Loop:**

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

#include <iostream>

using namespace std;

int main ()

{

for( ; ; )

{

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the for(;;) construct to signify an infinite loop.

**NOTE:** You can terminate an infinite loop by pressing Ctrl + C keys.