

OOPs Notes

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. The programming paradigm or style where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language. It simplifies the software development and maintenance by providing some concepts:

[Like](#)

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

C++ Class

Class is a user defined data type that have its own properties and behaviors.

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

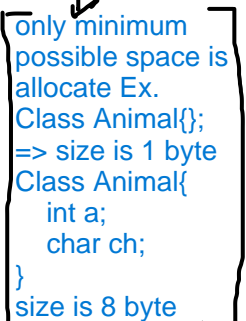
An object is an instance of a Class. It is an identifiable entity with some characteristics and behavior. Objects are the basic units of object-oriented programming. It may be any real-world object like a person, chair, table, pen, animal, car, etc.

A simple example of an object would be a car.

Logically, you would expect a car to have a model number or name. This would be considered the property of the car. You could also expect a car to be able to do something, such as starting or moving. This would be considered a method or properties of the car.

Code in object-oriented programming is organized around objects.

You need to have a class before you can create an object. When a class is defined, ~~memory~~ is allocated, but memory is allocated when it is instantiated (i.e., an object is created).



only minimum possible space is allocate Ex.
Class Animal{};
=> size is 1 byte
Class Animal{
int a;
char ch;
}
size is 8 byte

Syntax to create an object statically in C++:

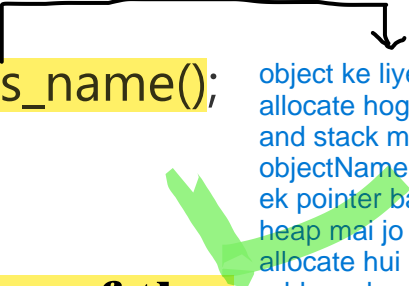
```
class_name objectName;
```

Syntax to create an object dynamically in C++:

```
class_name * objectName = new class_name();
```

Here,

❖ **objectName:** It is the name of the object created by class_name. The class's default constructor is called, and it dynamically allocates memory for one object of the class. The address of the memory allocated is assigned to the pointer, i.e., objectName.



object ke liye memory allocate hogi heap mai and stack mai
objectName name ka ek pointer banega jo ki heap mai jo memory allocate hui hai uske address ko store karega

Features of OOPs:

Four major object-oriented programming features make them different from non-OOP languages:

- **Abstraction: Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing.
- **Inheritance: When one object acquires all the properties and behaviors of parent object** i.e. known as inheritance. where a base class gives its behavior and attributes to a derived class. It provides code reusability. It is used to achieve runtime polymorphism.
- **Polymorphism**: ensures that it will execute the proper method based on the calling object's type. When **one task is performed by different ways** i.e. known as polymorphism.

- **Encapsulation: Binding (or wrapping) code and data together into a single unit is known as encapsulation.** For example: capsule, it is wrapped with different medicines.

Why do we need object-oriented programming?

- ❖ To make the development and maintenance of projects more effortless.
- ❖ OOPs provide the feature of data hiding that is good for security concerns whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
- ❖ We can solve real-world problems if we are using object-oriented programming.
- ❖ It increase code reusability, understandability, and maintainability of the code.

- ❖ It helps us to write generic code: which will work with a range of data, so we do not have to write basic stuff over and over again.
- ❖ Problems can be divided into subparts.

Example of OOPs in the Industry Example of OOPs using Cars: Consider the example of where you don't want to use just one car but 100 cars. Rather than describing each one in detail from scratch, you can use the same car class to create 100 objects of the type 'car'. You still have to give each one a name and other properties, but the basic structure of what a car looks like is the same. Here we will make Car class, and it will work as a basic template for other objects. We will make car class objects (Ferrari, BMW, and Mercedes). Each Car Object will have its own, Year of Manufacture, model, Top Speed, color, Engine Power, efficiency, etc. The car class would allow the programmer to store similar information unique to each car

(different models, colors, top speeds, etc.) and associate the appropriate information.

The disadvantage of OOPs?

- ❖ Requires pre-work and proper planning.
- ❖ In certain scenarios, programs can consume a large amount of memory.
- ❖ Not suitable for a small problem.
- ❖ Proper documentation is required for later use.

What is the difference between class and structure?

Class: User-defined blueprint from which objects are created.

- It consists of methods or sets of instructions that are to be performed on the objects.
- In class default access modifier is private.

- It is normally used for data abstraction and further inheritance.

Structure: A structure is basically a user-defined collection of variables of different data types. In Structure default access modifier is public.

- It does not support concept of inheritance.
- It may have only parameterized constructor.
- It is normally used for the grouping of data.
- What is the main reason for using structure?
- A structure is used **to represent information about something more complicated than a single number, character, or Boolean can do** (and more complicated than an array of the above data types can do). For example, a student can be defined by his or her name, gpa, age, uid, etc.

What is the difference between a class and an object?

Class:

Class is the blueprint of an object. It is used to create objects.

No memory is allocated when a class is declared.

A class is a group of similar objects.

Class is a logical entity.

A class can only be declared once.

An example of class can be a car.

Objects:

An object is an instance of the class.

Memory is allocated as soon as an object is created.

An object is a real-world entity such as a book, car, etc.

An object is a physical entity.

Objects can be created many times as per requirement.

Objects of the class car can be BMW, Mercedes, Ferrari, etc.

C++ Constructor

A constructor is a special member function automatically called when an object is created. In C++, the constructor is automatically called when an object is created. It is a special class method because it does not have any return type. It has the same name as the class itself. It is used to initialize the data members of new object generally.

The constructor must be placed in the public section of the class because we want the class to be instantiated anywhere. For every object in its lifetime constructor is called only once at the time of creation.

- Constructors can be overloaded.
- Constructor cannot be declared virtual.

- The virtual mechanism works only when we have a base class pointer to a derived class object.
- In C++, the constructor cannot be virtual, because when a constructor of a class is executed, there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.
- But virtual destructor is possible.
- Constructor cannot be inherited.
 - Addresses of Constructor cannot be referred.
 - Constructor make implicit calls to **new** and **delete** operators during memory allocation.
 - Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.

There are three types of constructors in C++:

★ **Default constructor**

- ★ Parameterized Constructor

- ★ Copy Constructor

Default constructor:

A constructor that doesn't take any argument or has no parameters is known as a default constructor.

Note: If we have not defined a constructor in our class, the C++ compiler will automatically create a default constructor with an empty code and no parameters, which will initialize data members with garbage values. When we write our constructor explicitly, the inbuilt constructor will not be available for us.

Note: When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as.

Note: 1. if we define any constructor in class other than copy constructor default constructor generated by compiler become vanish but a default copy constructor will be present.

2. if we define copy constructor in a class than default constructor as well as default copy constructor generated by compiler become vanish.

Ex:

Student s;

Will flash an error

Parameterized Constructor:

This is another type of Constructor with parameters. The parameterized constructor takes its arguments provided by the programmer. These

arguments help initialize an object when it is created.

Using this Constructor, you can provide different values to data members of different objects by passing the appropriate values as arguments.

Copy Constructor:

These are a particular type of constructor that takes an object as an argument and copies values of one object's data members into another object. We pass the class object into another object of the same class in this constructor. As the name suggests, you Copy means to copy the values of one Object into another Object of Class. This is used for Copying the values of a class object into another object of a class, so we call them Copy constructor and for copying the values.

We must pass the object's name whose values we want to copy, and when we are using or passing an

object to a constructor, we must use the & ampersand or address operator.

If we do not define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects.

- Copy constructor takes a reference to an object of the same class as an argument.
- 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.

Syntax:

```
Class class_name{  
    int data_member1;
```



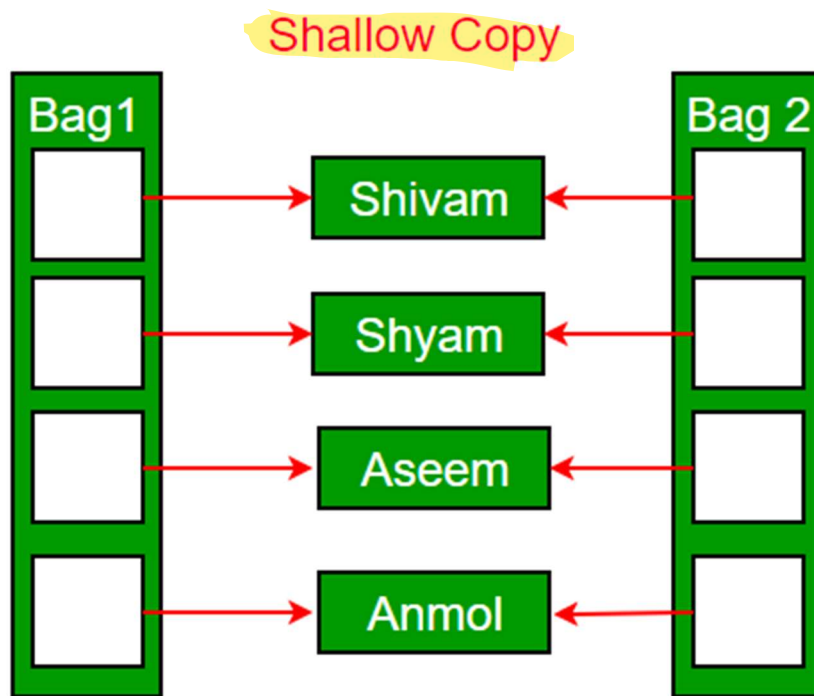
```
    stringdata_member2;  
  
    //copy constructor  
public:  
    |  
    class_name(class_name &obj){  
        // copies data of the obj parameter  
        data_member1 = obj.data_member1;  
        data_member2 = obj.data_member2;  
    }  
};
```

➤ When is a user-defined copy constructor needed?

If we do not 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

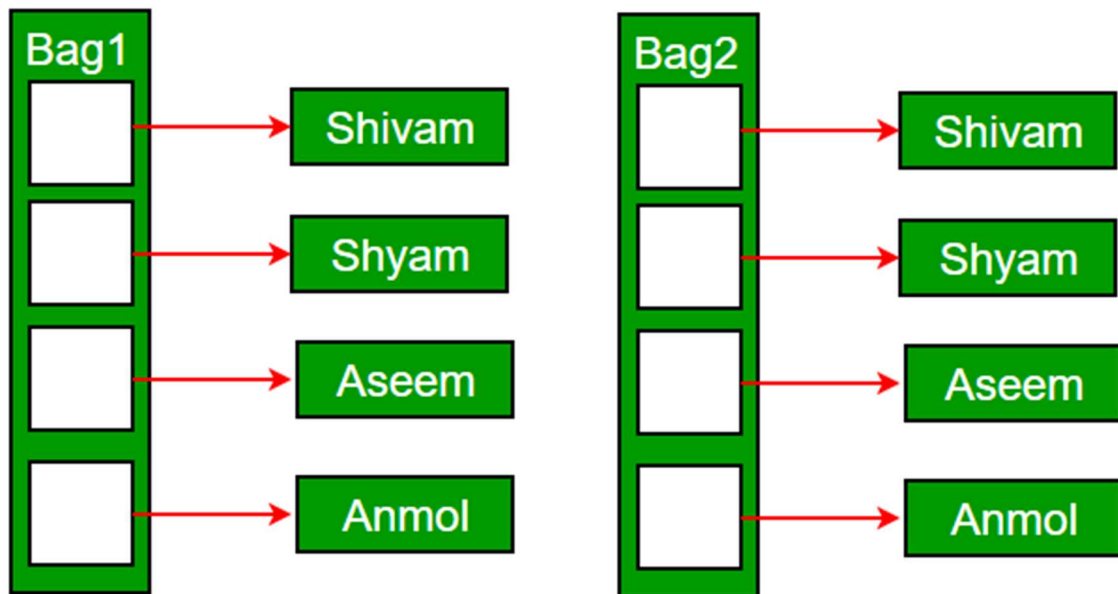
resource like a file handle, a network connection, etc.

- The default **constructor does only shallow copy.**



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

Deep Copy



➤ Copy constructor vs Assignment Operator

The main difference between [Copy Constructor](#) and [Assignment Operator](#) is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

➤ ***Which of the following two statements calls the copy constructor and which one calls the assignment operator?***

MyClass t1, t2;

MyClass t3 = t1; // ----> (1)

```
t2 = t1; // -----> (2)
```

A copy constructor is called when a new object is created from an existing object, as a copy of the existing object. The assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls the copy constructor and (2) calls the assignment operator.

➤ **Can we make the copy constructor private?**

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like the above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

➤ Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to the copy constructor would be made to call the copy constructor which becomes a non-terminating chain of calls.

Therefore, compiler doesn't allow parameters to be passed by value.

➤ Why argument to a copy constructor should be **const**?

One reason for passing **const** reference is, that we should use **const** in C++ wherever possible so that objects are not accidentally modified.

Constructor Overloading:

Constructor overloading can be defined as the concept of having more than one constructor with

different parameters so that every constructor can perform a different task.

As there is a concept of function overloading, similarly constructor overloading is applied when we overload a constructor more than a purpose.

The declaration is the same as the class name, but there is no return type as they are constructors.

The criteria to overload a constructor is to differ the number of arguments or the type of arguments. The corresponding constructor is called depending on the number and type of arguments passed.

Destructor:

A destructor is a special member function that works just opposite to a constructor;

Unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

```
~class_name() {
```

```
}
```

The destructor name should exactly match the class name. A destructor declaration should always begin with the tilde (~) symbol, as shown in the syntax above.

NOTE: The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory. Or in other words, for a statically created class object, destructor called by default itself but when we create an object

dynamically, then we must to call a destructor manually like this.

Ex: `delete Hero` => After this statement destructor is called.

When is a destructor called?

A destructor function is called automatically when:

- the object goes out of scope
- the program ends
- a scope (the {} parenthesis) containing local variable ends.
- a delete operator is called

Destructor rules

- ❖ The name should begin with a tilde sign (~) and match the class name.
- ❖ There cannot be more than one destructor in a class.
- ❖ Unlike constructors that can have parameters, destructors do not allow any parameter.
- ❖ They do not have any return type, not even void.
- ❖ A destructor should be declared in the public section of the class.
- ❖ The programmer cannot access the address of the destructor.
- ❖ When you do not specify any destructor in a class, the compiler generates a default destructor and inserts it into your code.
- ❖ It cannot be declared static or const.
- ❖ The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a

destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

- **Private Destructor in C++**
- **What is the use of private destructor?**

ANS: Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
```

```
private:
    ~Test() {}
};
int main() {}
```

The above program compiles and runs fine. Hence, we can say that: It is **not** a compiler error to create private destructors.

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test t; }
```

➔ for this compiler gives an error.

```
// CPP program to illustrate
```

```
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

int main() { Test* t; }
```

➔ The above program works fine. There is no object being constructed, the program just creates a pointer of type “Test*”, so nothing is destructed.

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
```

```
};  
int main() { Test* t = new Test; }
```

- ➔ **The above program also works fine. When something is created using dynamic memory allocation, it is the programmer's responsibility to delete it. So, compiler doesn't bother.**
 - ➔ **In the case where the destructor is declared private, an instance of the class can also be created using the malloc () function. The same is implemented in the below program.**
 - ➔ **Program fails in the compilation. When we call delete, destructor is called.**
- **We noticed in the above programs when a class has a private destructor, only dynamic objects of that class can be created. Following is a way to create classes with private destructors and have a function as a friend of the class. The function can only delete the objects.**

```
➤ // CPP program to illustrate
➤ // Private Destructor
➤ #include <iostream>
➤
➤ // A class with private destructor
➤ class Test {
➤ private:
➤     ~Test() {}
➤
➤ public:
➤     friend void destructTest(Test*);
➤ };
➤
➤ // Only this function can destruct objects of
➤ Test
➤ void destructTest(Test* ptr) { delete ptr; }
➤
➤ int main()
➤ {
➤     // create an object
➤     Test* ptr = new Test;
➤
➤     // destruct the object
➤     destructTest(ptr);
➤
➤     return 0;
➤ }
```

Another way to use private destructors is by using the class instance method.

```
#include <iostream>

using namespace std;

class parent {
    // private destructor
    ~parent() { cout << "destructor
called" << endl; }

public:
    parent() { cout << "constructor
called" << endl; }
    void destruct() { delete this; }
};

int main()
{
    parent* p;
    p = new parent;
    // destructor called
    p->destruct();

    return 0;
}
```

Interview Questions

1. Does C++ compiler create a default constructor when we write our own?

Ans: In C++, compiler by default creates a default constructor for every class. But, if we define our own constructor, compiler does not create the default constructor.

2. Explain constructor in C++

A constructor is a special member function automatically called when an object is created. A constructor initializes the class data members with garbage value if we do not put any value to it explicitly.

3. What do you mean by constructor overloading?

The concept of having more than one constructor with different parameters to perform a different task is known as constructor overloading.

4. Explain Destructor in C++

A destructor is a special member function that works just opposite to a constructor; unlike constructors that are used for initializing an object, destructors destroy (or delete) the object. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

5. What is a copy constructor?

These are a particular type of constructor that takes an object as an argument and copies values of one object's data members into another object. In this constructor, we pass the class object into another object of the same class.

6. How many types of constructors are there?

7. When should the destructor use delete to free the memory?

If the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

8. What is the return type of constructor and destructor?

They have no return type, not even void.

this Pointer

this pointer holds the address of the current object. in simple words, you can say that this pointer points to the current object of the class.

There can be three main usages of this keyword in C++.

- It can be used to refer to a current class instance variable.
- It can be used to pass the current object as a parameter to another method.
- It can be used to declare indexers.

Let us take an example to understand this concept.

```
#include <bits/stdc++.h>
```

```
Using namespace std;
```

```
Class mobile {
```

```
    string model;
```

```
    int year_of_manufacture;
```

```
    public:
```

```
void set_details( string model, int  
year_of_manufacture){
```

```
    this->model = model;
```

```
    this->year_of_manufacture =  
    year_of_manufacture;
```

```
}
```

```
void print(){
```

```
    cout<<this->model <<endl;
```

```
    cout<<this->year_of_manufacture <<endl;
```

```
}
```

```
};
```

```
int main () {
```

```
    mobile redmi;
```

```
    redmi.set_details("Note 7 Pro",2019);
```

```
    redmi.print();
```

```
}
```

Output: 1Note 7 Pro 2019 Here you can see that we have two data members model and year_of_manufacture. In member function set_details (), we have two local variables with the same name as the data members' names. Suppose you want to assign the local variable value to the data members. In that case, you won't be able to do until unless you use this pointer because the compiler won't know that you are referring to the object's data members unless you use this pointer. This is one of example where you must use this pointer.

Explain the situation where this pointer is used:

- As we know each object gets its own copy of data members and all objects share a single copy of member functions.
- Then now, question is that if only one copy of each member function exists which is used by multiple objects, how are the proper data members accessed and updated.
- Ans: The compiler supplies an implicit pointer along with the names of the functions as "this". Where "this" refers to the address of current.
- The 'this' pointer is available only within the non-static member functions of a class. If the member function is static, it will be common to all the objects, and hence a single object can't refer to those functions independently.

Static data member and member functions in C++:

Static members are class member that are declared using 'Static' key word.

A static member has certain special characteristic's

- Only one copy of that member is created for the entire class and that copy is shared by all the objects of that class. No matter how many objects are created.
- It is initialized before any object of this class is being created, even before main starts.
- It is visible only within the class, but its lifetime is in the entire program.
- `static data_type name_of_member;`
- Declared inside the class body.
- Defined outside the class.
- Static member variables are not belonging to any objects, but its belongs to whole class so these are called class member variable.

Advantage of C++ static keyword:

Memory efficient: Now we do not need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

When do we declared a member of a class is static:

- It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.
- It is used for recording the data which common to all. Ex You can use a static data member as a counter to store the number of objects of a particular type.
- Ex with code:

```
Class Truck {  
    private:  
        static int count = 0;
```



```
public:  
    static int getCount () {  
        return count;  
    }
```

```
Truck () {  
    Count++;  
}
```

```
};
```

Shallow Copy:

An object is created by simply copying the data of all variables of the original object. Here, the pointer will be copied but not the memory it points to.

It means that the original object and the created copy will now point to the same memory address, which is generally not preferred.

Since both objects will reference the exact memory location, then change made by one will reflect those change in another object as well.

This can lead to unpleasant side effects if the elements of values are changed via some other reference.

Since we wanted to create an object replica, the Shallow copy will not fulfill this purpose.

Note: C++ compiler implicitly creates a copy constructor and assignment operator to perform shallow copy at compile time.

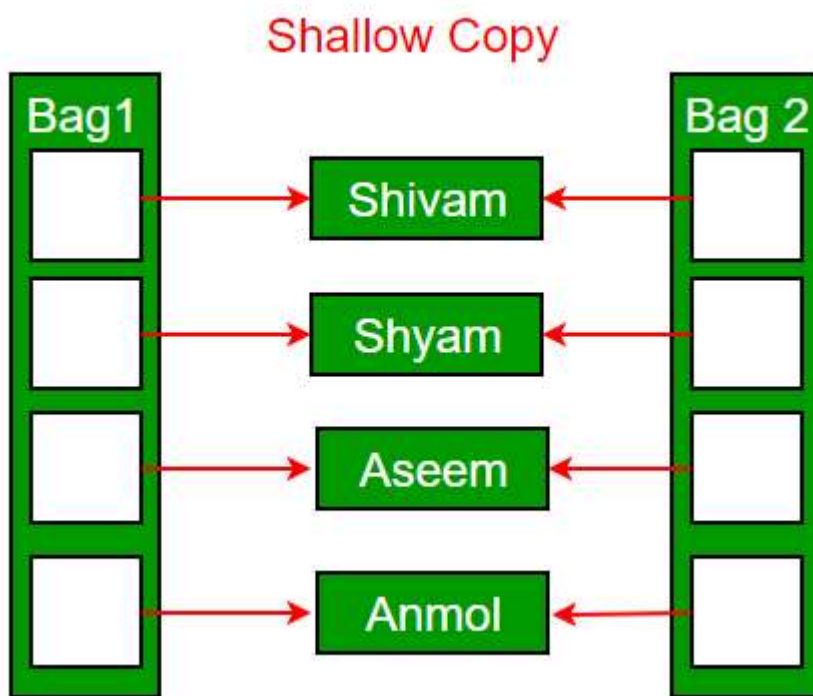
A shallow copy can be made by simply copying the reference.

Example:

```
class students(){  
    int age;  
    char * names;
```

```
public: students (int age, char *  
names) {  
    this->age = age; // shallow copy  
    this->names = names;  
}  
};
```

➤ Shallow Copy is faster than Deep copy.
we are just copying the reference.
The above code shows shallow copying.



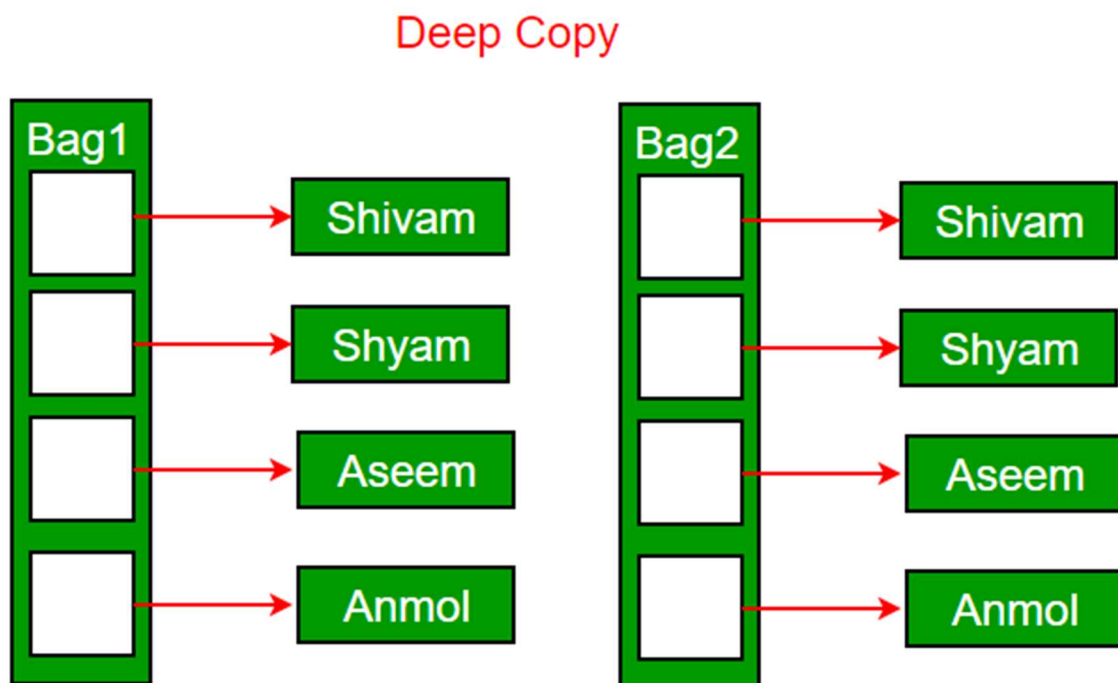
Deep Copy: An object is created by copying all the fields, and it also allocates similar memory resources with the same value to the object. To perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is necessary to allocate memory to the other constructors' variables dynamically. A deep copy means creating a new array and copying over the values. Changes to the array values referred to will not result in changes to the array data refers to.

Example:

```
class student () {  
    int age;  
    char * names;  
public:  
    student (int age, char * names) {  
        this->age = age; //deep copy  
        this->names = new char[strlen(names) +  
1];
```

```
        strcpy(this->names, names);  
        //Created new array and copied data  
    }  
};
```

The above code shows deep copying.



NOTE: Shallow copy works fine if none of the variable of the object is defined in heap section.

If some variable is dynamically allocated in heap memory, the copied object variable will also reference to the same memory location.

Interview Questions

1. What is this pointer?

Ans: this pointer is accessible only inside the member functions of a class and points to the object which has called this member function.

2. When is it necessary to use this pointer?

Ans: Suppose we have two local variables with the same name as the data members' names. Suppose you want to assign the local variable value to the data members. In that case, you will not be able to do until unless you use this pointer because the compiler won't know that you are referring to the object's data members unless you use this pointer.

3. What is similar between deep copy and shallow copy?

Ans: Both are used to copy data between objects.

4. What is the difference between deep copy and shallow copy?

Shallow Copy

- Shallow Copy stores the references of objects to the original memory address.
- Shallow Copy reflects changes made to the new/copied object in the original object.
- Shallow Copy stores the copy of the original object and points the references to the objects.
- Shallow copy is faster.

Deep Copy

- Deep copy stores copies of the object's value.
- Deep copy doesn't reflect changes made to the new/copied object in the original object.
- Deep copy stores the copy of the original object and recursively copies the objects as well.
- Deep copy is comparatively slower.

Access Modifiers in Java:

- Link => <https://www.geeksforgeeks.org/access-modifiers-java/>
- Default: When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.
- The data members, classes, or methods that are not declared using any access modifiers i.e., having default access modifier are accessible only within the same package.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- **Private** members are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend](#)

functions are allowed to access the private data members of the class in C++.

Access Modifiers in C++

➤ Link => <https://www.geeksforgeeks.org/access-modifiers-in-c/>

Types of Class Member Functions in C++

➤ **Simple Member functions in C++**

These are the basic member function, which don't have any special keyword like static etc. as prefix. All the general member functions, which are of below given form, are termed as simple and basic member functions.

➤ **Static Member functions in C++**

A function is made static by using **static** keyword with function name. These functions work for the class as whole rather than for a particular object of a class.

It can be called using the object and the direct member access operator. But, its more typical to call a static member function by itself, using class name and scope resolution **::** operator.

For example:

```
class X
{
    public:
        static void f()
        {
            // statement
        }
};

int main()
{
    X::f();    // calling member function
               directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only **static** data members and **static** member functions can be called inside them.

It doesn't have any "this" keyword which is the reason it cannot access ordinary members.

➤ **Const Member functions in C++**

We will study **Const** keyword in detail later([Const Keyword](#)), but as an introduction, Const keyword makes variables constant, that means once defined, their values can't be changed.

When used with member function, such member functions can never modify the object or its related data members.

basic syntax of const Member Function

```
void fun() const{  
  
    // statement  
  
}
```

➤ Inline functions in C++

All the member functions defined inside the class definition are by default declared as Inline.

➤ Friend functions in C++

Friend functions are not class member function. Friend functions are made to give **private** access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

For example:

```
class WithFriend {  
  
    int i;  
  
    public:  
  
        friend void fun (); // global function as  
friend
```

```
};

void fun () {

    WithFriend wf;

    wf.i=10;    // access to private data member

    cout << wf.i;

}

int main() {

    fun(); //Can be called directly

}
```

Hence, friend functions can access private data members by creating object of the class. Similarly, we can also make function of some other class as friend, or we can also make an entire class as **friend class**.

```
class Other{

    void fun();

};

class WithFriend{

    private:

    int i;
```

```
public:

    void getdata(); // Member function of class
WithFriend

    // making function of class Other as friend here

    friend void Other::fun();

    // making the complete class as friend

    friend class Other;

};
```

When we make a class as friend, all its member functions automatically become friend functions.

NOTE: Friend Functions is a reason, why C++ is not called as a pure [Object Oriented](#) language. Because it violates the concept of Encapsulation.

➤ Important Keywords:

Virtual Function in C++

- A virtual function is a member function which is declared within a base class and is re-defined (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 runtime.

Rules for Virtual Functions

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

Ex: Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

```
// CPP program to illustrate  
// concept of Virtual Functions
```

```
#include<iostream>
```

```
using namespace std;
```

```
class base {
```

```
public:
```

```
    virtual void print()
```

```
{
```

```
    cout << "print base class\n";
```

```
}
```



```
void show()
{
    cout << "show base class\n";
}
};
```

```
class derived : public base {
public:
    void print()
    {
        cout << "print derived class\n";
    }
}
```

```
void show()
```

```
    {  
        cout << "show derived class\n";  
    }  
};
```

```
int main()  
{  
    base *bptr;  
    derived d;  
    bptr = &d;  
  
    // Virtual function, binded at runtime  
    bptr->print();  
}
```

```
// Non-virtual function, binded at  
compile time
```

```
    bptr->show();
```

```
    return 0;
```

```
}
```

Output:

print derived class

show base class

NOTE: 1. Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

2. Late binding (Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile time) is done according to the type of pointer.

3. since print () function is declared with virtual keyword so it will be bound at runtime (output is *print derived class* as pointer is pointing to object of derived class) and show () is non-virtual so it will be bound during compile time (output is *show base class* as pointer is of base type).

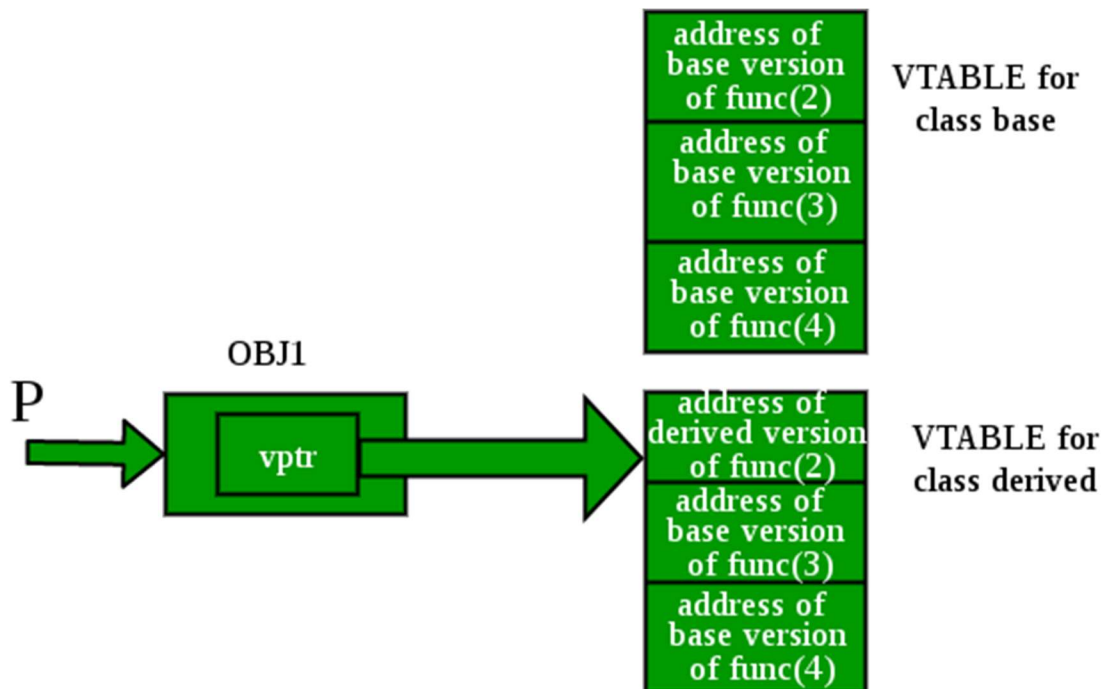
4. If we have created a virtual function in the base class and it is being overridden in the derived class then we do not need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

> Working of virtual functions (concept of VTABLE and VPTR)

As discussed, [here](#) if a class contains a virtual function then compiler itself does two things.

1) If object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

2) Irrespective of object is created or not, class contains as a member a static array of function pointers called VTABLE. Cells of this table store the address of each virtual function contained in that class.



```
// CPP program to illustrate
// working of Virtual Functions
#include<iostream>
using namespace std;
```

```
class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
```

```
};
```

```
class derived : public base {
```

```
public:
```

```
    void fun_1() { cout << "derived-1\n"; }
```

```
    void fun_2() { cout << "derived-2\n"; }
```

```
    void fun_4(int x) { cout << "derived-4\n"; }
```

```
};
```

```
int main()
```

```
{
```

```
    base *p;
```

```
    derived obj1;
```

```
    p = &obj1;
```

```
    // Early binding because fun1() is non-virtual
```

```
    // in base
```

```
    p->fun_1();
```

```
// Late binding (RTP)
```

```
p->fun_2();
```

```
// Late binding (RTP)
```

```
p->fun_3();
```

```
// Late binding (RTP)
```

```
p->fun_4();
```

```
// Early binding but this function call is
```

```
// illegal (produces error) because pointer
```

```
// is of base type and function is of
```

```
// derived class
```

```
// p->fun_4(5);
```

```
return 0;
```

```
}
```

Output:

base-1

derived-2

base-3

base-4

Explanation: Initially, we create a pointer of type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototypes of both the functions are different.

Limitations of Virtual Functions:

Slower: The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.

Difficult to Debug: In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

➤ What happens when we do not use virtual function

// C++ program to demonstrate how we will calculate

// area of shapes without virtual function

```
#include <iostream>
```

```
using namespace std;
```

```
// Base class
```

```
class Shape {
```

```
public:
```

```
// parameterized constructor
Shape(int l, int w)
{
    length = l;
    width = w;
}
int get_Area()
{
    cout << "This is call to parent class area\n";
    // Returning 1 in user-defined function means
true
    return 1;
}
```

protected:

```
    int length, width;
};
```

```
// Derived class
```

```

class Square : public Shape {
public:
    Square(int l = 0, int w = 0)
        : Shape(l, w)
    {
    } // declaring and initializing derived class
    // constructor
    int get_Area()
    {
        cout << "Square area: " << length * width <<
'\n';
        return (length * width);
    }
};

// Derived class
class Rectangle : public Shape {
public:
    Rectangle(int l = 0, int w = 0)
        : Shape(l, w)

```

```
{  
} // declaring and initializing derived class  
// constructor  
int get_Area()  
{  
    cout << "Rectangle area: " << length * width  
        << '\n';  
    return (length * width);  
}  
};
```

```
int main()  
{  
    Shape* s;  
  
    // Making object of child class Square  
    Square sq(5, 5);  
  
    // Making object of child class Rectangle
```

```

    Rectangle rec(4, 5);
    s = &sq; // reference variable
    s->get_Area();
    s = &rec; // reference variable
    s->get_Area();

    return 0; // too tell the program executed
    // successfully
}

```

Output

This is call to parent class area

This is call to parent class area

=====

➤ To resolve this problem, we must use virtual function

// C++ program to demonstrate how we will calculate

// the area of shapes USING VIRTUAL FUNCTION

```
#include <fstream>
```

```
#include <iostream>
```

```
using namespace std;
```

```
// Declaration of Base class
```

```
class Shape {
```

```
public:
```

```
    // Usage of virtual constructor
```

```
    virtual void calculate()
```

```
    {
```

```
        cout << "Area of your Shape ";
```

```
    }
```

```
    // usage of virtual Destuctor to avoid memory leak
```

```
    virtual ~Shape()
```

```
    {
```

```
        cout << "Shape Destuctor Call\n";
```

```
    }
```

```
};
```

```
// Declaration of Derived class
```

```
class Rectangle : public Shape {
```

public:

int width, height, area;

void calculate()

{

cout << "Enter Width of Rectangle: ";

cin >> width;

cout << "Enter Height of Rectangle: ";

cin >> height;

area = height * width;

cout << "Area of Rectangle: " << area << "\n";

}

// Virtual Destuctor for every Derived class

virtual ~Rectangle()

{

cout << "Rectangle Destuctor Call\n";


```
    }  
};
```

// Declaration of 2nd derived class

```
class Square : public Shape {
```

```
public:
```

```
    int side, area;
```

```
    void calculate()
```

```
{
```

```
    cout << "Enter one side your of Square: ";
```

```
    cin >> side;
```

```
    area = side * side;
```

```
    cout << "Area of Square: " << area << "\n";
```

```
}
```

// Virtual Destuctor for every Derived class

```
    virtual ~Square()
```

```
    {  
        cout << "Square Destuctor Call\n";  
    }  
};
```

```
int main()
```

```
{
```

```
    // base class pointer
```

```
    Shape* S;
```

```
    Rectangle r;
```

```
    // initialization of reference variable
```

```
    S = &r;
```

```
    // calling of Rectangle function
```

```
    S->calculate();
```

```
    Square sq;
```

```
// initialization of reference variable  
S = &sq;  
  
// calling of Square function  
S->calculate();  
  
// return 0 to tell the program executed  
// successfully  
return 0;  
}
```

Output:

Enter Width of Rectangle: 10

Enter Height of Rectangle: 20

Area of Rectangle: 200

Enter one side your of Square: 16

Area of Square: 256

What is the use?

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

Real-Life Example to Understand the Implementation of Virtual Function

Consider employee management software for an organization.

Let the code has a simple base class *Employee*, the class contains virtual functions

like *raiseSalary()*, *transfer()*, *promote()*, etc. Different types of employees like *Managers*, *Engineers*, etc., may have their own implementations of the virtual functions present in base class *Employee*.

In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don't need to worry about them because

if *raiseSalary()* is present for a specific employee type, only that function would be called.

// C++ program to demonstrate how a virtual function
// is used in a real-life scenario

```
class Employee {  
public:  
    virtual void raiseSalary()  
    {  
        // common raise salary code  
    }  
  
    virtual void promote()  
    {  
        // common promote code  
    }  
};
```

```
class Manager : public Employee {  
    virtual void raiseSalary()  
    {  
        // Manager specific raise salary code, may  
contain  
        // increment of manager specific incentives  
    }  
  
    virtual void promote()  
    {  
        // Manager specific promote  
    }  
};
```

// Similarly, there may be other types of employees

// We need a very simple function

// to increment the salary of all employees

// Note that emp[] is an array of pointers

```
// and actual pointed objects can  
// be any type of employees.  
// This function should ideally  
// be in a class like Organization,  
// we have made it global to keep things simple
```

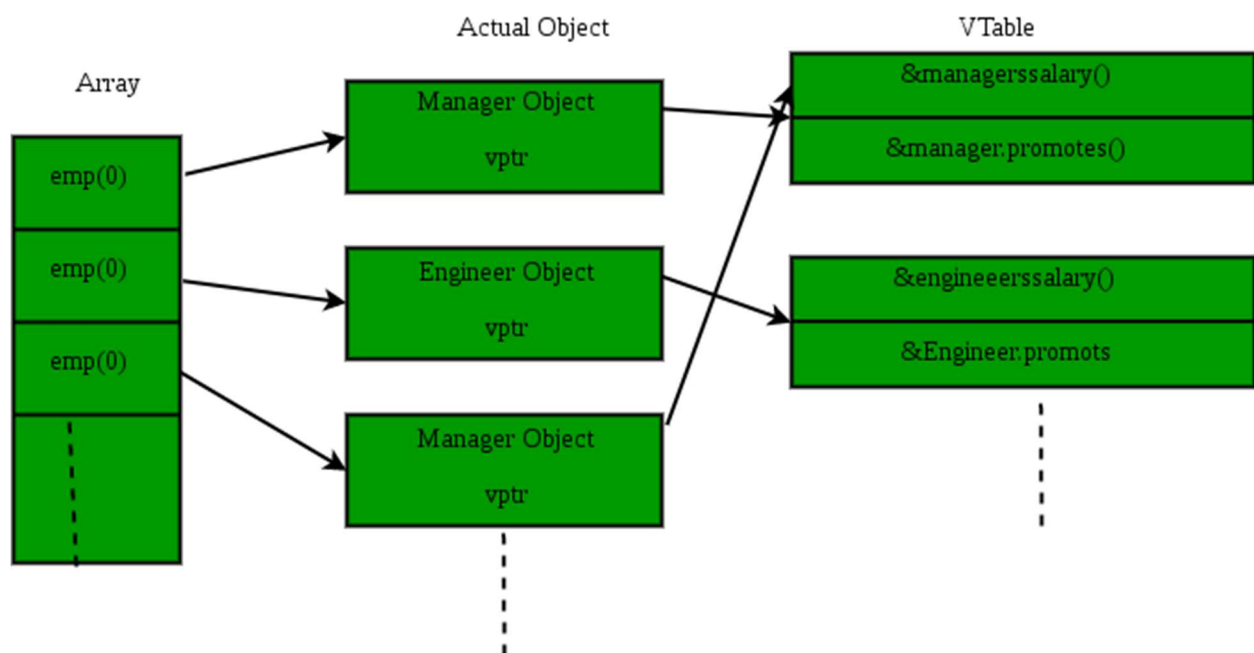
```
void globalRaiseSalary(Employee* emp[], int n)  
{  
    for (int i = 0; i < n; i++) {  
        // Polymorphic Call: Calls raiseSalary()  
        // according to the actual object, not  
        // according to the type of pointer  
        emp[i]->raiseSalary();  
    }  
}
```

Like the '***globalRaiseSalary()***' *function*, there can be many other operations that can be performed on a list of employees without even knowing the type of the object instance.

➤ How does the compiler perform runtime resolution?

1) **vtable**: A table of function pointers, maintained per class.

2) **vptr**: A pointer to vtable, maintained per object instance (see [this](#) for an example).



➤ The compiler adds additional code at two places to maintain and use vp_{tr}.

1. Code in every constructor. This code sets the vp_{tr} of the object being created. This code sets vp_{tr} to point to the vtable of the class.

2. Code with polymorphic function call (e.g., bp->show () in above code). Wherever a polymorphic call is made, the compiler inserts code to first look for vptr using a base class pointer or reference /*(In the above example, since the pointed or referred object is of a derived type, vptr of a derived class is accessed)*/. Once vptr is fetched, vtable of derived class can be accessed. Using vtable, the address of the derived class function show () is accessed and called.

➤ **Virtual Table:** are created for all classes which at least one Virtual function or those classes which are derived from classes which have at least one virtual function.

HOW TO FILL VTABLE:

For each virtual function pointer, we see whether there is a function overridden in this class derived class if it is there, this function pointer points to this

Otherwise, it points to base class.

So, say suppose there is class A, from class A there is a derived class B, from B there is also a derived class C, while filling the Virtual Table will see for a particular function if there is in C, we will point to C, if it is not there in C, we will check if there is in B, if yes then we will point to this else we will point to A.

- These V-Tables are formed by compiler at compile time but there is a catch over here, these V-Tables are essentially static arrays,
- Since these V-Tables are static arrays that means all the objects' instances point to the same V-Table, so all the objects' point to one V-Table, now there should be something which points object to V-Table. For that compiler does whenever a class declares a new virtual function will add to that class that is called virtual pointer, now when new object is instantiated compiler adds some extra code to constructor

And it points to this virtual pointer to the V-Table according to object type and V-pointer points to V-Table at real time so run time polymorphism will achieved.

➤ V-Table is object independent but V-Pointer is object dependent.

➤ When this line will be executed

Emp* a = new engineer ();

V-Pointer will be assigned according to object not according to pointer. This is called run time polymorphism.

➤ This is called late binding because during the execution we know what is object type.

➤ **Virtual Destructor**

We cannot delete a derived class object using a base class pointer that has a non-virtual destructor. To delete the derived class object using the base class pointer, the base class must contain a virtual destructor. It will be clearer as we understand the following examples.

➔ Deleting a derived class object using a pointer of base class type that has a non-virtual destructor result in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
#include <iostream>
using namespace std;
class Base {
public:
    virtual void show(){
        cout<<"now we are in base class"<<endl;
    }
    Base()
    { cout << "Base created\n"; }
    ~Base()
    { cout<< "Base destroyed\n"; }
};
```

```
class Derived: public Base {
public:
void show(){
    cout<<"now we are in Derived class"<<endl;
}
Derived()
{ cout << "Derived created\n"; }
~Derived()
{ cout << "Derived destroyed\n"; }
};
int main()
{
    Base *b = new Derived;
    delete b;
    return 0;
}
```

OUTPUT:

Base created

Derived created

Base destroyed

C++ | Virtual Functions | Question 2

```
#include<iostream>

using namespace std;

class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
```

```
Base *bp = new Derived;
```

```
bp->show();
```

```
Base &br = *bp;
```

```
br.show();
```

```
return 0;
```

```
}
```

(A)

In Base

In Base

(B)

In Base

In Derived

(C)

In Derived

In Derived

(D)

In Derived

In Base

Answer: (C)

Explanation: Since show() is virtual in base class, it is called according to the type of object being referred or pointed, rather than the type of pointer or reference.

C++ | Virtual Functions | Question 3

```
#include<iostream>
```

```
using namespace std;
```

```
class Base
```

```
{
```

```
public:
```

```
    virtual void show() { cout<<" In Base \n"; }
```

```
};
```

```
class Derived: public Base
```

```
{
```


public:

```
    void show() { cout<<"In Derived \n"; }  
};
```

```
int main(void)
```

```
{
```

```
    Base *bp, b;
```

```
    Derived d;
```

```
    bp = &d;
```

```
    bp->show();
```

```
    bp = &b;
```

```
    bp->show();
```

```
    return 0;
```

```
}
```

(A)

In Base

In Base

(B)

In Base

In Derived

(C)

In Derived

In Derived

(D)

In Derived

In Base

Answer: (D)

Explanation: Initially base pointer points to a derived class object. Later it points to base class object,

C++ | Virtual Functions | Question 4

Which of the following is true about pure virtual functions?

- 1) Their implementation is not provided in a class where they are declared.
- 2) If a class has a pure virtual function, then the class becomes abstract class and an instance of this class

cannot be created.

(A) Both 1 and 2

(B) Only 1

(C) Only 2

(D) Neither 1 nor 2

Answer: (C)

C++ | Virtual Functions | Question 5

```
#include<iostream>
```

```
using namespace std;
```

```
class Base
```

```
{
```

```
public:
```

```
    virtual void show() = 0;
```

```
};
```

```
int main(void)
{
    Base b;
    Base *bp;
    return 0;
}
```

- (A) There are compiler errors in lines “Base b;” and “Base bp;”
- (B) There is compiler error in line “Base b;”
- (C) There is compiler error in line “Base bp;”
- (D) No compiler Error

Answer: (B)

Explanation: Since Base has a pure virtual function, it becomes an abstract class and an instance of it cannot be created.

So there is an error in line “Base b”.

Note that there is no error in line “Base *bp;”. We can have pointers or references of abstract classes.

C++ | Virtual Functions | Question 6

```
#include<iostream>

using namespace std;

class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
    Derived q;
    return 0;
}
```

- (A) Compiler Error: there cannot be an empty derived class
- (B) Compiler Error: Derived is abstract
- (C) No compiler Error

Answer: (B)

Explanation: If we don't override the pure virtual function in derived class, then derived class also becomes abstract class.

C++ | Virtual Functions | Question 7

```
#include<iostream>

using namespace std;

class Base
{
public:
    virtual void show() = 0;
};
```

```
class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};
```

```
int main(void)
{
    Derived d;
    Base &br = d;
    br.show();
    return 0;
}
```

(A) Compiler Error in line “Base &br = d;”

(B) Empty Output

(C) In Derived

Answer: (C)

C++ | Virtual Functions | Question 9

Can a destructor be virtual?

Will the following program compile?

```
#include <iostream>
using namespace std;
class Base {
public:
virtual ~Base() {}
};
int main() {
return 0;
}
```

(A) Yes

(B) No

Answer: (A)

Explanation: A destructor can be virtual. We may want to call appropriate destructor when a base class pointer

points to a derived class object and we delete the object. If destructor is not virtual, then only the base class destructor may be called.

C++ | Virtual Functions | Question 10

```
#include<iostream>

using namespace std;

class Base {
public:
    Base() { cout<<"Constructor: Base"<<endl; }
    virtual ~Base() { cout<<"Destructor : Base"<<endl; }
};

class Derived: public Base {
public:
    Derived() { cout<<"Constructor: Derived"<<endl; }
    ~Derived() { cout<<"Destructor : Derived"<<endl; }
};

int main() {
```

```
Base *Var = new Derived();  
delete Var;  
return 0;  
}
```

(A)

Constructor: Base

Constructor: Derived

Destructor : Derived

Destructor : Base

(B)

Constructor: Base

Constructor: Derived

Destructor : Base

(C)

Constructor: Base

Constructor: Derived

Destructor : Derived

(D)

Constructor: Derived

Destructor : Derived

Answer: (A)

Explanation: Since the destructor is virtual, the derived class destructor is called which in turn calls base class destructor.

C++ | Virtual Functions | Question 11

Can static functions be virtual? Will the following program compile?

```
#include<iostream>
```

```
using namespace std;
```

```
class Test
```

```
{
```

```
public:
```

```
    virtual static void fun() { }
```

```
};
```

(A) Yes

(B) No

Answer: (B)

Explanation: Static functions are class specific and may not be called on objects. Virtual functions are called according to the pointed or referred object.

C++ | Virtual Functions | Question 12

Predict the output of following C++ program. Assume that there is no alignment and a typical implementation of virtual functions is done by the compiler.

```
#include <iostream>
```

```
using namespace std;
```

```
class A
```

```
{  
public:  
    virtual void fun();  
};
```

```
class B  
{  
public:  
    void fun();  
};
```

```
int main()  
{  
    int a = sizeof(A), b = sizeof(B);  
    if (a == b) cout << "a == b";  
    else if (a > b) cout << "a > b";  
    else cout << "a < b";  
    return 0;  
}  
  
(A) a > b  
(B) a == b
```

(C) `a < b`

(D) Compiler Error

Answer: (A)

Explanation: Class A has a VPTR which is not there in class B.

In a typical implementation of virtual functions, compiler places a VPTR with every object. Compiler secretly adds some code in every constructor to this.

C++ | Virtual Functions | Question 13

```
#include <iostream>
```

```
using namespace std;
```

```
class A
```

```
{
```

```
public:
```

```
    virtual void fun() { cout << "A::fun() "; }
```

```
};
```

```
class B: public A
{
public:
void fun() { cout << "B::fun() "; }
};
```

```
class C: public B
{
public:
void fun() { cout << "C::fun() "; }
};
```

```
int main()
{
    B *bp = new C;
    bp->fun();
    return 0;
}
```

- (A) A::fun()
- (B) B::fun()
- (C) C::fun()

Answer: (C)

Explanation: The important thing to note here is B::fun() is virtual even if we have not uses virtual keyword with it.

When a class has a virtual function, functions with same signature in all descendant classes automatically become virtual. We don't need to use virtual keyword in declaration of fun() in B and C. They are anyways virtual.

C++ | Virtual Functions | Question 14

```
#include<iostream>
```

```
using namespace std;
```

```
class Base
```

```
{
```

```
public:
```



```
        virtual void show() { cout<<" In Base \n"; }  
};
```

```
class Derived: public Base  
{  
public:  
        void show() { cout<<"In Derived \n"; }  
};
```

```
int main(void)  
{  
        Base *bp = new Derived;  
        bp->Base::show(); // Note the use of scope  
        resolution here  
        return 0;  
}
```

- (A)** In Base
- (B)** In Derived
- (C)** Compiler Error
- (D)** Runtime Error

Answer: (A)

Explanation: A base class function can be accessed with scope resolution operator even if the function is virtual.

➔ Pure Virtual Destructor in C++

➤ Can a destructor be pure virtual in C++?

Yes, it is possible to have a pure virtual destructor.

Pure virtual destructors are legal in standard C++ and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor.

➤ Why a pure virtual function requires a function body?

The reason is that destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation. This means that a derived class destructor will be invoked first, then the base class destructor will be called. If the

definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore, the compiler and linker enforce the existence of a function body for pure virtual destructors.

```
// C++ program to demonstrate if the value of  
// of pure virtual destructor are provided  
// then the program compiles & runs fine.  
  
#include <iostream>  
  
// Initialization of base class  
class Base {  
public:  
    virtual ~Base() = 0; // Pure virtual  
destructor  
};  
Base::~~Base() // Explicit destructor call  
{  
    std::cout << "Pure virtual destructor is  
called";  
}  
  
// Initialization of derived class  
class Derived : public Base {  
public:
```

```

    ~Derived() { std::cout << "~Derived() is
executed\n"; }
};

int main()
{
    // Calling of derived member function
    Base* b = new Derived();
    delete b;
    return 0;
}

```

Output

~Derived () is executed

Pure virtual destructor is called

- It is ***important to note*** that a class becomes an abstract class(at least a function that has no definition) when it contains a pure virtual destructor.

```

// C++ program to demonstrate how a class becomes
// an abstract class when a pure virtual destructor is
// passed

#include <iostream>
class Test {
public:

```

```

    virtual ~Test() = 0;
    // Test now becomes abstract class
};
Test::~Test() {}

// Driver Code
int main()
{
    Test p;
    Test* t1 = new Test;
    return 0;
}

```

The above program fails in a compilation & shows the following error messages.

Virtual Constructor in C++

The virtual mechanism works only when we have a base class pointer to a derived class object.

The creation of a virtual constructor is not possible because of the reasons given below:

- There is no virtual memory table present while calling the constructor. So, the

construction of a virtual constructor is not possible.

- . Because the object is not created, virtual construction is impossible.
- . The compiler must know the type of object before creating it.

➤ Virtual Copy Constructor in C++

<https://www.tutorialspoint.com/virtual-copy-constructor-in-cplusplus>

read this brother.

- **The abstract keyword->** 'abstract' keyword is used to declare the method or a class as abstract.

➤ Abstract Class-> A class which contains the **abstract** keyword in its declaration is known as an abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without a body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations for the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method

to be determined by child classes, you can declare the method in the parent class as an abstract.

- The **abstract** keyword is used to declare the method as abstract.
- You must place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Example

```
public abstract class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public abstract double computePay();  
    // Remainder of class definition  
}
```

Declaring a method as abstract has two consequences –

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.
- If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them and use their data members and member functions (except pure virtual functions).

Note – Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

- **Final keyword in java:** The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Ex: **final int** speedlimit=90;//final variable.

2) Java final method

If you make any method as final, you cannot override it.

3) Java final class

If you make any class as final, you cannot extend it.

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it.

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Que) Can we initialize blank final variable?

Yes, but only in constructor.

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Use of explicit keyword in C++

Explicit Keyword in C++ is used to mark constructors to not implicitly convert types in C++. It is optional for constructors that take exactly one argument and work on constructors (with a single argument) since those are the only constructors that can be used in typecasting.

```
// C++ program to illustrate default
// constructor without 'explicit'
// keyword
#include <iostream>
using namespace std;
```

```
class Complex {
private:
    double real;
    double imag;

public:

    // Parameterized constructor
    Complex(double r = 0.0,
            double i = 0.0) : real(r),
                               imag(i)
    {
    }

    // A method to compare two
    // Complex numbers
    bool operator == (Complex rhs)
    {
        return (real == rhs.real &&
                imag == rhs.imag);
    }
};

// Driver Code
int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "Not Same";
}
```

```
    return 0;  
}
```

Output

Same

As discussed in [this article](#), in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes a conversion constructor because such a constructor allows conversion of the single argument to the class being constructed. In this case, when `com1 == 3.0` is called, 3.0 is implicitly converted to Complex type because the default constructor can be called with only 1 argument because both parameters are default arguments and we can choose not to provide them.

We can avoid such implicit conversions as these may lead to unexpected results. We can make the constructor explicit with the help of an **explicit keyword**. For example, if we try the following program that uses explicit keywords with a constructor, we get a compilation error.

```
// C++ program to illustrate  
// default constructor with
```

```
// 'explicit' keyword
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
    // Default constructor
    explicit Complex(double r = 0.0,
                     double i = 0.0) :
        real(r), imag(i)
    {
    }

    // A method to compare two
    // Complex numbers
    bool operator == (Complex rhs)
    {
        return (real == rhs.real &&
                imag == rhs.imag);
    }
};

// Driver Code
int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);
}
```

```
    if (com1 == 3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}
```

Output: no match for 'operator==' in 'com1 == 3.0e+0'

We receive an error here because to avoid any unexpected errors we have made our constructor an explicit constructor and 3.0 won't be converted to Complex by our constructor on its own.

We can still typecast the double values to Complex, but now we have to explicitly typecast it. For example, the following program works fine.

```
// C++ program to illustrate
// default constructor with
// 'explicit' keyword
#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
```



```

// Default constructor
explicit Complex(double r = 0.0,
                 double i = 0.0):
    real(r) , imag(i)
{
}

// A method to compare two
// Complex numbers
bool operator == (Complex rhs)
{
    return (real == rhs.real &&
            imag == rhs.imag);
}

};

// Driver Code
int main()
{
    // a Complex object
    Complex com1(3.0, 0.0);

    if (com1 == (Complex)3.0)
        cout << "Same";
    else
        cout << "Not Same";
    return 0;
}

```

Output

Same

Note: The explicit specifier can be used with a constant expression. However, if that constant expression evaluates to true, then only the function is explicit.

Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Super Keyword

1

Super can be used to refer immediate parent class instance variable.

2

Super can be used to invoke immediate parent class method.

3

super() can be used to invoke immediate parent class constructor.

Note: *super () is added in each class constructor automatically by compiler if there is no super () or this ().*

super example: real use

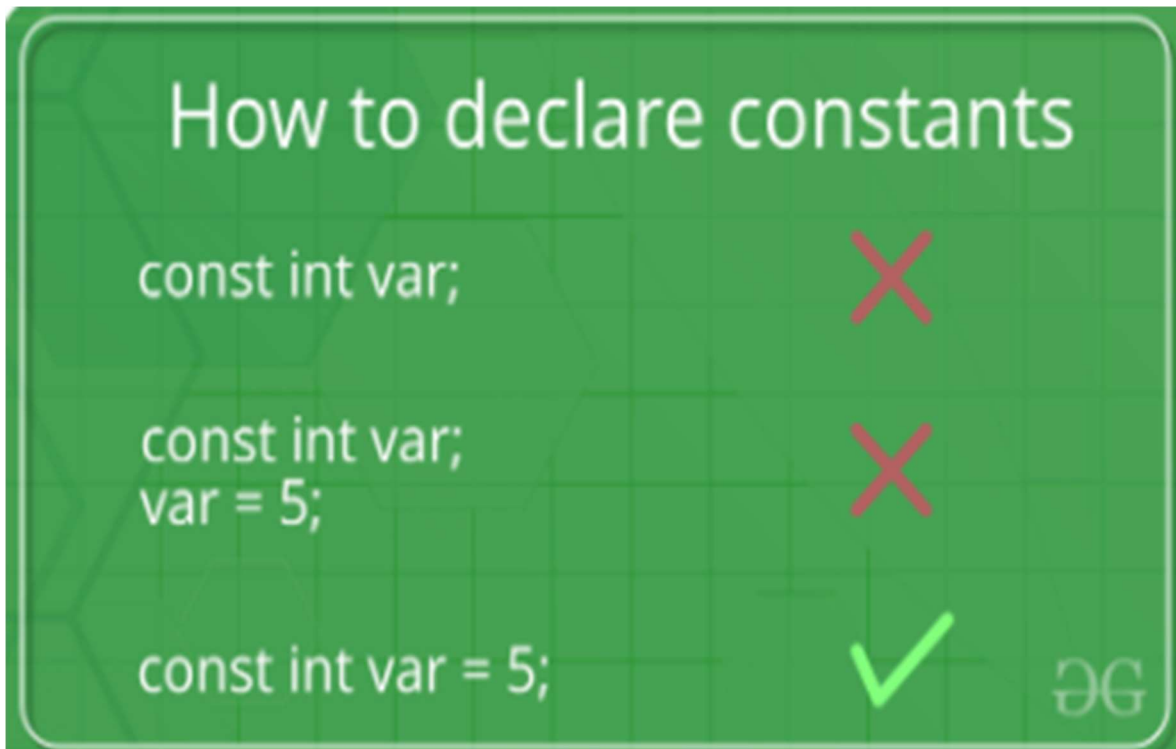
Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

Const keyword in C++

The various functions of the const keyword which is found in C++ are discussed. Whenever **const** keyword is attached with any **method()**, variable, pointer variable, and with the object of a class it prevents that specific object/method()/variable to modify its data items value.

There are a certain set of rules for the declaration and initialization of the constant variables:

- . The const variable cannot be left un-initialized at the time of the assignment.
- . It cannot be assigned value anywhere in the program.
- . Explicit value needed to be provided to the constant variable at the time of declaration of the constant variable.



Const Keyword With Pointer Variables:

Pointers can be declared with a const keyword. So, there are three possible ways to use a const keyword with a pointer, which are as follows:

When the pointer variable point to a const value:

<https://www.geeksforgeeks.org/const-keyword-in-cpp/>

use this documentation only.

Features of OOPs

➤ **Encapsulation:**

Encapsulation is process of binding the code and data into a single unit. whenever we need to restrict the data from outside of class people then we can make our data members as private.

Encapsulation refers to bundling data and the methods that operate on that data into a single unit. Many programming languages use encapsulation frequently in the form of classes. A class is an example of encapsulation in computer science in that it consists of data and methods that have been bundled into a single unit.

Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object.

Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.

➤ Encapsulation is combo of Data hiding & Abstraction.

In other words: Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.

The general idea of this mechanism is simple. For example, you have an attribute that is not visible from the outside of an object. You bundle it with methods that provide read or write access.

Encapsulation allows you to hide specific information and control access to the object's internal state.

➤ We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Ex:

```
#include <iostream>
```

```
using namespace std;
```

```
class Student{
```

```
private:
```

```
    string studentName;
```

```
    int studentRollno;
```

```
    int studentAge;
```

```
public:
```

```
    string getStudentName() {
```

```
return studentName;
```

```
    }
```

```
    void setStudentName(string studentName) {
```

```
this-> studentName = studentName;
```

```
}
```



```
int getStudentRollno() {  
    return studentRollno;  
}
```

```
void setStudentRollno (int studentRollno) {  
    this-> studentRollno = studentRollno;  
}
```

```
Int getStudentAge() {  
    returnstudentAge;  
}
```

```
Void setStudentAge(int studentAge) {  
    this-> studentAge = studentAge;  
}  
};
```

```
Int main () {  
    Student obj;  
    obj.setStudentName("Avinash");  
    obj.setStudentRollno(101);  
    obj.setStudentAge(22);  
    cout << "Student Name: " << obj.getStudentName()  
    << endl;  
    cout << "Student Rollno : " <<obj.getStudentRollno()  
    << endl;  
    cout << "Student Age : " << obj.getStudentAge();  
    }
```

Output:

Student Name : Avinash

Student Rollno : 101

Student Age : 22

➤ How can Encapsulation be achieved?

Ans: Using access modifier.

Advantage of Encapsulation

- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**.
Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is easy to test. So, it is better for unit testing.

How Encapsulation is achieved in a class

- Make all the data members private.

- Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.
- QUE 1: can we achieve encapsulation using public access modifier.
- QUE 2: how encapsulation is achieved in a class with example.
- QUE 3: can we achieve encapsulation without data hiding.

Abstraction: Implementation hiding

- An abstraction is a way of hiding the implementation details and showing only the functionality to the users. In other words, it ignores the irrelevant details and shows only the required one.

➤ Because the user is not interested to know the implementation.

➤ It is also safe from the security point of view.

➤ We can implement Abstraction in C++

1) Using classes, the class helps us to group data members and member functions using available access specifiers. A Class can decide which data members will be visible to the outside world and not. Access specifiers are the main pillar of implementing abstraction in C++.

2) One more type of Abstraction in C++ can be header files.

Ex: The `pow()` function is present in header file `math.h` to calculate power of a number, we can simply call a function, pass arguments without how it evaluate the answer.

➤ How to Achieve Abstraction in Java?

Using Abstract Class: Abstract classes are the same as normal Java classes the difference is only that an abstract class uses abstract keyword while the normal Java class does not use. We use the abstract keyword before the class name to declare the class as abstract.

Note: Using an abstract class, we can achieve 0-100% abstraction.

Remember that, we cannot instantiate (create an object) an abstract class. An abstract class contains abstract methods as well as concrete methods. If we want to use an abstract class, we have to inherit it from the base class.

NOTE: In C++ a class is abstract class when it contains at least one virtual function. Abstraction and Abstract class are same in java and C++, only is the difference in declaration.

If the class does not have the implementation of all the methods of the interface, we should declare the class as abstract. It provides complete abstraction. It means that fields are public static and final by default and methods are empty.

The syntax of abstract class is:

```
public abstract class ClassName {  
  
    public abstract methodName();  
  
}
```

Using Interface

In Java, an **interface** is similar to **Java classes**. The difference is only that an interface contains empty methods (methods that do not have method implementation) and variables. In other words, it is a collection of abstract methods (the method that does not have a method body) and static constants. The important point about an interface is that each method is **public** and **abstract** and does not contain any constructor. Along with the abstraction, it also helps to achieve multiple inheritance. The implementation of these methods provided by the clients when they implement the interface.

Note: Using interface, we can achieve 100% abstraction.

Separating interface from implementation is one way to achieve abstraction. The **Collection framework** is an excellent example of it.

Features of Interface:

- We can achieve total abstraction.
- We can use multiple interfaces in a class that leads to multiple inheritance.

- It also helps to achieve loose coupling.

Syntax:

1. **public interface** XYZ
2. {
3. **public void** method();
4. }

To use an interface in a class, Java provides a keyword called **implements**. We provide the necessary implementation of the method that we have declared in the interface.

Let's see an example of an interface.

Car.java

1. **interface** CarStart
2. {
3. **void** start();
4. }
5. **interface** CarStop
6. {
7. **void** stop();
8. }
9. **public class** Car **implements** CarStart, CarStop
10. {
11. **public void** start()
12. {
13. System.out.println("The car engine has been started.");
14. }
15. **public void** stop()


```

16.    {
17.    System.out.println("The car engine has been stopped.");

18.    }
19.    public static void main(String args[])
20.    {
21.    Car c = new Car();
22.    c.start();
23.    c.stop();
24.    }
25.    }

```

Output:

```

The car engine has been started.
The car engine has been stopped.

```

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.

3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: public abstract class Shape{ public abstract void draw(); }	Example: public interface Drawable{ void draw(); }

Abstraction is hiding complex code.

- Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Abstraction includes hiding the implementation part and showing only the required data and features to the user. It is done to hide the implementation complexity and details from the user. And to provide a good interface in programming.

- What is the difference between encapsulation and abstraction.

Inheritance:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can create new [classes](#) that are built upon existing classes. When you inherit from an existing class, you

can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

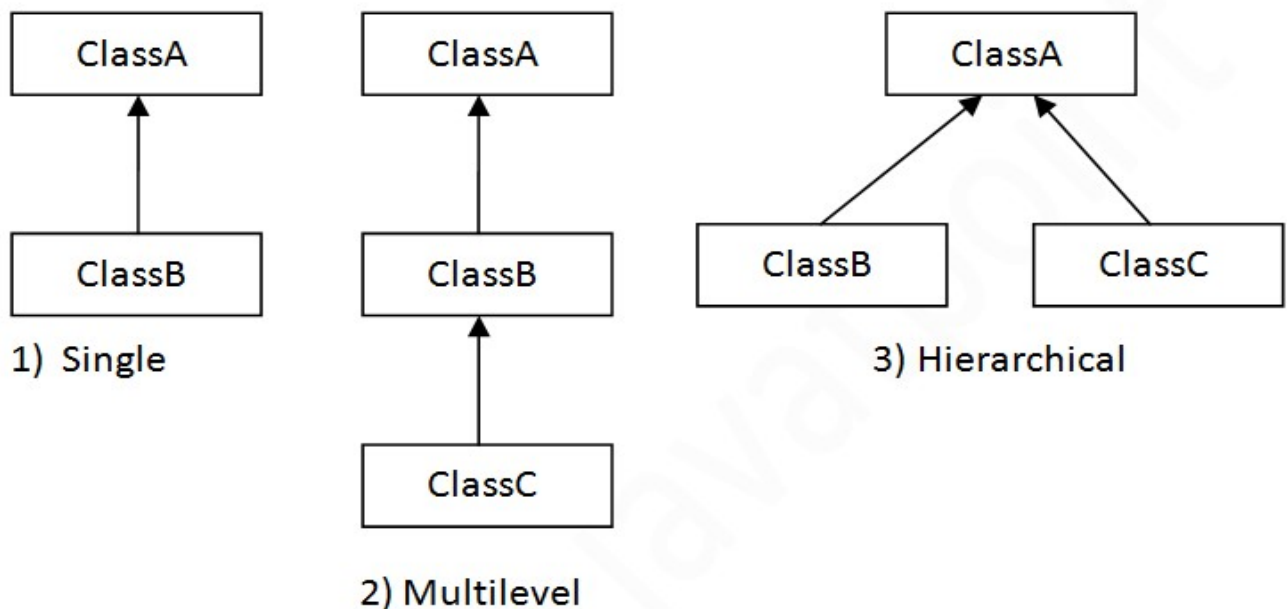
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in java:

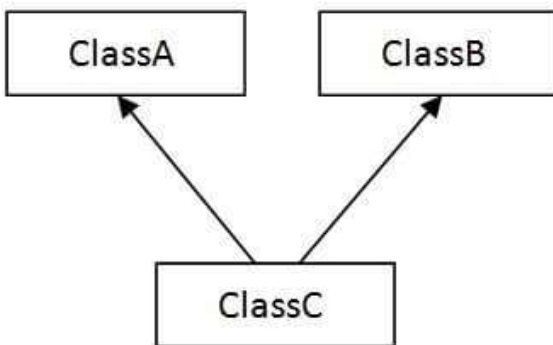
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

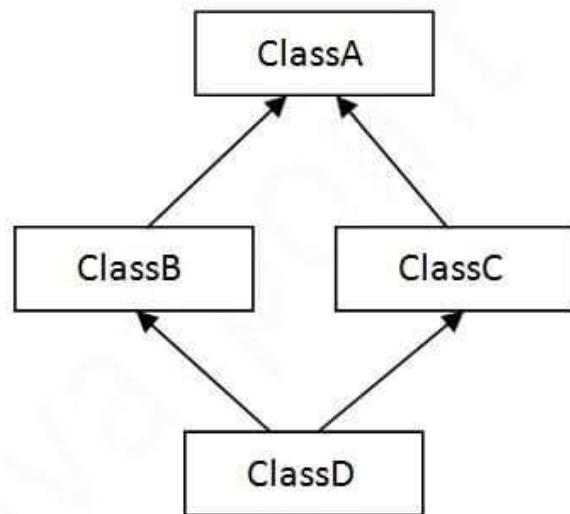


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



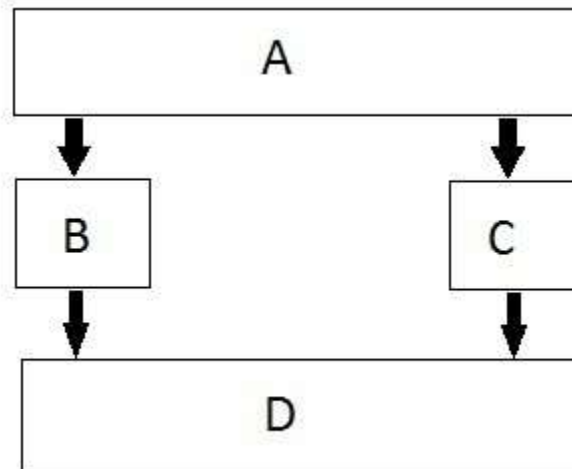
4) Multiple



5) Hybrid

Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit

2 classes. So whether you have same method or different, there will be compile time error.

Syntax:

```
class parent_class {  
    //Body of parent class  
};
```

```
class child_class: access_modifier parent_class  
{  
    //Body of child class  
};
```

➤ Modes of Inheritance

- ➔ 1. Public mode: If we derive a subclass from a public base class. Then, the base class's public members will become public in the derived class, and protected class members will become protected in the derived class.

- ➔ 2. Protected mode: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
- ➔ 3. Private mode: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.
Example:

➤ **what is the need of Inheritance?**

That is, we have utilized the concept of reusability as we have reused the code written in Person class over and over again.

It concludes that the main aim of inheritance is to implement the concept of reusability, saving our time and resources and also creating better connections between different classes, and achieve method overriding.

inheritance is basically used to add new features to an existing class. Sort of like how a child inherits properties of his parents with some new features of his own.

- ⇒ Polymorphism is a concept that allows you to perform a single action in different ways.
- ⇒ Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms. Let's understand polymorphism with a real-life example.
- ⇒ Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms. Let's understand polymorphism with a real-life example.

There are two types of polymorphism in C++

❖ Compile Time Polymorphism:

Compile-time polymorphism is also known as static polymorphism. This type of polymorphism can be achieved through function overloading or operator overloading.

a) Function overloading: When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of function overloading is that it increases the program's readability. Functions can be overloaded by using different numbers of arguments or by using different types of arguments. We have already discussed function overloading in detail in the previous module.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

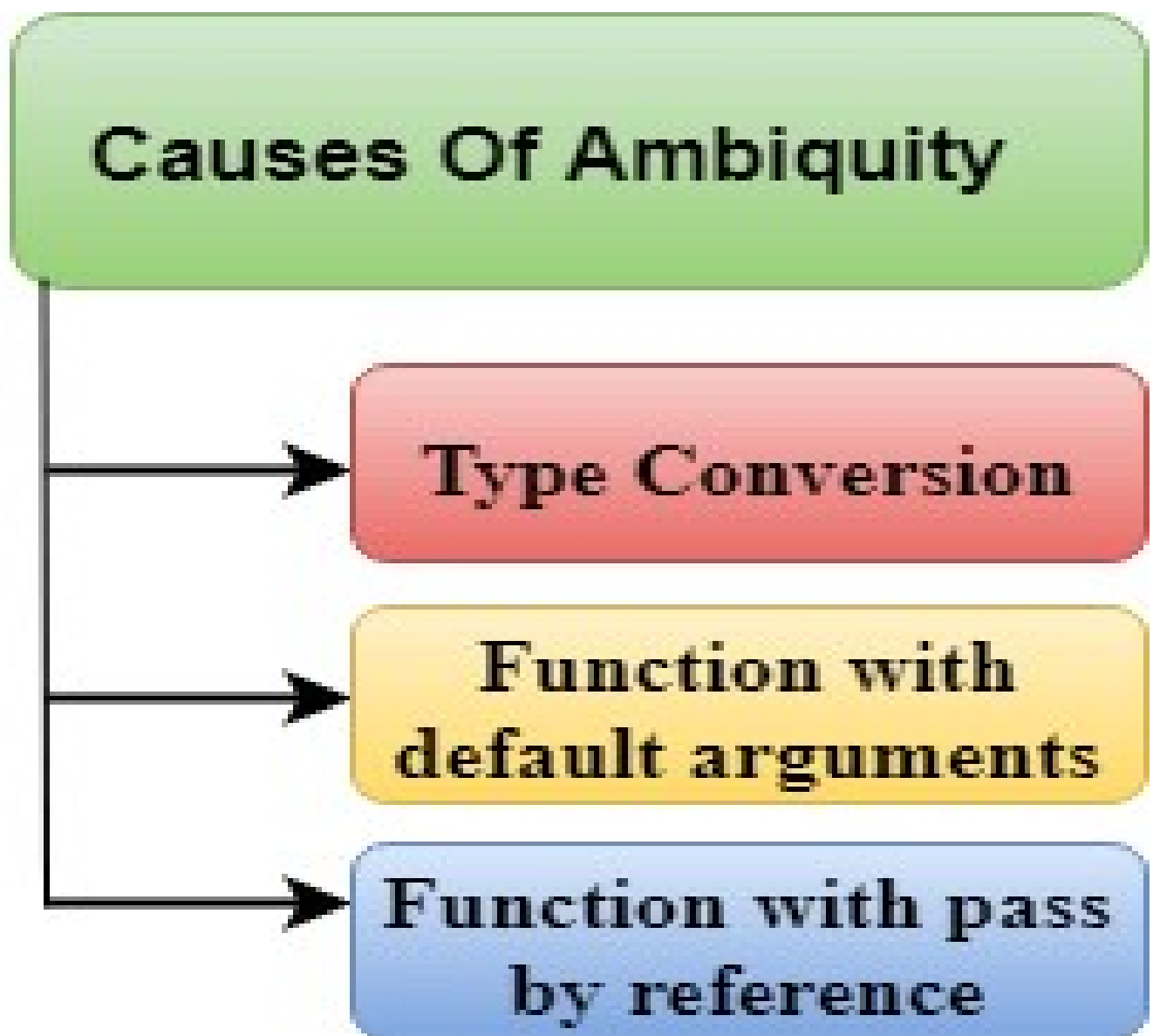
Function Overloading and Ambiguity:

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



- Type Conversion:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
    std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls

the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;

    std::cout << "Value of b is : " <<b<< std::endl;

}
```

```
int main()
{
    fun(12);

    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
    int a=10;
    fun(a); // error, which f()?
    return 0;
}
void fun(int x)
{
    std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
    std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument.

In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the `fun(int)` and `fun(int &)`.

b) Operator Overloading: C++ also provides options to overload operators. For example, we can make the operator ('+') for the string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. When placed between integer operands, a single operator, '+,' adds them and concatenates them when placed between string operands.

-> Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).

=> The overloaded operator contains at least 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.
-
- Operators = and & are already overloaded in C++ to avoid overloading them.
 - The precedence and associativity of operators remain intact.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

These operators cannot be overloaded because if we overload them it will make serious programming issues.

For an example the sizeof operator returns the size of the object or datatype as an operand. This is evaluated by the compiler. It cannot be evaluated during runtime. So we cannot overload it.

➤ Example: Perform the addition of two imaginary or complex numbers:

```
#include<iostream>
```

```
using namespace std;

class Complex{
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i = 0) {
            real = r;
            imag = i;
        }
}
```

```
// This is automatically called when '+' is used with
// between two Complex objects
```

```
Complex operator+ (Complex const& b) {
    Complex a;
    a.real = real + b.real;
    a.imag = imag + b.imag;
    return a;
}
```

```

}

void print() {
    cout << real << " + i" << imag << endl;
}

};

int main() {
    Complex c1(10,5), c2(2,4);
    Complex c3 = c1 + c2;
    // An example call to "operator+"
    c3.print();
}

```

Output: 12+ i9

❖ Runtime polymorphism:

Runtime polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymorphism.

Runtime polymorphism or Dynamic Method

Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

➤ **Method overriding:** Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. In other words, whatever methods the parent class has by default are available in the child class. But, sometimes, a child class may not be satisfied with parent class method implementation. The child class is allowed to redefine that method based on its requirement. This process is called method overriding.

➤ **Rules for method overriding:**

- The parent class method and the method of the child class must have the same name.
- The parent class method and the method of the child class must have the same parameters.

- It is possible through inheritance only.

Example:

```
#include<iostream>
using namespace std;
    class Parent{
        public:
            void show() {
                cout<<"Inside parent class"<<endl;
            }
    };

    class subclass1:public Parent {
        public:
            void show() {
                cout<<"Inside subclass1"<<endl;
            }
    };

    class subclass2:public Parent {
        public:
            void show() {
```



```
        cout<<"Inside subclass2";  
    }  
};
```

```
Int main() {  
    subclass1 o1;  
    subclass2 o2;  
    o1.show();  
    o2.show();  
}
```

Output:

```
Inside subclass1  
Inside subclass2
```

Interview Questions:

1. What is Encapsulation in C++? Why is it called Data hiding?

=> The process of binding data and corresponding methods (behavior) into a single unit is called encapsulation in C++. In other words, encapsulation is a programming technique that binds the class members (variables and methods)

together and prevents them from accessing other classes. Thereby we can keep variables and methods safe from outside interference and misuse. If a field is declared private in the class, it cannot be accessed by anyone outside the class and hides the fields. Therefore, Encapsulation is also called data hiding.

2. What is the difference between Abstraction and Encapsulation?

=>Abstraction

- > Abstraction is the method of hiding unnecessary details from the necessary ones.
- > Achieved through encapsulation.
- > Abstraction allows you to focus on what the object does instead of how it does it.
- > In abstraction, problems are solved at the design or interface level.

=>Encapsulation

- > Encapsulation is the process of binding data members and methods of a program together to do

a specific job without revealing unnecessary details.

-> You can implement encapsulation using Access Modifiers (Public, Protected & Private.)

-> Encapsulation enables you to hide the code and data into a single unit to secure the data from the outside world.

-> While in encapsulation, problems are solved at the implementation level.

4. Are there any limitations of Inheritance?

=> Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it also has limitations.

Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not

correctly implemented, this might lead to unexpected errors or incorrect outputs.

5. What is the difference between overloading and overriding?

=>Overloading is a compile-time polymorphism feature in which an entity has multiple implementations with the same name—for example, Method overloading and Operator overloading. Whereas Overriding is a runtime polymorphism feature in which an entity has the same name, but its implementation changes during execution. For example, Method overriding.

7. What are the advantages of Polymorphism?

=>There are the following advantages of polymorphism in C++:

a. Using polymorphism, we can achieve flexibility in our code because we can perform various

operations by using methods with the same names according to requirements.

b. The main benefit of using polymorphism is when we can provide implementation to an abstract base class or an interface.

8. What are the differences between Polymorphism and Inheritance in C++?

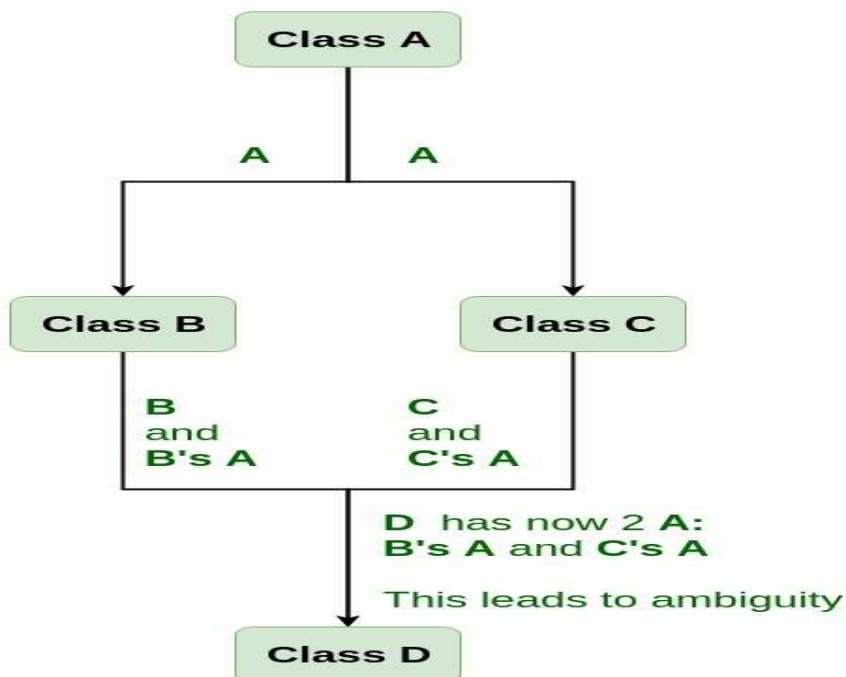
=>The differences between polymorphism and inheritance in C++ are as follows:

a. Inheritance represents the parent-child relationship between two classes. On the other hand, polymorphism takes advantage of that relationship to make the program more dynamic.

b. Inheritance helps in code reusability in child class by inheriting behavior from the parent class. On the other hand, polymorphism enables child class to redefine already defined behavior inside parent class. Without polymorphism, a child class can't execute its own behavior.

Virtual base class in C++

Need for Virtual Base Classes: Consider the situation where we have one class **A**. This class **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below. As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.



Note:

virtual can be written before or after the **public**.

Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
public:
```

```
    int a;
```

```
    A(){
```

```
        a = 10;
```

```
}
```

```
};
```

```
class B : public virtual A {
```

```
};
```

```
class C : public virtual A {
```

```
};
```

```
class D : public B, public C {
```

```
};
```

```
int main()
```

```
{
```

```
    D object;
```

```
    cout << "a = " << object.a << endl;
```

```
    return 0;
```

```
}
```


Can Virtual Functions be Private in C++?

// C++ program to demonstrate how a

// virtual function can be private

```
#include <iostream>
```

```
class base {
```

```
public:
```

```
    base() {
```

```
        std::cout << "base class constructor\n";
```

```
    }
```

```
    virtual ~base(){
```

```
        std::cout << "base class destructor\n";
```

```
    }
```

```
    void show(){
```

```
        std::cout << "show() called on base class\n";
```

```
    }
```

```
virtual void print(){  
    std::cout << "print() called on base class\n";  
}  
};
```

```
class derived : public base {  
public:  
    derived() : base(){  
        std::cout << "derived class constructor\n";  
    }
```

```
virtual ~derived(){  
    std::cout << "derived class destructor\n";  
}
```

```
private:  
virtual void print(){  
    std::cout << "print() called on derived  
class\n";
```

```
    }  
};  
  
int main(){  
    std::cout << "printing with base class pointer\n";  
  
    base* b_ptr = new derived();  
    b_ptr->show();  
    b_ptr->print();  
    delete b_ptr;  
}
```

Output

```
printing with base class pointer  
base class constructor  
derived class constructor  
show() called on base class  
print() called on derived class  
derived class destructor  
base class destructor
```

Explanation: '*b_ptr*' is a pointer of Base type and points to a Derived class object. When pointer '*ptr->print()*' is called, function '*print()*' of Derived is executed.

This code works because the base class defines a public interface and the derived class overrides it in its implementation even though the derived has a private virtual function.

➤ **Friend Function:** If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

A class's friend function is defined outside that class's scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions. A friend function in C++ is a function that is preceded by the keyword "friend."

```
class class_name{  
friend data_type function_name(argument);  
    // syntax of friend function.  
};
```

The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator.

```
#include <iostream>
using namespace std;
class Rectangle{
    private:
    int length;
    public:
    Rectangle() {
        length =10;
    }

    Friend int printLength(Rectangle);
//friend function
};
```

```
Int printLength(Rectangle b) {  
    b.length +=10;  
    return b.length;  
}  
  
int main() {  
    Rectangle b;  
    cout << "Length of Rectangle: " <<  
    printLength(b) << endl;  
    return 0;  
}
```

Output:

Length of Rectangle: 20

Characteristics of friend function:

- A friend function can be declared in the private or public section of the class.

- It can be called a normal function without using the object.
- A friend function is not in the scope of the class, of which it is a friend.
- A friend function is not invoked using the class object as it is not in the class's scope.
- A friend function cannot access the private and protected data members of the class directly. It needs to make use of a class object and then access the members using the dot operator.
- A friend function can be a global function or a member of another class.

Interview Questions:

1. Does every virtual function need to be always overridden?

No, It is not always mandatory to redefine a virtual function. It can be used as it is in the base class.

2. What is an abstract class?

An abstract class is a class that has at least one pure virtual function in its definition. An abstract class can never be instanced (creating an object). It can only be inherited, and the methods could be overwritten.

3. Can we have a constructor as Virtual?

Constructors cannot be virtual because they need to be defined in the class.

4. What is a pure virtual function?

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation. We only declare it. A pure virtual function is declared by assigning 0 in the declaration. See the following example.

5. What are the characteristics of Friend Function?

- ★ A friend function is not in the scope of the class, in which it has been declared as friend.
- ★ It cannot be called using the object of that class.
- ★ It can be invoked like a normal function without any object.
- ★ Unlike member functions, it cannot use the member names directly.
- ★ It can be declared in public or private parts without affecting its meaning.
- ★ Usually, it has objects as arguments.

6. What is the output of this program?

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

```
class Box {
    double width;
    public:
    friend void printWidth( Box box );
    void setWidth(doublewid );
};

Void Box::setWidth(double wid ) {
    width = wid;
}

Void printWidth( Box box ) {
    box.width = box.width *2;
    cout<<"Width of box : "<< box.width
<<endl;
}

Int main( ) {
    Box box;
```

```
    box.setWidth(10.0);  
    printWidth( box );  
    return 0;  
}
```

Answer:

20

Explanation: We are using the friend function for print width and multiplied the width value by 2, So we got the output as 20

First of all, **What is polymorphism?** In layman terms, it means “**Many Forms**”.

What we exactly mean in Java is that a single line of code (*Here, single line of code can refer to the name of the method, definition, variables etc*) can have multiple meaning, and which

meaning is to be interpreted depends on various factors.

- . Consider about your mobile phone. You can save contacts in it. Now suppose you want to save 2 numbers of a same person. You can do it by saving under the same name of that person.*

Similarly, In java, suppose you want to save two numbers of the same person. Suppose you do this by passing the mobile numbers as arguments to a method named void number(int parameter1, int parameter2).

Now its not that every person will have 2 numbers. It might be possible that others may have only 1 number. So instead of creating another method with different name which takes only 1 parameter, what we can do is we can have the same name of the method i.e number() but instead of taking 2 parameters ,

we can take only 1 parameter i.e void number(int parameter1).

Now, here comes the role of polymorphism. There is only 1 method named number(), but it has two definitions. Now which definition is to be executed depends upon the number of parameters passed.

This is also known as method overriding.

So as you can see, it makes the code more simple, more readable and thus reduces the complexity of reading and is easy to implement.

Lets see another example of polymorphism.

- . Suppose you go to an ice cream parlour and then you take one vanilla flavoured ice cream. Now you go to another ice cream parlour(different branch of the same company) and you find there vanilla with chocolate flavour. Now, you wanted to have the vanilla with chocolate flavour, but it was*

not present in the first branch while it was present in the other branch of the same ice cream company.

Now relating this with java,

Suppose you have a class named icecream which includes a method named dinshaws().

Now in this method, you have only vanilla flavour. Now since you have another branch of that ice cream company, suppose there is another class named Branch with extends Icecream . Now , this branch also has dinshaws icecream which has both vanilla as well as vanilla with chocolate flavour.

Now, instead of creating another method and then adding both the flavours definition in it, we can add definition of the vanilla with chocolate flavour in the same method i.e in dinshaws() only, which already included the flavour of vanilla. Thus , Now the method

named `dinshaws()` have two definitions- one with only vanilla flavour and one with both vanilla and vanilla with chocolate. Which method gets invoked depends upon the object type i.e. of which class the object has been made.

This is also known as Method Overriding.

Thus, Polymorphism makes the code more simpler. It also reduces the complexity of reading and also saves a lot of lines of codes.

AFTER THIS PLEASE GO THROUGH

<https://whimsical.com/object-oriented-programming-cheatsheet-by-love-babbar-YbSgLatbWQ4R5paV7EgqFw>

this ashish rathore