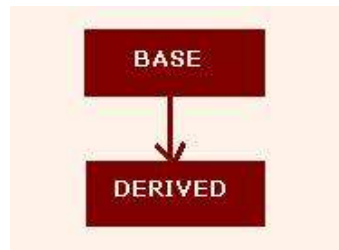


Chapter 05

Inheritance

5.1 Introduction

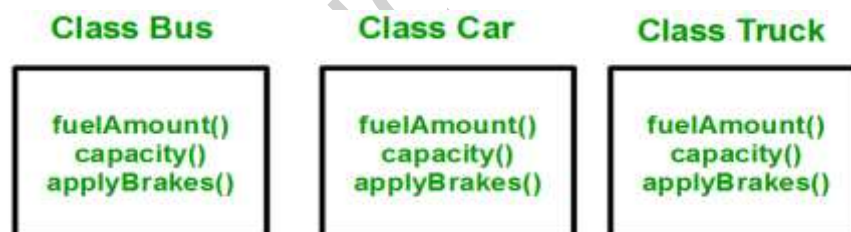
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object Oriented Programming. The process of deriving a new class from an already existing class is known as **inheritance**. The class that is derived is known as **derived class (sub-class)** and the class from which new class is derived is known as **base class (super-class)**.



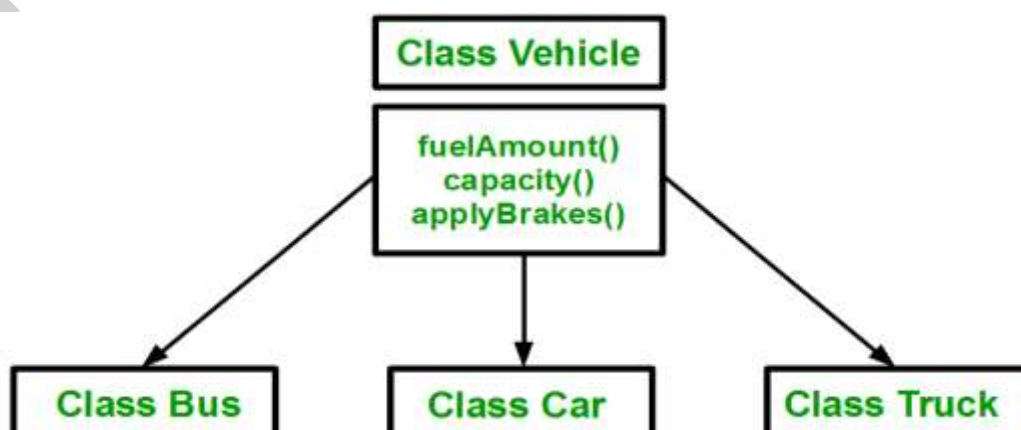
Sub Class: The class that inherits properties from another class is called Sub class.

Super Class: The class whose properties are inherited by sub class is called Super class.

Most important feature of the inheritance is **reusability**. It means if a class is designed, compiled and tested we can utilize that class in future. We can also add extra features as per our requirements.



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle 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.

Note: private member of the base class will never get inherited in the sub class.

```
#include <iostream>
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 << endl;
    cout << "Parent id is " << obj1.id_p << endl;

    return 0;
}
```

5.2 Derived Class:

The class that inherits properties or characteristics from another class is called as Derived class or Sub class.

How to Define Derive Class in Inheritance

First Define a base class as shown below:

```
class Base_Class_Name
{
    //some code snippets
}
```

Once the base class is defined you can define a derived class using following syntax:

```
class Derived_Class_Name: Visibility_Mode Base_Class_Name
{
    //some code snippets
}
```

Here, Derived_Class_Name is the name of the class that you want to derive. **Visibility_Mode** indicates the mode in which you want to derive a new class. It can be public, private or protected. **Base_Class_Name** is the name of the class from which you want to derive a new class. In the above syntax visibility mode determines how the data members of the base class are inherited in to derived class. Visibility mode can be public, private or protected. If you don't specify visibility mode then by default it is private.

5.3 Visibility Modes:

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. Private members of the base class will never get inherited in sub 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. Private members of the base class will never get inherited in sub 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. Private members of the base class will never get inherited in sub class.

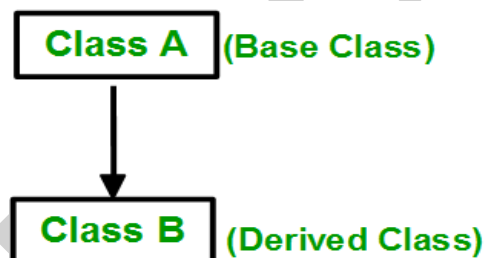
The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

	Derived Class	Derived Class	Derived Class
Base class	Public Mode	Private Mode	Protected Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

5.4 Types of Inheritance

1. Single Inheritance:

The process of deriving a new class from already existing class is known as single inheritance. Thus in single inheritance there is one base class and one derived class. 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.



Syntax:

```

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

Example:

```

#include <iostream>
using namespace std;

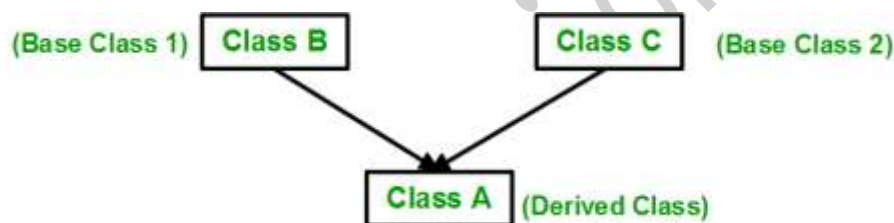
// base class
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
  
```

```
// sub class derived from base class
class Car: public Vehicle{

};

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

- 2. Multiple Inheritances:** The process of deriving a class from more than one base class is known as multiple inheritances. Thus in multiple inheritance there is **one derived class** but **more than one base classes**. 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**.



Syntax:

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

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

Example:

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

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

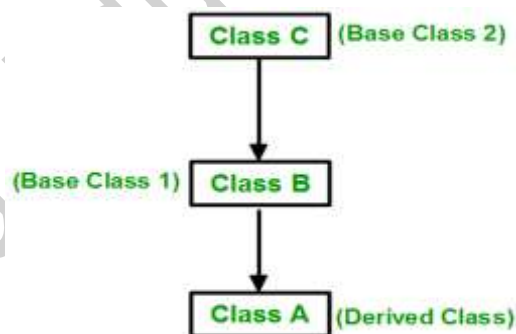
```
// second base class
class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// 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;
}
```

3. Multilevel Inheritance: The process of deriving a new class from an already existing class and then again derive a new class from previously derived class is known as multilevel inheritance. In this type of inheritance, a derived class is created from another derived class.



Example:

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

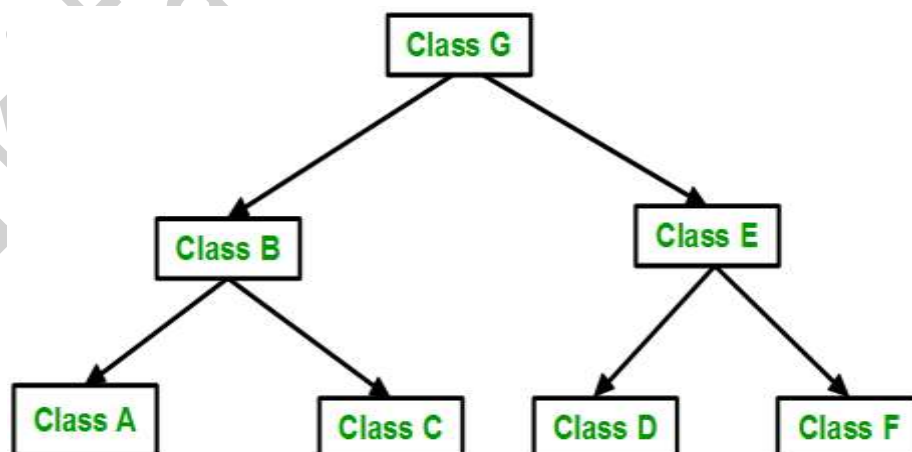
// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
```

```
class fourWheeler: public Vehicle
{
public:
    fourWheeler()
    {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};

// sub class derived from two base classes
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;
}
```

- 4. Hierarchical Inheritance:** The process of deriving more than one class from single base class is known as Hierarchical Inheritance. Thus in Hierarchical Inheritance there is **one base class** but **more than one derived classes**. 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.



Example:

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

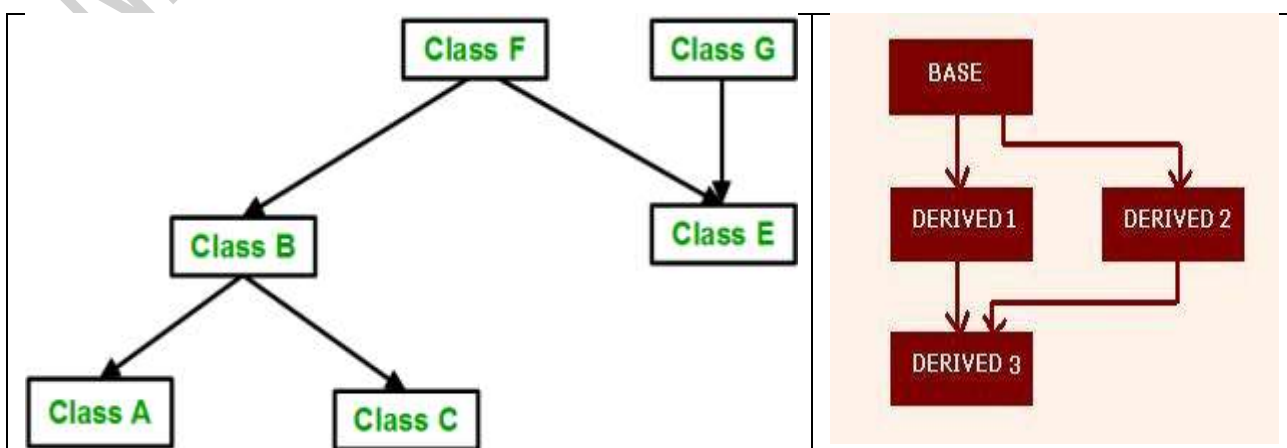
```
// base class
class Vehicle
{
    public:
        Vehicle()
        {
            cout << "This is a Vehicle" << endl;
        }
};

// 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;
}
```

5. Hybrid Inheritance: The combination of more than one inheritance is known as hybrid 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:



Example:

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

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

//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;
}
```

5.5 Programs on types of inheritance

1. Single Inheritance:

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

class student
{
    int rollno;
    char name[20];
public:
    void inputst()
    {
        cout<<"Enter Roll Number:";
        cin>>rollno;
        cout<<"Enter Name:";
        cin>>name;
    }
    void outputst()
    {
        cout<<"RollNumber:"<<rollno<<endl;
        cout<<"Name:"<<name<<endl;
    }
};

class result:public student
{
    int totalmark;
public:
    void inputmark()
    {
        cout<<"Enter marks:";
        cin>>totalmark;
    }
    void outputmark()
    {
        outputst();
        cout<<"Total Mark:"<<totalmark;
    }
};

int main()
{
    result r1;
    r1.inputst();
    r1.inputmark();
    r1.outputmark();
    return 0;
}
```

2. Multiple Inheritance:

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

class Area
{
public:
    float area_calc(float l,float b)
    {
        return l*b;
    }
};

class Perimeter
{
public:
    float peri_calc(float l,float b)
    {
        return 2*(l+b);
    }
};

/* Rectangle class is derived from classes Area and Perimeter. */
class Rectangle : private Area, private Perimeter
{
private:
    float length, breadth;
public:
    Rectangle() : length(0.0), breadth(0.0) { }
    void get_data( )
    {
        cout<<"Enter length: ";
        cin>>length;
        cout<<"Enter breadth: ";
        cin>>breadth;
    }

    float area_calc()
    {
        /* Calls area_calc() of class Area and returns it. */
        return Area::area_calc(length,breadth);
    }
}
```

```
float peri_calc()
{
    /* Calls peri_calc() function of class Perimeter and returns it. */
    return Perimeter::peri_calc(length,breadth);
}
};
void main()
{
    Rectangle r;
    r.get_data();
    cout<<"Area = "<<r.area_calc();
    cout<<"\nPerimeter = "<<r.peri_calc();
    getch();
}
```

3. Multilevel Inheritance

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

class college
{
    char name[20];
public:
    void inputcl()
    {
        cout<<"Enter College Name:";
        cin>>name;
    }
    void outputcl()
    {
        cout<<"Name:"<<name<<endl;
    }
};

class branch:public college
{
    char BranchName[20];
    int intake;
public:
    void inputbr()
    {
        cout<<"Enter Branch Name:";
        cin>>BranchName;
        cout<<"Enter Intake:";
        cin>>intake;
    }
}
```

```
void outputbr()
{
    cout<<"Branch Name:"<<BranchName<<endl;
    cout<<"Intake:"<<intake<<endl;
}

};

class student:public branch
{
    int rollno;
    char stname[20];
public:
    void inputst()
    {
        cout<<"Enter Roll Number:";
        cin>>rollno;
        cout<<"Enter Name:";
        cin>>stname;
    }
    void outputst()
    {
        outputcl();
        outputbr();
        cout<<"Roll Number:"<<rollno<<endl;
        cout<<"Name:"<<stname<<endl;
    }
};

int main()
{
    student S1;
    S1.inputcl();
    S1.inputbr();
    S1.inputst();
    S1.outputst();
    return 0;
}
```

4. Hierarchical Inheritance

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

class campus
{
    char trustname[20];
    char chairman[20];
```

```
public:
    void inputcm()
    {
        cout<<"Enter Trust Name:";
        cin>>trustname;
        cout<<"Enter Chairman Name:";
        cin>>chairman;
    }
    void outputcm()
    {
        cout<<"Trust Name:"<<trustname<<endl;
        cout<<"Chairman:"<<chairman<<endl;
    }
};

class diploma: public campus
{
    char principal[20];
public:
    void inputdip()
    {
        cout<<"Enter Principal Name:";
        cin>>principal;
    }
    void display()
    {
        outputcm();
        cout<<"Diploma Principal:"<<principal<<endl;
    }
};

class degree:public campus
{
    char principal[20];
public:
    void inputdeg()
    {
        cout<<"Enter Principal Name:";
        cin>>principal;
    }
    void display()
    {
        outputcm ();
        cout<<"Degree Principal:"<<principal<<endl;
    }
};
```

```
int main()
{
    Diploma D1;
    D1.inputcm();
    D1.inputdip();
    D1.display();
    Degree D2;
    D2.inputcm();
    D2.inputdeg();
    D2.Display();
    return 0;
}
```

5. Hybrid Inheritance

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

class arithmetic
{
protected:
    int num1, num2;
public:
    void getdata()
    {
        cout<<"For Addition:";
        cout<<"\nEnter the first number: ";
        cin>>num1;
        cout<<"\nEnter the second number: ";
        cin>>num2;
    }
};

class plus:public arithmetic
{
protected:
    int sum;
public:
    void add()
    {
        sum=num1+num2;
    }
};
```

```
class minus
{
protected:
    int n1,n2,diff;
public:
    void sub()
    {
        cout<<"\nFor Subtraction:";
        cout<<"\nEnter the first number: ";
        cin>>n1;
        cout<<"\nEnter the second number: ";
        cin>>n2;
        diff=n1-n2;
    }
};

class result:public plus, public minus
{
public:
    void display()
    {
        cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
        cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
    }
};

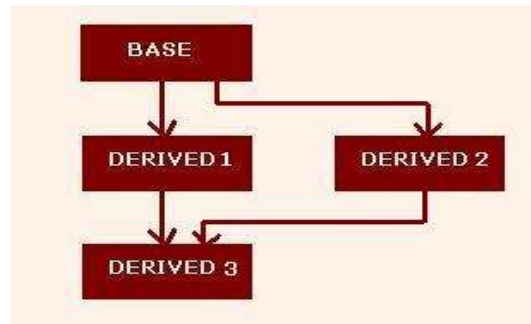
void main()
{
    clrscr();
    result z;
    z.getdata();
    z.add();
    z.sub();
    z.display();
    getch();
}
```

5.6 Virtual Base Classes

- An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.
- C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Consider Example of Hybrid Inheritance



Here, **BASE** is a base class from which two classes **DERIVED1** and **DERIVED2** are derived. Now class **DERIVED3** is derived from two classes **DERIVED1** and **DERIVED2**. In this case the public data member of class **BASE** is inherited in class **DERIVED1** and **DERIVED2**. Class **DERIVED3** is derived from two base classes **DERIVED1** and **DERIVED2** so the data member of class **BASE** is inherited into class **DERIVED3** twice, one through class **DERIVED1** and one through class **DERIVED2**.

Thus public and protected data member of class **BASE** inherited twice in class **DERIVED3**. To avoid duplication of data member of common base class we have to declare it as a virtual base class. The general syntax for declaring common base class as virtual is given below:

```

class A
{
    .....
    .....
};

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

class C : virtual public A
{
    .....
    .....
};

class D : public B, public C
{
    .....
    .....
};
  
```

Examples:

Example without using virtual base class

```
#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 occur
    //C:\Temp\demo.cpp|33|error: request for member 'a' is ambiguous|
    obj.a = 100;          //Statement 2, Error occur

    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;
    return 0;
}
```

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**.

Statement 1 and 2 in above example will generate error, bco'z compiler can't differentiate between two copies of **ClassA** in **ClassD**. To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB** and **ClassC** as **virtual** class.

Above Example using virtual base class

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

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;
    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;
return 0;
}
```

Output:

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

According to the above example, **ClassD** have only one copy of **ClassA** and statement 4 will overwrite the value of **a**, given in statement 3.

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

```
class student
{
    int r_no;
    char name[20];
public:
    void instudent()
    {
        cout<<"Enter Roll Number:";
        cin>>r_no;
        cout<<"Enter Name:";
        cin>>name;
    }

    void outstudent()
    {
        cout<<"Roll Number:"<<r_no<<endl;
        cout<<"Name:"<<name<<endl;
    }
};
```

```
class TW: virtual public Student
{
    int tmark;
public:
    void intmark()
    {
        cout<<"Enter Term work mark:";
        cin>>tmark;
    }
}
```

```
void outtmark()
{
    cout<<"Term Work Mark:"<<tmark;
}
};

class EXT: virtual public Student
{
    int emark;
public:
    void inemark()
    {
        cout<<"Enter External mark:";
        cin>>emark;
    }
    void outemark()
    {
        cout<<"External Mark:"<<emark;
    }
};

class Result: public TW, public EXT
{
    int total;
public:
    void displayResult ()
    {
        total = tmark + emark;
        outstudent ();
        outtmark();
        outemark ();
        cout<<"Total="<<total;
    }
};

int main ()
{
    Result R[60];
    int i;
    for (i=0;i<60;i++)
    {
        R[i].instudent ();
        R[i].intmark ();
        R[i].inemark();
    }
}
```

```
for (i=0;i<60;i++)  
{  
    R[i].display ();  
}  
return 0;  
}
```

5.7 Abstract Classes

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function; otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Up-casting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one **pure virtual function** or **abstract function** is called abstract class.

Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body
- We must implement all abstract functions in derived class.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

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

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

class BaseClass          //Abstract class
{
    public:
        virtual void Display1()=0;
        //Pure virtual function or abstract function
        virtual void Display2()=0;
        //Pure virtual function or abstract function

        void Display3()
        {
            cout<<"\n\tThis is Display3() method of Base Class";
        }
};

class DerivedClass : public BaseClass
{
    public:
        void Display1()
        {
            cout<<"\n\tThis is Display1() method of Derived Class";
        }

        void Display2()
        {
            cout<<"\n\tThis is Display2() method of Derived Class";
        }
};

void main()
{
    DerivedClass D;

    D.Display1();
    // This will invoke Display1() method of Derived Class
    D.Display2();
    // This will invoke Display2() method of Derived Class
    D.Display3();
    // This will invoke Display3() method of Base Class
}
```

Output :

This is Display1() method of Derived Class
This is Display2() method of Derived Class
This is Display3() method of Base Class

5.8 Constructors in derived Classes

Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember

1. Whether derived class's default constructor is called or parameterized is called, base class's default constructor is always called inside them.
2. To call base classes parameterized constructor inside derived class's parameterized constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Base class Default Constructor in Derived class Constructors

```
class Base
{ int x;
  public:
    Base() { cout << "Base default constructor"; }
};

class Derived : public Base
{ int y;
  public:
    Derived() { cout << "Derived default constructor"; }
    Derived(int i) { cout << "Derived parameterized constructor"; }
};

int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```

You will see in the above example that with both the object creation of the Derived class, Base class's default constructor is called.

Base class Parameterized Constructor in Derived class Constructor

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

```
class Base
{
    int x;
    public:
    Base(int i)
    {
        x = i;
        cout << "Base Parameterized Constructor\n";
    }
};

class Derived : public Base
{
    int y;
    public:
    Derived(int j) : Base(j)
    {
        y = j;
        cout << "Derived Parameterized Constructor\n";
    }
};

int main()
{
    Derived d(10) ;
    getch();
}
```

Output:

Base Parameterized Constructor
Derived Parameterized Constructor

Why is Base class Constructor called inside Derived class ?

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

Constructor call in Multiple Inheritance

Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

class A : public B, public C ;

In this case, first class B constructor will be executed, then class C constructor and then class A constructor.

Chapter 06

Polymorphism

6.1 Introduction

Polymorphism means ability to represent more than one form. Polymorphism is the technique of using same thing for different purpose.

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. Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So, same person posses have different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

Example:

```
class Base
{
    public:
        void show()
        {
            cout << "Base class";
        }
};

class Derived:public Base
{
    public:
        void show()
        {
            cout << "Derived Class";
        }
}

int main()
{
    Base b;           //Base class object
    Derived d;        //Derived class object
    b.show();
    d.show();
}
```

In above example, show() method is used in both super and sub class with different behaviors hence it is called as polymorphism.

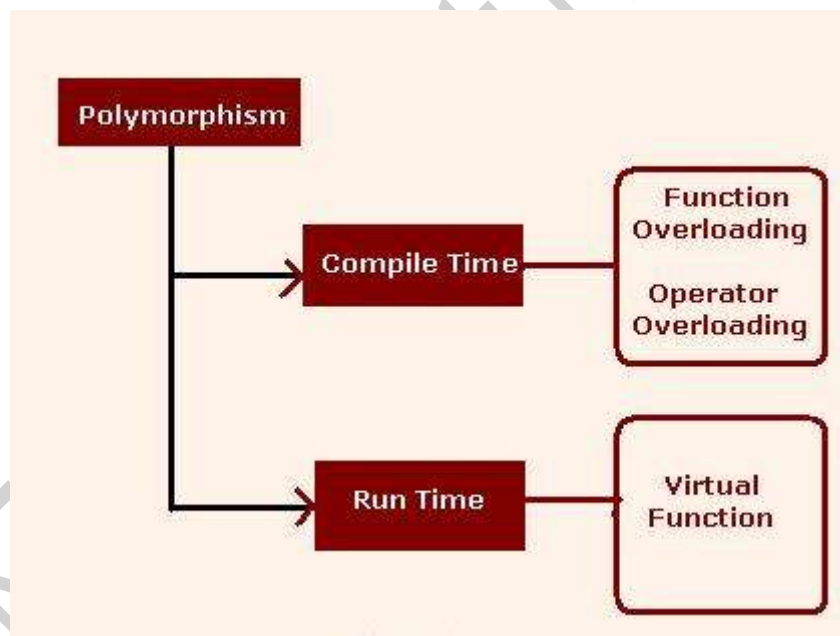
6.2 Types of polymorphism

(A) Compile time polymorphism

Compile time polymorphism is also known as static binding or early binding. **Function overloading** and **Operator overloading** are the example of compile time polymorphism. It is called compile time polymorphism because which version of function to invoke is determined by the compiler at compile time based on number and types of the argument. Thus in compile time polymorphism which function to invoke is determined at compile time so it is called **static or early binding**.

(B) Run time polymorphism

Run time polymorphism is also known as **dynamic binding or late binding**. **Virtual function** is the example of run time polymorphism. While inheriting derived class from base class if both classes contain same function then we have to declare that function as a virtual in the base class. In order to invoke function from the appropriate class you need to declare a pointer of base class and then invoke the function using that pointer. If pointer contains the address of the base class object then base class version is invoked and if pointer contains the address of derive class object then derived class version is invoked. Thus in run time polymorphism which function to invoke is determined at runtime so it is called **dynamic binding or late binding**.



6.3 Function Overloading

Whenever same method name is existing multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

- Overloading means using same thing for different purpose.
- The process of using same function for different purpose is known as Function Overloading.
- In Function Overloading you can create more than one function having same name but with
 - (1) Different Number of Arguments
 - (2) Different Data Types

- When you call the function, appropriate version of the function is invoked at compile time depending upon the number and type of arguments.
- Function overloading is the best example of **compile time polymorphism**
- **Note:** The scope of overloading is within the class only.
- You can have multiple definitions for the same function name in the same scope.
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
- You cannot overload function declarations that differ only by return type.

Why method Overloading?

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int, int) for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they cannot identify purpose of method. So we use method overloading to easily figure out the program. For example above two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.

Syntax:

```
class class_Name
{
    Returntype method()
    {
        .....
        .....
    }
    Returntype method(datatype1 variable1)
    {
        .....
        .....
    }
    Returntype method(datatype1 var1, datatype2 var2)
    {
        .....
        .....
    }
};
```

6.3.1 By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

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

```
class Addition
{
    public:
        void sum(int a, int b)
        {
            cout<<a+b;
        }
        void sum(int a, int b, int c)
        {
            cout<<a+b+c;
        }
};

void main()
{
    clrscr();
    Addition obj;
    obj.sum(10, 20);
    cout<<endl;
    obj.sum(10, 20, 30);
}
```

6.3.2 By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

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

using namespace std;

class Addition
{
    public:
        void sum(int a, int b)
        {
            cout<<a+b;
        }
        void sum(float a, float b)
        {
            cout<<a+b;
        }
};
```

```
void main()
{
    clrscr();
    Addition obj;
    obj.sum(10, 20);
    cout<<endl;
    obj.sum(10.5, 20.03);
}
```

Example:

```
#include <iostream>
using namespace std;
class FunctOver
{
public:
    void print(int i) {
        cout << " Here is int " << i << endl;
    }
    void print(double f) {
        cout << " Here is float " << f << endl;
    }
    void print(string c) {
        cout << " Here is string " << c << endl;
    }
    void print(int i,int j) {
        cout << " Here is i+j " << (i+j) << endl;
    }
};

int main() {
    FunctOver obj;
    obj.print(10);
    obj.print(10.10);
    obj.print("ten");
    obj.print(5,5);
    return 0;
}
```

6.4 Operator Overloading

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.

object of **ostream** class string

cout << "This is test string";

overloaded insertion operator

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- sizeof - sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

The process of giving special meaning to the existing c++ operator is known as "**Operator Overloading**". Using the concept of operator overloading we can use operators with object of the class. **For example:** We can use + to perform addition of two integer or floating point numbers. In the same way we can overload + operator so that we can perform addition of two objects of the class. When you overload the existing operator the basic syntax rules of the operator is not changed. It remains as it is. In order to overload particular operator you need to use special function known as **operator function**.

How to define Operator Function

The general syntax for defining operator overloading is given below:

Return-type operator op (Argument List)

```
{
    .....
}
```

Here,

Return-type indicates the type of the value that is return by the function.

Operator is the name of the operator function.

Op is the operator to which you want to overload

Argument List indicates number of arguments need to be passed.

The operator function can be either member function or friend function.

Keyword Operator to be overloaded

```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
    \\Function body
}
```

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function. Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Unary Operator Overloading:

Unary operator having one operand so generally we need to pass one argument to the operator function. If you define operator function as a member function then it will accept no argument. Because the object that is used to invoke the operator function is passed implicitly to the operator function.

If you define operator function as a friend function then it will accept one argument. Because friend functions is not a member function so it is not invoked using object of the class. Thus we need to pass object as an argument explicitly. Once the operator function is defined you can invoke operator function in following different ways:

```
op ObjectName;  
  
or  
  
// For Member Function  
  
ObjectName.operator ();  
  
// For Friend Function  
  
Operator op (ObjectName);
```


Binary Operator Overloading:

Binary operator having two operands so generally we need to pass two arguments to the operator function. If you define operator function as a member function then it will accept one argument. Because the object that is used to invoke the operator function is passed implicitly to the operator function and other object is passed explicitly to the function.

If you define operator function as a friend function then it will accept two arguments. Because friend functions is not a member function so it is not invoked using object of the class. Thus we need to pass two objects as an argument explicitly. Once the operator function is defined you can invoke operator function in following different ways:

```
ObjectName1 op ObjectName2;
```

Or

```
// For Member Function
```

```
ObjectName1.operator (ObjectName2);
```

```
// For Friend Function
```

```
Operator op (ObjectName1, ObjectName2);
```

Operator Overloading Rules

There are some rules or restrictions for overloading operators. These rules are as given below:

1. You can overload only existing operator. New operator cannot be created.
2. You cannot overload following operators:
sizeof operator
Conditional operator (? :)
Scope resolution operator (::)
Class member access operator (., .*)
3. By overloading operator you can not change the syntax rules of operator.
4. You cannot use friend function to overload following operators:
= Assignment operator.
() Function call operator.
{ } Subscripting operator.
-> Class member access operator.
5. When you overload unary operator using member function it will take no explicit argument. But if you overload unary operator using friend function then it will take one explicit argument.
6. When you overload binary operator using member function it will take one explicit argument. But if you overload binary operator using friend function then it will take two explicit arguments.
7. Overloaded operators cannot have default arguments.

6.5 Example on operator overloading

A simple and complete example

```
#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; }
};

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

What is the difference between operator functions and normal functions?

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

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

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

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

// The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};

Complex operator + (Complex const &c1, Complex const &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Overloading Arithmetic Operator

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the **+** operator, to add to Time(hh:mm:ss) objects.

Example: overloading '+' Operator to add two time object

```
#include< iostream.h>
#include< conio.h>
class time
{
    int h,m,s;
public:
    time()
    {
        h=0, m=0; s=0;
    }
    void getTime();
    void show()
    {
        cout<< h<< ":"<< m<< ":"<< s;
    }
    time operator+(time); //overloading '+' operator
};
```

```
time time::operator+(time t1)    //operator function
{
    time t;
    int a,b;
    a=s+t1.s;
    t.s=a%60;
    b=(a/60)+m+t1.m;
    t.m=b%60;
    t.h=(b/60)+h+t1.h;
    t.h=t.h%12;
    return t;
}

void time::getTime()
{
    cout<<"\n Enter the hour(0-11) ";
    cin>>h;
    cout<<"\n Enter the minute(0-59) ";
    cin>>m;
    cout<<"\n Enter the second(0-59) ";
    cin>>s;
}

void main()
{
    clrscr();
    time t1,t2,t3;
    cout<<"\n Enter the first time ";
    t1.getTime();
    cout<<"\n Enter the second time ";
    t2.getTime();
    t3=t1+t2; //adding of two time object using '+' operator
    cout<<"\n First time ";
    t1.show();
    cout<<"\n Second time ";
    t2.show();
    cout<<"\n Sum of times ";
    t3.show();
    getch();
}
```

Unary operators:

- Increment (++) Unary operator.
- Decrement (--) Unary operator.
- The minus (-) unary.
- The logical not (!) operator.

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

class complex {
    int a, b, c;
public:
    complex() {
    }

    void getvalue() {
        cout << "Enter the Two Numbers:";
        cin >> a>>b;
    }

    void operator++() {
        a = ++a;
        b = ++b;
    }

    void operator--() {
        a = --a;
        b = --b;
    }

    void display() {
        cout << a << "+\t" << b << "i" << endl;
    }
};

void main() {
    clrscr();
    complex obj;
    obj.getvalue();
    obj++;
    cout << "Increment Complex Number\n";
    obj.display();
    obj--;
    cout << "Decrement Complex Number\n";
    obj.display();
    getch();
}
```

Binary Operator Overloading Example Program

```
#include<iostream.h>

class complex {
    int a, b;
public:
    void getvalue() {
        cout << "Enter the value of Complex Numbers a,b:";
        cin >> a>>b;
    }
    complex operator+(complex ob) {
        complex t;
        t.a = a + ob.a;
        t.b = b + ob.b;
        return (t);
    }
    complex operator-(complex ob) {
        complex t;
        t.a = a - ob.a;
        t.b = b - ob.b;
        return (t);
    }
    void display() {
        cout << a << "+" << b << "i" << "\n";
    }
};

void main() {
    clrscr();
    complex obj1, obj2, result, result1;

    obj1.getvalue();
    obj2.getvalue();

    result = obj1 + obj2;
    result1 = obj1 - obj2;

    cout << "Input Values:\n";
    obj1.display();
    obj2.display();

    cout << "Result:";
    result.display();
    result1.display();
}
```

Chapter 07

Pointers & Virtual Functions

7.1 Introduction

Pointer:

Pointer is a special type of variable which stores address of another variable or array or structure or object of a class.

A **pointer** is known as derived data type. It is derived from built in data types. A pointer variable always contains the address of another variable in it. The address represents memory location of computer memory. A pointer allows us to access and manipulate memory addresses. Following symbols used in pointers:

Symbol	Name	Description
& (ampersand sign)	Address of operator	Give the address of a variable
* (asterisk sign)	Indirection operator	Gives the contents of an object pointed to by a pointer.

Syntax:

```
data_type *pointer_name;
```

Example:

```
int *ptr1;
char *ptr2;
```

- | | |
|-----------------------------|--|
| – int a1 = 10; | int *ptr = &a1; |
| – float farr[5]; | float *fptr = &farr[0]; |
| – struct student s1; | struct student *sptr = &s1; |

Pointer to Objects

Pointer to object means a special type of class variable can store the address of object declared using class type and asterisk (*) symbol. Pointer of class type holds address of object of a class. Then we can access public or protected members of class using pointer and pointer to member operator (→).

Syntax:

```
Class_type * pointer_name;
Pointer_name = &object_of_class;
```

Example:

```
MyClass obj;
MyClass *ptr_to_obj;
ptr_to_obj = &obj;
```

How to access class members using pointer:

```
ptr_to_obj → variable_name;
ptr_to_obj → function_name;
```

- In C++ you can declare a pointer that contains the address of the object of type class.
- Suppose we have created a class named base as shown below:

```
class Base
{
    public:
        int x;
        void display ()
        {
            cout<<"X="<<x<<endl;
        }
};
```

- Now you can declare a pointer that contains the address of the object of class base as shown below:

```
Base *ptr;
Base B1;
ptr = &B1;
```

- Using this pointer you can access the public member of the base class as shown below:

```
ptr->x = 10;
ptr->display ();
```

Example 'Pointer to Object':

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

class Base
{
    public:
        int x;
        void display ()
        {
            cout<<"X="<<x<<endl;
        }
};

int main ()
{
    Base B1;
    Base *ptr;
    ptr = &B1;
    ptr->x = 10;
    ptr->display();
    return 0;
}
```


7.2 this Pointer:

this is a special kind of pointer in C++. It contains an address of the object using which the function is invoked. **this** is default pointer available in class which contains address of current object on which function or constructor is invoked.

For example If you invoke Display () function of Base class using its object as shown below:

B1.Display ();

Then by default address of object B1 is contained in **this** pointer and passed implicitly to **display()** function.

Following are the uses of this pointer:

- (1) To access the private member directly in the member function.
- (2) In operator overloading when we overload unary operator using member function there is no need to pass any argument to the operator function because it is passes using this pointer.
- (3) It is also used to return the object using which the function is invoked.

Example:

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

class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
        cout<<"x="<<x<<endl;
        this->print();
    }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj,obj1,obj2;
    obj.setX(53);
    obj1.setX(10);
    obj2.setX(75);
    return 0;
}
```

Example: returning current object using **this** pointer

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0)
    {
        this->x = x;
        this->y = y;
    }

    Test & setX(int a)
    {
        x = a;
        return *this;
    }

    Test & setY(int b)
    {
        y = b;
        return *this;
    }

    void print()
    {
        cout << "x = " << x << " y = " << y << endl;
    }
};

int main()
{
    Test obj1(5, 5);
    obj1.print();

    Test obj2(50,60);
    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj2 = obj1.setX(10);
    obj2 = obj1.setY(20);

    obj2.print();
    return 0;
}
```

7.3 Pointer to Derived Class

It is possible to assign **address of object of** derived class to the **pointer** of base class type. This is called as pointer to derived class.

For example:

C++ allows you to assign the address of the derived class object to the base class pointer as shown below:

```
MyBase *ptr;  
MyDerived obj1;  
ptr = &obj1;           //pointer to derived class
```

Examples:

```
#include <iostream>  
using namespace std;  
  
class Base  
{  
    public:  
    int x;  
    void display ()  
    {  
        cout<<"X="<<x<<endl;  
    }  
};  
class Derive: public Base  
{  
    public:  
    int y;  
    void display ()  
    {  
        cout<<"X="<<x<<endl;  
        cout<<"Y="<<y<<endl;  
    }  
};  
int main ()  
{  
    Base *ptr1;  
    Derive D1;  
    ptr1 = &D1;  
    ptr1->x = 10;  
    ptr1->display ();  
    //this will call display() of base class  
}
```

Example:

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

class Base
{
    public:
    int x;
    void display ()
    {
        cout<<"X="<<x<<endl;
    }
};

class Derive: public Base
{
    public:
    int y;
    void display ()
    {
        cout<<"X="<<x<<endl;
        cout<<"Y="<<y<<endl;
    }
};

int main ()
{
    Base B1;
    Base *ptr;
    ptr = &B1;
    ptr->x = 5;
    ptr->display();    //display() of base class
    Derive D1;
    Derive *ptr1;
    ptr1 = &D1;
    ptr1->x = 10;
    ptr1->y = 20;
    ptr1->display (); //display() of derived class
}
```

7.4 Virtual Function

In pointer to derived class, the pointer of base class contains an address of object of derived class. But when we invoke members of a class using base pointer then it invoke members of base class even it holds address of derived class. If both classes i.e. base and derived class having same function names and prototype and if we wanted to call derived version using base pointer when it holds address of object of derived class but it is not possible in above case. Because base pointer invokes member of base class even it contains object of derived class. So, the solution to above problem is virtual function.

A **virtual function** is a member function of class that is declared within a base class and re-defined in derived class. Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function. Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

Virtual Keyword is used to make a member function of the base class Virtual.

When you want to use same function name in both the base and derived class, then the function in base class is declared as virtual by using the **virtual** keyword and again re-defined this function in derived class without using virtual keyword.

```
virtual return_type function_name()
{
    .....
    .....
}
```

Early Binding

In Early Binding function call is resolved at compile time. Hence, now compiler determines the type of object at compile time, and then binds the function call. Early Binding is also called as **Static Binding** or **Compile-time Binding**.

Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic Binding** or **Runtime Binding**.

```
class Base {
    public:
    void display () {
        cout <<"Base Class";
    }
};
class Derive: public Base {
    public:
    void display () {
        cout <<"Derived Class";
    }
};
```

- In above example if we create an object of derived class.
- And then if we stored address of object of derived class in pointer of base class type
- Then pointer will invoke the display function of base class rather than derived class
- Hence to avoid above problem of calling right function according to object of class
- We use concepts of virtual function
- Virtual function creates difference between base class version and derived class version of function

Rules for Defining Virtual Function

- If prototype of base class and derived class functions are identical then and only then you can declare a function as virtual in base class.
- Virtual function must be member function of a class.
- You can not declare constructor as a virtual in base class.
- Virtual function cannot be static.
- In order to access virtual function we must have to use pointer of base class.

Example:

Problem without Virtual Keyword:

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

class Base
{
    public:
    void display ()
    {
        cout <<"Base Class";
    }
};

class Derive: public Base
{
    public:
    void display ()
    {
        cout <<"Derived Class";
    }
};

int main()
{
    Derive d;           //derived class object
    Base *p;            //base class pointer
    p = &d;              //base pointer to derived class
    p->display();        //here early binding is done
    //hence it will call display() of base class
    //but if you want to call display() of derived class
    //then make display() function virtual in base.
    return 0;
}
```

Using Virtual Keyword:

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

class Base
{
    public:
    virtual void display ()
    {
        cout <<"Base Class";
    }
};

class Derive: public Base
{
    public:
    void display ()
    {
        cout <<"Derived Class";
    }
};

int main()
{
    Derive d;
    Base *p;
    p = &d;
    p->display();           //calls derived class function

    Base s;
    p = &s;
    p->display();           //calls base class function
    return 0;
}
```

7.5 Pure Virtual Function

Abstract Class

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. A pure virtual function (or abstract function) in C++ is a [virtual function](#) for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. Here is the syntax for a pure virtual function,

```
virtual return-type function-name() = 0;

// An abstract class
class Test
{
    public:
        // Pure Virtual Function
        virtual void show() = 0;
};
```

Example:

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

class Base
{
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    Base *p;
    p = &d;
    p -> fun();
    return 0;
}
```


7.6 Static and Dynamic Binding

Early Binding

In Early Binding function call is resolved at compile time. Hence, now compiler determines the type of object at compile time, and then binds the function call. Early Binding is also called as **Static** Binding or **Compile-time** Binding.

Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Example (Static Binding)

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

class Base
{
    public:
    void display ()
    {
        cout <<"Base Class";
    }
};

class Derive: public Base
{
    public:
    void display ()
    {
        cout <<"Derived Class";
    }
};

int main()
{
    Derive d;
    Base *p;
    p = &d;
    p->display();    //static binding
    return 0;
}
```

Example (Dynamic Binding)

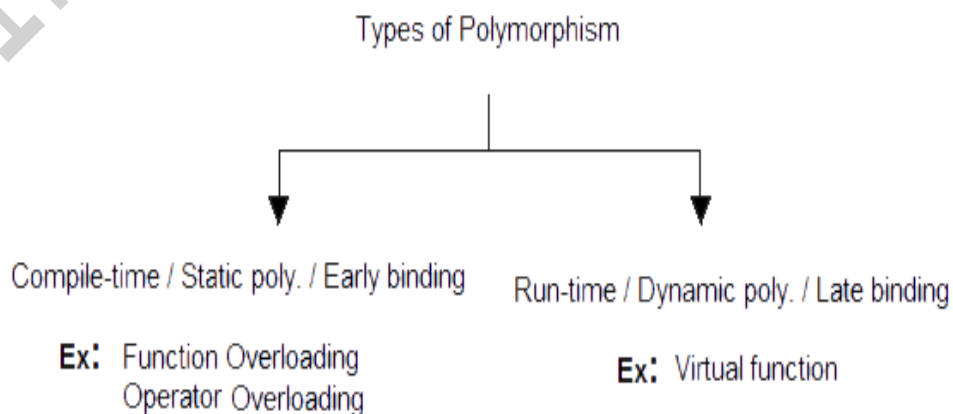
```
#include<iostream>
using namespace std;

class Base
{
    public:
    virtual void display ()
    {
        cout <<"Base Class";
    }
};

class Derive: public Base
{
    public:
    void display ()
    {
        cout <<"Derived Class";
    }
};

int main()
{
    Derive d;
    Base *p;
    p = &d;
    p->display();           //dynamic binding

    Base s;
    p = &s;
    p->display();           //dynamic binding
    return 0;
}
```



Difference between static and dynamic binding:

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time is "Static Binding".	Events occur at run time is "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Chapter 08

I/O & File Systems

8.1 Introduction

File:

- A file is an object on a computer
- File stores data, information, settings, or commands
- Files are stored permanently on hard disk
- It stores data in the form of text or binary
- It is a sequence of bytes

File Handling:

- **File handling** is a set of operations used in C++ programs to store program data permanently on computer's secondary memory.
- **File Operations:**
 - Creating & Naming a file
 - Opening a file
 - Reading data from file
 - Writing data into file
 - Closing a file

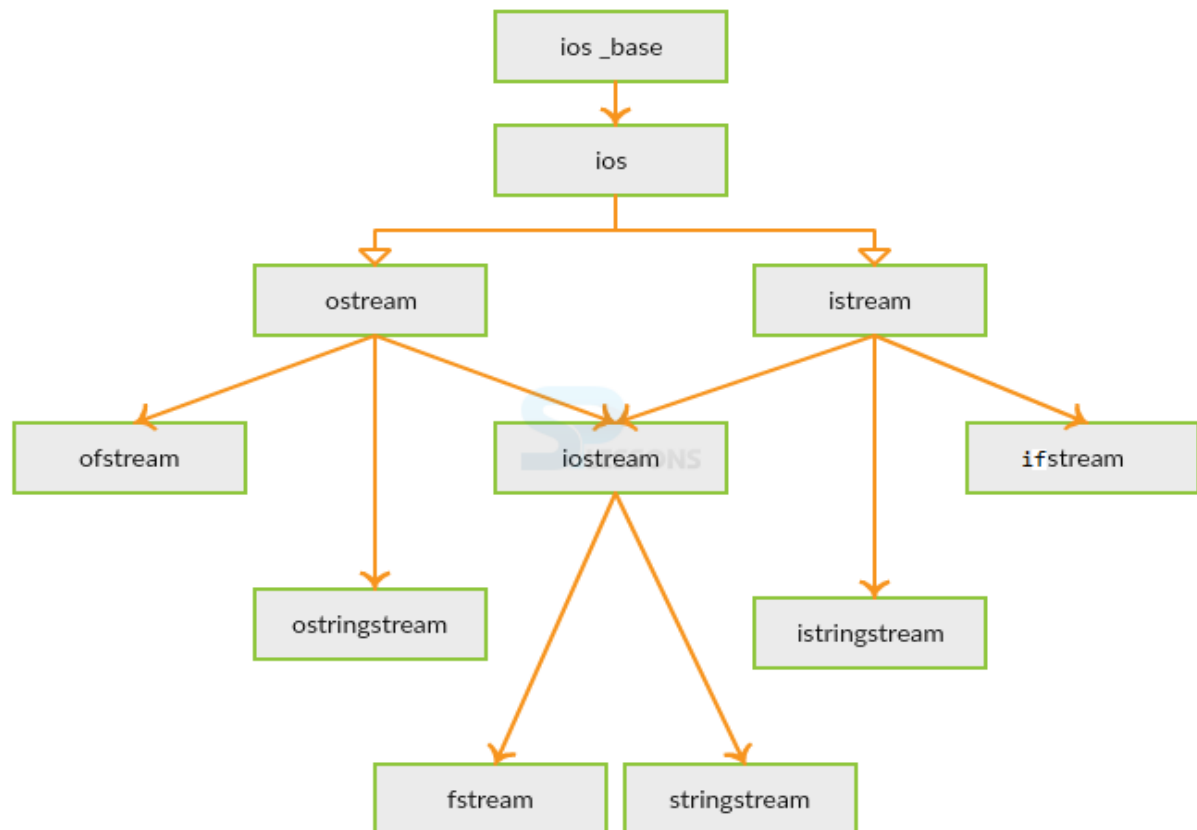
C++ Streams:

- Stream means sequence of bytes
- It represents continuous flow of bytes
- The term stream is an abstraction of a construct that allows you to send or receive an unknown number of bytes.
- Stream is linear queue that connects a file to the program and maintain the flow of data in both direction.
- Basically stream is of two type
 - Text Stream: sequence of characters
 - Binary stream: sequence of bits
- **iostream:**
 - This stream accepts input from keyboard or writes output on console.
 - **cin** and **cout** objects are used to accept input or write output
- **fstream:**
 - Reading and writing into file can be done using **fstream** standard library.
 - **fstream** must be used as header file while doing input and output operations to a file along with **iostream**.
- **fstream** provides following classes to perform various operations on file:

Class	Description
ofstream	This is used to create a file and write data on files

ifstream	This is used to read data from files
fstream	This is used to both read and write data from/to files

Hierarchy of Stream Classes



8.2 File Operations

- Creating a file: **open()**
- Reading data: **read()**
- Writing new data: **write()**
- Closing a file: **close()**

Open () Function:

Open () function is used to create, name, and open a file in computers primary memory for reading or writing. An open function check whether file is exists in secondary memory or not on given path. If file not exists then first it creates a file with given name and loads it in primary memory for reading or writing a data. If file is already exists in secondary memory then it only loads file in primary memory for reading or writing data.

Syntax: `void open(const char *filename, ios::openmode mode);`

Example:

```

ofstream outfile ;
outfile.open("file.txt", ios::out | ios::trunc );

fstream afile ;
afile.open("file.txt", ios::out | ios::in );
  
```

File Modes:

Modes	Description
in	Opens the file to read(default for ifstream)
out	Opens the file to write(default for ofstream)
binary	Opens the file in binary mode
app	Opens the file and appends all the outputs at the end
ate	Opens the file and moves the control to the end of the file
trunc	Removes the data in the existing file
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exists

Close () Function

This function is used to close file handler after finishing operations. This function saves changes in file, store file permanently on secondary memory and then frees the primary memory allocated for file.

Syntax: void close()

Example: afile.close();

Read () Function

The information can be read from a file using stream extraction operator >> or other reading functions provided by built-in classes.

Write () Function

The information can be written to a file by using stream insertion operator << or other writing functions provided by built-in classes.

Examples on File Operations**1. Writing into the file using ofstream:**

```
#include<iostream>
#include<fstream>

using namespace std;
```

```
int main()
{
    ofstream ofile;

    ofile.open("file.txt");

    ofile << "This is a line in a file" << endl;
    ofile << "This is another line" << endl;

    cout << "Data written to file" << endl;

    ofile.close();

    return 0;
}
```

2. Reading from file using ifstream:

```
#include<iostream>
#include<fstream>

using namespace std;

int main()
{
    char data[100];

    ifstream ifile;

    ifile.open("file.txt");

    cout << "Reading data from a file :-" << endl << endl;

    while (!ifile.eof())
    {
        ifile.getline(data, 100);
        cout << data << endl;
    }
    ifile.close();
    return 0;
}
```

3. Writing into the file using fstream with append mode.

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

int main()
{
    char line[100];
```

```
        fstream file;
        file.open("file.txt", ios :: out | ios :: app);
        if (file.fail())
        {
            cout << "Error Opening file ... " << endl;
        }
        else
        {
            cout << "Enter a line : ";
            cin.getline(line, 100);
            file << line << endl;
            cout << "Line written into the file" << endl;
        }
        return 0;
    }

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

int main()
{
    char data[100];
    fstream infile;
    infile.open("file.txt");

    cout << "Reading from the file" << endl;
    infile >> data;

    cout << data << endl;

    infile >> data;
    cout << data << endl;

    infile.close();

    return 0;
}
```

4. Writing object data into file:

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

class student {
    int roll;
    char name[30];
    float marks;
```



```
public:
    student() { }
    void getData();
    void displayData();
};

void student :: getData() {
    cout << "\nEnter Roll No. : ";
    cin >> roll;
    cin.ignore();
    cout << "Enter Name : ";
    cin.getline(name, 30);
    cout << "Enter Marks : ";
    cin >> marks;
}

void student :: displayData() {
    cout << "\nRoll No. : " << roll << endl;
    cout << "Name : " << name << endl;
    cout << "Marks : " << marks << endl;
}

int main() {
    student s[3];
    fstream file;
    int i;

    file.open("objects.txt", ios :: out | ios::app);
    cout << "\nWriting Student information:- " << endl;
    for (i = 0; i < 3; i++) {
        s[i].getData();
        file.write((char *)&s[i], sizeof(s[i]));
    }
    file.close();

    file.open("objects.txt", ios :: in);
    cout << "\nReading Student information:- " << endl;
    for (i = 0; i < 3; i++) {
        file.read((char *)&s[i], sizeof(s[i]));
        s[i].displayData();
    }
    file.close();

    return 0;
}
```

8.3 File Pointer & Its Manipulations

File Pointers:

- Each file has two associated pointers known as the file pointers.
- One of them is called the input pointer (or get pointer) and
- The other is called the output pointer (or put pointer).
- The input pointer is used for reading the contents of a given file location and
- The output pointer is used for writing to a given file location.

Functions for Manipulations of File Pointers:

When we want to move file pointer to desired position then use these function to manage the file pointers.

Seekg ()	:	moves get pointer (input) to a specified location
Seekp ()	:	moves put pointer (output) to a specified location
tellg ()	:	gives the current position of the get pointer
tellp ()	:	gives the current position of the put pointer

Examples:

fout . seekg(0, ios :: beg)	:	go to start
fout . seekg(0, ios :: cur)	:	stay at current position
fout . seekg(0, ios :: end)	:	go to the end of file
fout . seekg(m, ios :: beg)	:	move to m+1 byte in the file
fout . seekg(m, ios :: cur)	:	go forward by m bytes from the current position
fout . seekg(-m, ios :: cur)	:	go backward by m bytes from the current position
fout . seekg(-m, ios :: end)	:	go backward by m bytes from the end

put() and get() function

The function put() write a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

read() and write() function

```
file . read ((char *)&V , sizeof (V));  
file . Write ((char *)&V , sizeof (V));
```

- These function take two arguments.
- The first is the address of the variable V ,
- and the second is the length of that variable in bytes .
- The address of variable must be cast to type char * (i.e pointer to character type) .

Example:

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

```
int main() {
    fstream fp;
    char buf[100];
    int pos;

    fp.open("random.txt", ios :: out | ios :: ate | ios::app);
    cout << "\nWriting to a file ... " << endl;
    fp << "This is a line" << endl;
    fp << "This is a another line" << endl;
    pos = fp.tellp();
    cout << "Current position of put: " << pos << endl;
    fp.seekp(-10, ios :: cur);
    fp << endl << "Writing at a random location ";
    fp.seekp(7, ios :: beg);
    fp << " Hello World ";
    fp.close(); // file write complete
    cout << "Writing Complete ... " << endl;

    fp.open("random.txt", ios :: in | ios :: ate);
    cout << "\nReading from the file ... " << endl;
    fp.seekg(0);
    while (!fp.eof()) {
        fp.getline(buf, 100);
        cout << buf << endl;
    }
    pos = fp.tellg();
    cout << "\nCurrent Position of get: " << pos << endl;
    return 0;
}
```

8.4 Command Line Arguments

- If any input value is passed through command prompt at the time of running of program is known as **command line argument**.
- It is a concept of passing the arguments to the main() function by using command prompt.
- In command line arguments, main() function of program will takes two arguments that is;

-- argc

-- argv

- **argc:** argc is an integer type variable and it holds total number of arguments which is passed into main function. It take Number of arguments in the command line including program name.
- **argv[]:** argv[] is a char* type variable, which holds actual arguments which is passed to main function.

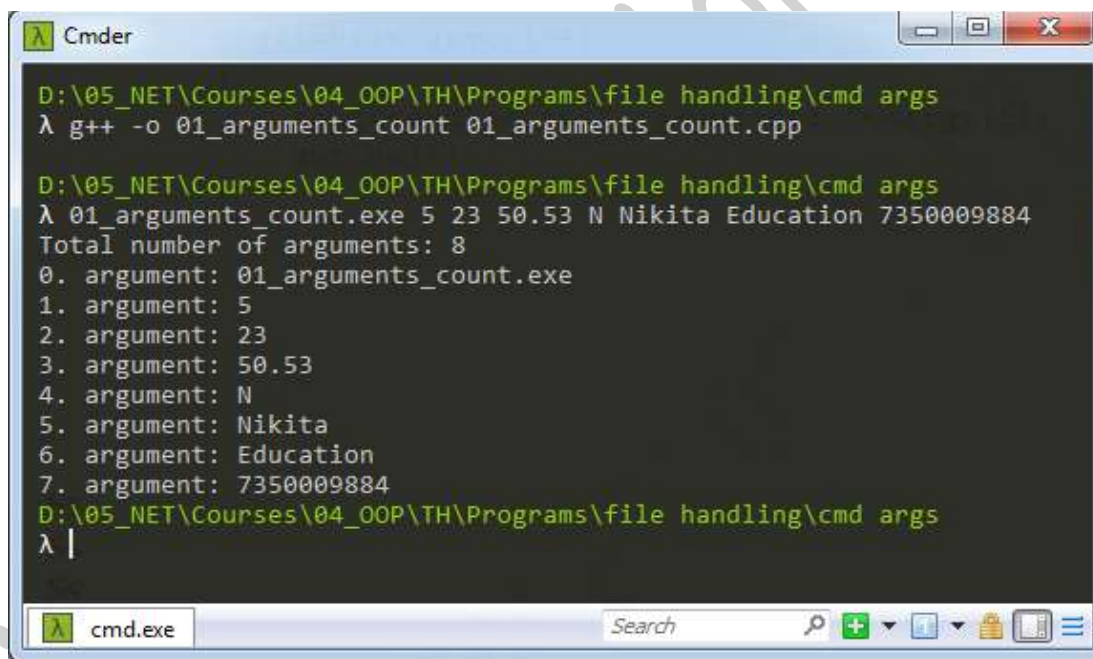
Compile and run program

Command line arguments are not compile and run like normal C++ programs, these programs are compile and run on command prompt. To Compile and Link Command Line Program we need TCC Command.

- First open command prompt
- Follow you directory where your code saved.
- For compile -> C:\TC\BIN\TCC mycmd.cpp
- For run -> C:\TC\BIN\mycmd 10 20
- **Explanation:** Here mycmd is your program file name and **TCC** is a Command. In "mycmd 10 20" statement we pass two arguments.

Example

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(int argc, char* argv[])
{
    int i;
    cout<<"Total number of arguments: "<<argc;
    for(i=0;i< argc;i++)
    {
        cout<<endl<< i<<" . argument: "<<argv[i];
        getch();
    }
}
```



```
D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ g++ -o 01_arguments_count 01_arguments_count.cpp

D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ 01_arguments_count.exe 5 23 50.53 N Nikita Education 7350009884
Total number of arguments: 8
0. argument: 01_arguments_count.exe
1. argument: 5
2. argument: 23
3. argument: 50.53
4. argument: N
5. argument: Nikita
6. argument: Education
7. argument: 7350009884
D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ |
```

```
#include <iostream>
#include<cstdlib>
using namespace std;
int main(int argc, char *argv[])
{
    int a,b, result;
    char opr;
    if(argc!=4)
    {
        cout<<"Invalid arguments...\n"<<endl;
        return -1;
    }
}
```

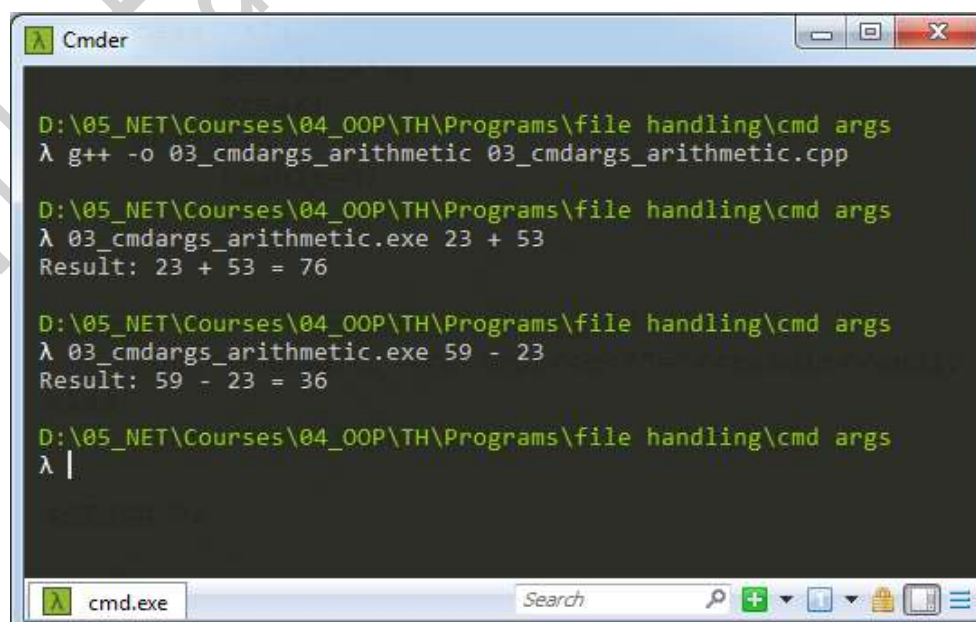
```
//get values
a = atoi(argv[1]);
b = atoi(argv[3]);

//get operator
opr=argv[2][0];

//calculate according to operator
switch(opr)
{
    case '+':
        result=a+b;
        break;
    case '-':
        result=a-b;
        break;
    case '*':
        result=a*b;
        break;
    default:
        result=0;
        break;
}

if(opr=='+' || opr=='-' || opr=='*')
    cout<<"Result:"<<a<<opr<<b<<"="<<result<<endl;
else
    cout<<"Undefined Operator...\n"<<endl;

return 0;
}
```



```
Cmder

D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ g++ -o 03_cmdargs_arithmetic 03_cmdargs_arithmetic.cpp

D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ 03_cmdargs_arithmetic.exe 23 + 53
Result: 23 + 53 = 76

D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ 03_cmdargs_arithmetic.exe 59 - 23
Result: 59 - 23 = 36

D:\05_NET\Courses\04_OOP\TH\Programs\file handling\cmd args
λ |
```

8.5 Templates

- **Template** is a feature which provides support for generic programming.
- It involves writing code in a way that is independent of any particular type.
- A template is a blueprint or formula for creating a generic class or a function.
- There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.
- You can use templates to define functions as well as classes

Function Templates

Function Templates provide a means to make a function generic.

Syntax:

```
template <class type>
return-type function-name(parameter list)
{
    // body of function
}
```

Class Templates

Class Templates provide a means to make a class generic.

Syntax:

```
template <class type>
class class-name
{
    ...
}
```

Examples:

Function Template

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

template <class T>
void swapValues(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 5, b = 7;
    float x = 65.39, y = 27.89;
    char p = 'A', q = 'B';
```

```

        cout << "Before Swapping :- " << endl;
        cout << "a = " << a << "    b = " << b << endl;
        cout << "x = " << x << "    y = " << y << endl;
        cout << "p = " << p << "    q = " << q << endl;

        /* swap the values stored in variables */
        swapValues(a, b); // swap two integer values
        swapValues(x, y); // swap two float values
        swapValues(p, q); // swap two character values

        cout << "\nAfter Swapping :- " << endl;
        cout << "a = " << a << "    b = " << b << endl;
        cout << "x = " << x << "    y = " << y << endl;
        cout << "p = " << p << "    q = " << q << endl;

        return 0;
    }

```

```

Before Swapping :-
a = 5    b = 7
x = 65.39    y = 27.89
p = A    q = B

```

```

After Swapping :-
a = 7    b = 5
x = 27.89    y = 65.39
p = B    q = A

```

```

Process returned 0 (0x0)    execution time : 0.056 s
Press any key to continue.

```

Class Template

```

#include<iostream>
using namespace std;

/* Define a generic class to perform relational operation */
template <class T>
class relational {
    T x, y;
public :
    relational(T var1, T var2) {
        x = var1;
        y = var2;
    }
    T getMin() {
        return ((x < y) ? x : y);
    }
    T getMax() {
        return ((x > y) ? x : y);
    }
};

```

```
int main() {
    relational<int> r1(11, 17);
    // object r1 with integer data members
    relational<double> r2(7.1, 6.9);
    // 'T' is replaced with 'double' type
    int min = r1.getMin(); // get min value of object r1
    double max = r2.getMax(); // get max value of object r2
    cout << "Minimum Element of object r1 : " << min << endl;
    cout << "Maximum Element of object r2 : " << max << endl;
    return 0;
}
```

```
Minimum Element of object r1 : 11
Maximum Element of object r2 : 7.1

Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

8.6 Exception Handling

- **Exceptions** are unusual conditions or problems that arises during the execution of a program like divide by zero, running out of memory or accessing an array out of its bounds.
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's Instructions.
- **Exception Handling** is a mechanism to handle such conditions at run time and prevent abnormal termination or crashing of the program.
- Handling the exception is nothing but converting system error message into user friendly error message. Use Three keywords for Handling the Exception in C++ Language, they are;
 - try
 - catch
 - throw

Syntax for handling the exception

```
try {
    // causes executions code
} catch( ExceptionName e1 ){
    // catch block
} catch( ExceptionName e2 ) {
    // catch block
} catch( ExceptionName eN ) {
    // catch block
}
```

Try Block

- It is one of the block in which we write the block of statements which causes executions at run time in other words try block always contains problematic statements.

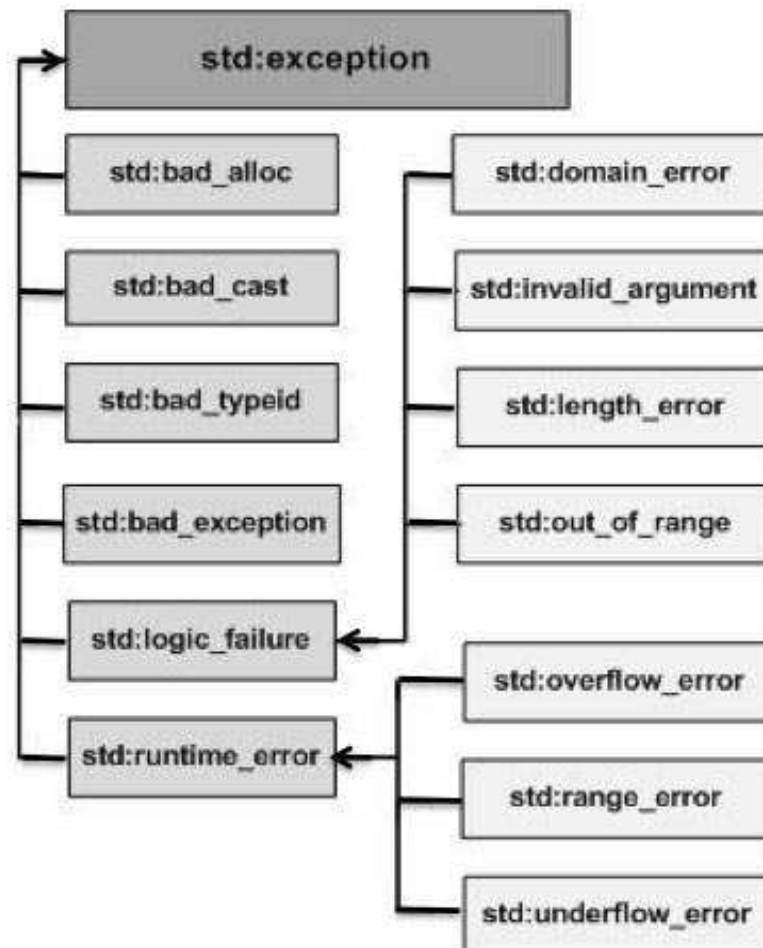
Catch block

- It is one of the block in which we write the block of statements which will generates user friendly error messages in other words catch block will support system error messages.

Throw keyword

- We can throw an exception from any part of the program using **throw** keyword whenever an anomalous situation occurs at runtime and then appropriate action is taken after catching the exception.
- Syntax: throw "string message";

C++ Standard Exceptions



Examples:

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

int main()
{
    int a,b;
    float c;

    cout<<"enter a :";
    cin>>a;
    cout<<"enter b :";
    cin>>b;
```

```
try{
    if (b==0)
    {
        throw exception();
    }
    c=a/b;
    cout<<"\ndivision is : "<<c<<endl;
}catch(exception & e){
    cout<<"Exception : "<<e.what()<<endl;
}

char s[20];
cout<<"\nEnter string : ";
cin>>s;
cout<<"\nString is : "<<s<<endl;
return 0;
}
```

Program 2:

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

float divide(float num, float denom) {
    if ( denom == 0 ) {
        throw "Divide by Zero Error";
    }
    else {
        return (num / denom);
    }
}

int main() {
    float x = 5, y = 0;
    /* protect the piece of code using try ... catch block
    and prevent crashing of the program */
    try {
        float result = divide(x, y);
        cout << "Result : " << result << endl;
    } catch (const char *err_msg) {
        cout << "Exception Caught : " << err_msg << endl;
    }

    char *s;
    cout<<"\nEnter string : ";
    cin>>s;
    cout<<"\nString is : "<<s<<endl;
    return 0;
}
```

Program 3:

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

int main()
{
    int a[]={1,2,3};
    int i;
    try
    {
        cout<<"array elements : "<<endl;
        for(i=0;i<5;i++)
        {
            if(i>2)
            {
                throw "Error: array index out of bound";
            }
            cout<<a[i]<<endl;
        }
    }
    catch(const char *msg)
    {
        cout<<msg<<endl;
    }
    cout<<"\n.....\n\n\n";
    cout<<"\nrest of the part of program"<<endl;
}
```