The Structured Approach & Object-Orientation Approach

The Structured Approach:

- Gained currency with the advent of structured programming.
- Was among the first methodologies that brought in rigour and discipline to the documentation of Systems Analysis and Design through the use of graphical notations.
- Used the Top-Down approach to the task of Systems Development.
- Laid more emphasis on the functions in the system.
- Was ideal to the analysis, design, and development of systems of smaller scope.
- Could not handle large and complex systems.
- Systems developed using this approach were not resilient to change.
- Could not therefore, evolve and adapt to changing business requirements.
- This would escalate system maintenance costs.
- A small change in system functionality would entail rewriting huge portions of code.
- This is because modules in a system developed using this approach were tightly coupled, thereby increasing the interdependency between modules.
- It was this interdependency that prevented the reusability of generic modules across systems.

Object-Orientation – A Paradigm Shift:

- Uses real-world concepts when dealing with the issue of complexity inherent in large systems.
- System structured along “objects”.
- An object can be defined as a real-world entity (physical or conceptual), that has attributes, and exhibits well-defined behaviours.
- It has a state, has a well-defined boundary, and has a unique identity.

Classification:

- One of the ways of dealing with the myriad of objects is to classify them.
- Classification is done based on common attributes and behaviors.
- Objects within the scope of a system are classified.
- Classification leads to the definition of a set of classes.

Class:

- A Class is a general description of a set of objects with common attributes and common behaviors.
- Classes generally tend to be related in the form of a hierarchy.
- Classes at a certain level in the hierarchy (except the class at the top of the hierarchy) are derived from the classes at the immediately preceding level.

Inheritance:

- A class that is derived from an already existing class is called as a Derived Class/Subclass.
- The class from which other classes are derived is called the Base class/Superclass
- A subclass not only inherits attributes and behaviors from its superclass, but also adds its own unique attributes and behaviours giving it its unique identity

Generalization/Specialization:

- Superclasses and subclasses help implement the Generalization/Specialization mechanism that is so very typical and characteristic of a hierarchy.
- Increasing levels of generalization are observed as we ascend the hierarchy.
- Increasing levels of specialization are observed as we come down the hierarchy.

OBJECT ORIENTED PROGRAMMING (C++)

In C language there is no concept of objects. C language follows structured programming. C++ follows object oriented programming.

In object oriented programming the entire environment can be considered as the collection of objects. An object is a real world entity.

Object oriented programming is a data centered view of programming in which data and behavior are strongly linked.

A widely accepted object identification or orientation approach is the famous **Grady Booch**'s grammatical analysis. According to his, an object oriented programming language (OOP) must have the following features.

1. Class & Objects
2. Abstraction
3. Encapsulation
4. Heirarchy (inheritance)

In addition to the above four features different object oriented programming languages can have additional features.

C++ Features:

C++ have so many additional features like

1. Function over loading
2. Inline functions
3. Reference variables
4. Default parameters
5. Templates
6. Operator overloading
7. Inheritance
8. Polymorphism
9. Data hiding
10. Streams
11. Exception handling etc.,

Class & Objects:

- A class is a user defined data type acts as a blue print of objects.
- A class can also defined as the generalization of similar type of objects.
- An object is real world entity. An object is an instance of class. The variable declared by using a class can be called as object.
- It is the responsibility of the programmer to define the class, once the class was defined we can declare any number of objects of that type.

Abstraction:

- Abstraction is the principle of ignoring the aspects of an object that are not relevant to the current purpose in order to concentrate morefully on those that are required.
- By using abstraction we can represent only the required attributes and required operations (functions) with in the class.

Abstraction can be considered in 2 ways.

1. Data Abstraction : To represent attributes
2. Procedural abstraction : to represent function

Encapsulation:

Encapsulation is the idea of packaging data and operations together in a well defined programming unit. In C++ the class structure provides for encapsulation by packaging both data and functions that operate upon data into a single class unit.

The data and function are wrapped together in a class therefore, the data of the object is hidden from others.

The advantages of encapsulation are as follows.

1. Data Security: Protection of data from accidental changes by external module.
2. Changes to data and functions of an object can be made without effecting other objects.
3. Objects are independent of each other. Therefore each object can be studied properly for better understanding.

Class Diagram is

Student
No :integer Name : String branch : String year :integer college : string
Void getdata ( ) Void putdata ( )

Class Definition is

```
Class classname
{
        member variable declarations;
        member function declarations;
};
```

Hierarchy:

- It is a feature of OOP, which allows making use of the existing class without making changes to it. This can be done by deriving a class from existing class.

- By using this feature all the related classes of a problem can be represented in a hierarchical form.

Function Overloading:

- Function overloading means providing new meanings or additional meaning to one function.

- In C++, there can be more than over function with same name. ie more than one function with same name and with different definitions.

- Even though the names of the functions are same, there should be some differences in their parameter lists to distinguish them. The parameter list can differ in 3 ways as

1. No of parameters
2. Datatypes of parameters
3. order of parameters.

Ex:

```
int add (int x, int y)
{
        return (x+y);
}

int add (int x, int y, int z)
{
        return (x+y+z);
}

float add (float a, float b)
{
        return (a+b);
}
```

In C++, function prototype is

returntype functionname (parameter list);

Ex:-

```
int add (int, int);
int add (int, int, int);
float add(float, float);
```

In C++, function name and parameter list can be combinely used as function signature to identify the function.

- **Ambiguity in function overloading:**

- Main cause of ambiguity involves C++'s automatic type conversions as well as providing default arguments
- C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function.
  - For ex: func(10) call will cause an ambiguity if the function is actually receiving a float or double.
  - func(int x=10, int y=100) may conflict with func()

Inline Functions:

Whenever a function f2() is called from the body of f1(), control goes to f2() function, executes the body of f2() and finally control returns to f1 function. But while jumping system control from f1() to f2(), some of the information like next executable statement's address, register variable values of f1(), intermediate results generated can be stored in stack. Finally while returning the control all those details can be fetched from stack. For this storing and fetching process some of the system time can be wasted. To overcome this problem we can use inline functions.

If we define the function as inline functions then the definition of that function can be stored at calling statement at the time of compilation. So during running no need to jump the system control. Hence time can be saved.

An inline function can be defined as

```
inline returntype functionname (parameter list)
{
    body;
}
```

An inline function must satisfy some rules

1. It should be small function.
2. It should not contain complex statements like selective statements, loops, jump control elements, function calls etc.,

If any of the above 2 rules are not satisfied then compiler treats that function as normal function (even though if we use keyword inline in function definition). Thus inline function is only a request to compiler but not order.

- Inline function execution is faster than ordinary function execution.

```
Ex: inline void f1()
{
        cout << "Hyderabad";
}
```

```
main ()
{
        int a=3;
        int b=4;
        f1();     // replaced by body
        cout << a << b;
        f1();    // replaced by body
}
```

♥ **Inline Vs. Macros:**

- Inline functions are functionally similar to #define macros. In both the cases, during compilation time, the body of the macro or the function is expanded.
- But inline functions are preferred over macros because of two main reasons:
- First, in the case of inline functions, the types of the arguments are checked against the parameter list in the declaration for the function. As a result, any mismatch in the parameters can be detected at the time of compilation.This allows inline functions to be overloaded, which is not possible in the case of macros.
- Second, there are certain situations where a macro does not behave in the same manner as a function call, which may lead to unpredictable results.

To compute the square of a given number, either a macro or an inline function can be used as follows:

```
#define square(x) x*x

        OR

inline int square(int x)
 {return(x*x);}
```

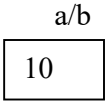
```
void main( )
 {
cout << square( 1 + 2) << "\n";
}
```

- If square( ) is a macro, then the above program prints 5.
- But if square( ) is an inline function , it prints the correct output 9.

- This is because if square( ) is a macro, all occurrences of square(x) are replaced with "x*x".
- So, in this case square (1+2) is replaced with 1+2*1+2 which yields 5 as the result due to precedence rules of operators.
- In the case where square( ) is an inline function, the parameter expression is evaluated first, and then sent down the function.
- So, the function body will evaluate the expression 3*3.
- All the functions declared as inline need not be treated as inline by the compiler.
- The compiler ignores the inline specification if the function uses recursive calls, or is too large.

Reference Variables:

Consider the declaration, int a=10;

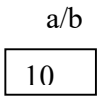


If we want to give another name for the memory cell a, it can be with reference variable. as

int &b =a;

&b means b is a referencing variable ie b is alias name given to the existing variable 'a'.

Ex:-




```
int a=10;
int &b=a;
a=a+5;
cout<<b; o/p is 12
```

changes to a are as good as changes to b and viceversa.

It is not possible to separate declaration and initialization of reference variables as

```
int &b;
b=a;
```

✕

Reference variables has to initialized in the declaration itself.

Call by reference:

Here actual parameters must be the memory variables, formal parameters are the alias names given to the memory variables. Both actual and formal parameters refers to a single memory location.

```
Void swap (int &x, int &y)
{
                                    int temp;
                                    temp=x;
```

```
        x=y;
        y=temp;
}
main ()
{
        int a=100,b=200;
        clrscr();
        swap(a,b);
        cout<<a<<b;
}
```

Here a, b are the memory variables and x, y are alias names to them. Here both calling and called functions are accessing common location called function with original names and calling function with alias names.

If any modifications are made by the calling function those will automatically reflected to called function.

No need of extra memory to formal parameters.

Description of 3 parameter passing techniques

No.	Description	Call be value	Call by pointer	Call by reference
1	Actival parameters	Variables or constants	Addresses or picture variables	Memory variables
2	Formal parameters	Mem variables	Pointer variables	Reference variables
3	Mem to formal parameters	Mem to hold the input data	Mem to hold the addresses of inputs	No need of memory
4	Reflection of modifications	Not reflected	Reflected	reflected

Streams:

A stream is a flow of data from one unit of computer to another unit

I/p device to memory - input stream
Memory to o/p device - output stream
File to memory - input file stream
Memory to file - output file stream

Different streams can be used to move data from i/p devices to memory and from memory to o/p devices.

In c++, we have all the predefined class definitions to streams.
istream is a predefined class in “iostream.h” cin is an object of that class.
By using cin we can read any no of inputs to memory variables of any time.

Syntax is :
Cin >> var1>>var2>>var3……;

Var1, var2, var3, …. Etc., are memory variables of any type.

Ex:-

```
int a;
float b;
char c;
cin >> a>> b>>c;
```

- By using a single cin we can read any no of inputs of any type.
- The no of >> must be equals to no of memory variables.
- >> show the direction of flow of data from input devices to memory variables.

Ostream is the predefined class in "iostream.h". Cout is an object of ostream.
By using cout we can print any no of messages or values of variables or constants of any type.

Cout << arg1 << arg2 <<arg3 ….;

Here arg1, arg2, arg3 etc are either variables or constants of any type.
Ex:
int a=10;
int b=20;
cout << "Values of a and b are " << a << "and" <<b;
In sting constants we can include backslash characters for escape sequencing.

Default parameters:

```
Float SI (float p, float t, float r)
{
      return (p*t*r/100);
}
```

```
{
Float SI(float, float b=0.25, float-c=0.10);
float ans1= SI (1000,0.20,0.12);
float ans2= SI (2000,0.22);
float ans3= SI (3000);
cout << ans1 << ans2 << ans3;
}
```

main()

- In the prototyping statement default values are given to 2nd and 3rd parameters.
- In the first call given 3 values can be taken as values to 3 parameters
- In the 2nd call given 2 values can be taken as values to first 2 parameters and default value can be considered to 3rd parameter.

- In the 3rd call, one value to first parameter for the remaining 2 default values can be considered.

Thus whenever actival values are missing default values are send to the function. The following call gives error

Ans4=s1();
Since there is no default value for p

1. Default values should be specified only in the function prototype.
2. Default value parameters should appear at the end of the parameter list.

Ex: Calculate a^b default value for b=2

```
main ()
{       char ch;
        float b;
        int b;
        double power (float a, int b=2);
        cout <<"Enter base:";
        cin>>a;
        cout<<"Are you entering power";
        cin >>ch;
        if (ch=='Y' || ch == 'y')
        {
                cout << "enter power";
                cin >> b;
                        ans=power(a,b);
        }
        else
                ans=power(a);
        cout "result is" <<ans;
        getch();
}
```

Templates:

```
Void swap (int &x, int &y)
{
        int temp;
        temp = x;
        x=y;
        y=temp;
}
```

This function can be used to swap 2 integers. Suppose if we want to swap floats, chars or objects then it cannot be used. In order to do so separate functions has to be implemented. If we observe all these functions logic, body is same only differences in their datatypes. Writing separate functions with same logic is a tedious process. To over come this problem, only one function can be given by templates and used for all types.

```
Template < class t>
Void swap (t &x, t &y)
{
        t temp;
```

```
        temp = x;
        x=y;
        y=temp;
}
```

T is a generic datatype. According to our requirement system can replace T with any valid data type and creates copies internally for different types.
Based on the actual parameter datatypes, T will be replaced and the individual functions can be created to swap ints, floats , chars etc.,

Operator Overloading:
For all the operators of C++ there are predefined meanings or operations to work with particular type of data that meaning is not applicable to work with user defined data types.

int + int is valid
float + float is valid
char + char is valid
object + object is invalid
book + book is invalid

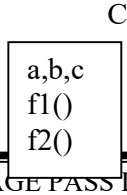
Operator overloading is the feature of c++ used to provide new meanings to work with objects. Once we define the meaning of operator to work with particular type of data then that operator can be used directly on objects just like fundamental type of data.

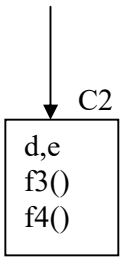
Suppose if we define the meanings of +, -, *, / with complex no then we can write the statements like

c1 + c2 * c3 + c4 etc.,
with operator overloading we can give new meanings but not possible to change the syntaxes and priorities etc.

Inheritance:

Inheriting or deriving one class contents into another class is known as inheritance.





Here class c2 contains 5 variables and 4 functions ie 3 variables a,b,c, and 2 functions f1(), f2() are inherited from class c1, 2 functions f3(), f4() and 2 variables d,e were newly added to it.
Here class c1 is base class and class c2 is derived class.

Polymorphism:
Poly means many. Forms means behaviors.
For a single function call we can find more than one behaviour (functionality) within a single program.
By using base class pointer we can hold either base class object or derived class object and also we can call the member functions of base class or derived class.

```
C1 * p;
C1 x;
P = &x;
P->  Display();              // base class display

C2  y;
P = &y;
P -> display();                    // c2 class display

C3 z;
P= &z;
p-> display();                     // c3 class display
```

Here for p-> display() three function definitions and can be executed for 3 calls.
Here calling statement is same but behaviours are different.

Data Hiding :

In C++ we can hide the data of an object from the non member functions. Function interfaces can be given to non member functions to communicate with objects.

While defining the classes all the variables can be declared as private members and all the functions can be declared as public members.

By representing member variables (data) under private we can hide them from non member functions.

Strong Typing:

- C++ is a strongly typed language.
- The C++ compiler enforces type checking of each assignment made in a program at compile time.
- Both the argument list and the return type of each function are type checked during compilation. An implicit conversion will be applied if possible.
- If this is not possible, or if the number of arguments is incorrect, then a compile time error is issued

Const Qualifiers:

These are read-only variables

- must be initialized in declaration
  const float NORMAL_TEMP = 98.6;
- better than using pre-processor #define to declare a constant (goes in compiler symbol table) not just values, but pointers, references and member functions can be declared const also

Type Cast Operator :

The C++ cast operator is used to explicitly request a type conversion. The cast operation has two forms.

```
int      intVar;
float   floatVar  =  104.8 ;
intVar  =  int ( floatVar ) ;          // functional notation, OR
intVar  = ( int ) floatVar ;           // prefix notation uses ()
```

How to define our own classes:

Syntax to define a class is as follows

class class name
{
access specifier:
data declaration
function declaration

};

Class and access specifiers are keywords. Variables declared inside class are called **member variable** and functions declared inside the class are called **member functions**

- In C++, Functions can be defined inside the class.

Scope:

- Every variable has an associated scope. The scope defines the context of the variable. The scope of a variable specifies where the variable may be accessed.
- C++ provides for three types of scope: **file**, **class** and **local** scope.

♥ **File Scope:**

- File scope is considered to be the outermost scope.
- It is defined as the space outside all functions.
- Variables that have file scope are accessible to all the functions in the program file, and are referred to as global variables.
- Such variables are defines outside main( ).

♥ **Local Scope:**

- Local scope is defined as being limited to the braces of a function, or control structure like if, while, and for.
- Variables that have local scope are not accessible outside the function or the control structures.
- Nesting of local scope is possible.

Accessibility of Class Members:

- On the other hand, when a class declaration is used for the Point data type as depicted earlier, the data members and the member functions are accessible from only within the class.
- Data and methods within the class declaration will no longer be visible to functions outside the class point. The member functions to get and set the x and y coordinates can no longer be called from main( )
- Therefore, all members in a class are by default private, thereby not being accessible outside the class.

Access Specifiers:

- C++ offers the designer of the class the flexibility of deciding which member data or methods should be accessible from outside the class, and which should not.
- Access specifiers serve the important purpose of drawing the line between the accessible and the inaccessible parts of a class.

- It is the class designer's prerogative to demarcate the part of the class that needs to be hidden, and the part that needs to be offered to the user as an interface to the class

- The **private** access specifier is generally used to encapsulate or hide the member data in the class.
- The **public** access specifier is used to expose the member functions to the outside world, that is, to outside functions as interfaces to the class.

Constructors

- Every time an object is created, memory is allocated for it. But allocation of memory does not automatically ensure initialization of member data within the object.
- Constructor is a special number function inside the class. Its job is initializing the object. In other words initializing the member variables of the object.
- Constructors are used to perform the initialization activities on objects whenever those were created.

Properties of constructors are

1. The name of the constructor is the class name.
2. Constructor should be declared in public part only.
3. Constructors are implicitly called i.e. whenever the object is declared in a function immediately a constructor is called. Thus other member functions are explicitly called when as constructors are implicitly called.
4. Constructors can not have any return type even void also.

   There are three types of constructors:
   a. Default constructor.
   b. Parameter constructor.
   c. Copy constructor.

Default constructor:

It has no parameters.

The body of the constructor can be according to our requirement

Consider the following class

```
Class C1
    {
        private:
                int a;
                int b;
                int c;
      public:
                C1( );  // prototype of the default constructor
                -----
                -----
};
```

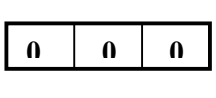
definition of the default constructor is as follows

```
C1 :: C1 ( )
      {
         a = 0;
         b = 0;
         c = 0;
      }
```

Functions call is as follows

```
C1 Θ 1;
```

Θ 1 object is created and default constructor is called,



Ex: -

```
C1 : : C1 ( )
{
      Cout <<" Enter any three integers";
      Cin >>a>>b>>c;
}
```

call: C1 Θ 1;

Θ 2 is created and a, b, c are accepts values form the keyboard.

It is not compulsory to initialize all the variables in default constructor. We can initialize a subset of the variables.

Parameter Constructor:

Parameter constructor is a constructor with certain no of parameters

Consider the following class

```
Class C1
   {
      private:      int a;
                    int b;
                    int c;
        public:
                    C1( int ,int, int );  // prototype of the  3 parameter constructor
                    C1(int); // prototype of the  1 parameter constructor
                    -----
   };
```

definition of parameter constructor is as follows

```
C1 : : C1(int p, intq , int r)
    {       a = p;
            b = q;
            c = r;
    }
```

Function call: -

C1 Θ 3 (10, 20, 30);

Θ 3 is crated and parameter constructor is called.

```
C1::C1(int x)
  {
        a=x;
        b=x;
        c=x;
  }
```

call for the above constructor is as follows

```
C1 x(15);
```

Suppose the function prototype contains default values for all the three parameters p, q, r, then the declarations **C1 y ;** gives error. This is because there is confusion has to which constructor is to be called. Ie default or parameter constructors. If there is no default constructor in the program then there is no error and Pc is called by taking default values for all the three arguments.

Copy Constructor:

Copy constructor job is copying the contents of one object into another object. Copy constructor will have only one parameter which is the object of same type. It must be a reference object for faster accessing.

```
Class C1
    {
      private:
            int a;
            int b;
            int c;
       public:
            C1( C1 & );  // prototype of the  copy constructor
            -----
            -----
            -----
    };
```

Definition of copy constructor is as follows

```
C1 : : C1(C1 & Θ 5)
        {
                a = Θ 5.a;
                b = Θ 5.b;
                c = Θ 5.c;
        }
```

call to copy constructor is as follows

C1 Θ 2(Θ 1)

Θ 2 is created and copy constructor is called to copy contents of Θ 1 to Θ 2.

Note:

- If we have the default constructor and also parameter constructors with default values then ambiguity errors can be raised
- The constructors with only one parameters can be invoked with assignment operator also.

We can call the one parameter constructor as follows also

```
C1 x(10);//1PC
    or
C1 x=10;//1PC
    or
C1 x;//DC

X=10;//1PC
```

We can call the copy constructor as follows also

```
C1 Θ 2(Θ 1) //CC
    or
C1 Θ 2=Θ 1// CC
    or
C1 Θ 2;   // dc executed
Θ 2 = Θ 1;    // cc executed
```

Destructors

Destructor is special member function inside the class. Its job is destroying the objects which are crated inside the block . destructors can be used to deallocate the resources given to object.

Ex: -
```
C1 : : ~ C1 ( )
{
        Cout<<" object is destroyed";
}
```

- The name of the destructors is ~ Class name
- Destructors should be written under public only.
- Destructors are called implicitly, when the block or function is completed.
- If n objects are created inside the block, then destructor is called n times when the block is completed.
- The objects are destroyed in the reverse order of creation.
- Objects which are created statically or dynamically can be destroyed.
- Destructors can not have return type not even void.

```
Class C1
{
        private :
                int a, b, c;
        public :
                C1 ( )
                {
                        Cout<<" objects is created";
                }
                ~C1 ( )
                {
                        Cout<<" object destroyed";
                }
};
void main ( )
{
        C1 Θ 1, Θ 2;
        {
                C1 Θ 3;
                {
                        C1, Θ 4, Θ 5, Θ 6;
                }
        }
}
```

Differences between delete & destructor:

- Delete operator is used to destroy only those objects which are created dynamically. If an object is created statically we can not use delete operator for destroying it.

Ex: - **correct** **wrong**

```
C1 * a;                 C1 Θ 1;
a = new C1;             C1 * p = & Θ 1;
   -                        -
   -                        -
delete ( a);            delete ( p);
```

- Destructor can destroy all the objects that are created inside the block. It does not matter whether they are crated statically or dynamically.

The this pointer:

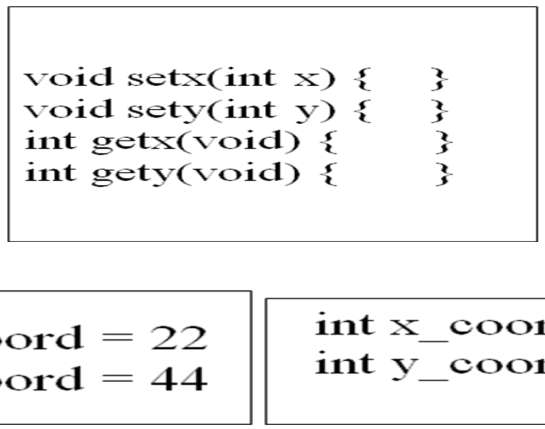
❖ **this pointer** holds the address of the object which invokes the function

Consider the following code:

```
class point
 {
   private:
    int x_coord;
    int y_coord;

   public:
    void setx( int x)
    {    x_coord = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
    void sety (int y)
     { y_coord = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
    int getx( void)
     {  return x_coord; }
    int gety( void)
     { return y_coord;}
}; // end of class
```

```
main( )
 {
  int a, b;
   // an object p1 of class type point, class keyword not required
   point p1,p2;
   p1.setx(22); // set the value of x_coord  of object p1
   p1.sety(44); // set the value of y_coord of object p1
   a = p1.getx( ); // return the value of the x_coord member of object p1
   b = p1.gety( ); // return the value of the y_coord member of p1
   p2.setx(10);
   p2.sety(20);
   }
```



If every instance of an object has its own copy of all member functions within it, it will be a considerable constraint on memory overhead. Therefore, each object maintains its own copy of member data. Only one copy of member functions exists in memory. If only one copy of a member function exists, how are the data members of an object bound to the references to the data members within the functions? For instance, if setx( ) were to be invoked, how does it know which copy of x_coord should be manipulated, p1.x_coord or p2.x_coord? The answer to the above question is the this pointer.

- Each class member function contains an implicit pointer of its class type, named **this.**
- The **this** pointer, created automatically by the compiler, contains the address of the object through which the function is invoked.
- Therefore, when the member function setx( ) is invoked through p1, the function setx( ) implicitly receives the address of the object p1 (***this**), and therefore, the x_coord of p1 is set.

Scope Resolution Operator :: :

- Defining member functions within the body of the class will be inline function bydefault.
- C++ provides the scope resolution operator **::** that allows the body of the member functions to be separated from the body of the class.
- Using the **::** operator, the programmer can define a member function outside the class definition, without the function losing its connection to the class.

The inline keyword needs to be explicitly used when the body of the function has been separated from the class definition using the scope resolution operator.

Example:

```
class a
{
int x;
public:
inline void getx();
};
void a::getx()
{
cin>>x;
}
```

When local variable and global variable names are same we can also use scope Resolution Operator (::) to access Global variable in a local scope

Example:

```
int global;
main()
{
int  global=10;
cout<<global; //it prints value of local scope global variable ie 10
cout<<::global; //it prints value of global  scope global variable ie. 0;
}
```

Static Class Members – Static Data Members

- Both function and data members of a class can be made static.
- When you precede a member variable's declaration with the keyword static, you are telling the compiler that only one copy of that variable will exist, and that all objects of that class will share that variable.

Static Data Members:

```
#include <iostream.h>
class static_demo
 {
  private:
   static int a;
   int b;
  public:
   void set ( int i, int j)
   {a = i; b = j; }
   void show( );
  };
int static_demo::a; // define the
static variable a
void static_demo::show( )
 {
   cout << "this is static a:  " << a;
   cout << this is non-static b:  " << b;
}
```

```
int main( )
 {
   static_demo x, y;
   x.set(1, 1); //set a to 1
   x.show( );
   y.set(2, 2); // change a to 2
   y.show( );
   x.show( ); /* Here, a has been
changed for both x and y because a is
shared  by both objects */
   return 0;
  }
```

Static Data Members – Uses:

- An interesting use of a static member variable is to keep track of the number of objects of a particular class type that is in existence. Consider the following example:

```
#include <iostream.h>
class counter_test
  {
    public:
     static int count;
     counter_test ( ) { count++; }
     ~counter_test ( ) { count--;}
   };

int counter_test::count;
```

```
void f( )
 {     counter temp;
    cout << objects in existence:  " <<
counter_test::count << "\n";
    // temp is destroyed when f( )
returns
  }
```

```
int main( )
 {    counter_test ob1;
   cout << objects in existence:  " <<
counter_test::count << "\n";
   counter_test ob2;
   cout << objects in existence:  " <<
counter_test::count << "\n";
   f( );
   cout << objects in existence:  " <<
counter_test::count << "\n";
   return 0; }
```

Static Member Functions:

- Member functions may also be declared static.
- Static member functions are subject to several restrictions.
- They may only directly refer to other static members of the class.
- A static member function does not have a this pointer.
- There cannot be a static and a non-static version of the same function.
- A static member function may not be virtual.
- One good use for them is to "pre-initialize" private static data before any object is actually created.

```
#include <iostream.h>
class static_type
  {
   private:
    static int i;
   public:

static void init ( int x)
      { i = x; }
    void show ( )
     { cout << i;}
    };
int static_type::i; // define i

int main( )
 {
   // initialize static data before object
creation
  static_type::init(100);
  static_type x;
  x.show( ); // displays
  return 0;
 }
```

Arrays of Objects:

- In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for arrays of primitive data types.

This example uses a three-element array of objects:

```
#include<iostream.h>
class c1
 {
   private:
    int i;
   public:
    void set_i (int j)
     { i = j; }
int get_i( )
  { return i; } };
int main( )
  {
    c1 ob[3];
    int i;
    for (int i = 0; i < 3; i++)
     ob[i].set_i(i+1);
    for ( i = 0; i < 3; i++)
       cout << ob[i].get_i( ) << "\n";
    return 0;
   }
```

- If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for arrays of primitive data types.
- However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructor function.
- For objects, whose constructor functions have only one parameter, you can simply specify a list of initial values, using the normal array initialization syntax.
- As each element in the array is created, a value from the list is passed to the constructor's parameter. The following example illustrates this point:

```
#include<iostream.h>
class c1
 {
   private:
    int i;
   public:
    c1( int j)
     { i = j; }
int get_i( )
  { return i; } };
int main( )
 {
   c1 ob[3] = {1, 2, 3};
   int i;
   for ( i = 0, i < 2, i++)
    cout << ob[i].get_i( ) << "\n";
   return 0; }
```

As before, this program displays the numbers 1, 2 and 3 on the screen. Actually, the initialization syntax shown in the preceding program is a shorthand for this longer form:

```
C1 ob[3] = { c1(1), c1(2), c1(3) };
```

Here, the constructor for c1 is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form.

❖ The following example illustrates passing list to a two parameter constructor

```
#include<iostream.h>
class c1
{
  private:
   int h, i;
  public:
   c1( int j, int k)
    { h = j;
      i = k; }
    int get_i( )
     { return i; }

int get_h( )
  { return h; }
};

int main ( )
{
  c1 ob[3] = {c1(1, 2), c1(3, 4), c1(5, 6)} // initialize
  int i;
  for ( i = 0; i < 3; i ++)
   {
     cout << ob[i].get_h( ) << ", "<< ob[i].get_i( ) << "\n";
   }
  return 0
}
```

Here, c1's constructor has two parameters, and therefore, requires two arguments. This means that the shorthand initialization format cannot be used, and the long form, shown above, must be employed.

Pointers to Objects:

- Just as you can pointers to primitive data types, you can have pointers to objects as well. When accessing members of a class given a pointer to an object, use the arrow operator (->)
- When a pointer is incremented, it points to the next element of its type in an array. All pointer arithmetic is relative to the base type of the pointer, i.e., it is relative to the type of data that the pointer is declared as pointing to.
- The same is true of pointers to objects.

```
#include<iostream>
using namespace std;
class c1
 {
   private:
    int i;
   public:
    c1 ( )
     { i = 0; }
    c1 (int j)
     { i = j; }
    int get_i( )
      { return i; }
   };
```

```
int main( )
 {
   c1 ob[3] = {1, 2, 3 };
   c1 *p; // pointer p of class type c1
   int i;
   p = ob; // get starting offset of array
  for ( i = 0; i < 3; i++)
   {
     cout << p->get_i( ) << "\n";
     p++; // point to the next object in
the array
    }
   return 0;
}
```

Pointers to Derived Types:

- Assume two classes B and D. Further, assume that D is derived from the base class B. In this situation, a pointer of type B may also point to an object of type D.
- To generalize, a base class pointer can also be used as a pointer to an object of any class derived from that base. The reverse, however, does not hold true.
- Using a base class pointer, you can access only the members of the derived type that were inherited from the base. But, using a base class pointer, you cannot access members added by the derived class.

Consider the following program:

```
#include<iostream.h>
class base
```

```
{
  private:
   int i;
  public:
   void set_i( int num)
    { i = num; }
   int get_i( )
    { return i; }
};
```

```
class derived : public base
{private:
   int j;
  public:
   void set_j( int num)
    { j = num; }
   int get_j( )
    { return j; } };
int main( )
{ base *bp;
  derived d;
  bp = &d; // base pointer points to derived object
  bp->set_i( 10); //access inherited members from derived object
  cout << bp->get_i( ) << " ";
  bp->set_j(22); //ERROR
 cout << bp->get_j( ); // ERROR
 return 0; }
```

Passing Object References :

- When an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called.
- If you do not want the destructor function to be called, simply pass the object by reference. When you pass by reference, no copy of the object is made.
- This means that no object used as a parameter is destroyed when the function terminates, and the parameter's (object's) destructor is not called.

```
#include<iostream.h>
class c1
{
  int id;
  public:
   int i;
   c1( int num)
   ~c1( );
   void negate(c1 &ob)
     {ob.i = -ob.i; } };
c1::c1(int num)
 {
   cout << "constructing " << num;
   id = num;
  }

c1::~c1( )
 { cout << "destructing " << id << "\n";
}

int main( )
 {c1 o(1);
  o.i = 10;
  o.negate(o);
  cout << o.i << "\n";
  return 0; };
```

Here is the output of the program
Constructing 1
-10

Destructing 1

Only one call is made to c1's destructor function. Had o been passed by value, a second object would have been created inside negate(), and the destructor would have been called a second time when that object was destroyed at the time negate( ) terminated.

The Dot . Operator

- When you access a member of a class through a reference, you use the dot operator.
- The arrow operator is reserved for use with pointers only.
- Passing all but the smallest objects by reference is faster than passing them by value. Arguments are usually pushed onto the stack.
- Thus, large objects take a considerable amount of CPU cycles to push onto, and pop from the stack.

Returning References

- A function may return a reference. This has the startling effect of allowing a function to be used on the left hand side of an assignment statement. Consider the following program:

```
#include<iostream.h>
char& replace (int i); //returns a reference to a character
char s[80] = "hello there";
int main( )
 {replace(5) = 'x' //assign x to space after hello
   cout << s;
   return 0; }
char& replace( int i)
 { return s[i]; }
```

Independent References

- You can declare a reference that is simply a variable. This type of reference is called an independent reference. An independent reference is another name for an object variable.

- All independent variables must be initialized when they are created. Apart from initialization, you cannot change what object a reference variable points to.

```
#include<iostream.h>
int main( )
{ int a;
 int &ref = a; // independent
reference
 a = 10;
 cout << a << " " << ref << "\n";
 ref = 100
 cout << a << " " << ref << "\n";
 int b = 19;
 ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref - -
cout << a << " " << ref << "\n";
 return 0;
}
```

References to Derived Types

- A base class reference can be used to refer to an object of a derived class of that base class.
- The most common application of this is found in function parameters.
- A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions on References:

- You cannot reference another reference. In other words, you cannot obtain the address of a reference.
- You cannot create arrays of references.
- You cannot create a pointer to a reference.
- A reference variable must be initialized. Null references are prohibited.

Inheritance

- Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications.

- Using inheritance, you can create a general class that defines traits common to a set of related objects, i.e, objects with common attributes and behaviours.
- This class may then be inherited by other, more specific classes, each adding only those attributes and behaviours that are unique to the inheriting class.

Code Reuse

- Inheritance leads to the definition of generalized classes that are at the top of an inheritance hierarchy. Inheritance is thus the implementation of generalization.
- Inheritance, in an object-oriented language like C++ makes the data and methods of a superclass or base class available to its subclass or derived class.
- Inheritance has many advantages, the most important of them being the reusability of code. Once a class has been created, it can be used to create new subclasses.

Generalization/Specialization

- In keeping with C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is referred to as the derived class.
- Each instance of a derived class includes all the members of the base class. The derived class inherits all the properties of the base class.
- Therefore, the derived class has a large set of properties than its base class. However, a derived class may override some of the properties of the base class.

Advantages: -

- Reusability of the code: - Here C1 class functions are inherited into C2 class. Hence C1 functions can be used in C2 class without re writing the code.
- Time is saved: - The time involved in designing C1 class functions is eliminated. Since they are inherited to class C2.
- Memory Saved: - The cost incurred in developing C1 class functions is eliminated.
- Reliability: - Since C1 class is existing class, we can replay on C1 class functions.

Code Syntax for Inheritance

- Class inheritance uses this general form:

  class derived-class-name : **access specifier** base-class-name

  {

Code for new qualities and to redefine the qualities of base class

};

- The access status of the base class members inside the derived class is determined by access specifier.
- The base class access specifier must either be public, private, or protected.
- If no access specifier is present, the access specifier is private by default.

Inheritance – private members:

In inheritance private members are not inherited. We know that variables are written in private part and functions are written is public part generally. Hence variables will not be inherited and function is inherited. In order to inherit variables also they should be written under public in such case data abstraction is loosed.
To reaction data abstraction and perform inheritance a new scope called protected is used.

Protected Scope: - The protected members of base class can be inherited to derived class. Further the protected members can not be accessed in non-member functions. They can be accessed only in member functions. Thus protected is enhanced form of private.

Characteristic	**Private**	**Protected**	**Public**
Data abstraction	The private and protected member can be accessed only with in member function they can not be accessed with in non - member function.		Accessed both in member and non - member functions.
Inheritance	Can not be inherited to derived class	The protected and public members can be inherited to derived class.	

Types of inheritance:

By depending upon number of base classes ,number of derived classes and number of levels of inheritance we can classify the inheritances into different types . those are

1. Simple inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. hybrid inheritance

Simple Inheritance:

One base class, one derived class and one level of inheritance.
All the qualities of base class can be given to derived class.

Multilevel Inheritance:

Here we can have morethan one level of inheritance.
A is the base class to B and grand base class to C
B is the deraved class of A and Blasé class to C
C is the deraved class of B and Grand deraved class to A
B can have its own qualities and also the qualities of A
C can have its own qualities, qualities of B and qualities of A.

Multiple Inheritance

Morethan 1 base class to a single deraved class.
Here A and B are base classes to C
C can have its own qualities and also the qualities of A and B

Hieratical Inheritance
One base class to morethan one deraved classes

Hybread Inheritance
It is the combination of morethan of one type of Inheritance

Access Specifier – Public:

- When the access specifier for a base class member is public, all public members of the base class become public members of the derived class.
- All protected members of the base class become protected members of the derived class.
- In all cases, the base class' private elements remain private to the base, and are not directly accessible by members of the derived class.

Access Specifier – Public (Code):

```
#include<iostream.h>
class base
{
  private:
   int i, j;
  public:
    void set( int a, int b)
     { i = a;
       j = b; }
void show( )
  {
    cout << i << "  " << j << "\n";
  }};
class derived : public base
  {
    private:
     int k;
    public:
     derived (int x)
      { k = x; }
void showk( )
  { cout << k << "\n";} };
int main( )
  {
    derived ob(3);
    ob.set(1,2); // access member of
base from derived object
    ob.show( ); // access member of
base
    ob.showk( ); // uses member of
derived class
    return 0; }
```

Access Specifier – Private:

When the base class is inherited by using the private access specifier, all public and protected members of the base class become private members of the derived class.

The following program will not even compile because both set( ) and show( ) are now private members of derived:

```
#include<iostream.h>
class base
{
  private:
   int i, j;
  public:
    void set( int a, int b)
     { i = a;
       j = b; }

void show( )
  { cout << i << "  " << j << "\n"; }
};
class derived : private base
  {
    private:
     int k;
    public:
     derived (int x)
      { k = x; }
void showk( )
  { cout << k << "\n";}
};
```

```
int main( )
  {
    derived ob(3);
    ob.set(1,2); //error, can't access
set( ) from outside derived
    ob.show( ); // error, can't access
show( ) from outside derived
    return 0;
  }
```

Inheritance and Protected Members:

- The protected keyword is included in C++ to provide greater flexibility in the inheritance mechanism.
- With one important exception, access to a protected member is the same as access to a private member. The sole exception is when a protected member is inherited.
- If the base class is inherited as public, then the base class' protected members become protected members of the derived class, and are therefore, accessible by the derived class.

```
#include<iostream.h>
class base
{
  protected:
   int i, j;
  public:
    void set( int a, int b)
     { i = a;
       j = b; }
void show( )
  {
    cout << i << "  " << j << "\n"; }
};
class derived : public base
  {
    private:
     int k;
    public:
Void setk (  )
 { k = i * j; } // access to protected
members
void showk( )
  { cout << k << "\n";} };
int main( )
  {
    derived ob(3);
    ob.set(1,2); //OK, known to derived
    ob.show( ); // OK, known to
derived
    ob.setk( );
    ob.showk( );
    return 0;  }
```

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited by the first derived class may also be inherited as protected again by a second derived class.
For example, the following program is correct, and derived2 does indeed have access to i an j

```
#include<iostream.h>
class base
{
  protected:
   int i, j;
  public:
    void set( int a, int b)
     { i = a;
       j = b; }
void show( )
```

```
 {
   cout << i << "  " << j << "\n"; } };
class derived1 : public base
 {
   private:
    int k;
   public:
Void setk (  )
 { k = i * j; }
void showk( )
 { cout << k << "\n";} };


class derived2 : public derived1
 {
  private:
   int m;
  public:
   void setm ( )
   { m = i – j; }
   void showm( )
    {cout << m << "\n";} };
int main( )
 {
  derived1 ob1;
  derived2 ob2;
  ob1.set(2, 3);
  ob1.show( );


ob1.setk( );
ob1.showk( );
ob2.set(3,4)
ob2.show ( );
ob2.setk( );
ob2.setm( );
ob2.showk( );
ob2.showm( );
return 0;
}
```

- If however, base were inherited as private, then all members of base would become private members of derived1, which means they would not be accessible by derived2

This is illustrated by the following program

```
#include<iostream.h>
    class base
 {
  protected:
  int i, j;
  public:
   void set( int a, int b)
    { i = a;
      j = b; }
   void show( )
   {cout << i << "  " << j << "\n"; }
  };


class derived1 : private base
 {
   private:
    int k;
```

```
    public:
void setk ( )
 { k = i * j; }
void showk( )
  { cout << k << "\n";} };
class derived2 : private derived1
 {
   private:
    int m;
   public:
    void setm ( )
     { m = i – j; } // i and j private in
derived1, will not compile


void showm( )
    {cout << m << "\n";}
  };
int main( )
 {
   derived ob1;
   derived ob2;
   ob1.set (1, 2); // error, can't use
set( )
   ob1.show( ); //error, can't use
show( )
   ob2.set(3, 4); // error, can't use set(
)
   ob2.show( ); // error, can't use
show( )
   return 0;
  }
```

Protected Base-Class Inheritance:

- It is possible to inherit a base class as protected. When this is done, all public and protected members of the base class become protected members of the derived class.

```
#include<iostream.h>
class base
{
  protected:
   int i, j;
  public:
    void setij( int a, int b)
     { i = a;
       j = b; }

void showij( )
  {
    cout << i << "  " << j << "\n"; } };
class derived : protected base
  {
    private:
     int k;
    public:
    void setk( )
    {
      setij(10,12);
      k = i * j;  }
void showall( )
  { cout << k << "\n";
```

```
    showij( ); }
};

int main( )
 {
  derived ob;
  ob.setij( ); // illegal, setij( ) is
protected member of derived
  ob.setk( ); // OK, public member of
derived
  ob.showall( ); // ok, public member
of derived
  ob.showij( ); illegal, showij( ) is
protected member of derived
   return 0;
 }
```

Constructors, Destructors & Inheritance:

- It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions.
- Constructors are the only exception to the inheritance rule in that they are not inherited by the derived class. The same holds true for destructors.
- The language therefore, has to provide for the explicit invocation of a base class constructor at the moment of creation of a base class object.
- It is important to understand the order in which the constructors and destructors are invoked when an object of the derived class comes into existence, and when it ceases to exist.

To begin, consider this short program:

```
#include<iostream.h>
class base
{
  public:
   base( )
  { cout << " Constructing base\n";  }

~base ( )
  { cout << "Destructing base\n"; } };

int main( )
class derived : public base
 {
   public:
    derived( )
     { cout << "Constructing
derived\n"; }
     ~derived( )
      { cout << "Destructing
derived\n"; }};
```

```
{
  derived ob;  // do nothing but construct and destruct ob
   return 0;  }
```

- Therefore, constructor functions are executed in their order of derivation.
- Destructor functions are executed in reverse order of derivation.

Passing Parameters to Base Class Ctors:

- Calling constructor explicitly

  -when base class contains parameterised constructor
- use an expanded form of the derived class' constructor declaration that passes along arguments to one or more base class constructors. The general form of this expanded derived class constructor is as follows:
- derived-constructor (arg-list) : base-constructor (arg-list)

{ body of derived class constructor}

Consider the following program:

```
#include<iostream.h>
class base
 {
   protected:
    int i;
   public:


    base (int x)
     { i = x;
       cout << "Constructing base\n"; }
~base( )
  { cout << "destructing base\n"; } };


class derived : public base
 {
  private:
   int j;
  public:
  // derived uses x; y is passed along to base
   derived (int x, int y) : base(y)
```

```
    {j = x;
     cout << "Constructing derived\n";
}
~derived( )
int main( )
 {
   derived ob(3,4)
   ob.show( ); // displays 4,3
   return 0; }
```

```
  { cout << "destructing derived\n"; }
void show( )
  { cout << i << " " << j << "\n"; } };
```

Multipath inheritance and virtual base classes

- Here the class C can have two instances of the members of class A
- These two instants can be derived through multiple paths.
- Due to the presence of two instances of the members of A in class C ambiguity can be raised and also unpredictable errors can be generated.
- To overcome this problem you need to maintain only one copy of members of class A in class C. For this while using the class A as Base class we can declare it has virtual base class.

Virtual Base Class:

- Properties of the virtual base class are made virtual for inheritance in the subsequent derived classes.

```
class B1:public virtual A
{
```

```
Body
}
```

```
class B2:public virtual A                    Body
{                                            }

class C:public B1, public B2
{
Body
}
```

- only one copy of the properties of class A will be inherited by class C via class B1 and class B2.

Inheritance Vs. Composition

- Mechanism to make use of existing classes to define new and more complex classes.
- The internal data (the state) of one class contains an object of another class.
- The host class has access to public members of contained class.
- It can be an alternative to multiple inheritance.
- Provides an additional layer of data protection.

Virtual Functions

- A virtual function is a member function that is declared within a base class, and is redefined by a derived class.
- To create a virtual function, precede the function's declaration in the base class with the keyword virtual.
- When a class containing a virtual function is inherited, the derived class redefines or overrides the virtual function to fit its own needs.
- Virtual functions implement the "single method; multiple implementations" paradigm intrinsic to polymorphism.
- The virtual function within the base class defines the form of the interface to the function.

- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.
- When a base class pointer points to a derived class object that contains a virtual function, C++ determines which version of that function to call based upon the type of the object pointed to by the pointer.
- And this determination is made at runtime.
- Thus, when different derived class objects are pointed to by the base class pointer at different points of time, different versions of the virtual function are executed.

```
#include<iostream.h>
class base
 {
   public:
    virtual void vfunc( )
      { cout << "this is base's vfunc(
)\n"; } };
class derived1: public base
 {
   public:
    void vfunc( )
     {  cout << "this is derived1's vfunc(
)\n"; } };
class derived2 : public base
 {
   public:
    void vfunc( )
      { cout << "this is derived2's vfunc(
)\n"; } };
int main( )
 { base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base's
vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1's
vfunc( )
   p = &d2;
   p->vfunc( ); // access derived2's
vfunc( )
   return 0;  }
```

- The key point here is that the kind of object to which p points to determines which version of vfunc( ) is executed.
- Further, this determination is made at runtime, and this process forms the basis of runtime polymorphism.
- Although you can call a virtual function in the normal manner by using a object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that runtime polymorphism is achieved.
- A function declared as virtual in the base class, and redefined in the derived classes is the implementation of overridden functions.
- The prototype for a redefined or overridden virtual function must exactly match the prototype specified in the base class.

- Prototype encompasses not only signature, but also the return data type of a function.

Calling a Virtual Function Through a Base Class Reference

- The polymorphic nature of a virtual function is also available when called through a base class reference.
- Thus a base class reference can be used to refer to an object of the base class, or any object derived from that base.
- When a virtual function is called through a base class reference, the version of the function executed is determined by the object being referred to at the time of call.
- The most common situation in which a virtual function is invoked through a base class reference is when the reference is defined as a function parameter. Consider the following program:

```
#include<iostream.h>
class base
 {
   public:
    virtual void vfunc( )
      {  cout << "this is base's vfunc(
)\n"; } };


class derived1: public base
 {
   public:
    void vfunc( )
    { cout << "this is derived1's vfunc( )\n"; } };

class derived2 : public base
 {
   public:
    void vfunc( )
    { cout << "this is derived2's vfunc( )\n"; } };


void f(base &r)
 {  r.vfunc( ); }


int main( )
 { base b;
   derived d1;
   derived d2;
   f(b); // pass a base object to f( )
   f(d1); // pass a derived object to f( )
   f(d2); // pass a derived object to f( )
   return 0; }
```

The Virtual Attribute is Inherited

- When a virtual function is inherited, its virtual nature is also inherited.

- This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

- No matter how many times a virtual function is inherited, it remains virtual.

```
#include<iostream.h>
class base
 {
   public:
    virtual void vfunc( )
      { cout << "this is base's vfunc(
)\n"; } };
class derived1: public base
 {
   public:
    void vfunc( )
      { cout << "this is derived1's vfunc(
)\n"; } };


class derived2 : public derived1
 {
   public:
    void vfunc( )
      { cout << "this is derived2's vfunc(
)\n"; } };
int main( )
 { base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base's
vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1's
vfunc( )
   p = &d2;
   p->vfunc( ); // access derived2's
vfunc( )
   return 0;  }
```

Virtual Functions are Hierarchical

- When a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden.

- When a derived class fails to override a virtual function, then, when an object of the derived class accesses that function, the function defined by the base class is used.

Consider this program in which class derived2 does not override vfunc( )

```
#include<iostream.h>
class base
 {
   public:
    virtual void vfunc( )
      { cout << "this is base's vfunc( )\n"; } };


class derived1: public base
 {
   public:
```

```
    void vfunc( )
     { cout << "this is derived1's vfunc( )\n"; } };


//derived2 inherits virtual function from derived1
class derived2 : public derived1
 { // vfunc( ) not overridden by derived2; base's is used  };
int main( )
 { base *p, b;
   derived1 d1;
   derived2 d2;
   p = &b;
   p->vfunc( ); //access to base's vfunc( )
   p = &d1;
   p->vfunc( ); // access derived1's vfunc( )
   p = &d2;
   p->vfunc( ); // access base's vfunc( ) since derived2 does not override vfunc( )
   return 0; }
```

Pure Virtual Functions

- When a virtual function is not redefined by the derived class, the version defined in the base class will be used. However, in many situations, there cannot be any meaningful definition of a virtual function within a base class.
- For example, a base class may not be able to define an object sufficiently to allow a base class virtual function to be created.
- Further, in some situations, you will want to ensure that all derived classes compulsorily override a virtual function.

- To handle these two situations, C++ supports the Pure Virtual Function. A pure virtual function is a virtual function that has no definition in the base class.
- To declare a pure virtual function, use this general form:
- virtual type func_name (parameter_list) = 0;
- When a virtual function is declared pure, any derived class must provide its own definition of the virtual function. If the derived class fails to override the pure virtual function, a compile-time error will ensue.

The base class number contains an integer called val, the function setval( ) and the pure virtual function show( )
The derived class hextype, dectype, and octtype inherit number, and redefine show( ) so that it outputs the value of val in each number base (i.e., hexadecimal, decimal or octal).

```
#include<iostream.h>
class number
 {
   protected:
    int val;
   public:
    void setval( int i)
     { val = i;}
    virtual void show( ) = 0; };


class hextype : number
 {
   public:
    void show( )
     { cout << hex << val << "\n"; } };
class dectype : number
 { public:
    void show( )
     { cout << val << "\n"; } };
class octtype : number
 { public:
    void show( )
     { cout << oct << val << "\n"; } }


int main( )
 {
  number *p;
  dectype d;
  hextype h;
  octtype o;
  p = &d;
  d.setval(20);
  p->show( ); // displays 20 – decimal
  p = &h;
  h.setval(20);
  p->show( ); // displays 14 –
hexadecimal
  p = &o;
 o.setval(20);
 p->show( ); // displays 24 –octal
 return 0; }
```

Abstract Classes

- A class that contains at least one pure virtual function is said to be abstract. An abstract class cannot be instantiated since it has one or more pure virtual functions.
- Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class.
- This allows abstract classes to support runtime polymorphism, which relies upon base class pointers or references to select the proper virtual function.

Concreate Classes

- It is the class with complited definition which can be used to create objects directly.

Using Virtual Functions

- One of the central aspects of Object-Oriented Programming is the principle of "one interface, multiple methods".
- This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations.
- In concrete C++ terms, the base class can be used to define the nature of the interface to a general class.
- Each derived class then implements the specific operations as they relate to the type of data used by the derived type.
- One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use abstract base classes, pure virtual functions, and base class references or pointers.
- Using these features, you can define a class hierarchy that moves from general to the specific (base to derived).
- define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function.
- Therefore, in the base class, you create and define everything you can that relates to the general class. The derived class fills in the specific details.
- Consider the following example. A class hierarchy is created that performs conversions from one system of units to another (for example, litres to gallons).

```
#include<iostream.h>
class convert
 {
  protected:
   double val1;
   double val2;
  public:
   convert (double i)
   { val1 = i; }
   double getconv( )
    { return val2; }
   double getinit( )
     { return val1; }
   virtual void compute( ) = 0;};

//litres to gallons
class l_to_g : public convert
 { l_to_g( double i) : convert( i ) {    }
   void compute( )
    { val2 = val1 / 3.7854;  } };
// fahrenheit to celsius
class f_to_c : public convert
 {
   public:
    f_to_c( double i) : convert( i)  {     }
    void compute
      { val2 = (val1 -32) / 1.8; } };
```

```
int main( )
 {
   convert *p // pointer to abstract
base class
  l_to_g lgob(4);
  f_to_c fcob(70);
  // use virtual function mechanism to
convert
  p = &lgob;
  cout << p->getinit( ) << litres is “;
  p->convert( );
cout << p->getconv( ) << “gallons\n”;
p = &fcob;
cout << p->getinit( ) << in fahrenheit
is “;
p->compute( );
cout << p->getconv( ) << “celsius\n”;
return 0;
}
```

Early Vs. Late Binding

- Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is know at compile time.
- In other words, early binding means that an object and a function call are bound during compilation.
- Examples of early binding includes normal function calls (including standard library functions), overloaded function calls, and overloaded operators.
- The advantage of early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

- The opposite of early binding is late binding. As it relates to C++, late binding refers to function calls that are not resolved until runtime.
- Virtual functions are used to achieve late binding.
- When access is via a base class pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer.
- Because in most cases, this cannot be determined at compile time, the object and the function are not linked until runtime.
- The main advantage to late binding is flexibility.
- Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes, without having to create contingency or conditional code.

- Because a function is not resolved until runtime, late binding can make for somewhat slower execution times.

Virtual Destructor

- When using dynamic binding all the instances of a class may not be properly disposed off. As delete to a pointer to a base class will call the destructor of the base class only.
- But if destructor is declared virtual it will call the inherited class destructor as well, thus properly disposing the class instances.

Friend Functions

- An external function cannot access the non-public members of an object.
- The only way to access the data within objects is through messages, which involve a call to a public member function that in turn accesses the non-public data.
- These function calls to access the non-public data slows down the execution of code owing to the overhead of additional function calls.
- To enable the programmer to avoid the overhead of using function calls, C++ provides the friend facility.
- Syntax:

```
friend return type Function Name()
{ statements}
```

- Using friend function we can access all the members of a class from outside the class

Example:

```
class mks_distance
  {private:
     int meter;
     int cm;
public:
  mks_distance( int, int);
  void disp_distance( void);
 friend int compare( mks_distance &, mks_distance &); };
mks_distance:: mks_distance( const int mt = 0, const int cmt = 0)
 {meter = mt;
  cm = cmt; }

void mks_distance::disp_distance( void)
 {  cout << "the distance = " << meter << '.' << cm  << "metres\n"; }
int compare( mks_distance &m1, mks_distance &m2)
```

```
{
  // Accessing private members of objects of mks_distance
 int m_diff = m1.meter – m2.meter;
 int cm-diff = m1.cm – m2.cm;
 return( m_diff * 100 + cm_diff);
}
```

By declaring function compare( ) as a friend function of the class mks_distance, the function can access the non-public (private and protected) members of all instances of the class.
The declaration of the friend function must be declared within the class of which it is a friend.

In the above example , compare function is accesing private data of a class mks_distance.
Note: friend function is not a member of a class .so no need to use scope resolution operator while defining compare function

- Delaring a function as a friend within a class does not make it a member of the class.
- Since a friend function is not a member of the class, it can be declared in the public, private, or protected section of the class without affecting its visibility.
- A function that is a friend of a particular class could be either a global function, or a member function of another class.
- Friend function can access the non-public members of more than one class.

- A friend function can act as a bridge between unrelated classes
- By declaring a function as a friend in more than one class, it can be made to access their non-public members.
- Bridging relationship between two classes is possible by using inheritence but if these two classes are unrelated then friend function will be used.
- For instance, consider the two classes fps_distance and mks_distance.
- You may want to copy the distance from a fps_distance object to a mks_distance object.
- For this, the copy function needs to access the private members of the instances of both the classes.
- The function can hence be declared a friend of both classes.
- This friend function can receive an object of both fps_distance class and mks_distance class as parameters, and convert feet and inches to meters and centimeters.

Example:

- **class mks_distance; // forward declaration**

```
class fps_distance
 {private:
    unsigned int feet;
    float inch;
 public:
   fps_distance(unsigned int, float); // constructor
  void disp_distance (void);
/* the friend function feet_to_meter( ) can access the private members of this class */


friend void feet_to_meter( fps_distance &, mks_distance & ); };
fps_distance :: fps_distance( unsigned int ft = 0, float in = 0)
 {feet = ft;
   inch = in;}
void fps :: disp_distance( void)
 {cout << "The distance = " << feet << "'" << inch << "," << '\n'; }
class mks_distance
  {private:
     int meter;
     int cm;
public:
  mks_distance( int, int);
  void disp_distance( void);
};


mks_distance:: mks_distance( const int mt = 0, const int cmt = 0)
 {meter = mt;
  cm = cmt; }

void mks_distance::disp_distance( void)
 {cout << "the distance = " << meter << '.' << cm  << "metres\n";}
friend void feet_to_meter(fps_distance &, mks_distance &);
};
/*Function to convert feet and inches to meter and centimeters. The function receives
an object of the fps_distance and the mks_distance class as parameters. */
void feet_to_meter( fps_distance &fps, mks_distance &mks)
 {
  int temp = ((( fps.feet  12 + fps.inch ) / 39 * 100);
  mks meter = temp / 100;
  mks.meter = temp % 100;
 }
```

- Note that a class cannot be referred to until it has been declared.

- In the above example, when the function feet_to_meter ( ) is declared as a friend in class fps_distance, class mks_distance is referred to in the parameter list.
- The definition of class mks_distance comes later.
- Even if we shift the definition of class mks_distance before that of class fps_distance, the same problem would persist with fps_distance.
- In such cases, the class must be declared (not defined) before a reference is made to it.
- This is known as a forward declaration. Once the forward declaration is made, the class can be referred to. The body of the class can be defined later in the program.

Forward Declaration:

- Declaring a class, without having it's implementation immediately followed by.
- Forward declaration tells the compiler that class definition comes later in the program.
- The forward declaration is made globally (outside any class declaration) before a reference is made to the class.

Friend function can be a member function of another class.

Example

```
Class A
{
int x;
Public:
void getx()
{
cin>>x;
}
 void disx()
{
 cout<<x;
}
 friend  void B:: sety();//declaring sety
function of class B as friend of class A;
};
 class B
{
 int y;
 public :
 void sety()
{
  y=20;
  A obj;
     obj.x=200;//assigning value to a
private data of a class A directly
 cout<<obj.x; // displaying the value of
private member of class a
}
 void disy()
{
 cout<<y;
}
```

```
};
 main()
{

}
```

```
 B obj;
Obj.sety();
```

Note: only **sety** function can access private data of class A but not **disy** function.

Friend class:

- Class can be declared as a friend of another class.
- Friend class can access all the members (including private and protected) of another class directly using object.
- Friend declarations can go in either the public, private, or protected section of a class.it doesn't matter where they appear.
- Access is granted for a class using the class **keyword** as **friend class aclass**;
- When we declared a class as friend of another class , then all the member function of a friend class can access all the members of another class [class which is having a friend declaration] including private and public members.

Example:

```
class A
{
private:
int X;
void setx(int p)
{
X=p;
}
void disx()
{
cout<<X;
}
friend class B;
};
```

```
class B
{
private:
int Y;
public:
void setxy()
{
A obj;
Obj.setx(10);//accessing private data
of class A
Obj.disx();
}
};
main()
```

```
{                                   Obj.setxy();
B Obj;                              }
```

In the above example class B is a friend of class A ie. all the member function of class B can access a all the members of **class A** directly using a instance of **class A**. **setx** is a private member function of class A, is accessed by setxy function of class B

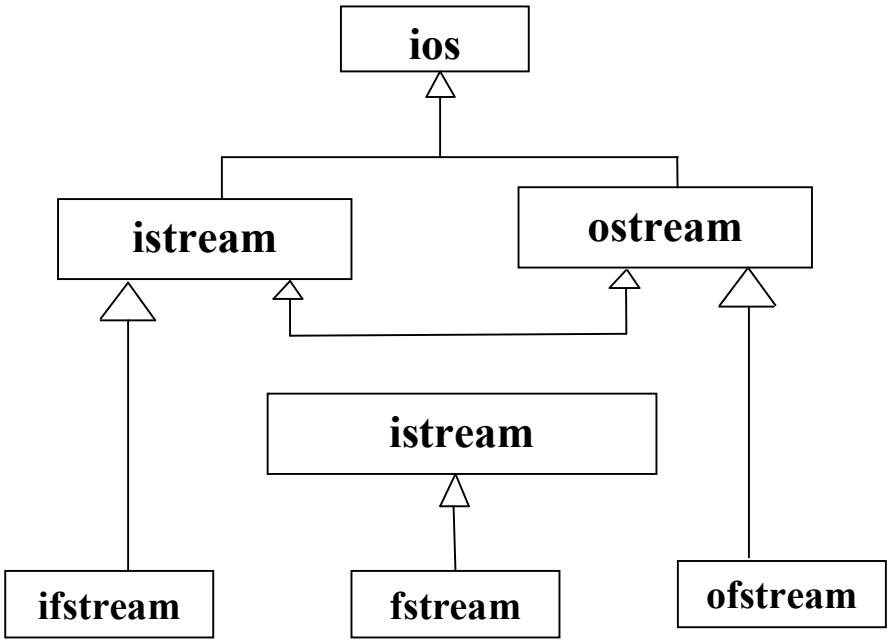
Streams

- A stream is a source or a destination for a collection of characters, or a flow of data. Output streams allow you to store (write) characters, and characters are fetched (read) from input streams.
- The stream classes form a powerful set of classes that can be modified, extended, or expanded to include support for user-defined data types or objects.
- They are fully buffered to reduce disk access.

Advantages of the Stream Classes

- They encapsulate their internal workings from the user. Thus, the programmer need not specify the type of data that is to be input or output; it is automatically determined by the stream class.
- They offer a rich set of error handling facility

The Stream Class Hierarchy



Standard Output

- Stream classes have their member data, functions, and definitions.
- Class ostream contains functions defined for output operations. These operations are called stream insertions. The << operator is called the inserter.
- cout is an object of type ostream defined in the header file iostream, and is attached to the standard output device, i.e., the screen.

Stream Extraction & Standard Input

- The opposite of insertion is extraction, which is the fetching of data from an input stream. Input stream operations are defined in istream class.
- The overloaded operator >> (right-shift bitwise operator) is called the extractor.
- The >> operator is overloaded to accept three data types, namely integers, floating points, and characters.

Integer Extraction

- cin is an object of type istream defined in the header file iostream, and associated with the standard input device, i.e., the keyboard.
- int i;
- cin >> i;
- If the input from the user through the keyboard is <space> <space> 787w33, then i will contain 787.
- The input buffer will now contain w33, and the pointer will be positioned at w. this is because integer extraction bypasses white spaces, and reads input characters until it encounters a character that cannot be part of that type.

Character Extraction

- The character extractor might not work the way one might expect it to. It reads the next character in the stream after skipping white spaces.
- If you enter <space> A, the extractor would ignore the space and return 'A'.
- A direct consequence is that char * (or string) extractors may often yield unpredictable results.
- char str1[20], str2[20];
- cin >> str1;
- cin >> str2;
- If the user enters <space> james <space> gosling <return>, then str1 will contain "james", and str2 will contain "gosling".
- This is because extraction terminates the moment a white space is encountered.

Implicit File Opening and Closing

- In c++ , file can be opened without using additional open function.

  ofstream out("OBJ.TST");

- It means that we create an object called out of the ofstream class, and its constructor is invoked with the value "OBJ.TST".
- When the scope of an object is over, its destructor is automatically called.
- Likewise, when the scope of a stream object in a program gets over, the destructor of the stream class is automatically called, and the file is closed.

Extraction & Insertion of Fundamental Data Types

- File I/O using Fundamental Data Types:
- Consider a situation where you want to write some integers to a file called INT.TST

```
#include<iostream.h>
void main( )
 {
  ofstream out("int.tst");
  out << 25 << ' ' << 4567 << ' ' <<
8910
}
```

```
#include<iostream.h>
void main( )
 {
   ifstream in("int.tst");
   int i, j, k;
   in >> i >> j >> k;
   cout << i << ' ' << j << ' ' << k;
  }
```

Strings in File I/O

```
#include <fstream.h>
void main( )
 { ofstream out("STR.TST");
   out <<"This is a test string"; }
```

```
#include <fstream.h>
void main( )
 {
   ifstream in("STR.TST");
   char str[30];
   in >> str;
   cout << str;
}
```

File I/O Using Objects

Consider the following code:

```
#include<fstream>
#include<string.h>
class vehicle
 {
   public:
    int serialno;
    char model[8];
    double price; };
void main( )
 { ofstream out("obj.tst");
    vehicle car;
   car.serialno = 22;
   strcpy(car.model, "astra");
   car.price = 600000.00;
   out << car.serialno << car.model << car.price; }
```

The following program shows the input of objects from a file:

```
#include<fstream>
class vehicle
 {
   public:
    int serialno;
    char model[8];
    double price; };
void main( )
  { ifstream in("obj.tst");
    vehicle car;
    in >> car.serialno;
    in >> car.model;
    in >> car.price;
    cout << '\n' << car.serialno << '\n' << car.model << '\n' << car.price; }
```

Binary I/O

- The I/O operations discussed so far are text or character-based. That is, all information is stored in the same format as it would be displayed on screen.
- So, 'A' would be written as 'A' on to the file. And the number -12345.678 will be written as -12345.678. This means that you can type the file and see the contents.
- Binary I/O entails that the number -12345.678 will be written as a float representation taking up to 4 bytes of storage.
- When reading text files using the >> operator, certain translations will occur. For example, white space characters are omitted.

- The C++ binary I/O functions such as get( ), getline( ), and read( ) can be used when the programmer wants to read white space characters as well.
- So with binary I/O, the problem of accepting and writing character strings with white space characters will be solved.

Binary I/O – Character I/O

The stream classes have many member functions for file I/O. Two of these are get( ) and put( ).

The get( ) function reads a character from a file, while the put( ) function writes a character on to a file.

```
#include<fstream>
#include<string.h>
void main( )
 {
   ofstream outfile("chrfile.tst");
   char str[ ] = "this is a test";
   for (int i = 0; i < strlen(str); i++)
   outfile.put(str[i];
   outfile.put('\0');
 }
```

Program to read the string back from the file:

```
#include<fstream.h>
void main( void)
  {
    const int MAX = 80;
    ifstream infile("CHRFILE.TST");
   char chstr[ MAX ];
   while ( infile ) // EOF check
    {
      infile.get(chstr[i++];
    }
   cout << chstr;
  }
```

- The get( ) function has the following overloaded form that can read a complete string:

- get(char * str, int len, char delim = '\n');
- Fetches characters from the input stream into the array str.
- Fetching is stopped if len number of characters have been fetched, or the delimiter is encountered, whichever is earlier.
- The terminating character is not extracted.

```
#include<iostream>
void main( )
 {
  char str[20];
  cin.get (str, 20);
  cout << str;
 }
```

This program gets a string (along with white spaces) from the standard input, and displays the string on standard output.
Another function available for string input is:
getline(char * str, int len, char delim = '\n')

The getline( ) function:
is similar to the get( ) function
extracts the terminator also
Here is a program segment that uses the getline( ) function.

```
#include<iostream>
Void main( )
 {
  char str[20];
  cin.getline(str,20);
  cout << str;
 }
```

In this case, str will also contain the '\n' character.

read( ) called with two arguments

- ❖ The address of the buffer into which the file is to be read
- ❖ The size of the buffer

- ➢ write( ) requires similar arguments.
- ➢ Program to get a series of values from the user, and write it to a file.

```
#include<fstream>
void main( )
 {
   ofstream outfile("intfile.tst");
   int number = 0;
   while ( number != 999)
    {
     cout << "please input an integer";
     cin >> number;
     outfile.write((char *) &number, sizeof(number));
    }
```

```
    #include<fstream>
    void main( )
     {
       ifstream intfile("INTFILE.TST");
       int number = 0;
       intfile.read((char *)&number, sizeof(number));
       while(intfile) // EOF check
        {
         cout << number;
         intfile.read((char *)&number, sizeof(number));
        }
      }
```

Explicit File Opening and Closing

- ➢ The open( ) function: There is another way to open files. Each file stream has an open( ) member function.

- ifstream ifile; // create an unopened input stream
- ifile.open("file"); // and associate it to a file
- Each file stream has a close( ) function.
- ofstream ofile;
- ofile.open("obj.tst");
- ofile.close( );

Opening a File for Input and Output

- If we want to open a file in both input as well as output mode, an fstream object is used.
- fstream inherits from iostream, that in turn inherits from both istream and ostream, and hence supports read/write capability.
- The statement fstream file("IOFILE"); will not be enough for the file to be used for both input and output.
- We need to explicitly state this by using the Open Mode bits.

The Open Mode Bits

- Each stream has a series of bits, or flags associated with its operations. These bits are defined in the ios class.
- Some of these bits are used for formatting input and output. Some others are used for error handling. By inheritance, these are available for the fstream classes.
- Some of these bits are associated with the opening of files. Associated with every stream is a series of bits called the Open Mode Bits.
- These open mode bits represent the mode in which the file is opened. Collectively, these bits state the mode in which the file is to be opened.
- fstream file("IOFILE", ios::in | ios::out);
- When the program encounters the aforesaid statement, these two bits are set on for the stream.
- Therefore, the file is available for input as well as output.

Mode	Explanation
app	All data written out is appended to the stream
ate	The file pointer starts at the end of the stream, i.e.,

	causes a seek to the end of the file. I/O operations can still occur anywhere within the file
in	The stream is opened for input.
out	The stream is opened for output
trunc	If the file exists, it is truncated, i.e., all data is erased before writing or reading
nocreate	Operation fails when opening, if the file does not already exist. If the file we are opening does not already exist, the open fails.
noreplace	Operation fails when opening for output, if the file already exists, unless app or ate is set.

Member Functions for Accessing Stream Status Bits

Name	Description
int good( )	returns a non-zero value if the stream is ok, zero otherwise.
int bad( )	returns a non-zero value if the badbit or hardfail bits are set, else returns 0.
int eof( )	returns a on-zero value if the eofbit is set, else returns 0.
int fail( )	returns a non-zero value if the failbit, badbit, or hardfail bit
int rdstate( )	returns the current value of the iostate variable
void clear(int ef=0)	set the error flags equal to ef. by default, ef equals 0, which resets all the error bits

Random Access

- The C++ I/O system manages two integer values associated with a file. One is the get pointer, which specifies where in the file the next input or read operation will occur.
- The other is the put pointer, which specifies where in the file the next output or write operation will occur.
- In other words, these are the current positions for read and write respectively.

- The seekg( ) and the tellg( ) functions allow you to set and examine the get pointer.
- The seekp( ) and the tellp( ) functions allow you to set and examine the put pointer.

- One can, therefore, access the file in a random access mode using these functions.
- All objects of the iostream classes can be manipulated using either the seekg( ), or the seekp( ) member functions.
- The seekg( ) member function takes two arguments:
- file.seekg(10, ios::beg); means position the get pointer 10 bytes relative to the beginning of the file
- The first argument is a long integer, specifying the number of byte positions (also called offset). The second argument is the reference point.

  There are three reference points defined in the ios class:

Name	Description
ios::beg	The beginning of the file
ios::cur	The current position of the file pointer
ios::end	The end of file

- If supplied with only one argument, ios::beg is assumed by default. For example, in the statement: file.seekg(16), ios::beg will be the reference point by default.
- The tellg( ) member function, on the other hand, does not have any arguments. It returns the current byte position of the get pointer relative to the beginning of the file.

```
long position = file.tellg( );
```

- Would result in the variable position taking the value of the current position of the get pointer.

Random Access Scenario

Example : to find out the how many Drug records are there in the file.

```
void main( )
 {
  drug drugobj;
```

```
  fstream ifile("drugs.dat", ios::in);
  ifile.seekg(0, ios::end);
  long endpos = ifile.tellg( );
  cout << "\n the size of the file is " << endpos;
  cout << "\n the size of a Drug record is " << sizeof(Drug);
  int n = endpos/ sizeof(drug)
  cout << "\n" << n << "drug records are in the the file";
    }
```

Random Access – Querying a File

Example: to retrieve a particular record from the file DRUGS.DAT.

```
void main( )
 {
  drug drugobj;
  fstream ifile("drugs.dat", ios::in);
  ifile.seekg(0, ios::end);
  long endpos = ifile.tellg( );
cout << "\n the size of the file is " << endpos;
cout << "\n the size of a Drug record is " << sizeof(Drug);
int n = endpos/ sizeof(drug)
cout << "\n" << n << "drug records are in the the file";
cout << "\n drug record query";
cout << "\n which record do you want to query";
cout << "\n please enter the record number";
int num;
cin >> num;
int seekpos = (num – 1) * sizeof(drug);
ifile.seekg(seekpos);
ifile.read((char *)&drugobj, sizeof(drug);
drugobj.showdata( ); }
```

Templates

Template Functions

- A generic function defines a general set of operations that will be applied to various types of data.

- The type of data that the function will operate upon is passed to it as a parameter.
- Through a generic function, a single general procedure can be applied to a wide range of data.
- Once you have defined a generic function, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function.
- In essence, when you create a generic function, you are creating a function that can automatically overload itself.
- A generic function is created using the keyword template. The normal meaning of the word "template" accurately reflects its use in C++.
- It is used to create a template that describes what a function will do, leaving it to the compiler to fill in the details as needed.
- When the compiler creates a specific version of this function, it is said to have created a specialization.
- This is also called a generated function. The act of generating a function is referred to as instantiating it.
- This is also called as template instantiation.
- By using templates, you can reduce this duplication to a single function template:
- template <class T> T min(T a, T b)
- return (a < b) ? a : b;
- Templates can significantly reduce source code size, and increase code flexibility without reducing type safety.

Template Functions – Implementation

```
include <iostream.h>
template <class T>
void swap(T &a, T &b)
{
        T temp=a;
        a=b;
```

```
        b=temp;
} void main()
    {
        int x=10,y=20;
        swap(x,y);
        cout<<x<<"   "<<y<<endl;
        char *s1="Hello",*s2="Hi";
        swap(s1,s2);
        cout<<s1<<"  "<<s2<<endl;
    }
```

Multiple Generic Types

```
template<class T, class S, class Z>
void fun(T a, S b, Z c)
{
        cout<<a<<endl<<b<<endl<<c;
}
void main()
{
        int i=10;
        float j=3.14;
        char ch='A';
        fun (i, j, ch);
}
```

Explicitly Overloading a Generic Function

- Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called explicit specialization.
- If you overload a generic function, that overloaded function overrides (or hides) the generic function relative to that specific version.

Consider the following example:

```
// Overriding a template function

#include <iostream.h>
template <class X> void swapargs (X &a, X &b)
  {X temp;
    temp = a;
    a = b;
```

```
        b = temp;
        cout << "Inside template swapargs \n"; }
    //This overrides the generic version of swapargs ( ) for ints.
    void swapargs( int &a, int &b)
      {int temp;
        temp = a;
        a = b;
        b = temp;
        cout << "Inside swapargs int specialization\n"; }

Int main( )
  {int i = 10, j = 20;
    double x = 10.1, y = 23.3;
    char a = 'x', b = 'z';
    cout << "Original i, j:  " << i << " " << j << '\n';
    cout << "Original x, y:  " << x << " " << y << '\n';
    cout << "Original a, b:  " << a << " " << b << '\n';
    swapargs( i, j); // calls explicitly overloaded swapargs
    swapargs( x, y); // calls generic swapargs
    swapargs( a, b); // calls generic swapargs
    cout << "Swapped i, j:  " << i << " " << j << '\n';
    cout << "Swapped x, y:  " << x << " " << y << '\n';
    cout << "Swapped a, b:  " << a << " " << b << '\n';
    return 0; }
```

Overloading a Template Function

template specification itself can be overloaded
To do so, simply create another version of the template that differs from any others in its parameter list. For example,

```
// Overload a function template declaration

#include <iostream.h>
//First version of f( ) template
template <class X> void f( X a)
```

```
{ cout << "Inside f( X a)\n"; }

//Second version of f( ) template
Template <class X, class Y> void f( X a, Y b)
 {
  cout << Inside f( X a, Y b) \n";
 }
Int main( )
 {
  f(10); // calls f( X)
  f( 10, 20); // calls f( X, Y)
  return 0;
 }
```

Templates Vs Macros

- Macros in C were also independent of data type but this feature of template is more bug free.
- In macros there is no type checking. The type of return value isn't specified, therefore the compiler cannot check whether we are assigning it to the correct return type or not.

Template Classes

- Like generic functions, you can also define a generic class.
- When you define a generic class, you create a class that defines all the algorithms used by that class.
- However, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

- Generic classes are useful when a class uses logic that can be generalized.

The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.
The general form of a generic class declaration is shown here:

```
template <class Ttype> class class-name
 {..... };
```

Once you have created a generic class, you create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Member functions of a generic class are themselves automatically generic. You need not use the keyword template to explicitly specify them as such.

In the following example, a generic stack class is used to store objects of any type.

```
#include <iostream.h>
const int size = 10;
// create a generic stack class
template <class StackType> class stack
 {private:
    StackType stck[ size];
    int tos; // index to top-of-stack

public:
    stack( )
     { tos = 0; // initialize stack }
void push( StackType ob); // push object on stack
StackType pop(  ); // pop object from stack };
template <class StackType> void stack< StackType>::push( StackType ob)
 {if (tos = =  size)
    {cout << "Stack is full\n";
      return; }
      stck[ tos ] = ob;
      tos + +; }
template <class StackType> StackType stack<StackType>::pop( )
 {if (tos = = 0)
    {cout << "Stack is empty\n";
      return 0; // return null on empty stack }
      tos - -;  return stck[ tos ]; }




int main( )
 {
   //demonstrating character stacks
   stack<char> s1, s2; // create two character stacks
   int i;
   s1.push( 'a');
   s2.push( 'x');
   s1.push( 'b');
   s2.push( 'y');
   s1.push( 'c');
   s2.push( 'z');
   for (int i = 0; i < 3; i+ +) cout << "pop s1:  " << s1.pop( ) << "\n";
   for (int i = 0; i < 3; i+ +) cout << "pop s2:  " << s2.pop( ) << "\n";
```

```
// demonstrate double stacks
 stack<double> ds1, ds2; // create two double stacks
 ds1.push(1.1);
 ds2.push(2.2);
 ds1.push(3.3);
 ds2.push(4.4);
 ds1.push(5.5);
 ds2.push(6.6);
 for (int i = 0; i < 3; i+ +) cout << "pop ds1:  " << ds1.pop( ) << "\n";
 for (int i = 0; i < 3; i+ +) cout << "pop ds2:  " << ds2.pop( ) << "\n";
 return 0;
}
```

Template Class

- Generic class independent of data type.
- Can be instantiated using type-specific versions.
- Can create an entire range of related overloaded classes called template classes.
- Usually used for data storage (container) classes like stacks etc.
- Class templates cannot be nested.
- Template classes can be inherited.

Inheritence and Template class

- Template class can be inherit
- Three ways of template class inheritece
  - Template class(base) to template class(Derived)
  - Template class (base)to normal class(Derived)
  - normal class(Base) to Template class(Derived)

Example :Template class to template class

```
template<class type>
class base
{
type x;
public:
void getx()
{ cin>>x; }
void putx()
```

```
    {cout<<x; }
    };
template<class type>
    class Der:public base<type>
    {
    main()
    {
    Der<int> obj;
    obj.getx();
    obj.putx();
    obj.gety();
    obj.puty();
    }
```

Example :Template class to normal class

```
    template<class type>
    class base
    {
    type x;
    public:
    void getx()
    { cin>>x; }
    void putx()
    {cout<<x; }
    };

    main()
    {
    Der obj;
    obj.getx();
    obj.putx();
    obj.gety();
    obj.puty();
    }
```

```
    type y;
    public:
    void gety()
    {cin>>y;}
    void puty()
    {cout<<y; }
    };
```

Example :Normal class to Template class

```
    class base
    {
    int x;
    public:
    void getx()
    { cin>>x; }
```

```
    void putx()
    {cout<<x; }
    };

    template<class type>
```

```
    class Der:public base<int>
    {
    int y;
    public:
    void gety()
    {cin>>y;}
    void puty()
    {cout<<y; }
    };
```

```
class Der:public base
{
type y;
public:
void gety()
main()
{
Der<int> obj;
obj.getx();
obj.putx();
obj.gety();
obj.puty();
}
{cin>>y;}
void puty()
{cout<<y; }
};
```

Exception Handling

What is an Exception?

- An exception is an error that occurs during runtime.
- Such runtime errors can cause the applications to behave unpredictably, or can cause the application to crash.
- Exception handling allows you to manage runtime errors in an orderly fashion.
- Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

Common Exceptions

- Falling short of memory
- Inability to open files
- Exceeding bounds of an array

Exception Handling

- Provides a structured means by which your program can handle abnormal events.
- Automatically invokes an error handling routine when an error occurs.
- Can handle only synchronous exceptions like “overflow”, “out of range” etc.

Exception Handling in C++

- C++ uses a special language-supported exception handling feature to signal such anomalous events. The exception mechanism uses three new keywords:
  - try
  - catch
  - throw
- The function in which error is expected is kept inside
- the try block. Exception is thrown from the try block.
- This is handled by the catch block immediately following the try block.

Exception Handling - Mechanics

- When an exception is thrown, it is caught by the corresponding catch statement, which processes the exception.
- There can be more than one catch statement associated with a try.
- Which catch statement is used is determined by the type of the exception.

The throw Statement

- The general format of the throw statement is as follows: throw exception
- throw generates the exception specified by exception.
- If the exception is to be caught, then throw must be executed from either within try block itself, or from any function called from within the try block.

Uncaught Exceptions

- If an exception is thrown for which there is no applicable catch statement, an abnormal program termination occurs.
- Throwing an unhandled exception causes the standard library function terminate( ) to be invoked.
- By default, terminate( ) calls abort( ) to stop your program.

Restricting Exceptions

- The throw clause in a function definition specifies the type of exceptions the function is expected to throw.
- Using the throw clause, you can restrict the type of exceptions that a function can throw outside of itself by specifying a comma-separated list of exception types within the throw clause in the function definition.
- Throwing any other exception will cause abnormal program termination.
- Special function unexpected( ) is called when you throw something other than what appears in the exception specification.
- By default, unexpected( ) causes abort( ) to be called, which causes abnormal program termination.
- Presence of throw( ) with empty parenthesis indicates that function is not expected to throw any exception.

Catching Any Exception

- If your function has no exception specification, any type of exception can be thrown. One solution to this problem is to create a handler that catches any type of exception.

- This is done by using the ellipses in the argument list of catch().
  - catch(...) {
  - cout << "an exception was thrown" << endl }

Re-throwing an exception

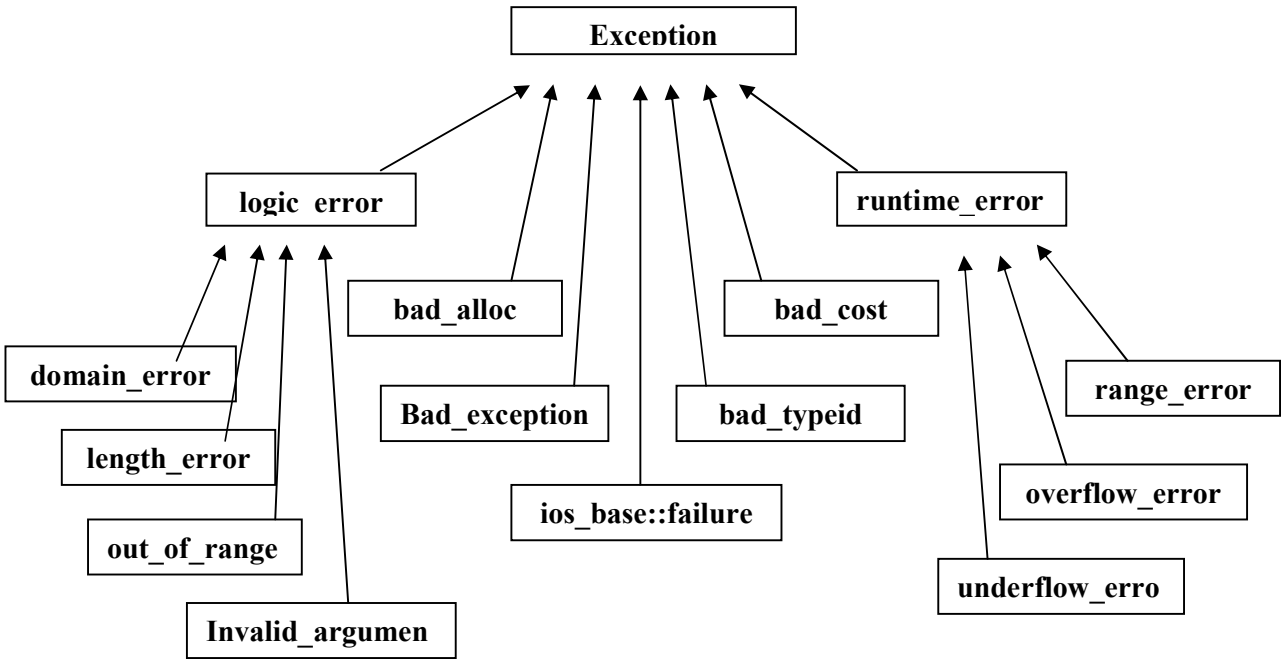
- Sometimes, you'll want to rethrow the exception that you just caught, particularly when you use the ellipses to catch any exception, because there's no information available about the exception.
- This is accomplished by saying throw with no argument:
  - catch(...)
  - {
  - cout << "an exception was thrown" << endl;
  - throw;
  - }
- the throw causes the exception to go to the exception handlers in the next-higher context.

Handling Derived Class Exceptions

Since a catch's argument type applies to objects of base, as well as classes derived from that base, the order of catch handlers is important

```
catch( CBaseException& ex )
{
        // handle a base type of exception
}
catch( CDerivedException& ex )
{
       // this handler would never "run" because its
       // base would catch everything it would…
}
```

Standard Library Exception Hierarchy



Standard Library Exceptions

- Declares many std exception objects, e.g. :

bad_alloc, thrown when new fails
bad_cast, thrown when dynamic_cast fails
bad_typeid, thrown if dynamic_cast is applied to a NULL pointer
bad_exception, thrown when an unexpected (non-specified) exception is thrown

- also in std namespace
    - throw std::range_error;

Programming With Exceptions

- Avoid exceptions :
    - for asynchronous events.
    - for ordinary error conditions.
    - for flow-of-control.
- Use exceptions to:
    - fix the problem and resume the program.
    - do whatever you want in the current context, and throw a different exception to a higher context.
    - terminate the program.

Operator Overloading

- Most fundamental data types have pre-defined operations associated with them.
- To make a user-defined data type as natural as a fundamental data type, the appropriate set of operators must be associated with it.
- C++ allows to Define additional meaning to a existing operator without changing its basic meaning

Operator overloading

- To add two objects of the same class , a member function needs to be invoked.
  - Obj3.addFun(obj1,obj2)
- User have to remember functions names for all operations
- Understanding of operations will be more easier when we use operators
- For example,
- object3.add_distance(object1, object2);
- In this case, the distance in object1 is added to the distance in object2, and the result is stored in object3.
- You might also want to compare two distances. The following function call compares two objects of the fps_distance class. For example,
- object1.compare_distance(object2);
- Hence, the member function compare_distance of object1 is invoked with object2 as an argument. This function compares the data members of object1 with those of object2, and returns the difference between them.
- adding the two objects and storing the result in a third object could be carried out as follows:
  - object3 = object1 + object2;
- The user can understand the operation more easily as compared to the function call because it is closer to a real-life implementation.

Thus by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simple form. Associating operators with ADTs involves overloading the operators.

For example, the '+' operator used to add two integers behaves differently when applied to the ADT fps_distance. Is this not similar to overloading where depending on the number, type and sequence of parameters, the same function behaves differently. Operator overloading refers to giving additional meaning to the normal C++ operators when applied to ADTs.

- Only the predefined set of C++ operators can be overloaded, no new operator can be introduced.
- An operator can be overloaded by defining a function for it.
- The function for the overloaded operator is declared using the **operator** keyword.
- For example, if the = = operator is overloaded in the fps_distance class, the declaration for the operator function is as follows:
- int fps_distance :: operator = =(fps_distance);

- This instruction tells the compiler to call the operator = =( ) function whenever the = = operator is encountered, provided the left hand side operand is of type fps_distance.
- The right hand side operand is passed on to the operator function as an argument.
- For example, the expression:
- X = R = = F translates to
- X = R.operator= =(F);
- The return value of type int of the operator function is stored in X.
- When operators are overloaded, it does not change the predefined sequence of execution of the operators. This is true regardless of the class or implementation.
- The expression syntax of the C++ operators cannot be overloaded. For example, it is not possible to overload the '%' operator as a unary operator.
- Neither is it possible to overload the ++ operator as a binary operator.

In C++ there are **4** types of operators.

Binary operators:	+, -, *, /, %, >, ≥, <, ≤, = = , !, =, &&, ||
Unary operators:	++, --, unary -, !
Assignment:	=, +=, -=, /=, *=, %=
Redirect ional operators:	**>>, <<**

Operator overloading is nothing but redefining the operators ie giving one more meanings to the operators ie adding new functionalities to the operators. We know that variable + variable is valid. However struct + struct, union + union, array + array, matrix + matrix are not valid. However in C++, obj + obj is valid & provided , if we define operator + function, thus all these operators which are defined on variables can be defined on objects also.

Thus user should feel that objects are nothing new. In C++ we have a special member function inside the class is operator (op) where op is any operator.

- It should be defined under public only.
- It can be declared as friend or member function.
- Usually the return type is not void.
- The operands can be obj, obj (or) obj, variable (or) variable, obj.

We can overload all the operators except few operators like.

1. Conditional ? :
2. Space resolution : :
3. Member access *

and 4 size of ( )

All the remaining operators can be overloaded. When we overload operator symantics can be changed but syntax can not be changed.

Ex: - ϴ 1 + ϴ 2 * ϴ 3

Computer evaluates Θ 2 * Θ 3 and then adds the result with Θ 1. We can not ask the computer to change procedure. However in operator + function we can write subtraction logic.

Operator	Friend function	Member function
Binary	2 parameters	1 parameter
Unary	1 parameter ++a → P1	0 parameters ++a→ owner
Simple Assignment	Not possible	1 parameter a = b owner P1
Shortcut Assignment operators	2 parameters	One parameter
Redirectional operators	2 parameters	Not possible

Overloading Unary Operators

- Unary operators act on one operand. As an example, you will now overload the + + operator for the fps_distance to increment the data member feet by 1.

```
#include(iostream>
class fps_distance
 { private:
   int feet;
   float inch;
  public:
   fps_distance (int f, float i)
   {feet = f;
    inch = i; }
operator ++( )
 { feet++;  }
void disp_distance( )
 { cout << "distance = " << feet << " "
<< inch << " " ";  } };

void main ( )
 {
   fps_distance fps(10,10)
   ++fps;
   fps.disp_distance( );
  }
```

- The data member feet of the fps object is incremented using the overloaded operator ++.

- Since the operator ++ is a member function in class fps_distance, it can access all members of the class.
- Unary operators work with one operand. Hence, the operator function does not need any parameter. Note that the + + operator has been used as a prefix operator.
- When used with fundamental data types, the prefix application of the operator causes the variable to be incremented first before it is used in an expression.
- The postfix application of the operator causes the value to be incremented after the value is used in an expression.
- The ++ operator used in the fps_distance class earlier cannot be used as a postfix operator.
- An operator function with no arguments is invoked by the compiler for the prefix application of the operator.
- The compiler invokes the operator function with an int argument for the postfix application of the operator.
- int operator ++(int)

Overloading Overloaded Operators:

- A class can have more than one function for the same operator, i.e., the same operator can be overloaded.
- For example, you might want to add a constant distance to the object of the fps_distance class. The constant distance can be in inches.
- fps2 = fps1 + 20; //adding a constant
- The program for overloading the '+' operator given earlier cannot handle this case. We have to define another + operator function with a different kind of parameter.

The following are the two operator functions:

```
fps_distance fps_distance ::  operator + (fps_distance &fps2)
 {int f = feet + fps2.feet;
  float i = inch + fps2.inch;
  if ( i >= 12)
   { i - = 12;
     f++; }
     return fps_distance( f, i); //creating an unnamed object and returning it }
fps_distance fps_distance :: operator + (int inches)
{int f = feet + (inches / 12);
  int i = inch + (inches % 12);
  if ( i >= 12.0)
  {i- = 12;
   f++; }
 return fps_distance( f, i); }
```