# C++ Object Oriented

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of object-oriented programming:

Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction

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.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

# C++ Basic Syntax

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods and Instance variables mean.

* **Object** − Objects have Properties and Behaviors. Example: A dog has Properties - color, name, breed as well as Behaviors - wagging, barking, eating. An object is an instance of a class.
* **Class** − A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
* **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
* **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

## 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;

}

Let us look various parts of the above program:

* The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
* The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
* The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
* The line **int main()** is the main function where program execution begins.
* The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.
* The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

## Compile & Execute C++ Program

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

* Open a text editor and add the code as above.
* Save the file as: hello.cpp
* Open a command prompt and go to the directory where you saved the file.
* Type 'g++ hello.cpp ' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate a.out executable file.
* Now, type ' a.out' to run your program.
* You will be able to see ' Hello World ' printed on the window.

$ g++ hello.cpp

$ ./a.out

Hello World

Make sure that g++ is in your path and that you are running it in the directory containing file hello.cpp.

You can compile C/C++ programs using makefile. For more details, you can check Makefile Tutorial.

## Semicolons & Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements −

x = y;

y = y+1;

add(x, y);

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

{

cout << "Hello World"; // prints Hello World

return 0;

}

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where on a line you put a statement. For example:

x = y;

y = y+1;

add(x, y);

is the same as

x = y; y = y+1; add(x, y);

## C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers −

mohd zara abc move\_name a\_123

myname50 \_temp j a23b9 retVal

## C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| asm | Else | new | this |
| auto | Enum | operator | throw |
| bool | Explicit | private | true |
| break | Export | protected | try |
| case | Extern | public | typedef |
| catch | False | register | typeid |
| char | Float | reinterpret\_cast | typename |
| class | For | return | union |
| const | Friend | short | unsigned |
| const\_cast | Goto | signed | using |
| continue | If | sizeof | virtual |
| default | Inline | static | void |
| delete | Int | static\_cast | volatile |
| do | Long | struct | wchar\_t |
| double | Mutable | switch | while |
| dynamic\_cast | namespace | template |  |

## Trigraphs

A few characters have an alternative representation, called a trigraph sequence. A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.

Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.

Following are most frequently used trigraph sequences −

|  |  |
| --- | --- |
| **Trigraph** | **Replacement** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

## Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it.

Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the statement,

int age;

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the statement

fruit = apples + oranges; // Get the total fruit

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

CLASS AND OBJECT

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

## C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

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 the 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;

}

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

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

## Classes & Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| **Class member functions** | A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. |
| **Class access modifiers** | A class member can be defined as public, private or protected. By default members would be assumed as private. |
| **Constructor & destructor** | A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| **C++ copy constructor** | The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| **C++ friend functions** | A **friend** function is permitted full access to private and protected members of a class. |
| **C++ inline functions** | With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| **The this pointer in C++** | Every object has a special pointer **this** which points to the object itself. |
| **Pointer to C++ classes** | A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. |
| **Static members of a class** | Both data members and function members of a class can be declared as static. |

# 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.

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

# 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:

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

## Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

|  |  |
| --- | --- |
| **S.N.** | **Operators and Example** |
| 1 | **Unary operators overloading** |
| 2 | **Binary operators overloading** |
| 3 | **Relational operators overloading** |
| 4 | **Input/Output operators overloading** |
| 5 | **++ and -- operators overloading** |
| 6 | **Assignment operators overloading** |
| 7 | **Function call () operator overloading** |
| 8 | **Subscripting [] operator overloading** |
| 9 | **Class member access operator -> overloading** |

# 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 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.

# 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.

# Interfaces in C++ (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

class Box {

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

## Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

// pure virtual function providing interface framework.

virtual int getArea() = 0;

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived classes

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

class Triangle: public Shape {

public:

int getArea() {

return (width \* height)/2;

}

};

int main(void) {

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

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

Tri.setWidth(5);

Tri.setHeight(7);

// Print the area of the object.

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

return 0;

}

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

Total Rectangle area: 35

Total Triangle area: 17

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

## Designing Strategy

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly.

The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.

This architecture also allows new applications to be added to a system easily, even after the system has been defined.

# C++ Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

* **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
* **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

// protected code

}catch( ExceptionName e1 )

{

// catch block

}catch( ExceptionName e2 )

{

// catch block

}catch( ExceptionName eN )

{

// catch block

}

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

try {

// protected code

}catch( ExceptionName e ) {

// code to handle ExceptionName exception

}

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

try {

// protected code

}catch(...) {

// code to handle any exception

}

The following is an example, which throws a division by zero exception and we catch it in catch block.

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

}catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

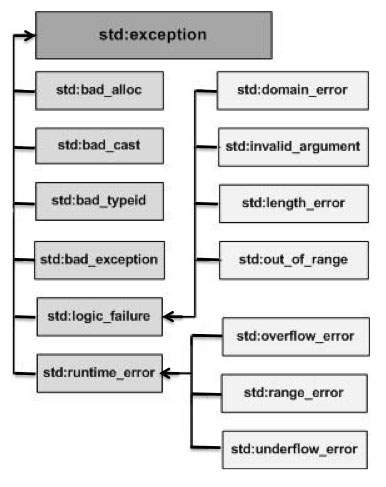
}

Because we are raising an exception of type **const char\***, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

|  |  |
| --- | --- |
| **Exception** | **Description** |
| **std::exception** | An exception and parent class of all the standard C++ exceptions. |
| std::bad\_alloc | This can be thrown by **new**. |
| std::bad\_cast | This can be thrown by **dynamic\_cast**. |
| std::bad\_exception | This is useful device to handle unexpected exceptions in a C++ program |
| std::bad\_typeid | This can be thrown by **typeid**. |
| **std::logic\_error** | An exception that theoretically can be detected by reading the code. |
| std::domain\_error | This is an exception thrown when a mathematically invalid domain is used |
| std::invalid\_argument | This is thrown due to invalid arguments. |
| std::length\_error | This is thrown when a too big std::string is created |
| std::out\_of\_range | This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[](). |
| **std::runtime\_error** | An exception that theoretically can not be detected by reading the code. |
| std::overflow\_error | This is thrown if a mathematical overflow occurs. |
| std::range\_error | This is occured when you try to store a value which is out of range. |
| std::underflow\_error | This is thrown if a mathematical underflow occurs. |

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception {

const char \* what () const throw () {

return "C++ Exception";

}

};

int main() {

try {

throw MyException();

}catch(MyException& e) {

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

} catch(std::exception& e) {

//Other errors

}

}

This would produce the following result:

MyException caught

C++ Exception

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

# C++ Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts:

* **The stack:** All variables declared inside the function will take up memory from the stack.
* **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

## The new and delete operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements:

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

double\* pvalue = NULL;

if( !(pvalue = new double )) {

cout << "Error: out of memory." <<endl;

exit(1);

}

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the delete operator as follows:

delete pvalue; // Release memory pointed to by pvalue

Let us put above concepts and form the following example to show how new and delete work:

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}

If we compile and run above code, this would produce the following result:

Value of pvalue : 29495

## Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

char\* pvalue = NULL; // Pointer initialized with null

pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this:

delete [] pvalue; // Delete array pointed to by pvalue

Following is the syntax of new operator for a multi-dimensional array as follows:

int ROW = 2;

int COL = 3;

double \*\*pvalue = new double\* [ROW]; // Allocate memory for rows

// Now allocate memory for columns

for(int i = 0; i < COL; i++) {

pvalue[i] = new double[COL];

}

The syntax to release the memory for multi-dimensional will be as follows:

for(int i = 0; i < ROW; i++) {

delete[] pvalue[i];

}

delete [] pvalue;

## Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

#include <iostream>

using namespace std;

class Box {

public:

Box() {

cout << "Constructor called!" <<endl;

}

~Box() {

cout << "Destructor called!" <<endl;

}

};

int main( ) {

Box\* myBoxArray = new Box[4];

delete [] myBoxArray; // Delete array

return 0;

}

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:

Constructor called!

Constructor called!

Constructor called!

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!

# C++ Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

## Function Template

The general form of a template function definition is shown here:

template <class type> ret-type func-name(parameter list) {

// body of function

}

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

#include <iostream>

#include <string>

using namespace std;

template <typename T>

inline T const& Max (T const& a, T const& b) {

return a < b ? b:a;

}

int main () {

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

If we compile and run above code, this would produce the following result:

Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World

## Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

template <class type> class class-name {

.

.

.

}

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace std;

template <class T>

class Stack {

private:

vector<T> elems; // elements

public:

void push(T const&); // push element

void pop(); // pop element

T top() const; // return top element

bool empty() const{ // return true if empty.

return elems.empty();

}

};

template <class T>

void Stack<T>::push (T const& elem) {

// append copy of passed element

elems.push\_back(elem);

}

template <class T>

void Stack<T>::pop () {

if (elems.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

}

// remove last element

elems.pop\_back();

}

template <class T>

T Stack<T>::top () const {

if (elems.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

// return copy of last element

return elems.back();

}

int main() {

try {

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate int stack

intStack.push(7);

cout << intStack.top() <<endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

}catch (exception const& ex) {

cerr << "Exception: " << ex.what() <<endl;

return -1;

}

}

If we compile and run above code, this would produce the following result:

7

hello

Exception: Stack<>::pop(): empty stack

# C++ Multithreading

Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

This tutorial assumes that you are working on Linux OS and we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

## Creating Threads

There is following routine which we use to create a POSIX thread:

#include <pthread.h>

pthread\_create (thread, attr, start\_routine, arg)

Here, **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| thread | An opaque, unique identifier for the new thread returned by the subroutine. |
| attr | An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. |
| start\_routine | The C++ routine that the thread will execute once it is created. |
| arg | A single argument that may be passed to start\_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed. |

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

## Terminating Threads

There is following routine which we use to terminate a POSIX thread:

#include <pthread.h>

pthread\_exit (status)

Here **pthread\_exit** is used to explicitly exit a thread. Typically, the pthread\_exit() routine is called after a thread has completed its work and is no longer required to exist.

If main() finishes before the threads it has created, and exits with pthread\_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

## Example

This simple example code creates 5 threads with the pthread\_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread\_exit().

#include <iostream>

#include <cstdlib>

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid) {

long tid;

tid = (long)threadid;

cout << "Hello World! Thread ID, " << tid << endl;

pthread\_exit(NULL);

}

int main () {

pthread\_t threads[NUM\_THREADS];

int rc;

int i;

for( i=0; i < NUM\_THREADS; i++ ){

cout << "main() : creating thread, " << i << endl;

rc = pthread\_create(&threads[i], NULL, PrintHello, (void \*)i);

if (rc){

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

Compile the following program using -lpthread library as follows:

$gcc test.cpp -lpthread

Now, execute your program which should generate result something as follows:

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Hello World! Thread ID, 0

Hello World! Thread ID, 1

Hello World! Thread ID, 2

Hello World! Thread ID, 3

Hello World! Thread ID, 4

## Passing Arguments to Threads

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example:

#include <iostream>

#include <cstdlib>

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

struct thread\_data{

int thread\_id;

char \*message;

};

void \*PrintHello(void \*threadarg) {

struct thread\_data \*my\_data;

my\_data = (struct thread\_data \*) threadarg;

cout << "Thread ID : " << my\_data->thread\_id ;

cout << " Message : " << my\_data->message << endl;

pthread\_exit(NULL);

}

int main () {

pthread\_t threads[NUM\_THREADS];

struct thread\_data td[NUM\_THREADS];

int rc;

int i;

for( i=0; i < NUM\_THREADS; i++ ){

cout <<"main() : creating thread, " << i << endl;

td[i].thread\_id = i;

td[i].message = "This is message";

rc = pthread\_create(&threads[i], NULL, PrintHello, (void \*)&td[i]);

if (rc){

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

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

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Thread ID : 3 Message : This is message

Thread ID : 2 Message : This is message

Thread ID : 0 Message : This is message

Thread ID : 1 Message : This is message

Thread ID : 4 Message : This is message

## Joining and Detaching Threads

There are following two routines which we can use to join or detach threads:

pthread\_join (threadid, status)

pthread\_detach (threadid)

The pthread\_join() subroutine blocks the calling thread until the specified threadid thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

This example demonstrates how to wait for thread completions by using the Pthread join routine.

#include <iostream>

#include <cstdlib>

#include <pthread.h>

#include <unistd.h>

using namespace std;

#define NUM\_THREADS 5

void \*wait(void \*t) {

int i;

long tid;

tid = (long)t;

sleep(1);

cout << "Sleeping in thread " << endl;

cout << "Thread with id : " << tid << " ...exiting " << endl;

pthread\_exit(NULL);

}

int main () {

int rc;

int i;

pthread\_t threads[NUM\_THREADS];

pthread\_attr\_t attr;

void \*status;

// Initialize and set thread joinable

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

for( i=0; i < NUM\_THREADS; i++ ){

cout << "main() : creating thread, " << i << endl;

rc = pthread\_create(&threads[i], &attr, wait, (void \*)i );

if (rc){

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

// free attribute and wait for the other threads

pthread\_attr\_destroy(&attr);

for( i=0; i < NUM\_THREADS; i++ ){

rc = pthread\_join(threads[i], &status);

if (rc){

cout << "Error:unable to join," << rc << endl;

exit(-1);

}

cout << "Main: completed thread id :" << i ;

cout << " exiting with status :" << status << endl;

}

cout << "Main: program exiting." << endl;

pthread\_exit(NULL);

}

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

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Sleeping in thread

Thread with id : 0 .... exiting

Sleeping in thread

Thread with id : 1 .... exiting

Sleeping in thread

Thread with id : 2 .... exiting

Sleeping in thread

Thread with id : 3 .... exiting

Sleeping in thread

Thread with id : 4 .... exiting

Main: completed thread id :0 exiting with status :0

Main: completed thread id :1 exiting with status :0

Main: completed thread id :2 exiting with status :0

Main: completed thread id :3 exiting with status :0

Main: completed thread id :4 exiting with status :0

Main: program exiting.

# C++ Web Programming

What is CGI?

* The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script.
* The CGI specs are currently maintained by the NCSA and NCSA defines CGI is as follows:
* The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
* The current version is CGI/1.1 and CGI/1.2 is under progress.

Web Browsing

To understand the concept of CGI, let's see what happens when we click a hyperlink to browse a particular web page or URL.

* Your browser contacts the HTTP web server and demand for the URL ie. filename.
* Web Server will parse the URL and will look for the filename. If it finds requested file then web server sends that file back to the browser otherwise sends an error message indicating that you have requested a wrong file.
* Web browser takes response from web server and displays either the received file or error message based on the received response.

However, it is possible to set up the HTTP server in such a way that whenever a file in a certain directory is requested, that file is not sent back; instead it is executed as a program, and produced output from the program is sent back to your browser to display.

The Common Gateway Interface (CGI) is a standard protocol for enabling applications (called CGI programs or CGI scripts) to interact with Web servers and with clients. These CGI programs can be a written in Python, PERL, Shell, C or C++ etc.

CGI Architecture Diagram

The following simple program shows a simple architecture of CGI:



Web Server Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI directory and by convention it is named as /var/www/cgi-bin. By convention CGI files will have extension as **.cgi**, though they are C++ executable.

By default, Apache Web Server is configured to run CGI programs in /var/www/cgi-bin. If you want to specify any other directory to run your CGI scripts, you can modify the following section in the httpd.conf file:

<Directory "/var/www/cgi-bin">

AllowOverride None

Options ExecCGI

Order allow,deny

Allow from all

</Directory>

<Directory "/var/www/cgi-bin">

Options All

</Directory>

Here, I assumed that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell etc.

First CGI Program

Consider the following C++ Program content:

#include <iostream>

using namespace std;

int main () {

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Hello World - First CGI Program</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<h2>Hello World! This is my first CGI program</h2>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Output

<html>

<head>

<title>Hello World - First CGI Program</title>

</head>

<body>

<h2>Hello World! This is my first CGI program</h2>

</body>

</html>

Compile above code and name the executable as cplusplus.cgi. This file is being kept in /var/www/cgi-bin directory and it has following content. Before running your CGI program make sure you have change mode of file using **chmod 755 cplusplus.cgi** UNIX command to make file executable.

Hello World! This is my first CGI program

Above C++ program is a simple program which is writing its output on STDOUT file ie. screen. There is one important and extra feature available which is first line to be printed **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and specify the content type to be displayed on the browser screen. Now you must have understood basic concept of CGI and you can write many complicated CGI programs using Python. A C++ CGI program can interact with any other exernal system, such as RDBMS, to exchange information.

HTTP Header

The line **Content-type:text/html\r\n\r\n** is part of HTTP header, which is sent to the browser to understand the content. All the HTTP header will be in the following form

HTTP Field Name: Field Content

For Example

Content-type: text/html\r\n\r\n

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

|  |  |
| --- | --- |
| **Header** | **Description** |
| Content-type: | A MIME string defining the format of the file being returned. Example is Content-type:text/html |
| Expires: Date | The date the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. A valid date string should be in the format 01 Jan 1998 12:00:00 GMT. |
| Location: URL | The URL that should be returned instead of the URL requested. You can use this filed to redirect a request to any file. |
| Last-modified: Date | The date of last modification of the resource. |
| Content-length: N | The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file. |
| Set-Cookie: String | Set the cookie passed through the *string* |

CGI Environment Variables

All the CGI program will have access to the following environment variables. These variables play an important role while writing any CGI program.

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| CONTENT\_TYPE | The data type of the content. Used when the client is sending attached content to the server. For example file upload etc. |
| CONTENT\_LENGTH | The length of the query information. It's available only for POST requests |
| HTTP\_COOKIE | Return the set cookies in the form of key & value pair. |
| HTTP\_USER\_AGENT | The User-Agent request-header field contains information about the user agent originating the request. Its name of the web browser. |
| PATH\_INFO | The path for the CGI script. |
| QUERY\_STRING | The URL-encoded information that is sent with GET method request. |
| REMOTE\_ADDR | The IP address of the remote host making the request. This can be useful for logging or for authentication purpose. |
| REMOTE\_HOST | The fully qualified name of the host making the request. If this information is not available then REMOTE\_ADDR can be used to get IR address. |
| REQUEST\_METHOD | The method used to make the request. The most common methods are GET and POST. |
| SCRIPT\_FILENAME | The full path to the CGI script. |
| SCRIPT\_NAME | The name of the CGI script. |
| SERVER\_NAME | The server's hostname or IP Address |
| SERVER\_SOFTWARE | The name and version of the software the server is running. |

Here is small CGI program to list out all the CGI variables. Click this link to see the result Get Environment

#include <iostream>

#include <stdlib.h>

using namespace std;

const string ENV[ 24 ] = {

"COMSPEC", "DOCUMENT\_ROOT", "GATEWAY\_INTERFACE",

"HTTP\_ACCEPT", "HTTP\_ACCEPT\_ENCODING",

"HTTP\_ACCEPT\_LANGUAGE", "HTTP\_CONNECTION",

"HTTP\_HOST", "HTTP\_USER\_AGENT", "PATH",

"QUERY\_STRING", "REMOTE\_ADDR", "REMOTE\_PORT",

"REQUEST\_METHOD", "REQUEST\_URI", "SCRIPT\_FILENAME",

"SCRIPT\_NAME", "SERVER\_ADDR", "SERVER\_ADMIN",

"SERVER\_NAME","SERVER\_PORT","SERVER\_PROTOCOL",

"SERVER\_SIGNATURE","SERVER\_SOFTWARE"

};

int main () {

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>CGI Envrionment Variables</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<table border = \"0\" cellspacing = \"2\">";

for ( int i = 0; i < 24; i++ ) {

cout << "<tr><td>" << ENV[ i ] << "</td><td>";

// attempt to retrieve value of environment variable

char \*value = getenv( ENV[ i ].c\_str() );

if ( value != 0 ){

cout << value;

}else{

cout << "Environment variable does not exist.";

}

cout << "</td></tr>\n";

}

cout << "</table><\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

C++ CGI Library

For real examples, you would need to do many operations by your CGI program. There is a CGI library written for C++ program which you can download from ftp://ftp.gnu.org/gnu/cgicc/ and following the following steps to install the library:

$tar xzf cgicc-X.X.X.tar.gz

$cd cgicc-X.X.X/

$./configure --prefix=/usr

$make

$make install

You can check related documentation available at C++ CGI Lib Documentation.

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.

Passing Information using GET method:

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows:

http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation and you can pass upto 1024 characters in a request string.

When using GET method, information is passed using QUERY\_STRING http header and will be accessible in your CGI Program through QUERY\_STRING environment variable

You can pass information by simply concatenating key and value pairs alongwith any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example : Get Method

Here is a simple URL which will pass two values to hello\_get.py program using GET method.

/cgi-bin/cpp\_get.cgi?first\_name=ZARA&last\_name=ALI

Below is program to generate **cpp\_get.cgi** CGI program to handle input given by web browser. We are going to use C++ CGI library which makes it very easy to access passed information:

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Using GET and POST Methods</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("first\_name");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "First name: " << \*\*fi << endl;

}else{

cout << "No text entered for first name" << endl;

}

cout << "<br/>\n";

fi = formData.getElement("last\_name");

if( !fi->isEmpty() &&fi != (\*formData).end()) {

cout << "Last name: " << \*\*fi << endl;

}else{

cout << "No text entered for last name" << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Now, compile the above program as follows:

$g++ -o cpp\_get.cgi cpp\_get.cpp -lcgicc

Generate cpp\_get.cgi and put it in your CGI directory and try to access using following link:

/cgi-bin/cpp\_get.cgi?first\_name=ZARA&last\_name=ALI

This would generate following result:

First name: ZARA

Last name: ALI

Simple FORM Example: GET Method

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same CGI script cpp\_get.cgi to handle this input.

<form action="/cgi-bin/cpp\_get.cgi" method="get">

First Name: <input type="text" name="first\_name"> <br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

Here is the actual output of the above form, You enter First and Last Name and then click submit button to see the result.

Top of Form

First Name:    
Last Name:



Bottom of Form

Passing Information using POST method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

The same cpp\_get.cgi program will handle POST method as well. Let us take same example as above, which passes two values using HTML FORM and submit button but this time with POST method as follows:

<form action="/cgi-bin/cpp\_get.cgi" method="post">

First Name: <input type="text" name="first\_name"><br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

Here is the actual output of the above form, You enter First and Last Name and then click submit button to see the result.

Top of Form

First Name:    
Last Name:



Bottom of Form

Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes

<form action="/cgi-bin/cpp\_checkbox.cgi"

method="POST"

target="\_blank">

<input type="checkbox" name="maths" value="on" /> Maths

<input type="checkbox" name="physics" value="on" /> Physics

<input type="submit" value="Select Subject" />

</form>

The result of this code is the following form

Top of Form

 Maths  Physics



Bottom of Form

Below is C++ program, which will generate cpp\_checkbox.cgi script to handle input given by web browser through checkbox button.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

bool maths\_flag, physics\_flag;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Checkbox Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

maths\_flag = formData.queryCheckbox("maths");

if( maths\_flag ) {

cout << "Maths Flag: ON " << endl;

}else{

cout << "Maths Flag: OFF " << endl;

}

cout << "<br/>\n";

physics\_flag = formData.queryCheckbox("physics");

if( physics\_flag ) {

cout << "Physics Flag: ON " << endl;

}else{

cout << "Physics Flag: OFF " << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio button:

<form action="/cgi-bin/cpp\_radiobutton.cgi"

method="post"

target="\_blank">

<input type="radio" name="subject" value="maths"

checked="checked"/> Maths

<input type="radio" name="subject" value="physics" /> Physics

<input type="submit" value="Select Subject" />

</form>

The result of this code is the following form

Top of Form

 Maths  Physics



Bottom of Form

Below is C++ program, which will generate cpp\_radiobutton.cgi script to handle input given by web browser through radio buttons.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Radio Button Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("subject");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Radio box selected: " << \*\*fi << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box:

<form action="/cgi-bin/cpp\_textarea.cgi"

method="post"

target="\_blank">

<textarea name="textcontent" cols="40" rows="4">

Type your text here...

</textarea>

<input type="submit" value="Submit" />

</form>

The result of this code is the following form

Top of Form



Bottom of Form

Below is C++ program, which will generate cpp\_textarea.cgi script to handle input given by web browser through text area.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Text Area Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("textcontent");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Text Content: " << \*\*fi << endl;

}else{

cout << "No text entered" << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box

<form action="/cgi-bin/cpp\_dropdown.cgi"

method="post" target="\_blank">

<select name="dropdown">

<option value="Maths" selected>Maths</option>

<option value="Physics">Physics</option>

</select>

<input type="submit" value="Submit"/>

</form>

The result of this code is the following form

Top of Form



Bottom of Form

Below is C++ program, which will generate cpp\_dropdown.cgi script to handle input given by web browser through drop down box.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Drop Down Box Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("dropdown");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Value Selected: " << \*\*fi << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Using Cookies in CGI

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

* **Expires :** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
* **Domain :** The domain name of your site.
* **Path :** The path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
* **Secure :** If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
* **Name=Value :** Cookies are set and retrieved in the form of key and value pairs.

Setting up Cookies

This is very easy to send cookies to browser. These cookies will be sent along with HTTP Header before to Content-type filed. Assuming you want to set UserID and Password as cookies. So cookies setting will be done as follows

#include <iostream>

using namespace std;

int main () {

cout << "Set-Cookie:UserID=XYZ;\r\n";

cout << "Set-Cookie:Password=XYZ123;\r\n";

cout << "Set-Cookie:Domain=www.tutorialspoint.com;\r\n";

cout << "Set-Cookie:Path=/perl;\n";

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Cookies in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "Setting cookies" << endl;

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

Here, it is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html\r\n\r\n**.

Compile above program to produce setcookies.cgi, and try to set cookies using following link. It will set four cookies at your computer:

/cgi-bin/setcookies.cgi

Retrieving Cookies

This is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form.

key1=value1;key2=value2;key3=value3....

Here is an example of how to retrieving cookies.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc cgi;

const\_cookie\_iterator cci;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Cookies in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<table border = \"0\" cellspacing = \"2\">";

// get environment variables

const CgiEnvironment& env = cgi.getEnvironment();

for( cci = env.getCookieList().begin();

cci != env.getCookieList().end();

++cci ) {

cout << "<tr><td>" << cci->getName() << "</td><td>";

cout << cci->getValue();

cout << "</td></tr>\n";

}

cout << "</table><\n";

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Now, compile above program to produce getcookies.cgi, and try to get a list of all the cookies available at your computer:

/cgi-bin/getcookies.cgi

This will produce a list of all the four cookies set in previous section and all other cookies set at your computer:

UserID XYZ

Password XYZ123

Domain www.tutorialspoint.com

Path /perl

File Upload Example:

To upload a file the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type will create a "Browse" button.

<html>

<body>

<form enctype="multipart/form-data"

action="/cgi-bin/cpp\_uploadfile.cgi"

method="post">

<p>File: <input type="file" name="userfile" /></p>

<p><input type="submit" value="Upload" /></p>

</form>

</body>

</html>

The result of this code is the following form:

Top of Form

File:



Bottom of Form

**Note:** Above example has been disabled intentionally to save people uploading files on our server. But you can try above code with your server.

Here is the script **cpp\_uploadfile.cpp** to handle file upload:

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc cgi;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>File Upload in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

// get list of files to be uploaded

const\_file\_iterator file = cgi.getFile("userfile");

if(file != cgi.getFiles().end()) {

// send data type at cout.

cout << HTTPContentHeader(file->getDataType());

// write content at cout.

file->writeToStream(cout);

}

cout << "<File uploaded successfully>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

The above example is writing content at **cout** stream but you can open your file stream and save the content of uploaded file in a file at desired location.

Hope you enjoyed this tutorial. If yes, please send me your feedback at: