2023

OOPS

CYBER TRON

C++

(SECOND PART)

T.I.M. HAMIM

Index:

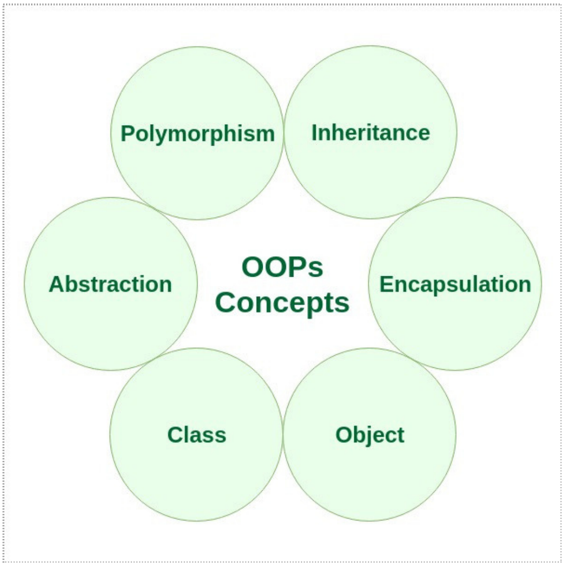
1. Introduction to OPP

OOP stands for Object-Oriented Programming.

C + OOP = C++

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. 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.

Characteristics of an Object Oriented Programming language

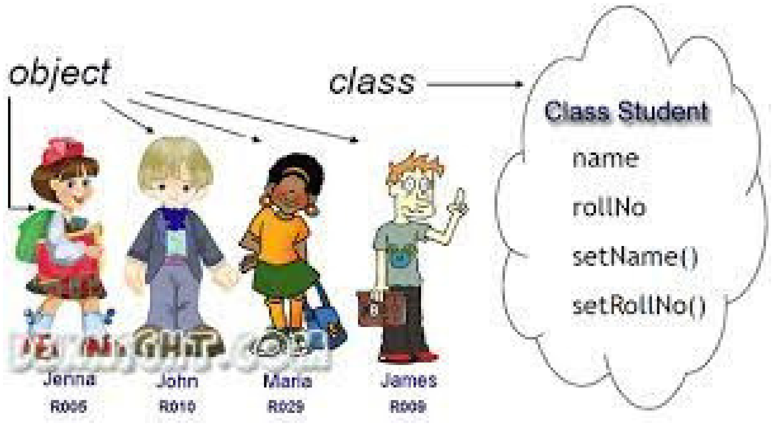


OOPS concepts are as follows:

1. Class
2. Object
3. Method and method passing
4. Pillars of OOPs
    - (a) Abstraction
    - (B) Encapsulation
    - (C) Inheritance
    - (D) Polymorphism
        - (i) Compile-time / static Polymorphism (Method Overloading, Constructor Overloading)
        - (ii) Runtime / dynamic Polymorphism (Method Overriding)

TABLE OF CONTENT:

class and object :



2. class

A class is a user-defined blueprint or prototype from which objects are created.

A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

We can say that a Class in C++ is a blue-print representing a group of objects which shares some common properties and behaviours.

Defining Class:

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

Syntax to declare a class:

```
class className
{
   Access Modifiers:    // can be private, public or
                                               protected
   Data members;
   Static Members
   Member Functions/Methods() { }
   Friend Functions();
   Constructors;
   Destructors
   Nested classes;
   Enums, Constants, and Typedefs;
};                // class name ends with a semicolon
```

A class in C++ can contains:

- **Data Members (Attributes):** These represent the class's state and hold information. Data members can be of various data types and can be private, protected, or public.

```
class MyClass {
private:
   int privateVar;
protected:
   double protectedVar;
public:
   std::string publicVar;
};
```

- **Member Functions (Methods):** Member functions define the behavior of the class. They can be used to manipulate the data members, perform actions, and provide the class's interface. Member functions can be private, protected, or public.

```
class MyClass {
private:
   void privateMethod() {
      // Implementation here
   }
protected:
   int protectedMethod() {
      // Implementation here
   }
public:
    void publicMethod() {
        // Implementation here
    }
};
```

- **Constructors:** Constructors are special member functions that are used to initialize objects of the class. They are called when an object is created. A class can have multiple constructors, including default and parameterized constructors.

  ```
  class MyClass {
  public:
      MyClass() {
          // Default constructor
      }
      MyClass(int value) {
          // Parameterized constructor
      }
  };
  ```

- **Destructors:** Destructors are used to clean up resources when an object is destroyed. They have the same name as the class with a tilde () prefix. There is only one destructor per class.

```
class MyClass {
public:
    ~MyClass() {
        // Destructor
    }
};
```

- **Static Members:** Static members (data members and member functions) belong to the class itself rather than an instance of the class. They are shared by all instances of the class.

```
class MyClass {
public:
    static int staticVar;
    static void staticMethod() {
        // Implementation here
    }
};
```

- **Friend Functions:** Friend functions are not part of the class but have access to its private and protected members. They are declared as friends using the friend keyword.

```
class MyClass {
private:
    int privateVar;
    friend void friendFunction(MyClass& obj);
};
```

```
void friendFunction(MyClass& obj) {
    obj.privateVar = 42; // Access to privateVar
}
```

- **Nested Classes:** A class can contain other classes as nested classes, which are classes defined within another class. These nested classes can have their own data members and member functions.

```
class OuterClass {
public:
    class NestedClass {
        // Implementation here
    };
};
```

- **Enums, Constants, and Typedefs:** Inside a class, you can define enumerations, constants, and typedefs.

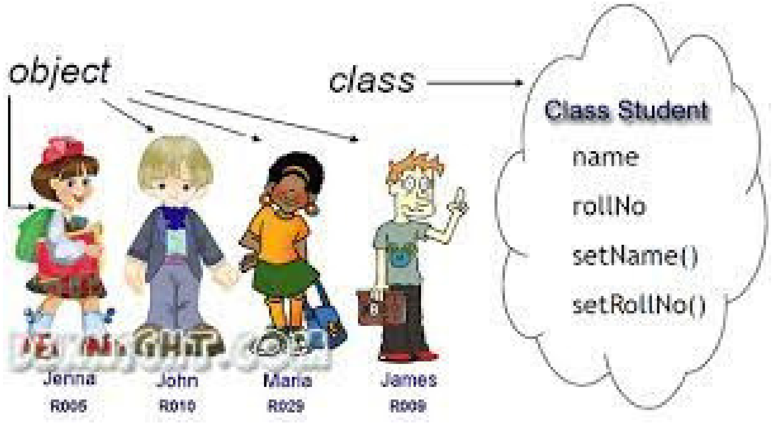
```
class MyClass {
public:
    enum Color { RED, GREEN, BLUE };
    static const int MAX_VALUE = 100;
    typedef int MyInt;
};
```

3. object

Any class type variable is called an object. An object is a basic unit of Object-Oriented Programming that represents real-life entities.

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

class and object :



class and object practical example:

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
 int id;
 double gpa;
};

int main()
{
 Student Hamim, Ava;

 Hamim.id = 5369;
 Hamim.gpa = 3.68;
 cout << Hamim.id << " " << Hamim.gpa << endl;

 Ava.id = 5368;
 Ava.gpa = 3.72;
 cout << Ava.id << " " << Ava.gpa << endl;
 return 0;
}
```

Output:

```
5369  3.68
5368  3.72
```

Here,
Hamim and Ava are two objects of Student class. Using these two objects we can access elements of the Student class.

adding function to the class:

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
  int id;
  double gpa;

  void display()
  {
  cout << id << "  " << gpa << endl;
  }
};

int main()
{
  Student Hamim, Ava;

  Hamim.id = 5369;
  Hamim.gpa = 3.68;
  Hamim.display();
  Ava.id = 5368;
  Ava.gpa = 3.72;
  Ava.display();
  return 0;
}
```

```
Output:
5369  3.68
5368  3.72
```

Here,
display() is a function of Student class.

adding parametrized function to the class:

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
  int id;
  double gpa;

  void display()
  {
  cout << id << "  " << gpa << endl;
  }
  void setValue(int x, double y)
  {
    id = x;
    gpa = y;
  }
};

int main()
{
  Student Hamim, Ava;

  Hamim.setValue(5369,3.68);
  Hamim.display();

  Ava.setValue(5368,3.75);
  Ava.display();

  return 0;
}
```

Output:

```
5369  3.68
5368  3.75
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
  int id;
  double gpa;

  void display()
  {
  cout << id << "  " << gpa << endl;
  }

  void setValue(int x, double y)
  {
    id = x;
    gpa = y;
  }
};

int main()
{
  Student Hamim, Ava;

  Hamim.setValue(5369,3.68);
  Hamim.display();

  Ava.setValue(5368,3.75);
  Ava.display();
  Hamim.display();

  return 0;
}
```

Output:

```
5369  3.68
5368  3.75
5369  3.68
```

Constructor:

Constructor is a special type of function that is used to initialize the object.

- Constructor is a special type of function.
- Constructor has the same name as that of the class it belongs.
- It has no return type not even void.
- It is called automatically.

There are two types of constructors:

1. Default constructor,
2. Parameterized constructor.

Parameterized constructor:

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
  int id;
  double gpa;

  void display()
  {
  cout << id << "  " << gpa << endl;
  }

  Student(int x, double y) // Constructor
  {
    id = x;
    gpa = y;
  }
};

int main()
{
  Student Hamim(5369,3.68);
  Hamim.display();

  Student Ava(5368, 3.78);
  Ava.display();
  return 0;
}
```

Output:
```
5369  3.68
5368  3.78
```

Here,
Student(int x, double y) is a constructor.

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

class Student
{
public:
  int id;
  double gpa;

  void display()
  {
  cout << id << "  " << gpa << endl;
  }

  Student(int x, double y) // Constructor
  {
    id = x;
    gpa = y;
  }
  void setValue(int x, double y)
  {
    id = x;
    gpa = y;
  }
};

int main()
{
  Student Hamim(12,45);
  Hamim.setValue(5369,3.68);
  Hamim.display();

  Student Ava(5368, 3.78);
  Ava.display();
  return 0;
}
```

Default constructor:

- **What is the output of the following C++ program fragment:**

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

class Student
{
public:
 int id;
 double gpa;

 void display()
 {
 cout << id << " " << gpa << endl;
 }
  // constructor overloading
 Student(int x, double y) // Parameterized
Constructor
 {
 id = x;
 gpa = y;
 }
 Student() // Default Constructor
 {
 cout<<"Default constructor"<<endl;
 }
};

int main()
{
 Student Hamim(5369,3.68);
 Hamim.display();

 Student Ava(5368, 3.78);
 Ava.display();

 Student DC;
 return 0;
}
```

Output:

```
5369  3.68
5368  3.78
Default constructor
```

creating separate files for class in CodeBlocks:

```
#include <iostream>
#include "myfirstclass.h"

using namespace std;

int main()
{
    MyFirstClass obj1;
    obj1.display();
    return 0;
}


#ifndef MYFIRSTCLASS_H
#define MYFIRSTCLASS_H

class MyFirstClass
{
    public:
        MyFirstClass();
        void display();
    protected:

    private:
};

#endif // MYFIRSTCLASS_H
```

```
#include "myfirstclass.h"
#include<iostream>
using namespace std;

MyFirstClass::MyFirstClass()
{
   cout<<"Inside the constructor"<<endl;
}

void MyFirstClass::display()
{
   cout<<"Inside the display method"<<endl;
}
```

Destructor :

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

- Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when object goes out of scope.
- Destructor release memory space occupied by the objects created by constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

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

Syntax:

Syntax for defining the destructor within the class

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

}
```

Syntax for defining the destructor outside the class

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

}
```

- **What is the output of the following C++ program fragment:**

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

class Test
{
public:
  Test()
 {
   cout<<"\n Constructor executed";
 }

 ~Test()
  {
   cout<<"\n Destructor executed";
  }
};
main()
{
 Test t;

 return 0;
}
```

Output:

```
 Constructor executed
 Destructor executed
```

- **What is the output of the following C++ program fragment:**

```
#include<iostream>
using namespace std;
class Test
{
    public:
        Test()
        {
            cout<<"\n Constructor executed";
        }

        ~Test()
        {
            cout<<"\n Destructor executed";
        }
};

main()
{
    Test t,t1,t2,t3;
    return 0;
}
```

Output:
```
 Constructor executed
 Constructor executed
 Constructor executed
 Constructor executed
 Destructor executed
 Destructor executed
 Destructor executed
 Destructor executed
```

- **What is the output of the following C++ program fragment:**

```
#include<iostream>
using namespace std;
int count=0;
class Test
{
    public:
        Test()
        {
            count++;
            cout<<"\n No. of Object created:\t"<<count;
        }

        ~Test()
        {
            cout<<"\n No. of Object destroyed:\t"<<count;
            --count;
        }
};
```

```
main()
{
    Test t,t1,t2,t3;
    return 0;
}
```

```
Output:
 No. of Object created: 1
 No. of Object created: 2
 No. of Object created: 3
 No. of Object created: 4
 No. of Object destroyed:      4
 No. of Object destroyed:      3
 No. of Object destroyed:      2
 No. of Object destroyed:      1
```

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:
(1) the function ends
(2) the program ends
(3) a block containing local variables ends
(4) a delete operator is called

Note: destructor can also be called explicitly for an object.
syntax:
object_name.~class_name()

How are destructors different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)
Destructors don't take any argument and don't return anything

- **What is the output of the following C++ program fragment:**

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

// Destructor never takes an argument nor does
it return any value
int count = 0;

class num   // Class
{
  public :
    num() // Constructor
    {
      count++;
      cout<<"This is the time when constractor is
called for object number"<<count<<endl;
    }

    ~num() // Destructor
    {
      cout<<"This is the time when my destructor is
called for object number"<<count<<endl;
      count--;
    }
};
```

```
 int main()
 {
   cout<<"We are inside our main function"<<endl;
   cout<<"Creating first object n1"<<endl;
   num n1;
   {
     cout<<"Entering this block"<<endl;
     cout<<"Creating two more objects"<<endl;
     num n2, n3;
     cout<<"Exiting this block"<<endl;
   }
   cout<<"Back to main"<<endl;
 }
```

Output:

```
We are inside our main function
Creating first object n1
This is the time when constractor is called for object
number1
Entering this block
Creating two more objects
This is the time when constractor is called for object
number2
This is the time when constractor is called for object
number3
Exiting this block
This is the time when my destructor is called for object
number3
This is the time when my destructor is called for object
number2
Back to main
This is the time when my destructor is called for object
number1
```

Selection Operator (->) :

Selection operator(->) is used to access pointer objects elements.

- **What is the output of the following C++ program fragment:**

```
#include<iostream>
using namespace std;
 class myclass
 {
  public:
    void display()
    {
      cout<<"I am Hameem"<<endl;
    }
 };

 int main()
 {
  myclass obj;
  myclass *p = &obj;
  p -> display();
  return 0;
 }
```

Output:

```
I am Hameem
```

- **What is the output of the following C++ program fragment:**

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

 int main()
 {
    const int x = 10; // constant variable
    cout<<x;
    return 0;
 }
```

Output:

```
10
```

Here,
x is a constant variable. If we try to change this variable value we will get error.

Constant Objects:

- **What is the output of the following C++ program fragment:**

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

class myclass
{
  public :
    void display1() const // constant function
    {
      cout<<"I am a constant function"<<endl;
    }
};

int main()
{
  const myclass object1; // constant object
  object1.display1();
  return 0;
}
```

Output:

```
I am a constant function
```

- **What is the output of the following C++ program fragment:**

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

class myclass
{
 public :
   void display1() const // constant function
   {
     cout<<"I am a constant function"<<endl;
   }

   void display2() // non-constant function
   {
     cout<<"I am a non-constant function"<<endl;
   }
};

int main()
{
 const myclass object1; // constant object
 object1.display1();
 myclass object2;
 object2.display2();
 return 0;
}
```

Output:

```
I am a constant function
I am a non-constant function
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
 public :
  const int admissionFee;
  Student(int x)
  : admissionFee(x) // constructor initializer
  {
   cout<<admissionFee<<endl;
  }
};

int main()
{
 Student s1(1500), s2(1200);
 return 0;
}
```

Output:

```
1500
1200
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
  public :
    const int admissionFee;
    const int examFee;
    Student(int x, int y)
    : admissionFee(x), examFee(y) // constructor
initializer
    {
      cout<<admissionFee<<endl;
      cout<<examFee<<endl;
    }
};

int main()
{
  Student s1(1500,500);
  return 0;
}
```

Output:

```
1500
500
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
  public :
    const int admissionFee;
    const int examFee;
    int id;
    Student(int x, int y, int z)
    : admissionFee(x), examFee(y) // constructor
initializer
    {
      cout<<admissionFee<<endl;
      cout<<examFee<<endl;
      id = z;
      cout<<"Id = "<<id<<endl;
    }
};

int main()
{
  Student s1(1500,500,5369);
  return 0;
}
```

Output:

```
1500
500
Id = 5369
```

4. Access specifiers/modifiers in C++

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as Data Hiding.

Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. public,
2. private,
3. protected.

Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be Private.

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

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
using namespace std;
// class definition
class Circle
{
 public:
  double radius;

  double compute_area()
  {
   return 3.14*radius*radius;
  }

};

// main function
int main()
{
 Circle obj;

 // accessing public datamember outside class
 obj.radius = 5.5;

 cout << "Radius is: " << obj.radius << "\n";
 cout << "Area is: " << obj.compute_area();
 return 0;
}
```

```
Output:
Radius is: 5.5
Area is: 94.985
```

Output:
In the above program, the data member radius is declared as public so it could be accessed outside the class and thus it was allowed access from inside main().

2. Private: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
 // private data member
}

 private:
  double radius;

 // public member function
 public:
  double compute_area()
  { // member function can access private
   // data member radius
   return 3.14*radius*radius;
  }

};

// main function
int main()
{
 // creating object of the class
 Circle obj;

 // trying to access private data member
 // directly outside the class
 obj.radius = 1.5;

 cout << "Area is:" << obj.compute_area();
 return 0;
}
```

Output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
        double radius;
              ^
31:9: error: within this context
    obj.radius = 1.5;
        ^
```

Here,
The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
 // private data member
 private:
  double radius;

 // public member function
 public:
  void compute_area(double r)
  { // member function can access private
   // data member radius
   radius = r;

   double area = 3.14*radius*radius;

   cout << "Radius is: " << radius << endl;
   cout << "Area is: " << area;
  }

};
```

```
// main function
int main()
{
 // creating object of the class
 Circle obj;

 // trying to access private data member
 // directly outside the class
 obj.compute_area(1.5);
 return 0;
}
```

Output:

```
Radius is: 1.5
Area is: 7.065
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
  private :
    string name;
  public :
    void setName(string x)
    {
      name = x;
    }
    string getName()
    {
      return name;
    }
};

int main()
{
  Student s1;
  s1.setName("Hamim");
  cout<<s1.getName();
}
```

Output:

```
Hamim
```

Friend class :

- **What is the output of the following C++ program fragment:**

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

class A
{
  private:
    int id = 101;
    string name = "Hameem";
  public :
    friend class B;
};

class B
{
  public:
    void display(A obj)
    {
      cout<<obj.id<<endl;
      cout<<obj.name<<endl;
    }
};
```

```
int main()
{
  A obj1;
  B obj2;
  obj2.display(obj1);
  return 0;
}
```

Output:

```
101
Hameem
```

3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
 // protected data members
 protected:
 int id_protected;

};
```

```
// sub class or derived class from public base class
class Child : public Parent
{
 public:
 void setId(int id)
 {

  // Child class is able to access the inherited
  // protected data members of base class

  id_protected = id;

 }

 void displayId()
 {
  cout << "id_protected is: " << id_protected << endl;
 }
};

// main function
int main() {

 Child obj1;

 // member function of the derived class can
 // access the protected data members of the base
class

 obj1.setId(81);
 obj1.displayId();
 return 0;
}
```

Output:

```
id_protected is: 81
```

5. Encapsulation

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

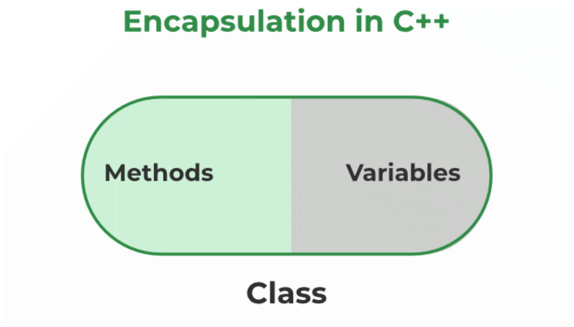
In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

This is what Encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Features of Encapsulation :

Below are the features of encapsulation:

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called encapsulation.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Increase in the security of data
5. It helps to control the modification of our data members.



Encapsulation also leads to data abstraction. Using encapsulation also hides the data, as in the above example, the data of the sections like sales, finance, or accounts are hidden from any other section.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate
// Encapsulation
#include <iostream>
using namespace std;

class Encapsulation {
private:
 // Data hidden from outside world
 int x;

public:
 // Function to set value of
 // variable x
 void set(int a)
  {
    x = a;
  }
 // Function to return value of
 // variable x
 int get()
  {
    return x;
  }
};
```

```
// Driver code
int main()
{
 Encapsulation obj;
 obj.set(5);
 cout << obj.get();
 return 0;
}
```

Output:

```
5
```

Here,
In the above program, the variable x is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

- **What is the output of the following C++ program fragment:**

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

// declaring class
class Circle {
   // access modifier
private:
   // Data Member
   float area;
   float radius;

public:
   void getRadius()
   {
      cout << "Enter radius\n";
      cin >> radius;
   }
   void findArea()
   {
      area = 3.14 * radius * radius;
      cout << "Area of circle=" << area;
   }
};
int main()
{
   // creating instance(object) of class
   Circle cir;
   cir.getRadius(); // calling function
   cir.findArea(); // calling function
}
```

Output:

```
Enter radius
5
Area of circle=78.5
```

- **What is the output of the following C++ program fragment:**

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

class Student
{
  private :
    string name;
  public :
    void setName(string x)
    {
      name = x;
    }
    string getName()
    {
      return name;
    }
};

int main()
{
  Student s1;
  s1.setName("Hamim");
  cout<<s1.getName();
}
```

Output;

```
Hamim
```

this keyword :

- **What is the output of the following C++ program fragment:**

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

class Student
{
  public :
    string name;
    Student (string name)
    {
      this -> name = name;
    }

    void display()
    {
      cout<<name<<endl;
    }
};

int main()
{
  Student s1("Hamim");
  s1.display();
}
```

Output:

```
Hamim
```

6. Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance.

Inheritance is one of the most important features of Object-Oriented Programming.

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

- Sub Class: The class that inherits properties from another class is called Subclass or Derived Class.
- Super Class: The class whose properties are inherited by a subclass is called Base Class or Superclass.

Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Derived Classes: A Derived class is defined as the class derived from the base class.

Syntax:

```
class  <derived_class_name> : <access-specifier>
<base_class_name>
{
       //body
}
```

Where
class   — keyword to create a new class
derived_class_name   — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

Note: A derived class doesn’t inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Example:

```
1. class ABC : private XYZ          //private derivation
        {            }
2. class ABC : public XYZ          //public derivation
        {           }
3. class ABC : protected XYZ    //protected derivation
        {          }
4. class ABC: XYZ          //private derivation by default
{         }
```

Note:

- When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.
- constructor can not be inherited.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate implementation
// of Inheritance
#include <bits/stdc++.h>
using namespace std;
// Base class
class Parent {
public:
 int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent {
public:
 int id_c;
};

// main function
int main()
{
 Child obj1;

 // An object of class child has all data members
 // and member functions of class parent
 obj1.id_c = 7;
 obj1.id_p = 91;
 cout << "Child id is: " << obj1.id_c << '\n';
 cout << "Parent id is: " << obj1.id_p << '\n';

 return 0;
}
```

```
Output:
Child id is: 7
Parent id is: 91
```

- **What is the output of the following C++ program fragment:**

```
#include<iostream>
using namespace std;
class Person
{
  public:
  string name;
  int age;
  void display_P()
  {
    cout<<"Name : "<<name<<endl;
    cout<<"Age : "<<age<<endl;
  }
};

class Student : public Person
{
  // name, age, display()
  public:
  int id;
  void display_S()
  {
    cout<<"ID : "<<id<<endl;
    cout<<"Name : "<<name<<endl;
    cout<<"Age : "<<age<<endl;
  }
};
```

```
int main()
{
  Student s1;
  s1.id = 5369;
  s1.name = "Hamim Talukder";
  s1.age = 22;
  s1.display_S();
}
```

or,

```
#include<iostream>
using namespace std;
class Person
{
  public:
  string name;
  int age;
  void display_P();
};

void Person :: display_P()
{
  cout<<"Name : "<<name<<endl;
  cout<<"Age : "<<age<<endl;
}

class Student : public Person
{
  // name, age, display()
  public:
```

```
  int id;
  void display_S()
  {
    cout<<"ID : "<<id<<endl;
    cout<<"Name : "<<name<<endl;
    cout<<"Age : "<<age<<endl;
  }
};

int main()
{
  Student s1;
  s1.id = 5369;
  s1.name = "Hamim Talukder";
  s1.age = 22;
  s1.display_S();
}
```

or,

```
#include<iostream>
using namespace std;
class Person
{
  public:
  string name;
  int age;
  void display_P();
};
```

```
void Person :: display_P()
{
  cout<<"Name : "<<name<<endl;
  cout<<"Age : "<<age<<endl;
}

class Student : public Person
{
  // name, age, display()
  public:
  int id;
  void display_S()
  {
    cout<<"ID : "<<id<<endl;
    display_P();
  }
};

int main()
{
  Student s1;
  s1.id = 5369;
  s1.name = "Hamim Talukder";
  s1.age = 22;
  s1.display_S();
}
```

Output:

```
ID : 5369
Name : Hamim Talukder
Age : 22
```

Modes of Inheritance: There are 3 modes of inheritance :

1. Public Mode: If we derive a subclass 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 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.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object.
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
};
```

- **What is the output of the following C++ program fragment:**

```
// Example: define member function without
argument within the class

#include<iostream>
using namespace std;

class Person
{
 int id;
 char name[100];

 public:
  void set_p()
  {
   cout<<"Enter the Id:";
   cin>>id;
   fflush(stdin);
   cout<<"Enter the Name:";
   cin.get(name,100);
  }

  void display_p()
  {
   cout<<endl<<id<<"\t"<<name<<"\t";
  }
};
```

```
class Student: public Person
{
 char course[50];
 int fee;

 public:
 void set_s()
  {
   set_p();
   cout<<"Enter the Course Name:";
   fflush(stdin);
   cin.getline(course,50);
   cout<<"Enter the Course Fee:";
   cin>>fee;
  }

  void display_s()
  {
   display_p();
   cout<<course<<"\t"<<fee<<endl;
  }
};

main()
{
 Student s;
  Person s1;
 s.set_s();
 s.display_s();
 return 0;
}
```

```
Output:
Enter the Id:5369
Enter the Name:Hamim Talukder
Enter the Course Name:CSE 225
Enter the Course Fee:5000

5369    Hamim Talukder  CSE 225 5000
```

- **What is the output of the following C++ program fragment:**

```
// Example: define member function without
argument within the class

#include<iostream>
using namespace std;

class Person
{
 int id;
 char name[100];

 public:
  void set_p()
  {
   cout<<"Enter the Id:";
   cin>>id;
   fflush(stdin);
   cout<<"Enter the Name:";
   cin.get(name,100);
  }
  void display_p()
  {
   cout<<endl<<id<<"\t"<<name<<"\t";
  }
};

class Student: private Person
{
 char course[50];
 int fee;

 public:
 void set_s()
  {
   set_p();
   cout<<"Enter the Course Name:";
   fflush(stdin);
   cin.getline(course,50);
   cout<<"Enter the Course Fee:";
   cin>>fee;
  }

  void display_s()
  {
   display_p();
   cout<<course<<"\t"<<fee<<endl;
  }
};
```

```
main()
{
 Student s;
 s.set_s();
 s.display_s();
 return 0;
}
```

```
Output:
Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: GCS
Enter the Course Fee:70000

101      Dev     GCS    70000
```

- **What is the output of the following C++ program fragment:**

```
// Example: define member function without
argument outside the class

#include<iostream>
using namespace std;

class Person
{
 int id;
 char name[100];

 public:
  void set_p();
  void display_p();
};

void Person::set_p()
{
 cout<<"Enter the Id:";
 cin>>id;
 fflush(stdin);
 cout<<"Enter the Name:";
 cin.get(name,100);
}

void Person::display_p()
{
 cout<<endl<<id<<"\t"<<name;
}
```

```
class Student: private Person
{
 char course[50];
 int fee;

 public:
  void set_s();
  void display_s();
};

void Student::set_s()
{
 set_p();
 cout<<"Enter the Course Name:";
 fflush(stdin);
 cin.getline(course,50);
 cout<<"Enter the Course Fee:";
 cin>>fee;
}

void Student::display_s()
{
 display_p();
 cout<<"\t"<<course<<"\t"<<fee;
}

main()
{
 Student s;
 s.set_s();
 s.display_s();
 return 0;
}
```

```
Output:
Enter the Id:5369
Enter the Name:Hamim talukder
Enter the Course Name:CSE 122
Enter the Course Fee:3000

5369    Hamim talukder  CSE 122 3000
```

- **What is the output of the following C++ program fragment:**

```
// Example: define member function with
argument outside the class

#include<iostream>
#include<string.h>
using namespace std;

class Person
{
 int id;
 char name[100];

 public:
  void set_p(int,char[]);
  void display_p();
};
```

```
void Person::set_p(int id,char n[])
{
 this->id=id;
 strcpy(this->name,n);
}

void Person::display_p()
{
 cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
 char course[50];
 int fee;
 public:
 void set_s(int,char[],char[],int);
 void display_s();
};

void Student::set_s(int id,char n[],char c[],int f)
{
 set_p(id,n);
 strcpy(course,c);
 fee=f;
}

void Student::display_s()
{
 display_p();
 cout<<"t"<<course<<"\t"<<fee;
}
```

```
main()
{
 Student s;
 s.set_s(1001,"Ram","B.Tech",2000);
 s.display_s();
 return 0;
}
```

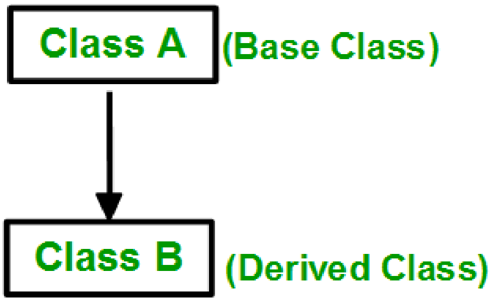
Output:

```
1001    RamtB.Tech      2000
```

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

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



Syntax:

```
class subclass_name : access_mode base_class
{
  // body of subclass
};
```

OR

```
class A
{
```

```
... .. ...
};

class B: public A
{
... .. ...
};
```

- **What is the output of the following C++ program fragment:**

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;

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

// sub class derived from a single base classes
class Car : public Vehicle {

};
```

```
// 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
```

- **What is the output of the following C++ program fragment:**

```
// Example:

#include<iostream>
using namespace std;

class A
{
 protected:
 int a;

 public:
  void set_A()
  {
   cout<<"Enter the Value of A=";
   cin>>a;

  }
  void disp_A()
  {
   cout<<endl<<"Value of A="<<a;
  }
};


class B: public A
{
 int b,p;

 public:
  void set_B()
  {
   set_A();
   cout<<"Enter the Value of B=";
   cin>>b;
  }

  void disp_B()
  {
   disp_A();
   cout<<endl<<"Value of B="<<b;
  }

  void cal_product()
  {
   p=a*b;
   cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
  }
};
```

```
main()
{

 B _b;
 _b.set_B();
 _b.cal_product();

 return 0;

}
```

```
Output:
Enter the Value of A=123
Enter the Value of B=34

Product of 123 * 34 = 4182
```

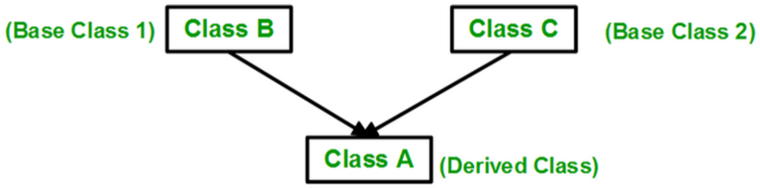
- **What is the output of the following C++ program fragment:**

```
// Example:

#include<iostream>
using namespace std;

class A
{
 protected:
 int a;
 public:
  void set_A(int x)
  {
   a=x;
  }

  void disp_A()
  {
   cout<<endl<<"Value of A="<<a;
  }
};

class B: public A
{
 int b,p;

 public:
  void set_B(int x,int y)
  {
   set_A(x);
   b=y;
  }

  void disp_B()
  {
   disp_A();
   cout<<endl<<"Value of B="<<b;
  }

  void cal_product()
  {
   p=a*b;
   cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
  }

};

main()
{
 B _b;
 _b.set_B(4,5);
 _b.cal_product();

 return 0;
}
```

Output:

```
Product of 4 * 5 = 20
```

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



Syntax:

```
class subclass_name : access_mode base_class1,
access_mode base_class2, ....
{
  // body of subclass
};


or,
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
{
... ... ...
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified.

- **What is the output of the following C++ program fragment:**

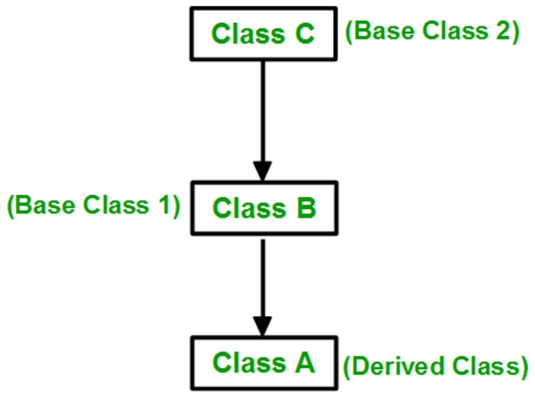
```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

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

// second base class
class FourWheeler {
public:
 FourWheeler()
 {
  cout << "This is a 4 wheeler Vehicle\n";
 }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};
```

```
// 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
This is a 4 wheeler Vehicle
```

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



Syntax:-

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

- **What is the output of the following C++ program fragment:**

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

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

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

```
// 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
```

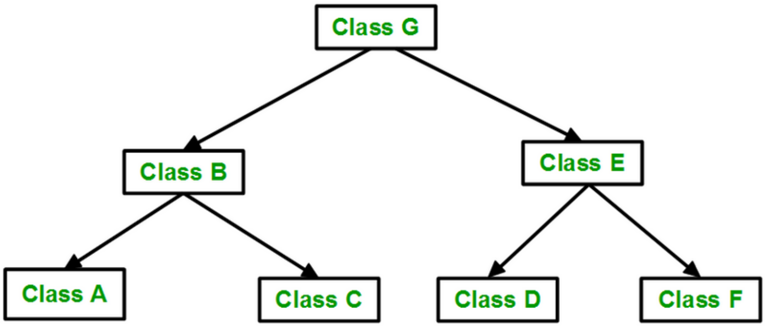
- **What is the output of the following C++ program fragment:**

```
// Example:

#include<iostream>
using namespace std;

class A
{
   protected:
   int a;

   public:
    void set_A()
    {
      cout<<"Enter the Value of A=";
      cin>>a;
    }

    void disp_A()
    {
      cout<<endl<<"Value of A="<<a;
    }
};

class B: public A
{
  protected:
    int b;

  public:
    void set_B()
    {
     cout<<"Enter the Value of B=";
     cin>>b;
    }


    void disp_B()
    {
     cout<<endl<<"Value of B="<<b;
    }
};

class C: public B
{
 int c,p;
 public:
  void set_C()
  {
    cout<<"Enter the Value of C=";
    cin>>c;
  }

  void disp_C()
  {
    cout<<endl<<"Value of C="<<c;
  }

   void cal_product()
   {
    p=a*b*c;
    cout<<endl<<"Product of "<<a<<" * "<<b<<" * "
<<c<<" = "<<p;
   }
};

main()
{
 C _c;
 _c.set_A();
 _c.set_B();
 _c.set_C();
 _c.disp_A();
 _c.disp_B();
 _c.disp_C();
 _c.cal_product();
 return 0;
}
```

Output:

```
Enter the Value of A=12
Enter the Value of B=34
Enter the Value of C=56

Value of A=12
Value of B=34
Value of C=56
Product of 12 * 34 * 56 = 22848
```

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



Syntax:-

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

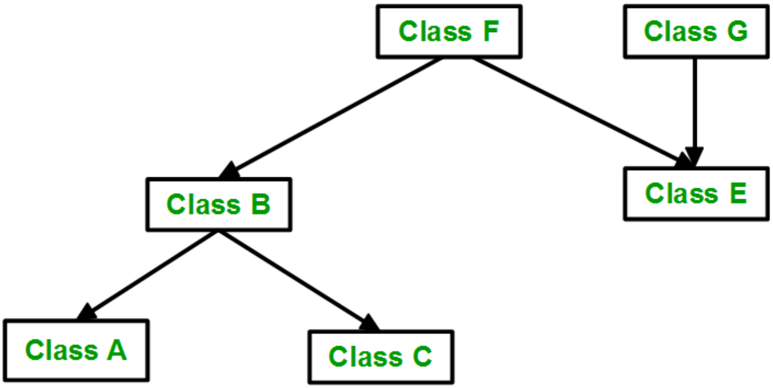
- **What is the output of the following C++ program fragment:**

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

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

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

// main function
int main()
{
 // Creating object of sub class will
 // invoke the constructor of base class.
 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 inheritances:



Syntax:-

```
class G
{
 // body of the class G.
}

class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
class E : public A, public G
{
 // body of class E.
}
```

- **What is the output of the following C++ program fragment:**

```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

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

// base class
class Fare {
public:
 Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle, public Fare {
};

// main function
int main()
{
 // Creating object of sub class will
 // invoke the constructor of base class.
 Bus obj2;
 return 0;
}
```

```
Output:
This is a Vehicle
Fare of Vehicle
```

- **What is the output of the following C++ program fragment:**

```
// Example:

#include <iostream>
using namespace std;

class A
{
 protected:
 int a;
 public:
 void get_a()
 {
 cout << "Enter the value of 'a' : ";
 cin>>a;
 }
};
```

```
class B : public A
{
 protected:
 int b;
 public:
 void get_b()
 {
 cout << "Enter the value of 'b' : ";
 cin>>b;
 }
};
class C
{
 protected:
 int c;
 public:
 void get_c()
 {
  cout << "Enter the value of c is : ";
  cin>>c;
 }
};

class D : public B, public C
{
 protected:
 int d;
 public:
 void mul()
 {
  get_a();
  get_b();
  get_c();
  cout << "Multiplication of a,b,c is : " <<a*b*c;
 }
};

int main()
{
 D d;
 d.mul();
 return 0;
}
```

```
Output:
Enter the value of 'a' : 14
Enter the value of 'b' : 56
Enter the value of c is : 5
Multiplication of a,b,c is : 3920
```

6. A special case of hybrid inheritance: Multipath inheritance:
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

- **What is the output of the following C++ program fragment:**

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

#include <iostream>
using namespace std;

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;
};

int 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 << " 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 << '\n';
}
```

Output:
```
 a from ClassB : 10
 a from ClassC : 100
 b : 20
 c : 30
 d : 40
```

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

There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

```
obj.ClassB::a = 10;  // Statement 3
obj.ClassC::a = 100; // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

```
#include<iostream>

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;
};
```

```
int main()
{
 ClassD obj;

 obj.a = 10;  // Statement 3
 obj.a = 100;  // Statement 4

 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 << '\n';
}
```

Output:
According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

Function Overriding (achieved at run time) :

It is the redefinition of base class function in its derived class with same signature i.e. return type and parameters.

- It can only be done in derived class.
- Example:

Syntax:

```
Class a
{
public:
      virtual void display()
      {
          cout << "hello";
      }
};

Class b : public a
{
public:
       void display()
       {
           cout << "bye";
       }
};
```

- **What is the output of the following C++ program fragment:**

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

class Person
{
  public:
    void display()
    {
      cout<<"I am a person"<<endl;
    }
};

class Student : public Person
{
  public:
    void display()
    {
      cout<<"I am a student"<<endl;
    }
};

class Teacher : public Person
{
  public:
    void display()
    {
      cout<<"I am a teacher"<<endl;
    }
};
```

```
int main()
{
  Person p;
 Student s;
 Teacher t;
 p.display();
 s.display();
 t.display();
}
```

Output:

```
I am a person
I am a student
I am a teacher
```

Function Overloading (achieved at compile time) :

Function Overloading provides multiple definitions of the function by changing signature i.e. changing number of parameters, change datatype of parameters, return type doesn’t play any role.

- It can be done in base as well as derived class

Syntax:

```
Class overload
{
public:
     void add(int a, int b)
     {
          cout <<  a + b;
     }

     void add(int a, int b, int c)
    {
        cout << a + b + c;
    }

    void add()
    {
       cout << "Nothing to add";
    }

};
```

- **What is the output of the following C++ program fragment:**

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

class overload
{
 public:
   void add(int a, int b)
   {
         cout <<  a + b<<endl;;
   }

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

   void add()
   {
      cout << "Nothing to add"<<endl;
   }

};

int main()
{
  overload h;
  h.add(1,2);
  h.add(3,4,5);
  h.add();
}
```

```
Output:
3
12
Nothing to add
```

- **What is the output of the following C++ program fragment:**

```
// CPP program to illustrate
// Function Overloading
#include <iostream>
using namespace std;

// overloaded functions
void test(int);
void test(float);
void test(int, float);

int main()
{
 int a = 5;
 float b = 5.5;

 // Overloaded functions
 // with different type and
 // number of parameters
 test(a);
 test(b);
 test(a, b);
 return 0;
}
```

```
// Method 1
void test(int var)
{
 cout << "Integer number: " << var << endl;
}

// Method 2
void test(float var)
{
 cout << "Float number: "<< var << endl;
}

// Method 3
void test(int var1, float var2)
{
 cout << "Integer number: " << var1;
 cout << " and float number:" << var2;
}
```

Output:

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number:5.5
```

Function Overloading	**Function Overriding**
Function Overloading provides multiple definitions of the function by changing signature.	Function Overriding is the redefinition of base class function in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don’t need inheritance.	In function overriding, we need an inheritance concept.

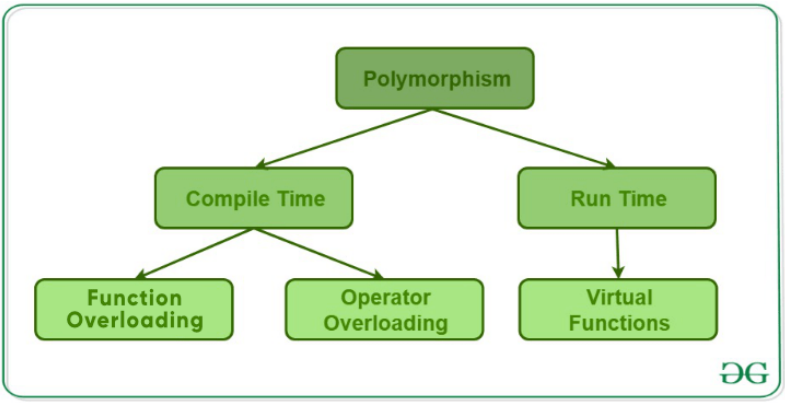
7. Polymorphism

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 is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- Compile-time Polymorphism.
- Runtime Polymorphism.



1. Compile-Time Polymorphism :

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

A. Function Overloading :

When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:

 // Function with 1 int parameter
 void func(int x)
 {
  cout << "value of x is " <<
    x << endl;
 }

 // Function with same name but
 // 1 double parameter
 void func(double x)
 {
  cout << "value of x is " <<
    x << endl;
 }

 // Function with same name and
 // 2 int parameters
 void func(int x, int y)
 {
  cout << "value of x and y is " <<
    x << ", " << y << endl;
 }
};

// Driver code
int main()
{
 Geeks obj1;
 obj1.func(7);

 // func() is called with double value
 obj1.func(9.132);
 // func() is called with 2 int values
 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
```

Output:
In the above example, a single function named function func() acts differently in three different situations, which is a property of polymorphism.

B. Operator Overloading :

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

- **What is the output of the following C++ program fragment:**

```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#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& obj)
 {
  Complex res;
  res.real = real + obj.real;
  res.imag = imag + obj.imag;
  return res;
 }
 void print()
 {
 cout << real << " + i" <<
   imag << endl;
 }
};

// Driver code
int main()
{
 Complex c1(10, 5), c2(2, 4);

 // An example call to "operator+"
 Complex c3 = c1 + c2;
 c3.print();
}
```

Output:

```
12 + i9
```

Here,
In the above example, the operator ‘+’ is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

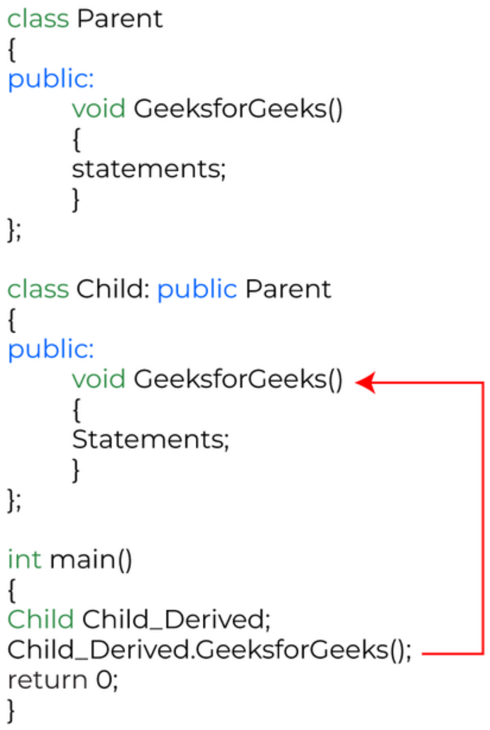
But I don't understand anything about the code.

2. Runtime Polymorphism :

This type of polymorphism is achieved by Function Overriding. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. 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.



```
class Parent
{
public:
        void GeeksforGeeks()
        {
        statements;
        }
};

class Child: public Parent
{
public:
        void GeeksforGeeks()
        {
        Statements;
        }
};

int main()
{
Child Child_Derived;
Child_Derived.GeeksforGeeks();
return 0;
}
```

- **What is the output of the following C++ program fragment:**

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

class Person
{
  public:
    void display()
    {
      cout<<"I am a person"<<endl;
    }
};

class Student : public Person
{
  public:
    void display()
    {
      cout<<"I am a student"<<endl;
    }
};

class Teacher : public Person
{
  public:
    void display()
    {
      cout<<"I am a teacher"<<endl;
    }
};
```

```
int main()
{
  Person p;
  Student s;
  Teacher t;
  p.display();
  s.display();
  t.display();
}
```

```
Output:
I am a person
I am a student
I am a teacher
```

Virtual Function :

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword “virtual” inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

- **What is the output of the following C++ program fragment:**

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

class Person
{
  public:
    virtual void display()
    {
      cout<<"I am a person"<<endl;
    }
};

class Student : public Person
{
  public:
    void display()
    {
      cout<<"I am a student"<<endl;
    }
};

class Teacher : public Person
{
  public:
    void display()
    {
      cout<<"I am a teacher"<<endl;
    }
};
```

```
int main()
{
  Person *p;
  Student s;
  Teacher t;
  p = &s;
  p -> display();
  p = &t;
  p->display();
}
```

Output:

```
I am a student
I am a teacher
```

- **What is the output of the following C++ program fragment:**

```
// C++ program for function overriding
#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:
 // print () is already virtual function in
 // derived class, we could also declared as
 // virtual void print () explicitly
 void print()
 {
  cout << "print derived class" <<
    endl;
 }
 void show()
 {
 cout << "show derived class" <<
   endl;
 }
};

// Driver code
int main()
{
 base* bptr;
 derived d;
 bptr = &d;

 // Virtual function, binded at
 // runtime (Runtime polymorphism)
 bptr->print();

 // Non-virtual function, binded
 // at compile time
 bptr->show();

 return 0;
}
```

Output:

```
print derived class
show base class
```

- **What is the output of the following C++ program fragment:**

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
 // virtual function
 virtual void display()
 {
  cout << "Called virtual Base Class function" <<
    "\n\n";
 }

 void print()
 {
 cout << "Called GFG_Base print function" <<
 "\n\n";
 }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {
public:
 void display()
 {
 cout << "Called GFG_Child Display Function" <<
 "\n\n";
 }
 void print()
 {
  cout << "Called GFG_Child print Function" <<
    "\n\n";
 }
};

// Driver code
int main()
{
 // Create a reference of class GFG_Base
 GFG_Base* base;

 GFG_Child child;
 base = &child;

 // This will call the virtual function
 base->GFG_Base::display();

 // this will call the non-virtual function
 base->print();
}
```

Output:

```
Called virtual Base Class function

Called GFG_Base print function
```

- **What is the output of the following C++ program fragment:**

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

class Shape
{
  public :
  double dim1, dim2;

  Shape(double dim1, double dim2)
  {
    this -> dim1 = dim1;
    this -> dim2 = dim2;
  }

  virtual double area()
  {
    return 0;
  }

};

class Triangle : public Shape
{
  // dim1, dim2, area()
  public:
    Triangle(double dim1, double dim2)
    : Shape(dim1, dim2)
    {

    }
    double area()
    {
      return 0.5 * dim1 * dim2;
    }
};

class Rectangle : public Shape
{
  // dim1, dim2, area()
  public:
    Rectangle(double dim1, double dim2)
    : Shape(dim1, dim2)
    {

    }

    double area()
    {
      return dim1 * dim2;
    }
};

int main()
{
  Shape *p;
  Triangle t(10,20);
  Rectangle r(10,20);

  p = &t;
  cout<<"Triangle Area = "<<p -> area()<<endl;
  p = &r;
  cout<<"Rectangle Area = "<<p -> area()<<endl;
}
```

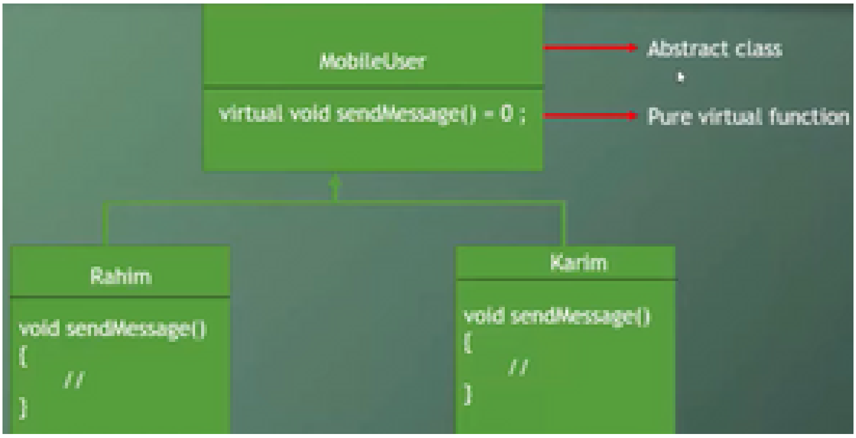
Output:

```
Triangle Area = 100
Rectangle Area = 200
```

8. Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. 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.
If we use pure virtual function in any class then it will be called an abstract class.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know 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 the accelerator, brakes, etc in the car. This is what abstraction is.


MobileUser
virtual void sendMessage() = 0 ;
Abstract class
Pure virtual function
Rahim
void sendMessage() { // }
Karim
void sendMessage() { // }

Properties of Abstract class :

- Object cant be created.
  Example: MovileUser m; (invalid)
- We can create pointer and reference of base abstract class points
  Example: MobileUser *m;
- It can have constructor.
- It can have non-virtual functions.

Note: If we use pure virtual function then we must make its body in another class .

- **What is the output of the following C++ program fragment:**

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

class MobileUser  // Abstract class
{
  public:
    void call()
    {
      cout<<"Hello !"<<endl;
    }

    virtual void sendMessage() = 0;  // pure virtual
function
};

class Rahim : public MobileUser
{
  public :
    void sendMessage()
    {
      cout<<"Hi, This is Rahim"<<endl;
    }
};

class Jim : public MobileUser
{
  public :
    void sendMessage()
    {
      cout<<"Hi, This is Jim"<<endl;
    }
};

int main()
{
  MobileUser *m;
  Rahim r;
  Jim j;
  m = &r;
  m->call();
  m->sendMessage();
  m = &j;
  m->call();
  m->sendMessage();
}
```

Output:

```
Hello !
Hi, This is Rahim
Hello !
Hi, This is Jim
```

9. Exception handling (try, catch, throw)

- **What is the output of the following C++ program fragment:**

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

int main()
{
  int num1, num2;
  cout<<"Enter 1st number : ";
  cin>>num1;
  cout<<"Enter 2nd number : ";
  cin>>num2;
  double result = (double) num1 /num2;
  cout<<"Result : "<<result<<endl;
  return 0;
}
```

```
Output:
Enter 1st number : 12
Enter 2nd number : 0
Result : inf
```

- **What is the output of the following C++ program fragment:**

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

int main()
{
 try{
 int num1, num2;
 cout<<"Enter 1st number : ";
 cin>>num1;
 cout<<"Enter 2nd number : ";
 cin>>num2;

 if(num2==0){
  throw -1;
 }
 double result = (double) num1 /num2;
 cout<<"Result : "<<result<<endl;
 }

 catch(int x)
 {
  cout<<"Divide by zero is not possible."<<endl;
  cout<<"Please try again."<<endl;
 }
 return 0;
}
```

or,

```
#include<iostream>
using namespace std;
int main()
{
  try{
  int num1, num2;
  cout<<"Enter 1st number : ";
  cin>>num1;
  cout<<"Enter 2nd number : ";
  cin>>num2;

  if(num2==0){
    throw -1;
  }
  double result = (double) num1 /num2;
  cout<<"Result : "<<result<<endl;
  }

  catch(...)
  {
    cout<<"Divide by zero is not possible."<<endl;
    cout<<"Please try again."<<endl;
  }
  return 0;
}
```

Here,
catch(...) will work for every type of data type.

or,

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

int main()
{
  try{
  int num1, num2;
  cout<<"Enter 1st number : ";
  cin>>num1;
  cout<<"Enter 2nd number : ";
  cin>>num2;

  if(num2==0){
    throw num2;
  }
  double result = (double) num1 /num2;
  cout<<"Result : "<<result<<endl;
  }

  catch(int x)
  {
    cout<<"Divide by "<< x << " is not possible."<<endl;
    cout<<"Please try again."<<endl;
  }
  return 0;
}
```

Output:

```
Enter 1st number : 10
Enter 2nd number : 0
Divide by zero is not possible.
Please try again.
```

Operator Precedence and Associativity:

- **Write a program that will show given bellow.**

Operator

Output:
Here,

→

Jim

Default constructor:

- **What is the output of the following C++ program fragment:**

C++ LANGUAGE
(THIRD PART)
T.I.M. HA-MEAM

ABC
PROKASHONI