OOPS

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

- 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.
- Class- Collection of objects is called class. It is a logical entity.
- Inheritance- When one object acquires all the properties and behaviors of the parent object. It provides code reusability. It is used to achieve runtime polymorphism.
- Polymorphism- In C++, we use Function overloading and Function overriding to achieve polymorphism.
- Abstraction- Hiding internal details and showing functionality is known as abstraction.
- Encapsulation- Binding (or wrapping) code and data together into a single unit is known as encapsulation.

Advantage of OOPs over Procedure-oriented programming language

https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf

1. OOPs makes development and maintenance easier whereas in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provides the ability to simulate real-world events much more effectively. We can provide the solution of real word problems if we are using the Object-Oriented Programming language.

Object

Is an entity that has state and behavior, it is created at runtime. Object is an instance of a class. All the members of the class can be accessed through object.

Class

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. Eg-

```
class Student
 {
     public:
     int id;  //field or data member
     float salary; //field or data member
     String name;//field or data member
 }
Student s1; //object of class student
```

```
#include <iostream>
using namespace std;
class Student {
   public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Here we used the object to display class data

```
#include <iostream>
using namespace std;
class Student {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       void insert(int i, string n)    //member function can be define inline like this
        {
            id = i;
            name = n;
        }
//or it can be declared in the class and then defined externally using scope resolution operator
       void display();
};
void Student :: display()
{
       cout<<id<<"  "<<name<<endl;
}
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```
Here we created a member function in class that is used to display data

Access Specifiers

Access specifiers define how the members (attributes and methods) of a class can be accessed. In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

Note- It is possible to access private members of a class using a public method inside the same class.
By default, all members of a class are private if you don't specify an access specifier.

Encapsulation

The meaning of Encapsulation is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public get and set methods.

```
#include <iostream>
using namespace std;
class Employee {
  private:
          // Private attribute
          int salary;
  Public:
           // Setter
           void setSalary(int s) {
          salary = s;
          }
           // Getter
          int getSalary() {
           return salary;
          }
};
int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

Constructors in C++

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure.

There can be many types of constructors in C++.

- Default constructor- A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.
- Parameterized constructor- A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.
- Copy constructor- A copy constructor is a member function which initializes an object using another object of the same class.
- Virtual constructor
- Virtual copy constructor

Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

Copy Constructor-

In C++, a Copy Constructor may be called in the following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

4. When the compiler generates a temporary object.

Eg-

```
#include <bits/stdc++.h>
using namespace std;
class MyClass
{
	int x;
public:
	MyClass(int s)
	{
	cout<<"Normal constructor called\n";
	x = s;
	}
	MyClass(const MyClass &ob)
	{
	cout<<"Copy constructor called\n";
	x = ob.x;
	}

};
MyClass f(MyClass ob)
{
	cout<<"H\n";
	return ob;
}
int main()
{
	MyClass ob1(5);
	MyClass ob2(ob1);
	MyClass ob3 = f(ob2);
	return 0;
}
```

Output:

```
Normal constructor called
Copy constructor called
Copy constructor called
H
Copy constructor called
```

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases.

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 copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore the compiler doesn’t allow parameters to be passed by value.

The compiler can create its own default copy constructor too if not provided with one. Default copy constructors can only do a shallow copy.

Shallow Copy: Shallow repetition is quicker. However, it's "lazy" it handles pointers and references. Rather than creating a contemporary copy of the particular knowledge the pointer points to, it simply copies over the pointer price. So, each the first and therefore the copy can have pointers that reference constant underlying knowledge.

Deep Copy: Deep repetition truly clones the underlying data. It is not shared between the first and therefore the copy.

Shallow Copy	**Deep Copy**
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect 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.	Deep copy stores the copy of the original object and recursively copies the objects as well.
Shallow copy is faster.	Deep copy is comparatively slower.

C++ Doesn't allow virtual constructors

Virtual copy constructor -

Virtual Copy Constructor in C++

Note- C++ constructors can be declared as private and then we may access them using:

- Friend function
- Singleton design pattern
- Named constructor idiom

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.
A destructor is defined like a constructor. It must have the same name as class. But it is prefixed with a tilde sign (~).

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
        Employee(int i, string n, float s)  //parameterized constructor
        {
            cout<<”Parameterized constructor invoked”<<endl;
            id = i;
            name = n;
            salary = s;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main()
{
    Employee e1; //creating an object of Employee
    Employee e2 = Employee(102, “Sonoo”, 98000);
    return 0;
}
```

Output:

```
Default Constructor Invoked
Parameterized constructor invoked
Destructor Invoked
Destructor Invoked
```

Private destructor- possible but can only be used with dynamic memory allocation, which makes the object the programmer's responsibility to delete. But then again if the programmer does try to delete it then it will throw an error unless done with a friend function. Refer for code examples-
Private Destructor

Virtual Destructor
Pure virtual destructor in C++

Structs in C

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.

Declaration	struct structure_name{
	type struct_member 1;
	type struct_member 2;
	type struct_member 3;
	.
	type struct_memberN;
	};
		class class_name{
		data member;
		member function;
		};

Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.
Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.

Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

Structs in C++ are able to perform inheritance and can also include member functions. They become much more similar to class in C++.
In C++, a structure is the same as a class except for a few differences.

- The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end-user while a class is secure and can hide its programming and designing details.
- Class can have null values but the structure can not have null values.
- When deriving a struct from a class/struct, the default access-specifier for a base class/struct is public. And when deriving a class, the default access specifier is private.

Enum in C++

A data type that contains a fixed set of constants. The C++ enum constants are static and final implicitly.

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
   week day;
   day = Friday;
   cout << "Day: " << day+1<<endl;
   return 0;
}
```

Output:

```
Day: 5
```

C++ 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. The declaration of a friend function should be done inside the body of the class starting with the keyword friend. 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.

Eg1-

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Eg2-

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
   int x;
   public:
   void setdata(int i)
   {
      x=i;
   }
   friend void min(A,B);         // friend function.
};
class B
{
   int y;
```

```
    public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);                    // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
    std::cout << a.x << std::endl;
    else
    std::cout << b.y << std::endl;
}
   int main()
{
   A a;
   B b;
   a.setdata(10);
   b.setdata(20);
   min(a,b);
    return 0;
 }
```

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Eg-

```
#include <iostream>
using namespace std;
class A
{
    int x =5;
    friend class B;           // friend class.
};
class B
{
  public:
    void display(A &a)
    {
        cout<<"value of x is : "<<a.x;
    }
};
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
Output:
Value of x is : 5
```

Real world analogy of classes and objects:

Real Life Examples of Object Oriented Programming

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations.

Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

- **Function Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments. Eg-

```
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }
    void func(int x, int y)
    {
        cout << "vvalue of x and y is " << x << ", " << y << endl;
    }
};
int main() {
    Geeks obj1;
    obj1.func(7);
    obj1.func(9.132);
    obj1.func(85,64);
    return 0;
}
```

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

- **Operator Overloading**: C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them. Eg-

```
#include<iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    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: This type of polymorphism is achieved by Function Overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden. Eg-

```
#include <bits/stdc++.h>
using namespace std;
class base
{
public:
   virtual void print ()
   { cout<< "print base class" <<endl; }

   void show ()
   { cout<< "show base class" <<endl; }
};
class derived:public base
{
public:
   void print () //print () is already virtual function in derived class, we could also declared as
virtual void print () explicitly
   { cout<< "print derived class" <<endl; }
   void show ()
   { cout<< "show derived class" <<endl; }
};
int main()
{
   base *bptr;
   derived d;
   bptr = &d;
   bptr->print();
   bptr->show();
   return 0;
}
```

Note:

Functions that cannot be overloaded in C++

Operators that cannot be overloaded in C++

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Inheritance

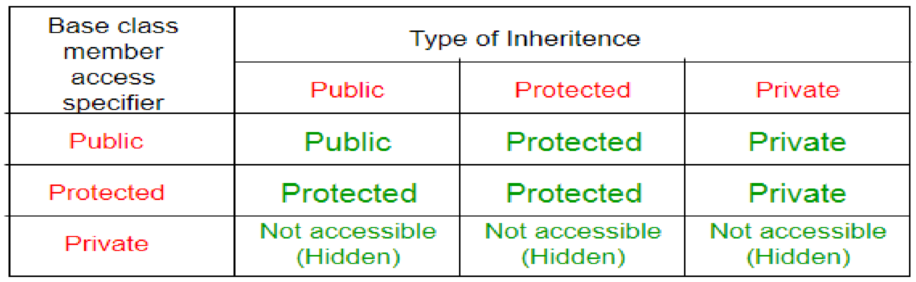
The capability of a class to derive properties and characteristics from another class is called Inheritance.
Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Syntax:
```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```
Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Modes of Inheritance

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

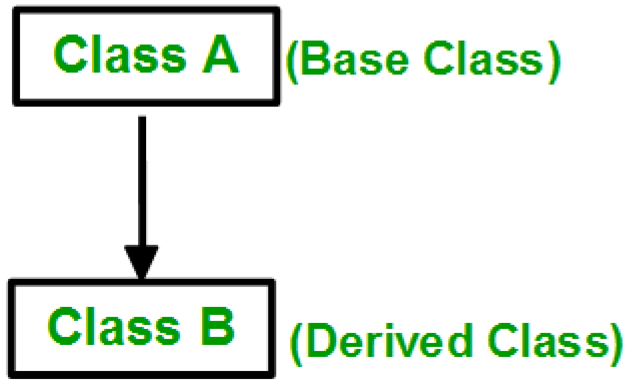
```
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};
class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};
class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};
class D : private A      // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

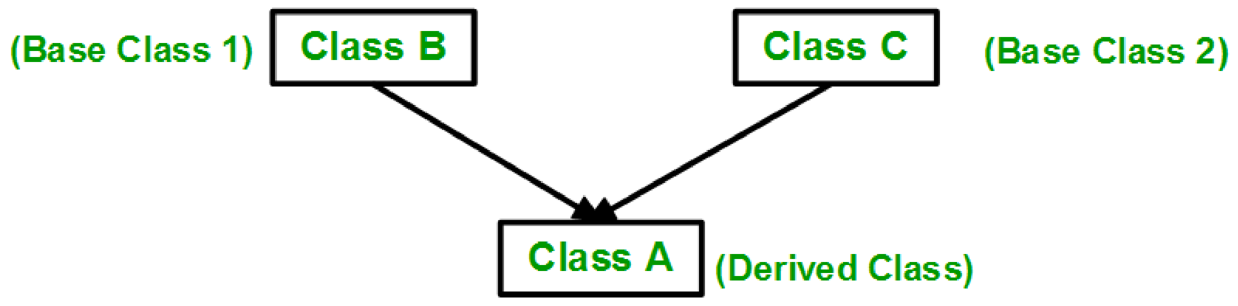
1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Eg-

```
#include <iostream>
using namespace std;
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class Car: public Vehicle{

};
int main()
{
    Car obj;
    return 0;
}
```

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

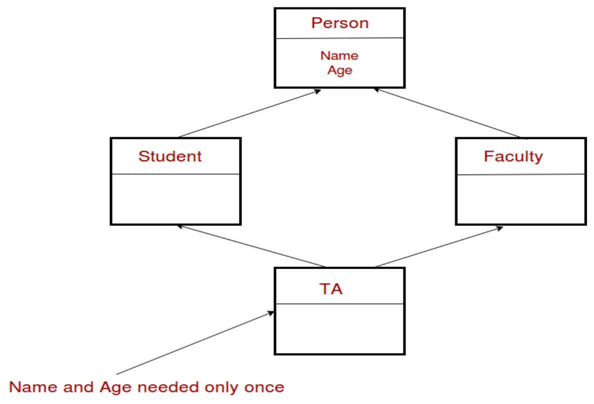


Eg-
```
#include<iostream>
using namespace std;
class A
{
public:
  A()  { cout << "A's constructor called" << endl; }
};
class B
{
public:
  B()  { cout << "B's constructor called" << endl; }
};
class C: public B, public A  // Note the order
{
public:
  C()  { cout << "C's constructor called" << endl; }
};
int main()
{
    C c;
    return 0;
}
```
Output:
```
B's constructor called
A's constructor called
C's constructor called
```

The constructors of inherited classes are called in the same order in which they are inherited.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



```
#include<iostream>
using namespace std;
class Person {
// Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : public Person {
// data members of Faculty
public:
    Faculty(int x):Person(x) {
    cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : public Person {
// data members of Student
public:
    Student(int x):Person(x) {
            cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
            cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

Output:

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities. *The solution to this problem is ‘virtual’ keyword*.

We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class. For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x) { cout << "Person::Person(int ) called" << endl; }
   Person()     { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x) {
   cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
         cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student {
public:
   TA(int x):Student(x), Faculty(x) {
         cout<<"TA::TA(int ) called"<< endl;
   }
};

int main() {
   TA ta1(30);
}
```

Output:

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In the above program, constructor of ‘Person’ is called once. One important thing to note in the above output is, *the default constructor of ‘Person’ is called*. When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class? The constructor has to be called in 'TA' class. For example, see the following program.

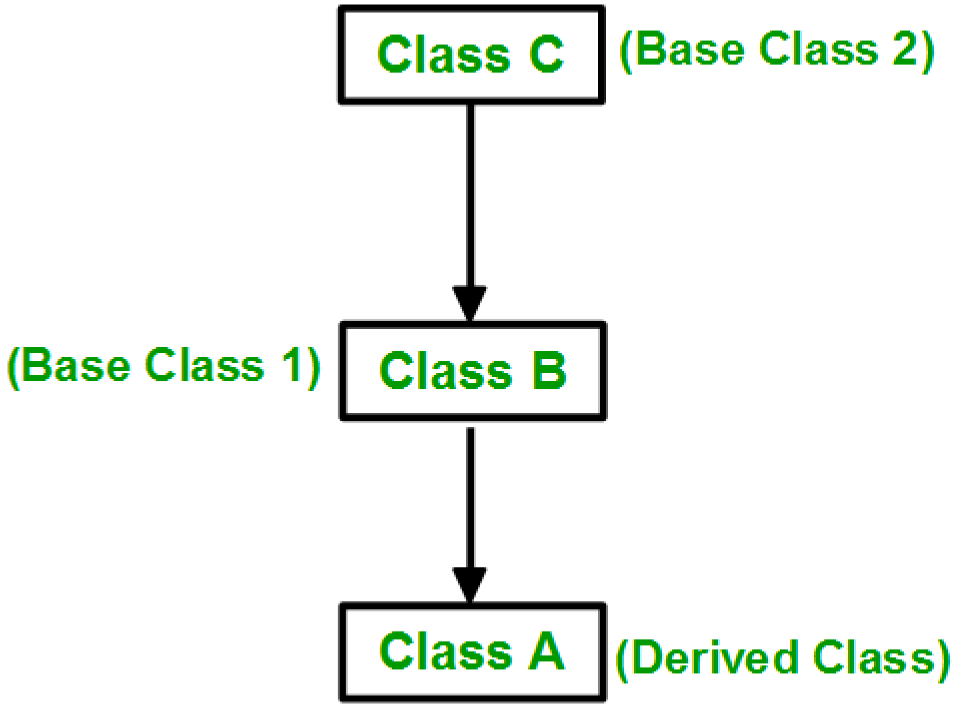
```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
   Person()     { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)   {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x), Person(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

Output:

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

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

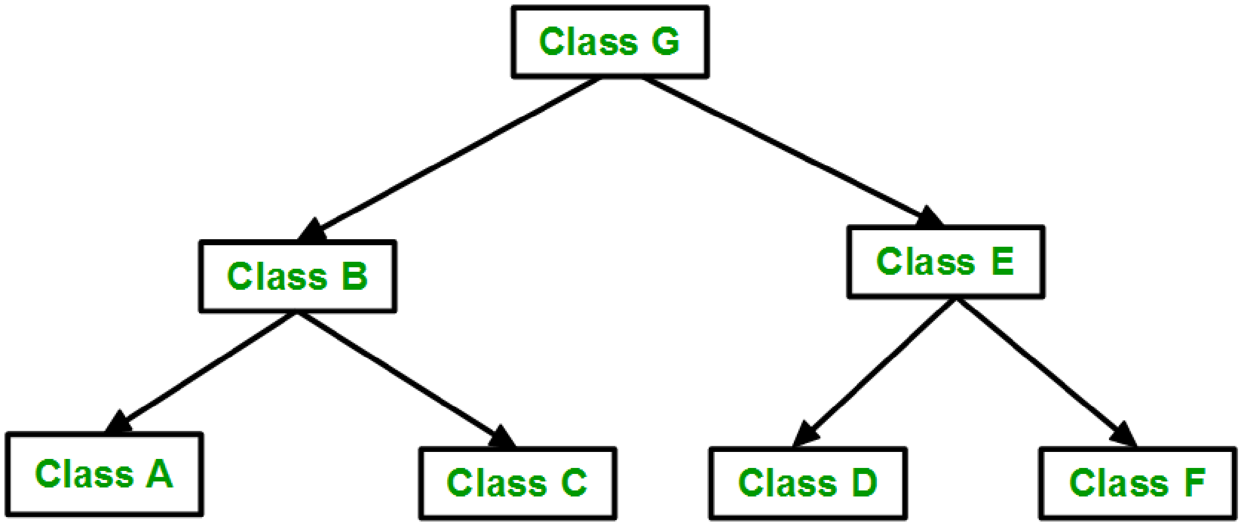


```
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from the derived base class fourWheeler
class Car: public fourWheeler{
   public:
     Car()
     {
       cout<<"Car has 4 Wheels"<<endl;
    }
};

// main function
int main()
{
   //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return 0;
}
```

Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

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

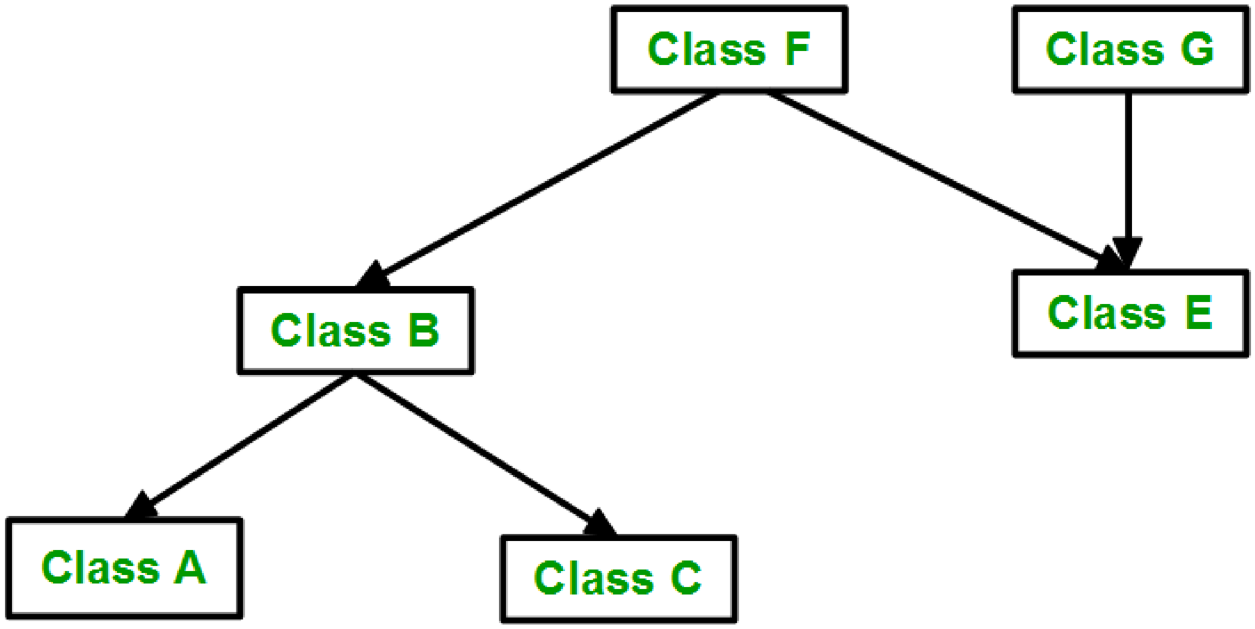


```
#include <iostream>
using namespace std;
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class Car: public Vehicle
{

};
class Bus: public Vehicle
{
};
int main()
{
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance.



```
#include <iostream>
using namespace std;
class Vehicle
{
  public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};
class Fare
{
   public:
   Fare()
   {
      cout<<"Fare of Vehicle\n";
```

```
    }
};
class Car: public Vehicle
{

};
class Bus: public Vehicle, public Fare
{

};
int main()
{
    Bus obj2;
    return 0;
}
```
Output:
```
This is a Vehicle
Fare of Vehicle
```

6. A special case of hybrid inheritance : Multipath inheritance(basically diamond problem again)

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arise in this type of inheritance.
Consider a program-

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <conio.h>
#include <iostream.h>
class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};
class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

void main()
{
    ClassD obj;

    obj.a = 10;                         //Statement 1, Error
    obj.a = 100;                        //Statement 2, Error
    //obj.ClassB::a = 10;        //Statement 3
    //obj.ClassC::a = 100;       //Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n A from ClassB : " << obj.ClassB::a;
    cout << "\n A from ClassC : " << obj.ClassC::a;

    cout << "\n B : " << obj.b;
    cout << "\n C : " << obj.c;
    cout << "\n D : " << obj.d;
}
```

Output: (if we comment out statement 1 and 2 and uncomment 3 and 4)

```
A from ClassB  : 10
A from ClassC  : 100
B : 20
C : 30
D : 40
```

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.
If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, because the compiler can’t differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:
Avoiding ambiguity using scope resolution operator:
Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example. Note : Still, there are two copies of ClassA in ClassD.

Avoiding ambiguity using virtual base class:

```
#include<iostream.h>
    #include<conio.h>

    class ClassA
    {
                    public:
                    int a;
    };

    class ClassB : virtual public ClassA
    {
```

```
                public:
                int b;
    };
    class ClassC : virtual public ClassA
    {
                public:
                int c;
    };

    class ClassD : public ClassB, public ClassC
    {
                public:
                int d;
    };

    void main()
    {

                ClassD obj;

                obj.a = 10;
                obj.a = 100;

                obj.b = 20;
                obj.c = 30;
                obj.d = 40;

                cout<< "\n A : "<< obj.a;
                cout<< "\n B : "<< obj.b;
                cout<< "\n C : "<< obj.c;
                cout<< "\n D : "<< obj.d;

    }
```

Output:

```
A : 100
B : 20
C : 30
D : 40
```

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

Questions on Inheritance:

Real life example of multiple inheritance-
A child inheriting features of its mother and father

Limitations of inheritance-

- Inherited functions work slower than normal functions as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.

Advantages of inheritance-

- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of the inherited class.
- Reusability enhances reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

Sealed modifier-
The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class. Sealed classes are used to restrict the inheritance feature of object oriented programming.

How to invoke base class function without creating an instance-
The method needs to be static, and then we can use the scope resolution operator(::) to call it

What is object slicing-
c++ - What is object slicing?

What is a local class-
A class declared inside a function becomes local to that function and is called local class.
In a local class all function definitions should be inline/inside the class.
It cannot contain static data members but it may contain static member functions tho.
The local class can only access static and enum variables of enclosing class.
Local classes can access global types, variables and functions. Can also access other local classes in the same function.

Nested class-
A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.
Refer Nested Classes in C++ for code example.

Simulating final class-
Simulating final class in C++

Encapsulation

Encapsulation is defined as binding together the data and the functions that manipulate them. We use access modifiers to implement encapsulation. Data members are kept private and member functions are made public.

```
#include<iostream>
using namespace std;
class Encapsulation
{
    private:
        int x;
    public:
        void set(int a)
        {
            x =a;
        }
        int get()
        {
            return x;
        }
};
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout<<obj.get();
    return 0;
}
```

X is made private and can only be accessed or changed through the member functions which are public. These member functions are accessible through objects of the class.
Encapsulation is used for hiding unnecessary data and implementation from the user.

Abstraction

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
Real life eg- A man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.
Abstraction using Classes: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in the math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Abstraction using access specifiers-

- Members declared as **public** in a class, can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

C++ Abstract Class

In C++ class is made abstract by declaring at least one of its functions as a pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.
A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class

```
#include <iostream>
using namespace std;
 class Shape
{
    public:
    virtual void draw()=0;
};
 class Rectangle : Shape
{
    public:
     void draw()
    {
        cout < <"drawing rectangle..." < <endl;
    }
};
class Circle : Shape
{
    public:
     void draw()
    {
        cout <<"drawing circle..." < <endl;
    }
};
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
   return 0;
}
```

Output:
```
drawing rectangle...
drawing circle...
```

Note: We cannot create objects of abstract classes.

Pure virtual function with implementation (virtual function can also have an implementation)

Encapsulation vs abstraction real world example

Virtual Functions and Runtime Polymorphism in C++ | Set 1 (Introduction)
Virtual Function in C++

Important Keywords

Static keyword in C++

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.
Static variables are allocated memory in data segment, not stack segment. Static variables (like global variables) are initialized as 0 if not initialized explicitly.

- **Static variables in a Function**: When a variable is declared as static, space for it gets allocated for the lifetime of the program. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.

eg-

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

void demo()
{
   // static variable
   static int count = 0;
   cout << count << " ";

   // value is updated and
   // will be carried to next
   // function calls
   count++;
}

int main()
{
   for (int i=0; i<5; i++)
      demo();
   return 0;
}
```

Output:

```
0 1 2 3 4
```

- **Static variables in a class**: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables in a class are shared by the objects. There can not be multiple copies of the same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

The following code gives an error:

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

class GfG
{
  public:
    static int i;

    GfG()
    {
      // Do nothing
    };
};

int main()
{
 GfG obj1;
 GfG obj2;
 obj1.i =2;
 obj2.i = 3;

 // prints value of i
 cout << obj1.i<<" "<<obj2.i;
}
```

This will work instead-

```
#include<iostream>
using namespace std;
class GfG
{
public:
   static int i;
```

```
    GfG()
    {
            // Do nothing
    };
};
int GfG::i = 1;
int main()
{
    GfG obj1, obj2;
    // prints value of i
    cout << obj1.i<<endl;
           cout<< obj2.i;
}
```

Output-
```
1
1
```

- **Class objects as static**: Just like variables, objects also when declared as static have a scope till the lifetime of the program.

Eg-
Consider this code with non static class object:
```
#include<iostream>
using namespace std;
class GfG
{
    int i;
    public:
        GfG()
        {
            i = 0;
            cout << "Inside Constructor\n";
        }
        ~GfG()
        {
            cout << "Inside Destructor\n";
        }
};
int main()
{
    int x = 0;
    if (x==0)
    {
        GfG obj;
    }
    cout << "End of main\n";
}
```

Output:
```
Inside Constructor
Inside Destructor
End of main
```

Now consider a static object:

```
#include<iostream>
using namespace std;
class GfG
{
    int i = 0;

    public:
    GfG()
    {
        i = 0;
        cout << "Inside Constructor\n";
    }

    ~GfG()
    {
        cout << "Inside Destructor\n";
    }
};
int main()
{
    int x = 0;
    if (x==0)
    {
        static GfG obj;
    }
    cout << "End of main\n";
}
```

Output:

```
Inside Constructor
End of main
Inside Destructor
```

- **Static functions in a class**: Just like the static data members or static variables inside the class, static member functions also does not depend on object of class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator. Static member functions are allowed to access only the static data members or other static member functions, they can not access the non-static data members or member functions of the class.

```
#include<iostream>
using namespace std;
class GfG
{
   public:
    static void printMsg()
    {
        cout<<"Welcome to GfG!";
    }
};
int main()
{
    GfG::printMsg();
}
```

Output:
Welcome to GfG!

C++ this Pointer

In C++ programming, ‘this’ is a keyword that refers to the current instance of the class. There can be 3 main uses of this keyword in C++.

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

```
#include <iostream>
using namespace std;
class Employee {
   public:
       int id; //data member (also instance variable)
       string name; //data member(also instance variable)
       float salary;
       Employee(int id, string name, float salary)
        {
             this->id = id;
            this->name = name;
            this->salary = salary;
        }
       void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Explicit keyword

C++ Const Keyword

new vs operator new in C++