C++ Classes and Objects

Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

A Class is a user defined data-type which has data members and member functions.

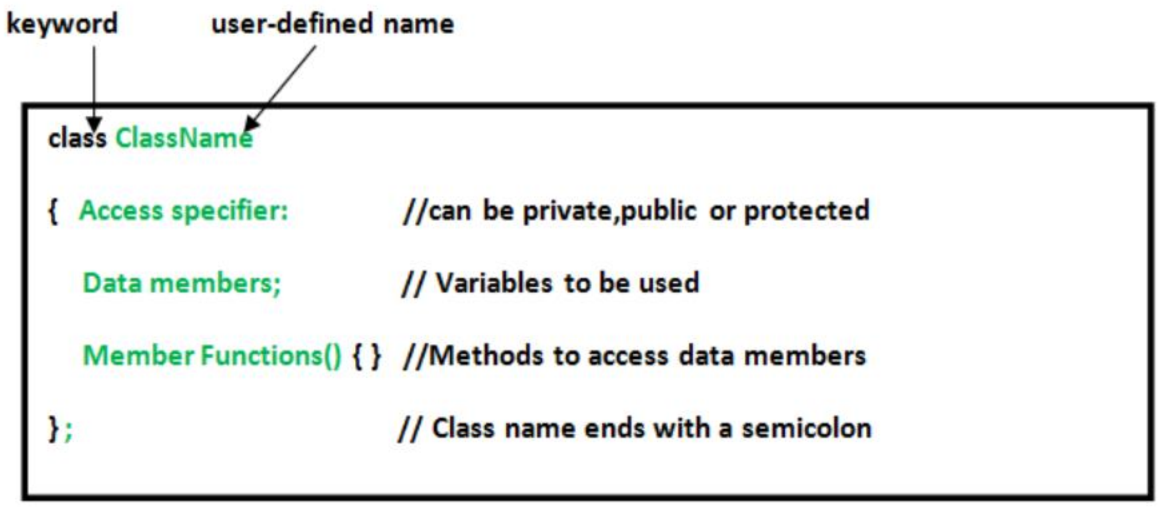
Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
class ClassName
{  Access specifier:           //can be private,public or protected
   Data members;               // Variables to be used
   Member Functions() { }      //Methods to access data members
};                             // Class name ends with a semicolon
```

classes-and-objects-in-c

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers : public, private and protected.

```
// C++ program to demonstrate

// accessing of data members

#include <bits/stdc++.h>

using namespace std;

class Geeks

{

   // Access specifier

   public:

   // Data Members

   string geekname;

   // Member Functions()

   void printname()

   {

     cout << "Geekname is: " << geekname;

   }
};
int main() {
      // Declare an object of class geeks
    Geeks obj1;
      // accessing data member
    obj1.geekname = "Abhi";
      // accessing member function
    obj1.printname();
    return 0;
}
```

Output:

```
Geekname is: Abhi
```

Member Functions in Classes

There are 2 ways to define a member function:

Inside class definition

Outside class definition

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

```
// C++ program to demonstrate function
// declaration outside class
  #include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    string geekname;
    int id;
    // printname is not defined inside class definition
    void printname();
        // printid is defined inside class definition
    void printid()
    {
        cout << "Geek id is: " << id;
    }
};
  // Definition of printname using scope resolution operator ::
void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}
int main() {
        Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;
        // call printname()
    obj1.printname();
    cout << endl;
        // call printid()
    obj1.printid();
    return 0;
}
```

Output:

```
Geekname is: xyz
Geek id is: 15
```

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.

Constructor

Variable initialization in C++

Constructor is used to initialize an object of the class and assign values to data members corresponding to the class. While destructor is used to deallocate the memory of an object of a class. There can be multiple constructors for the same class. In a class, there is always a single destructor. A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

The Class Constructor

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

A constructor in C++ is a special 'MEMBER FUNCTION' having the same name as that of its class which is used to initialize some valid values to the data members of an object. It is executed automatically whenever an object of a class is created. The only restriction that applies to the constructor is that it must not have a return type or void. It is because the constructor is automatically called by the compiler and it is normally used to INITIALIZE VALUES. The compiler distinguishes the constructor from other member functions of a class by its name which is the same as that of its class. ctorz is an abbreviation of constructor in C++.

The syntax for defining constructor inside the class body is as follows:

```
class CLASSNAME
{
   .........
  public :
              CLASSNAME([parameter_list])  // constructor definition
             {
               . . . . . .
             }
             . . . . . . . .
};
```

```
class CLASSNAME
 {
 . . . . . . . .
public:
         CLASSNAME ([parameter_list]);//Constructor declaration
        . . . . . . . . .
};
CLASSNAME: :CLASSNAME([parameter_list])//Constructor Definition
{
. . . . . . . . . . .
}
```

The above syntax shows the declaration and definition of a constructor. It is defined outside the class in the same way as we define a member function outside the class using the scope resolution operator.

One should note that the name of the constructor is the same as that of its class. The constructor parameter list enclosed in the square brackets is optional and may contain zero or more parameters.

We should declare the constructor in the public section of the class as they are invoked automatically when the objects are created.

Types of Constructors in C++

<u>Default Constructor</u>

A constructor to which no arguments are passed is called the Default constructor. It is also called a constructor with no parameters.

Using the default constructor, data members can be initialized to some realistic values in its definition even though no arguments are specified explicitly. Each time an object is created, a constructor is invoked. If we define objects and classes without defining any constructor for a class. So in such a situation, the compiler automatically generates a constructor of its own without any parameters i.e. Default Constructor.

This compiler generates a default constructor which is invoked automatically whenever any object of the class is created but doesn’t perform any initialization. However, if you define a default constructor explicitly, the compiler no longer generates a default constructor for you.

The following are the key points while defining constructors for a class:

A constructor has the same name as that of the class to which it belongs to.

If you don’t openly provide a constructor of your own then the compiler generates a default constructor for you.

A constructor can preferably be used for initialization and not for input/output operations.

Constructors may not be static.

Constructors are also used to locate memory at run time using the new operator.

Constructors cannot be virtual.

A constructor is executed repeatedly whenever the objects of a class are created.

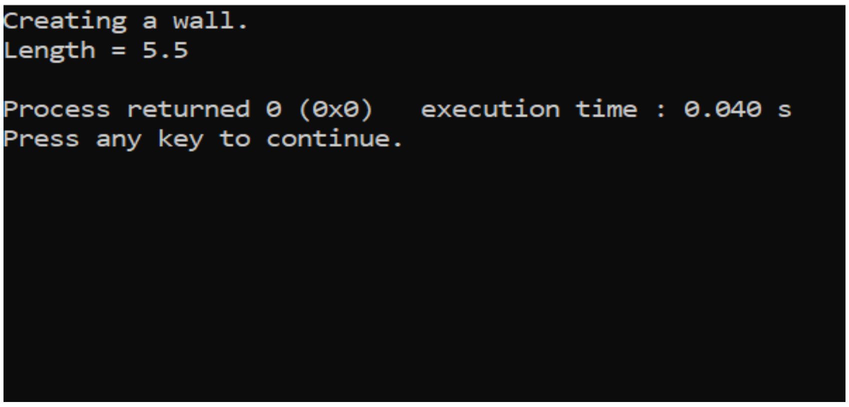
We can declare more than one constructor in a class i.e. constructors can be overloaded.

Sample Program

```
// C++ program to demonstrate the use of default constructor


#include <iostream>

using namespace std;


// declare a class

class  Wall {

  private:

    double length;


  public:

    // default constructor to initialize variable

    Wall() {

      length = 5.5;

      cout << "Creating a wall." << endl;

      cout << "Length = " << length << endl;

    }

};
```

```
int main() {
  Wall wall1;
  return 0;
}
```

Output:



```
Creating a wall.
Length = 5.5

Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

Parameterized Constructor

Unlike default constructors which do not take any parameters, it is however possible to pass one or more arguments to a constructor.

Constructors that can take arguments are known as parameterized constructors.

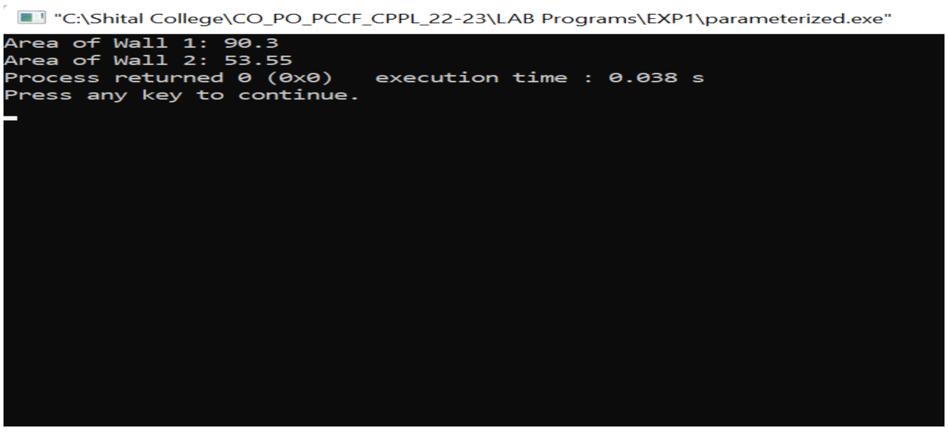
```
class class_name
{

public:
  class_name(variables) //Parameterized constructor declared.
  {

  }
};
```

The syntax for declaring parameterized construct outside the class:

```
class class_name
{
};
class_name :: class_name() //Parameterized constructor declared.
{
}
```

Example : C++ Parameterized Constructor

```
// C++ program to calculate the area of a wall

#include <iostream>
using namespace std;
// declare a class
class Wall {
  private:
    double length;
    double height;
  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }
    double calculateArea() {
      return length * height;
    }
};
int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();
  return 0;
}
```



```
"C:\Shital College\CO_PO_PCCF_CPPL_22-23\LAB Programs\EXP1\parameterized.exe"
Area of Wall 1: 90.3
Area of Wall 2: 53.55
Process returned 0 (0x0)   execution time : 0.038 s
Press any key to continue.
```

Copy Constructor in C++

copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

- Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- Copy constructor takes a reference to an object of the same class as an argument.

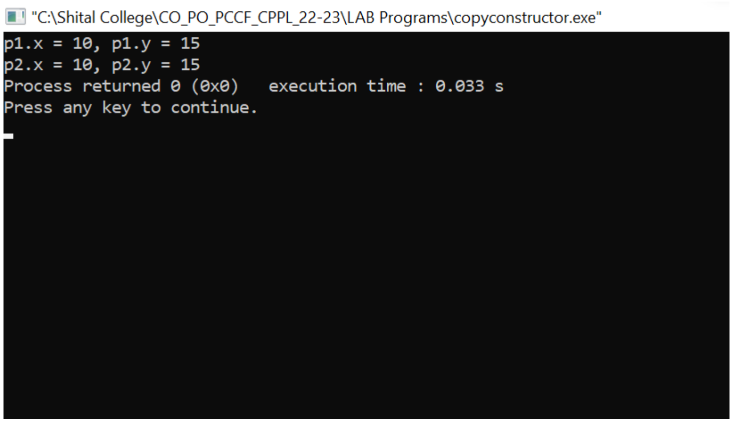
```
Sample(Sample &t)
{
        id=t.id;
}
```

- The process of initializing members of an object through a copy constructor is known as copy initialization.

- It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.
- The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

```
// C++ program to demonstrate the working of a COPY CONSTRUCTOR
#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& p1)
    {
        x = p1.x;
        y = p1.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;
}
```

Output:



```
"C:\Shital College\CO_PO_PCCF_CPPL_22-23\LAB Programs\copyconstructor.exe"
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

Finalization in C++

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object

What is a destructor?

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor.

Destructor has the same name as their class name preceded by a tiled (~) symbol.

It is not possible to define more than one destructor.

The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded.

Destructor neither requires any argument nor returns any value.

It is automatically called when object goes out of scope.

Destructor release memory space occupied by the objects created by constructor.

In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

Syntax:

Syntax for defining the destructor within the class

```
~ <class-name>()
{


}
```

Syntax for defining the destructor outside the class

```
<class-name>: : ~ <class-name>()
{
}
```

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

Parameterized constructors

Copy constructors

```
// C++ program to demonstrate constructors


#include <bits/stdc++.h>

using namespace std;

class Geeks

{

        public:

        int id;


        //Default Constructor

        Geeks()

        {

                cout << "Default Constructor called" << endl;

                id=-1;

        }


        //Parameterized Constructor

        Geeks(int x)

        {
                cout << "Parameterized Constructor called" << endl;
                id=x;
        }
};
int main() {

        // obj1 will call Default Constructor
        Geeks obj1;
        cout << "Geek id is: " <<obj1.id << endl;

        // obj2 will call Parameterized Constructor
        Geeks obj2(21);
        cout << "Geek id is: " <<obj2.id << endl;
        return 0;
}
```

Output:

```
Default Constructor called
Geek id is: -1
Parameterized Constructor called
Geek id is: 21
```

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 &){}
```

Destructors

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

```
// C++ program to explain destructors
#include <bits/stdc++.h>
```

```
using namespace std;
class Geeks
{
        public:
        int id;

        //Definition for Destructor
        ~Geeks()
        {
                cout << "Destructor called for id: " << id <<endl;
        }
};

int main()
{
        Geeks obj1;
        obj1.id=7;
        int i = 0;
        while ( i < 5 )
        {
                Geeks obj2;
                obj2.id=i;
                i++;
        } // Scope for obj2 ends here

        return 0;
} // Scope for obj1 ends here
```

Output:

```
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
```

Encapsulation in C++

In normal terms Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them. Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

We can not access any function from class directly. We need an object to access that function which is using the member the variable of that class.

The function which we are making inside the class, it must use the all member variable then only it is called encapsulation.

If we don’t make function inside the class which is using the member variable of the class then we don’t call it encapsulation.

Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section. In C++ encapsulation can be implemented using Class and access modifiers. Look at the below program:

Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

The data members should be labeled as private using the private access specifiers

The member function which manipulates the data members should be labeled as public using the public access specifier.

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.

public, protected and private inheritance in C++

public, **protected,** and **private** inheritance have the following features:

- **public inheritance** makes `public` members of the base class `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class.
- **protected inheritance** makes the `public` and `protected` members of the base class `protected` in the derived class.
- **private inheritance** makes the `public` and `protected` members of the base class `private` in the derived class.

Example 1: C++ public Inheritance

```
// C++ program to demonstrate the working of public inheritance
#include <iostream>
using namespace std;
class Base {
  private:
    int pvt = 1;
  protected:
    int prot = 2;
  public:
    int pub = 3;
    // function to access private member
    int getPVT() {
      return pvt;
    }
};
class PublicDerived : public Base {
  public:
    // function to access protected member from Base
    int getProt() {
      return prot;
    }
};
int main() {
  PublicDerived object1;
  cout << "Private = " << object1.getPVT() << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.pub << endl;
  return 0;
}
```

Output:

```
Private = 1
Protected = 2
Public = 3
```

Here, we have derived `PublicDerived` from `Base` in **public mode**.

As a result, in `PublicDerived`:

- `prot` is inherited as **protected**.
- `pub` and `getPVT()` are inherited as **public**.
- `pvt` is inaccessible since it is **private** in `Base`.

Since **private** and **protected** members are not accessible from `main()`, we need to create public functions `getPVT()` and `getProt()` to access them:

```
// Error: member "Base::pvt" is inaccessible
cout << "Private = " << object1.pvt;
// Error: member "Base::prot" is inaccessible
cout << "Protected = " << object1.prot;
```

Notice that the `getPVT()` function has been defined inside `Base`. But the `getProt()` function has been defined inside `PublicDerived`.

This is because `pvt`, which is **private** in `Base`, is inaccessible to `PublicDerived`.

However, `prot` is accessible to `PublicDerived` due to public inheritance. So, `getProt()` can access the protected variable from within `PublicDerived`.

Accessibility in public Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

```
// C++ program to demonstrate the working of protected inheritance
#include <iostream>
using namespace std;
class Base {
  private:
    int pvt = 1;
  protected:
    int prot = 2;
   public:
    int pub = 3;
    // function to access private member
    int getPVT() {
      return pvt;
    }
};
class ProtectedDerived : protected Base {
  public:
    // function to access protected member from Base
    int getProt() {
      return prot;
    }
    // function to access public member from Base
    int getPub() {
      return pub;
    }
};
int main() {
  ProtectedDerived object1;
  cout << "Private cannot be accessed." << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.getPub() << endl;
  return 0;
}
```

Output:

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `ProtectedDerived` from `Base` in **protected mode**.

As a result, in `ProtectedDerived`:

- `prot`, `pub` and `getPVT()` are inherited as **protected**.
- `pvt` is inaccessible since it is **private** in `Base`.

As we know, **protected** members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `ProtectedDerived`.

That is also why we need to create the `getPub()` function in `ProtectedDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in protected Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	

Example 3: C++ private Inheritance

```
// C++ program to demonstrate the working of private inheritance

#include <iostream>
using namespace std;

class Base {
  private:
    int pvt = 1;

  protected:
    int prot = 2;

  public:
    int pub = 3;

    // function to access private member
    int getPVT() {
      return pvt;
    }
};

class PrivateDerived : private Base {
  public:
```

```
  // function to access protected member from Base
  int getProt() {
    return prot;
  }

  // function to access private member
  int getPub() {
    return pub;
  }
};

int main() {
  PrivateDerived object1;
  cout << "Private cannot be accessed." << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.getPub() << endl;
  return 0;
}
```

Output:

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `PrivateDerived` from `Base` in **private mode**.

As a result, in `PrivateDerived`:

- `prot`, `pub` and `getPVT()` are inherited as **private**.
- `pvt` is inaccessible since it is **private** in `Base`.

As we know, private members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `PrivateDerived`.

That is also why we need to create the `getPub()` function in `PrivateDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();
// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in private Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

Inline Functions in C++

Inline function is one of the important feature of C++. So, let’s first understand why inline functions are used and what is the purpose of inline function?

- When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function.
- The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function.
- This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee).
- For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code.
- This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)

2) If a function contains static variables.

3) If a function is recursive.

4) If a function return type is other than void, and the return statement doesn’t exist in function

body.

5) If a function contains switch or goto statement.

Inline functions provide following advantages:

1) Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) When inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

Program:

```
#include <iostream>
using namespace std;
class operation
{
	int a,b,add,sub,mul;
	float div;
public:
	void get();
	void sum();
	void difference();
	void product();
	void division();
};
inline void operation :: get()
{
	cout << "Enter first value:";
	cin >> a;
	cout << "Enter second value:";
	cin >> b;
}

inline void operation :: sum()
```

```
{
        add = a+b;
        cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
        sub = a-b;
        cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
        mul = a*b;
        cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
        div=a/b;
        cout<<"Division of two numbers: "<<a/b<<"\n" ;
}
```

```
int main()
{
        cout << "Program using inline function\n";
        operation s;
        s.get();
        s.sum();
        s.difference();
        s.product();
        s.division();
        return 0;
}
```

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. **It is used to perform the operation on the user-defined data type.** For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.The advantage of Operators overloading is to perform different operations on the same operand.

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the return type is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

<u>Rules for Operator Overloading</u>

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
// program to overload the unary operator ++.

// Overload ++ when used as prefix

#include <iostream>

using namespace std;


class Count {

  private:

   int x;

  public:

  void getdata()

  {

      cout<<"enter value of x";

      cin>>x;

  }

   // Overload ++ when used as prefix

   void operator ++ () {

       ++x;

   }

   void display() {

       cout << "value of x= " << x << endl;

   }

};

int main() {

   Count count1;
   count1.getdata();
   // Call the "void operator ++ ()" function
  ++ count1;
   count1.display();
   return 0;
}
```

Output:

```
enter value of x 5
value of x= 6
```

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 –

::	.*	.	?:

Function Overloading in C++

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a polymorphism feature in C++.

The parameters should follow any one or more than one of the following conditions for Function overloading:

Parameters should have a different type

add(int a, int b)

add(double a, double b)

Below is the implementation of the above discussion:

```
#include <iostream>

using namespace std;




void add(int a, int b)

{

cout << "sum = " << (a + b);

}



void add(double a, double b)
```

```
{
        cout << endl << "sum = " << (a + b);
}


// Driver code
int main()
{
        add(10, 2);
        add(5.3, 6.2);


        return 0;
}
```

Output

```
sum = 12
sum = 11.5
```

Parameters should have a different number

add(int a, int b)

add(int a, int b, int c)

Below is the implementation of the above discussion:

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

.

```
void add(int a, int b)
{
cout << "sum = " << (a + b);
}

void add(int a, int b, int c)
{
        cout << endl << "sum = " << (a + b + c);
}

// Driver code
int main()
{
        add(10, 2);
        add(5, 6, 4);

        return 0;
}
```

Output

```
sum = 12
sum = 15
```

Parameters should have a different sequence of parameters.

.

add(int a, double b)

add(double a, int b)

Below is the implementation of the above discussion:

```
#include<iostream>

using namespace std;


void add(int a, double b)

{

        cout<<"sum = "<<(a+b);

}


void add(double a, int b)

{

        cout<<endl<<"sum = "<<(a+b);

}


// Driver code

int main()

{

        add(10,2.5);

        add(5.5,6);
        return 0;
}
```

Output

```
sum = 12.5
sum = 11.5
```

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
cout << " Here is int " << i << endl;
}
void print(double f) {
cout << " Here is float " << f << endl;
}
void print(char const *c) {
cout << " Here is char* " << c << endl;
}

int main() {
print(10);
print(10.10);

print("ten");

return 0;

}
```

Output

```
 Here is int 10

 Here is float 10.1

 Here is char* ten
```

How does Function Overloading work?

Exact match:- (Function name and Parameter)

If a not exact match is found:–

->Char, Unsigned char, and short are promoted to an int.

->Float is promoted to double

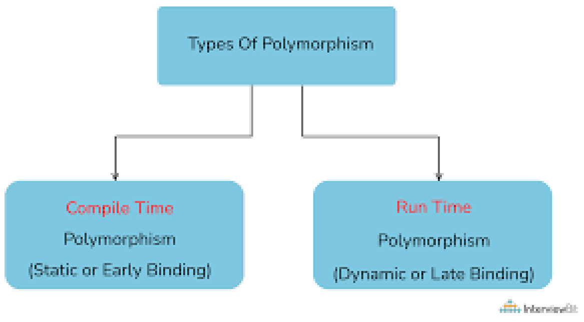
If no match is found:

->C++ tries to find a match through the standard conversion.

ELSE ERROR

There are two types of binding in C++: static (or early) binding and dynamic (or late) binding. Following is the differences between static and dynamic binding in C++.

1. The static binding happens at the compile-time, and dynamic binding happens at the runtime. Hence, they are also called early and late binding, respectively.
2. In static binding, the function definition and the function call are linked during the compile-time, whereas in dynamic binding, the function calls are not resolved until runtime. So, they are not bound until runtime.
3. Static binding happens when all information needed to call a function is available at the compile-time. Dynamic binding happens when the compiler cannot determine all information needed for a function call at compile-time.
4. Static binding can be achieved using the normal function calls, function overloading, and operator overloading, while dynamic binding can be achieved using the virtual functions.
5. Since all information needed to call a function is available before runtime, static binding results in faster execution of a program. Unlike static binding, a function call is not resolved until runtime for later binding, resulting in somewhat slower execution of code.
6. The major advantage of dynamic binding is that it is flexible since a single function can handle different types of objects at runtime. This significantly reduces the size of the codebase and also makes the source code more readable.


Types Of Polymorphism
Compile Time Polymorphism (Static or Early Binding)
Run Time Polymorphism (Dynamic or Late Binding)
InterviewBit

Dynamic Binding in C++

C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions. Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time using virtual table entry. So, Dynamic biding can be achieved in C++ via the Virtual keyword.

Prerequisite for Dynamic binding in C++

A virtual function is a member function which is declared within a base class and is redefined (Overridden) 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
- 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 but it cannot have a virtual constructor.

Class A is the 'base' class, whereas Class B is the 'derived class'. Inside both the classes, are one function with the same name 'display'. In parent class, there is another function named final_print() calling the display() function. In the main() function, we make two different objects for two different classes, calling the same display() function

```
#include <iostream>
using namespace std;
class A {
public:
    void final_print() // function that call display
    {
display();
    }
    void display() // the display function
    {
cout<< "Printing from the base class" <<endl;
    }
};
class B : public A // B inherit a publicly
{
public:
    void display() // B's display
    {
cout<< "Printing from the derived class" <<endl;
    }
};
int main()
{
    A obj1; // Creating A's pbject
    obj1.final_print(); // Calling final_print
    B obj2; // calling b
    obj2.final_print();
    return 0;
}
```

OUTPUT for Static Binding:

```
Printing from the base class
Printing from the base class
```

As expected, the output executes the base class’s display() function two times as it is defined at compile time (static binding). But this is not the output we wanted.

Now to convert the above code into a dynamic binding one, we need to use virtual functions. Let us see how to do that.

```
#include <iostream>
using namespace std;
class A {
public:
```

.

```
    void final_print() // function that call display
    {
display();
    }
virtual  void display() // the display function
    {
cout<< "Printing from the base class" <<endl;
    }
};
class B : public A // B inherit a publicly
{
public:
   virtual void display() // B's display
    {
cout<< "Printing from the derived class" <<endl;
    }
};
int main()
{
    A obj1; // Creating A's object
    obj1.final_print(); // Calling final_print
    B obj2; // calling b
    obj2.final_print();
    return 0;
}
```

OUTPUT for Dynamic Binding:

```
Printing from the base class
Printing from the derived class
```