Object-oriented programming (OOP) is nothing but that which allows the writing of programs with the help of certain classes and real-time objects. We can say that this approach is very close to the real-world and its applications because the state and behaviour of these classes and objects are almost the same as real-world objects.

**What Are Class & Object?**  
It is the basic concept of OOP; an extended concept of the structure used in C. It is an abstract and user-defined data type. It consists of several variables and functions. The primary purpose of the class is to store data and information. The members of a class define the behaviour of the class. A class is the blueprint of the object, but also, we can say the implementation of the class is the object. The class is not visible to the world, but the object is.

By default class variables are private but in case of structure it is public.

**Encapsulation and Data abstraction:** Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Data abstraction refers to, providing only needed information to the outside world and hiding implementation details. For example, consider a class Complex with public functions as getReal() and getImag(). We may implement the class as an array of size 2 or as two variables. The advantage of abstractions is, we can change implementation at any point, users of Complex class wont’t be affected as out method interface remains same.

**Inheritance:** inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.

**Polymorphism:** polymorphism means ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.  
C++ supports operator overloading and function overloading.  
The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.  
Function overloading is using a single function name to perform different types of tasks.  
Polymorphism is extensively used in implementing inheritance.

**Dynamic Binding:** In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has [virtual functions](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/) to support this.

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

Note: Declaring a[friend function](http://quiz.geeksforgeeks.org/friend-class-function-cpp/)is a way to give private access to a non-member function.

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

[**Constructors**](https://www.geeksforgeeks.org/constructors-c/)

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.  
There are 3 types of constructors:

* [Default constructors](http://quiz.geeksforgeeks.org/constructors-c/)
* Parametrized constructors
* [Copy constructors](http://quiz.geeksforgeeks.org/copy-constructor-in-cpp/)

A **Copy Constructor** creates a new object, which is exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.  
Syntax:

class-name (class-name &){}

#include<iostream>

using namespace std;

class Point

{

private:

    int x, y;

public:

    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor

    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }

    int getY()            {  return y; }

};

int main()

{

    Point p1(10, 15); // Normal constructor is called here

    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors

    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

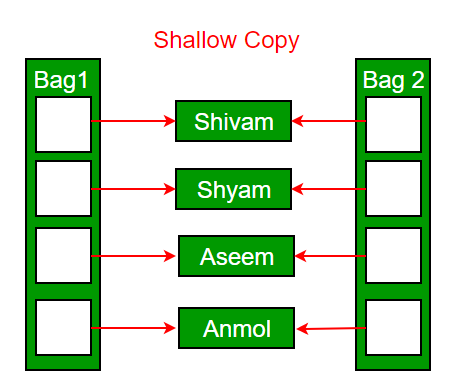
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;

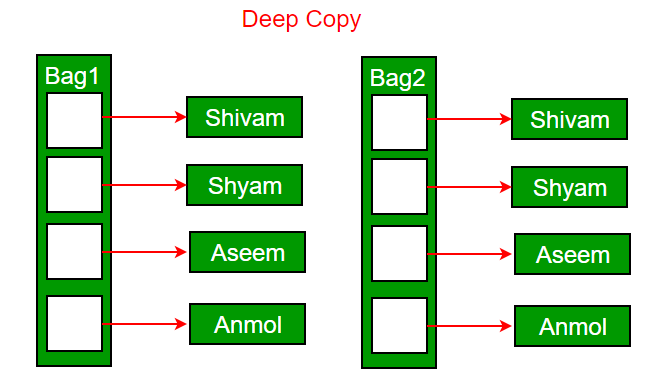
}

**When is copy constructor called?**  
In C++, a Copy Constructor may be called in following cases:  
1. When an object of the class is returned by value.  
2. When an object of the class is passed (to a function) by value as an argument.  
3. When an object is constructed based on another object of the same class.  
4. When the compiler generates a temporary object.

**When is user-defined copy constructor needed?**  
If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection..etc.

***Default constructor does only shallow copy.***  


***Deep copy is possible only with user defined copy constructor.*** In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



**Write an example class where copy constructor is needed?**  
Following is a complete C++ program to demonstrate use of Copy constructor. In the following String class, we must write copy constructor.

#include<iostream>

#include<cstring>

using namespace std;

class String

{

private:

    char \*s;

    int size;

public:

    String(const char \*str = NULL); // constructor

    ~String() { delete [] s;  }// destructor

    String(const String&); // copy constructor

    void print() { cout << s << endl; } // Function to print string

    void change(const char \*);  // Function to change

};

String::String(const char \*str)

{

    size = strlen(str);

    s = new char[size+1];

    strcpy(s, str);

}

void String::change(const char \*str)

{

    delete [] s;

    size = strlen(str);

    s = new char[size+1];

    strcpy(s, str);

}

String::String(const String& old\_str)

{

    size = old\_str.size;

    s = new char[size+1];

    strcpy(s, old\_str.s);

}

int main()

{

    String str1("GeeksQuiz");

    String str2 = str1;

    str1.print(); // what is printed ?

    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?

    str2.print();

    return 0;

}

**copy constructor must be passed as a reference and  copy constructor should be const**

[**Destructors**](https://www.geeksforgeeks.org/destructors-c/)

Destructor is another special member function that is called by the compiler when the scope of the object ends.

**Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

**Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of a class.

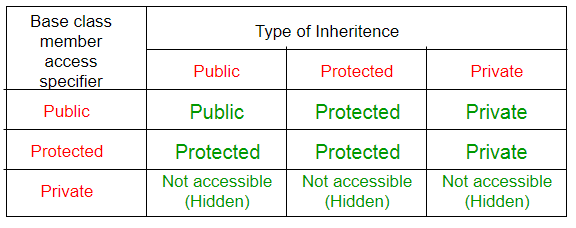
**Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class.

**Modes of Inheritance**

Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.



class A

{

public:

    int x;

protected:

    int y;

private:

    int z;

};

class B : public A

{

    // x is public

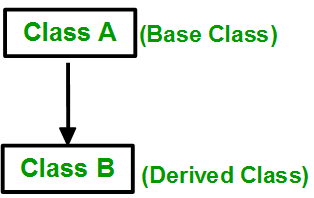
    // y is protected

    // z is not accessible from B

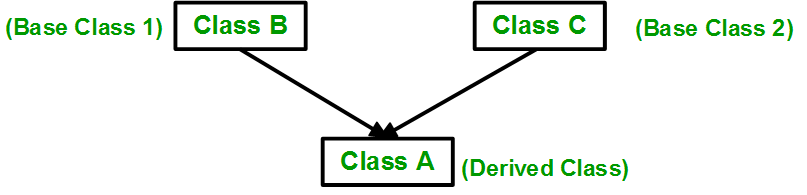
};

**Types of Inheritance in C++**

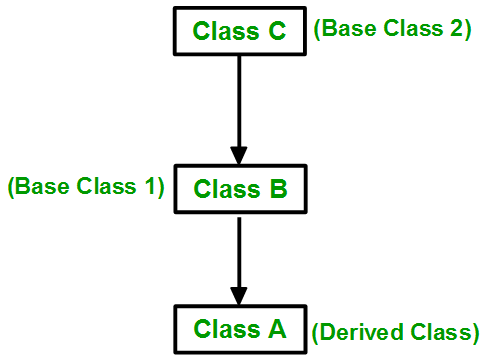
1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



1. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



1. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



1. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



**In C++ polymorphism is mainly divided into two types:**

* Compile time Polymorphism
* Runtime Polymorphism

1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.   
     
   * **Function Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.
   * [**Operator Overloading**](https://www.geeksforgeeks.org/operator-overloading-c/): C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.
2. [**Runtime polymorphism**](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/): This type of polymorphism is achieved by Function Overriding.

**Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Virtual Function in C++**

A virtual function a member function which is declared within a base class and is re-defined (Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve[Runtime polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Functions are declared with a **virtual**keyword in base class.
* The resolving of function call is done at Run-time.

**Rules for Virtual Functions**

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

|  |
| --- |
| // CPP program to illustrate  // concept of Virtual Functions  #include<iostream>  using namespace std;    class base  {  public:      virtual void print ()      { cout<< "print base class" <<endl; }        void show ()      { cout<< "show base class" <<endl; }  };    class derived:public base  {  public:      void print ()      { cout<< "print derived class" <<endl; }        void show ()      { cout<< "show derived class" <<endl; }  };    int main()  {      base \*bptr;      derived d;      bptr = &d;        //virtual function, binded at runtime      bptr->print();        // Non-virtual function, binded at compile time      bptr->show();  } |

Output

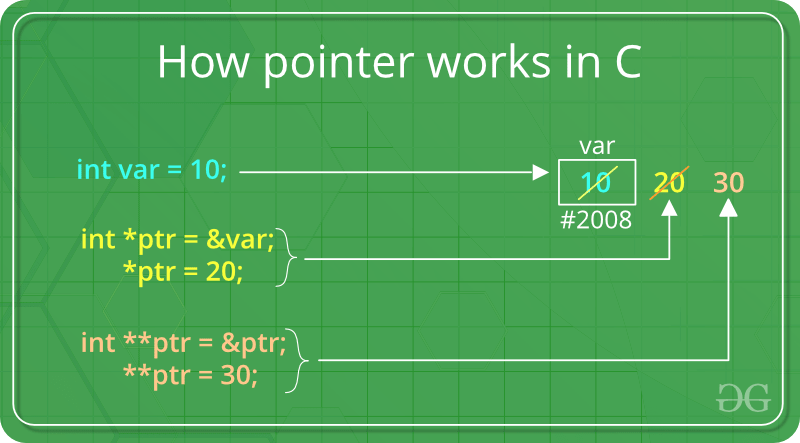
print derived class

show base class

**NOTE:** If we have created a virtual function in the base class and it is being overridden in the derived class then we don’t need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

**Pointers store address of variables or a memory location.**



Pointer arithmetic is meaningless unless performed on an array.  
Note : Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses**.**

**POINTER NOTATION ARRAY NOTATION VALUE**

\*(\*nums) nums[0][0] 16

\*(\*nums + 1) nums[0][1] 18

\*(\*nums + 2) nums[0][2] 20

\*(\*(nums + 1)) nums[1][0] 25

\*(\*(nums + 1) + 1) nums[1][1] 26

\*(\*(nums + 1) + 2) nums[1][2] 27