UNIT-I

pop vs oops, introduction of c++: input & output statement, compiling and running c++ program. pop c++ program, oops structure of c++ program, structure vs union vs class, scope resolution operator, inline function, class and object, access members, explicit and implicit member function declaration, access specifier, static data member and static member function, friend function & friend class, passing object to function, returning object by function and copying one object to another object.

UNIT-2

arrays of objects, pointer to object, type checking c++ pointers, *this* pointer, pointer to class member (.* and→*), references: independent reference, references as function parameters, reference as return value, passing reference to object, restrictions to references. dynamic allocation operator *new* and *delete:* initializing allocated memory, allocating arrays, allocating object.

constructor and destructor: default constructor, parameterize constructor, overloaded constructor (multiple constructors in a class), copy constructor, constructor with default argument. destructor

UNIT-3

function overloading, operator overloading: restricted operator, operator overloading using member function of class, unary operator overloading: ++, --. binary operator overloading: +,-,*,/, %, +=,-=,*=,/=, %=. operator overloading using friend function of class, overloading new and delete, overloading some special operators: [ ], ( ), ->, comma operator, <<, >>.

UNIT-4

inheritance: access protected member of base class, single, multilevel, multiple, hierarchical and hybrid inheritance (virtual base class). constructor and destructor used in inheritance, passing parameters to the base class constructor, granting access. run time polymorphism: dynamic binding, pointer to derived class, virtual function, pure virtual function, abstract base class, early vs late binding.

UNIT-5

the c++ input / output system basics: c++ streams, c++ predefined streams, unformatted i/o operations: overloaded operators >> and <<, get() and put() function, getline() and write() function. formatted console i/o operations, *ios* formatted functions:.*width, precision, fill, setf, ,unsetf.* manipulators: *setw, setprecision, setfill, endl,* creating your own manipulator functions

MISCELLANEOUS

Templates: class template, class template for multiple parameter types, function template.

File handling i/o operation in c++: opening and closing file, using << and >>, put() and get(),write() and read(), getline(), eof(), ignore(), peek(), putback(), flush(), seekg(), seekp(), tellg() and tellp().

Exception handling:

UNIT-I

POP (Procedure Oriented Programming)

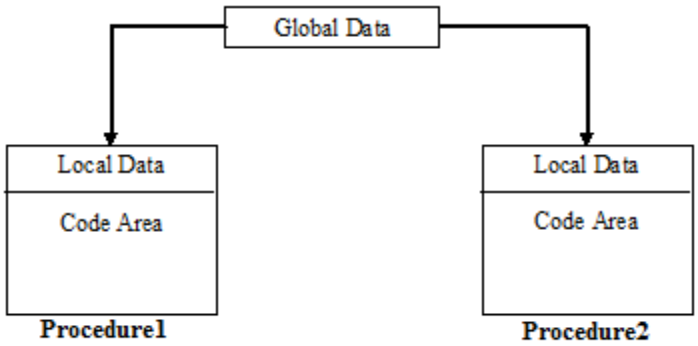
COBOL, FORTRAN and C are such programming language which commonly called POP based languages.

Basic approach / concept of POP:

In this approach for any requirement, sequence of reading, calculating and printing instructions are grouped in a unit called procedure (function or sub routine or method). For each requirement, a new procedure is created. In this way, a single program is oriented by a number of procedures. Such approach for designing program is called POP (Procedure Oriented Programming) and programming language that follow this approach called POP language like C.

In POP, Data can be used in two ways.

1. **Global Data:** Sharable for all the procedures that declared outside of all procedures.
2. **Local Data:** Accessible to that procedure where it is declared. For other procedure local data are unknown.



Following control structures can be used in each procedure to control instructions.

1. **Conditional Structure:** To skip running a set of instructions.
2. **Looping Structure:** To run a set of instructions more than one times.

Drawbacks of POP:

1. A procedure can also use global data that does not need and affect result of program.
2. For real world application, POP becomes fail. Because large scale program cannot be developed properly.

OOPs (Object Oriented Programming)

OOPs is a new approach to design a real world program and secure global data. C++, Java and C# are OOPs based languages.

Basic approach / concept of OOPs:

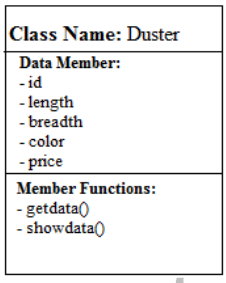
Everything is considered in the form of object and they are identified on the basis of their properties. Objects of any class can be communicated to each other called message passing in OOPs.

Characteristics of OOPs:

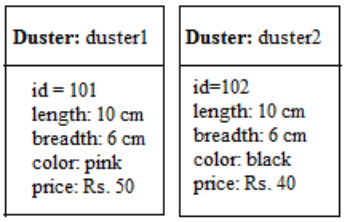
1. Class
2. Objects
3. Data encapsulation
4. Data abstractions
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message passing

Any programming language that support all above properties called OOPs based language like C++, Java and C#.

1. **Class:** Description of any object on the basis of data and functions called class. Or User defined data type of object called class.

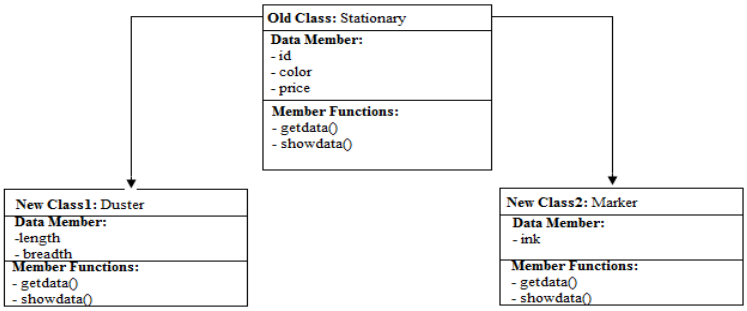


2. **Object:** One Instance of the class is called object. Or variable of the class is called object.



duster1 & duster2 are two object of Duster class data type

1. **Data encapsulation:** When all data of any object, are defined in a single unit called data encapsulation. It is done using making class. Ex: id, length, breadth, color and price are defined in a single unit for all objects (duster1, duster2).
2. **Data abstraction:** When it is possible to know about any object using a shorter name called data abstraction. It is done by defining the class name. Ex: Duster is a name of class that defines about duster1 and duster2.
3. **Inheritance:** When a new class (derived class) is enabled to access properties of old class (base class) called inheritance.



In this example stationary is a common class for Duster and Marker class. Both classes do not to redefine id, color and price data types. They are linked using inheritance.

4. **Polymorphism:** If two or more functions having same name then it is called polymorphism.
5. **Dynamic binding:** When it is possible to link a function to its function call at run time then it is called dynamic binding.
6. **Message passing:** Objects of any class can pass data to each other called message passing.

Advantages of OOPs:

1. **Data Hiding:** Private data of the class can't be accessed from outside.
2. **Security of data:** Data of the class can be accessed as a private, protected and public. Private for same class, protected for derived too and public for anywhere.
3. **Reusability:** Using inheritance, members of old class can be used into new class.
4. **Easy development:** Large size of real world application can be easily developed.
5. **Easy to debug:** All classes can be debugged separately.

Uses / Application of OOPs:

Now a days OOP is much popular because we can use their concept to solve in any type of problems some of them are much like:

1. Real time system
2. Simulation and modeling
3. Object oriented database
4. Hypertext and hypermedia
5. AI and expert system
6. Decision support and office automation
7. CAM /CAD System

Introduction of C++:

- C++ is an object-oriented programming language.
- It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's.
- C++ is an extension of C therefore initially it was called 'C with classes'. In1983, the name was changed to C++ due to idea comes from the C increment operator ++.
- C+ + is a superset of C. Almost all c programs are also C++ programs.
- C++ a truly object-oriented language because it provides the facilities of making class, objects, inheritance and polymorphism.

Input (cin) & Output(cout) statement in C++:

In c++, iostream.h header file is used to support all input output streams classes in our program.

Output statement: using "*cout*<<" stream object we can display data on monitor. (just like printf in c)

Input statement: using "*cin* >>" stream object we can assign keyboard value to the variable of the program.(just like scanf in c)

Example: *u1p1.cpp*

```
#include<iostream.h>
void main()
{
     int a;
     cout<<"\n Input Integer:";
     cin>>a;
     cout<<"\nInteger value is "<<a;
}
```

Output:

```
Input Integer: 11
Integer value is 11
```

If number of data are more than each data must be separated by << or >>.

```
cout << "a =" << a << "\n";
```

will display on monitor as- a=11

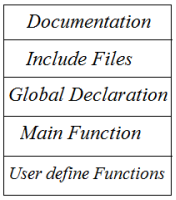
```
cin>>a>>b;
```

This statement reads two values from keyboard and store to first in variable a then to variable b.

Compiling and running C++ program:

1. Install Turbo C++ software in DOS based operating system.
2. Open TC.exe file from location c:\TC3 or TC\BIN\TC.exe.
3. Type C++ program and save.
4. Compile using Alt+F9.
5. If program is error free then run using Ctrl+F9.
6. Check output screen using Alt+F5.

POP (Procedure Oriented Programming) C++ Program:



Program: *u1p2.cpp*

```
// Documentation
      /*
      Program to demonstrate POP organization and Top to
      Down approach, Created by: www.LRsir.net
      */
//include files
      #include<iostream.h>
//Global declaration
      int a,b;
      void getdata(int,int);
      void showdata();
      int sum();
//Main function
      void main()
      {
            getdata(10,20);
            showdata();
            int c=sum();
            cout<<"\nSum="<<c;
      }
//User defined functions
      void getdata(int p1,int p2)
      {
            a=p1;
            b=p2;
      }
      void showdata()
      {
            cout<<"\n"<<a<<"\t"<<b;
      }
      int sum()
      {
            return a+b;
      }
```

Output:

```
a=10      b=20
Sum=30
```

Remark:

/*……*/ is multiline comment

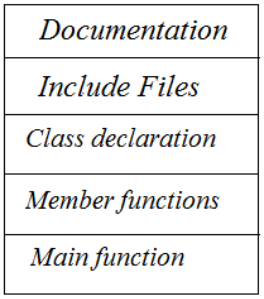
//….. is single line comment

Variables can be declared anywhere before use in C++.

In C++, program can be designed in POP or OOPs style.

OOPs Structure of C++ Program: (Bottom-up Approach)

The structure of C++ program can have four sections.



1. **Documentation:** In this section we write comment about program that ignore by compiler. In comments we can write program name, uses, developer name, date, any help etc. Comments are written in /*--------*/(multiline comment) and //------ (single line comment).

```
// Documentation

/* Program to demonstrate POP organization and Top to
Down approach, Created by: www.LRsir.net */
```

2. **Include file section:** In this section all needful files are included in the program that contains declaration of class and functions used in the program.

   Ex: `#include<iostream.h>`

   Contain declaration of cout and cin stream object.

3. **Class declaration section:** In this section we create classes. Every class contains data and function declaration. In class we can also define member functions.

   Ex:

```
class A
{
 int a,b;
 public:
   void getdata(int);
   void showdata();
   int sum();
};
```

4. **Member function definition:** In this section, all explicit member functions of the class are defined outside the class using scope resolution operator (::).

   Ex:

```
void A::getdata(int p1,int p2)
{
   a=p1;
   b=p2;
}
void A::showdata()
{
        cout<<"\n"<<a<<"\t"<<b;
}
```

```
int A::sum()
{
   return a+b;
}
```

5. **Main function section**: In this section we can declare object of class and call public member functions of the class.

   Ex:

```
void main()
{
   A ob;
   ob.getdata(10);
   ob.showdata();
   int c=ob.sum();
   cout<<"\nSum="<<c;
}
```

In this way a OOPs C++ program is designed in four sections.

OOPs(Object Oriented Programmings) C++ Program:

On the basis of above structure, an OOP's C++ will design in following order.

Program:*u1p3.cpp*

```
// Documentation
   /*  Program to demonstrate OOPs organization and
   Bottom-Up approach, Created by: www.LRsir.net
   */
//include files
   #include<iostream.h>
//class declaration
   class A
   {
      int a,b;
      public:
           void getdata(int);
           void showdata();
           int sum();
   };
//Member function definition
   void A::getdata(int p1,int p2)
   {
      a=p1;
      b=p2;
   }
   void A::showdata()
   {
      cout<<"\n"<<a<<"\t"<<b;
   }
   int A::sum()
   {
      return a+b;
   }
```

```
//Main function
void main()
{
   A ob;
   ob.getdata(10);
   ob.showdata();
   int c=ob.sum();
   cout<<"\nSum="<<c;
}
```

Output:

```
a=10      b=20
Sum=30
```

Structure vs Union vs Class

Structure	Union	Class
Structure body supports only data members.	Union body supports only data members.	Class body supports data members as well as member functions.
All data members are created when variable of structure is created.	Only one largest size of data member is created when variable of union is created.	All data members are created when *variable of class*(object) is created.
By default, structure members are public.	By default, union members are public.	By default, class members are private.
struct keyword is used to define structure.	*union* keyword is used to define union.	*class* keyword is used to define class.

Ex:
Program:*u1p4.cpp*
`struct A`
`{`
`int a;`
`int b;`
`};`
`void main()`
`{`
`A oa;`
`oa.a=10;`
`oa.b=20;`
`out<<oa.a<<"\n"`
`cout<<oa.b;`
`}`
Output:
`10`
`20`
	Ex:
	Program:*u1p5.cpp*
	`union A`
	`{`
	`int a;`
	`int b;`
	`};`
	`void main()`
	`{`
	`A oa;`
	`oa.a=10;`
	`oa.b=20;`
	`cout<<oa.a<<"\n"`
	`cout<<oa.b;`
	`}`
	Output:
	`20`
	`20`
		Ex: ***Program:****u1p6.cpp*
		`class A`
		`{`
		`int a;`
		`int b;`
		`public:`
		`void get(int p1,int p2)`
		`{`
		`a=p1;`
		`b=p2;`
		`}`
		`void show()`
		`{`
		`cout<<oa.a<<"\n"`
		`cout<<oa.b;`
		`}`
		`};`
		`void main()`
		`{`
		`A oa;`
		`oa.get(10,20);`
		`oa.show();`
		`}`
		Output:
		`10`
		`20`

Scope Resolution operator (::)

If a function that uses same name of global variable and local variable then by local variable will be accessed. But if we want to use global variable then use scope resolution operator (::) with variable name.

Example: ***Program:****u1p7.cpp*

```
#include<iostream.h>
int n=10;    //global variable
void main()
{
   int n=20; //local variable
   cout<<"\n Access local variable:"<< n;
   cout<<"\n Access global variable:"<< ::n;
}
```

Output:

```
Access local variable: 10
Access global variable: 20
```

In this program, name of local and global variable having the same name (n) therefore inside any function n will be consider as local variable and ::n will be consider as global variable.

Inline function:

A function that contain only one or two line of code and define using *inline* keyword called inline function.

Syntax:

```
inline fun_name(type arg1, type arg2,…)
{
    One/two line of code
}
```

Example: ***Program:****u1p8.cpp*

```
#include<iostream.h>
inline int addition(int a, int b)
{
   return a+b;
}
void show(int n)
{
   cout<<n;
}
void main()
{
   int x;
   x= addition(10,20);
   show(x);
}
```

Output:

```
30
```

In this example addition() is an inline function whereas show() is not.

Benefit of inline function:

At compile time, function calling statement is replaced by codes of inline function that save time consuming to jump on that function at run time. If inline function has loop or a number of statement then inline function behave normal function. Member functions that define inside the class and having only one or two line of code also called inline function.

What is class and object? How class is defined using data member and function? How object access members of the class?

Class declaration:

A description of any object is called class. Data members and functions of the object can be defined inside the class body.

Syntax of the class:

```
class cls_name
{
  Data Member's Declaration
  public:
       Member Function's Definitions
};
```

Data members of the class can be accessed only by member functions of the same class.
Member functions will be called by objects as per needs.
Ex:

```
class Duster
{
     int id;
     int length;
     int breadth;
     char color[10];
     float price;
public:
     void getdata()
     {
          cout<<"\nInput id,length,breadth of duster:";
          cin>>id>>length>>breadth;
          cout<<"Input color and price of duster:";
          cin>>color>>price;
     }
     void showdata()
     {
          cout<<"\nDuster record is:";
          cout<<"\nId="<<id;
          cout<<"\nlength="<<length;
          cout<<"\nbreadth="<<breadth;
          cout<<"\ncolor="<<color;
          cout<<"\nprice="<<price;
     }
};
```

In this example id, length, breadth, color and price are data members and used by getdata() and showdata() member functions. They are related to any Duster objects so suitable name of the class is Duster.

Object declaration:

Instances of class or variable of class data type is called object. When object is created then data members of the class is created.

Syntax:

```
cls_name obj_name1, obj_name2;
```

Example:

```
void main()
{
      Duster duster1, duster2;
      -----
      -----
}
```

Duster: duster1	**Duster:** duster2
id = 101 length: 10 cm breadth: 6 cm color: pink price: Rs. 50	id=102 length: 10 cm breadth: 6 cm color: black price: Rs. 40

duster1 & duster2 are two object of Duster class data type

When more than objects are declared then separate data members are created for each object.

Accessing members of the class:

Object of the class can access public members using .(dot) called period or membership operator.

Syntax:

```
obj_name1.member_name
obj_name2.member_name
```

Example:

```
void main()
{
      Duster duster1, duster2;
      duster1.getdata();
      duster2.getdata();
      duster1.showdata();
      duster2.showdata();

}
```

If duster1 call getdata() and showdata() member function then, these function access data of object duster1, similarly if duster2 object call these member function then they access data of duster2.

Program:u1p9.cpp

```
//program to demonstrate oops based duster information
#include<iostream.h>
class Duster
{
     int id;
     int length;
     int breadth;
     char color[10];
     float price;
public:
     void getdata()
     {
          cout<<"\nInput id,length,breadth of duster:”;
          cin>>id>>length>>breadth;
          cout<<”Input color and price of duster:";
          cin>>color>>price;
     }
     void showdata()
     {
          cout<<"\nDuster record is:";
          cout<<"\nId="<<id;
          cout<<"\nlength="<<length;
          cout<<"\nbreadth="<<breadth;
          cout<<"\ncolor="<<color;
          cout<<"\nprice="<<price;
     }
};
void main()
     {
          Duster duster1, duster2;
          duster1.getdata();
          duster2.getdata();
          duster1.showdata();
          duster2.showdata();
     }
```

Output:

```
Input id,length,breadth of duster:101   10   20(enter)
Input color and price of duster:Red     25(enter)

Duster record is:";
Id=101
length=10
breadth=20
color=Red
price=25;
```

Explicit and Implicit member function declaration:

There are two ways to declare member function of the class.

1. **Implicitly:** When member function is defined inside the class body called implicit function declaration.
2. **Explicitly:** When member function is defined outside the class body using scope resolution operator(::) called explicit member function declaration. This way improve the clarity of the class body.

Example: ***Program:u1p10.cpp***

```
//program to demonstrate oops based duster information
#include<iostream.h>
class Duster
{
     int id;
     int length;
     int breadth;
     char color[10];
     float price;
     public:
     //define inside class body-implicit function
     void getdata()
     {
          cout<<"\nInput id,length,breadth of duster:";
          cin>>id>>length>>breadth;
          cout<<"Input color and price of duster:";
          cin>>color>>price;
     }
     // define outside class body-explicit function
     void showdata();
};
//explicit function of Duster class
void Duster::showdata()
     {
          cout<<"\nDuster record is:";
          cout<<"\nId="<<id;
          cout<<"\nlength="<<length;
          cout<<"\nbreadth="<<breadth;
          cout<<"\ncolor="<<color;
          cout<<"\nprice="<<price;
     }
//Main function
void main()
     {
          Duster duster1, duster2;
          duster1.getdata();
          duster2.getdata();
          duster1.showdata();
          duster2.showdata();

     }
```

Output: same as previous

In this class getdata() and showdata() both are member functions of Duster class, but getdata() is defined inside class body wheareas showdata() is defined outside the class body using :: operator. Before defining explicit function, it must be declared in class body.

Access specifier /modifier of the class

(Visibility mode / label)

In C++, every data members and member functions are declared in following type of sections.

1. private
2. public
3. protected

They are used as

```
class cls_name
{
   private:
        members(data+function)
   protected:
        members(data+function)
   public:
        members(data+function)
};
```

Order of section can be changed. One labels can be repeated.

Uses of label:

1. *private:* members of private sections are accessible only by member functions of same class, friend functions of that class and its friend class.
2. *public:* members of public sections are accessible only by member functions of same class as well as all non member functions throughout in the program.
3. *protected:* members of protected sections are accessible only by member functions of same class as well as member functions of derived class.

Example: ***Program:u1p11.cpp***

```
//program to demonstrate private,protected and public
#include<iostream.h>
class A
{
     private:
          int a;
     protected:
          int b;
     public:
          int c;
          //member functions
          void getdata(int p1,int p2)
          {
               a=p1;
               b=p2;
          }
```

```
            void showdata()
            {
                  cout<<"\na="<<"\tb="<<b;
            }
};
void main()
{
      A ob;
      //ob.a=10;  not accessible due to private
      //ob.b=20;  not accessible due to protected
      ob.c=30;  //Accessible due to public
      ob.getdata(10,20); //Accessible due to public
      ob.showdata(); //Accessible due to public
      cout<<"\nc="<<c; //Accessible due to public
}
```

Output:

```
a=10
b=20
c=30
```

For one class, protected members behaves as private. In normal practice, data members are kept in private section and member functions in public section. In the absence of these labels, upper members will be considering private.

Static data member and static member function:

Static data member of the class is common for all objects of that class. That is static data is created only one times for objects. Default value of static data member is 0.
Static member function of the class can access only static data member of that class and it can be directly by class name.
static keyword is used to define static data member and static member function.
Example: ***Program:u1p12.cpp***

```
      #include<iostream.h>
      class A
      {
            static int n;
            public:
            static void counter()
            {
                  n++;
                  cout<<"\n"<< n;
            }
      };
      int A::n;
      void main()
      {
            A::counter();
            A::counter();
      }
```

Output:

```
1
2
```

In this program n is a static data member whereas counter() is a static member function that use static data n to count number of function calling. Class A can call directly static function counter() using :: operator.

Friend Function & Friend class:

By default, all data members of the class are private and only usable by member function of that class.

Friend Function:

A function which is not a member of the class but capable to use private data of that class called friend function. In c++ friend function is declared using friend keyword.

Example: ***Program:u1p13.cpp***

```
#include<iostream.h>
class A
{
     int a, b;
     public:
     int larger()
     {
          return (a>b ? a:b);
     }
     friend void fun();
};
void fun(int p1,int p2)//friend function of class A
{
     A ob;
     ob.a=p1;
     ob.b=p2;
     int n = ob.larger();
     cout<<"\n Large number is "<< n;
}
void main()
{
     larger(10,20);
}
```

Output:

```
Large Number is 20
```

In this program fun() is not a member function of class A but it is using private data in the form of ob.a & ob.b. Normally it is invalid to use but no error because fun() is publicly declared using friend keyword inside the class A.

Friend class:

Let A and B are two classes and if class B is capable to use all private data of class A then class B will be called friend class of A. Friend class is declared using friend class.

Example: ***Program:u1p14.cpp***

```
#include<iostream.h>
class B;
```

```
class A
{
     int a, b;;
     public:
     int larger()
     {
          return (a>b ? a:b);
     }
     friend class B;
};
Class B
{
     public:
     void fun(int p1, int p2)
     {
          A ob;
          ob.a=p1;  //class B uses private data of A
          ob.b=p2;
          int n = ob.larger();
          cout<<"\n Large number is "<< n;
     }
};
void main()
{
     B obj;
     obj.fun(10,20);
}
```

Output:

```
Large Number is 20
```

In this program class B is the friend class of class A, therefore any function of class B can uses private data of class A.

Passing object to function, returning object by function and copying one object to another object:

Passing object:

We can pass object to the function in the same way as we pass integer data but arguments of that function must be define with class type of passing objects.

Returning object:

Any function can also return one object in the same way as we return one integer data but function type must be defined with class type of returning object.

Copying object:

All data of one object can be copied into another object using single assignment operator (=) in the same way as we copy integer value to integer variable, but class type of both object must be similar.

Example: ***Program:u1p15.cpp***

```
#include<iostream.h>
class complex
{
     int a, b;
     public:
     void getdata(int p1, int p2)
     {
          a=p1;
          b=p2;
     }
     void showdata()
     {
          cout<<"\n"<<a<<"+"<<b<<"i";
     }
     friend complex sum(complex, complex);
};

complex sum(complex c1, complex c2)
{
     complex c3;
     c3.a=c1.a+c2.a;
     c3.b=c1.b+c2.b;
     return c3;
}

void main()
{
     complex oc1,oc2,oc3;
     oc1.getdata(2,3);   //to input 2+3i
     oc2.getdata(4,5);   //to input 4+5i
     oc3=sum(oc1,oc2);
     oc3.showdata();     // we get 6+8i
}
```

Output:

```
6+8i
```

In this example complex is a class to define any complex in the form of (a+bi). Here sum() is a friend function of complex class in which we pass two objects oc1 and oc2. After addition this function returns an object c3 of complex type. Using = we can assign returning object to oc3 object.

Unit-2

Arrays of objects:

In C++ it is possible to create array of objects of same class type. If we want to work on a number of objects of same class then arrays of objects are better option in place of declaring objects separately in the same way as we create array of any other type.

Example: ***Program:u2p1.cpp***

```
#include<iostream.h>
class A
{
     int a,b;
     public:
     void getdata()
     {
          cout<<"\n Input any two integer values:";
          cin>>a;
          cin>>b;
     }
     void showdata()
     {
          cout<<"\na="<<a;
          cout<<"\tb="<<b;
     }
};
void main()
{
     A ob[5]; //arrays of 5 objects
     for(int i=0;i<5;i++)
          Ob[i].showdata();
     cout<<"\nData of all objects are:";
     for(i=0;i<5;i++)
          Ob[i].showdata();
}
```

Output:

```
Input any two integer values:10    20(enter)
Input any two integer values:30    40(enter)
Input any two integer values:50    60(enter)
Input any two integer values:70    80(enter)
Input any two integer values:90    100(enter)
Data of all objects are:
a=10 b=20
a=30 b=40
a=50 b=60
a=70 b=80
a=90 b=100
```

In this example, ob[5] is arrays of 5 objects of same class A type. Each object can access their members using ob[i].getdata() or ob[i].showdata(). Here I denote index of array that vary from 0 to 4.

Pointer to object:

Address of object can be hold into pointer of same type of class. In this case, pointer to object can also access members of object using →(arrow) reference operator.

Example: ***Program:u2p2.cpp***

```
#include<iostream.h>
class A
{
      int a,b;
      public:
      void getdata()
      {
            cout<<"\n Input any two integer values:";
            cin>>a;
            cin>>b;
      }
      void showdata()
      {
            cout<<"\na="<<a;
            cout<<"\tb="<<b;
      }
};
void main()
{
      A *p;      //pointer to object
      A ob;      //object
      p=&ob;     //hold address
      p→getdata();
      p→showdata();
      ob.showdata();
}
```

Output:

```
Input any two integer values: 10   20(enter)
a=10 b=20 (by pointer to object)
a=10 b=20 (by object)
```

In this example, p is pointer to object ob. Therefore pointer p can also access members of object ob.

Type checking C++ pointers:

In C++, we must be same type of pointer while assigning one pointer into another pointer. C++ compiler checks same type of pointer.

For example:

```
int *pi;
float *pf;
```

in C++, the following assignment is illegal:

```
pi = pf; // error -- type mismatch
```

C++'s stronger type checking where pointers are involved differs from C, in which you may assign any value to any pointer.

this pointer:

When an object is called member function then pointer to this object also passed automatic internally, called ***this*** pointer. Inside member function *this* keyword represent pointer of calling object and **this* represent calling object.

Example: ***Program:u2p3.cpp***

```
#include<iostream.h>
class A
{
     int a,b;
     public:
     A getdata()
     {
          cout<<"\n Input any two integer values:";
          cin>>a;
          cin>>b;
          return *this;
     }
     void showdata()
     {
          cout<<"\na="<<a;
          cout<<"\tb="<<b;
     }
};
void main()
{
     A ob1,ob2;
     ob2=ob1.getdata(10,20);
     ob1.showdata();
     ob2.showdata();
}
```

Output:

```
a=10 b=20 (values of ob1)
a=10 b=20 (values of ob2)
```

getdata() member function return ***this** and it is an object ob1 inside member function.

Pointer to class member (.* and →*)

Offsets of public members can be assigned to pointer variable of class member, then they can be accessed by object using (.*) operator or pointer to object(→*).

Example: ***Program:u2p4.cpp***

```
#include<iostream.h>
class A
{
     public:
          int a;
          void show()
          {
               cout<<"\nwww.LRsir.net:"<<a;
          }
};
```

```
void main()
{
     int A::*data;  //pointer to data member
     void (A::*fun)()//pointer to member function
     data=&A::a;    //assign offset of a
     fun=&A::show;  //assign offset of show()
     A ob;      //object
     A *p;      //pointer to object
     p=&ob;     //assign address of ob to p

     ob.*data=10;   // ob.a=10;
     ob.*fun();     //ob.show();
     p→*data=20;    //ob.a=20;
     p→*fun(); //ob.show();
}
```

Output:

```
www.LRsir.net: 10
www.LRsir.net: 20
```

Here data is pointer to class data member that can hold offset of integer data member a and fun() is pointer to class member function that can hold offset of show() like member function. ob is object of A whereas p is its pointer. Therefore ".*" is applicable with ob and "→*" with pointer to ob.

References:

A reference is implicit pointer. It is declared using & operator. At the time of declaration, variable or object can be assign or passes then reference becomes alias, therefore we can use reference in place actual variable. It can be used in following ways.

1. Independent reference
2. References as parameters
3. Returning reference
4. Passing reference to object

Independent reference): When a variable is assign at the time of reference declaration then reference becomes alias of variable.

Example: ***Program:u2p5.cpp***

```
#include<iostream.h>
void main()
{
     int a;
     int &r=a;
     r=10;
     cout<<"\n"<<a;
}
```

Output:

```
10
```

Here r is standalone reference of a or alias of a, therefore r point to a.

References as function parameters:

When parameters of functions are references then they become alias of passing values. If function modifies references then this change takes on actual values.

Example: ***Program:u2p6.cpp*** swapping two number

```
#include<iostream.h>
void swap(int &x, int &y)
{
     int t;
     t=x;
     x=y;
     y=t;
}
void main()
{
     int a=10,b=20;
     swap(a,b);
     cout<<"\na="<<a<<"\tb="<<b;
}
```

Output:

```
a=20 b=10
```

In this example, x and y are reference parameters that can hold reference of a and b respectively.

Reference as return value:

Reference can also return from any function.

Example: ***Program:u2p7.cpp*** Replace R by small r.

```
#include<iostream.h>
char s[15]="www.LRsir.net"
char& replace(int i)
{
     return s[i];
}
void main()
{
     replace(5)='r';
     cout<<s;
}
```

Output:

```
www.Lrsir.net
```

"char &replace(int i)" return reference of given indexing character that replace by r.

Passing reference to object

We can also pass reference of any object to the function. If function change using reference then it is made on actual object, because no copy is made of passing object.

Example: ***Program:u2p8.cpp***

```
#include<iostream.h>
class A
{
     int a, b;
     public:
          void getdata(int p1, int p2)
          {
               a=p1;
               b=p2;
          }
          void showdata()
          {
               cout<<"\na="<<a;
               cout<<"\tb="<<b;
          }
          friend void sign(A &);
};
void sign(A &or)
{
     or.a=(-or.a);
     or.b=(-or.b);
}
void main()
{
     A ob;
     ob.getdata(10,-20);
     sign(ob);
     ob.showdata();
}
```

Output:

```
a=-10     b=20
```

It is clear that reference (or) is alias of passing object. Therefore sign changes on data members of passing object ob.

Restrictions to References

- A reference variable must be initialized when it is declared.
- Null references are prohibited.
- We cannot reference another reference.
- We cannot obtain the address of a reference.
- We cannot create arrays of references.
- We cannot create a pointer to a reference.

Dynamic allocation operator: (*new* and *delete*)

At run time we can allocate required amount of memory space using *new* operator (malloc in c) and released using *delete* operator (free in c). At run time we can allocate and releases memory space for:

1. One data allocation with initial value
2. Allocating array
3. Allocating object

new operator gives a base address of allocated memory space otherwise NULL, that assign to the pointer variable.

delete operator released memory space of passing pointer.

Initializing Allocated Memory

Assign a value at the time of memory allocation.

Syntax:

```
ptr = new type (value);
```

Example: ***Program:u2p9.cpp***

```
#include<iostream.h>
void main()
{
int *p;
p = new int(10); // initialize to 10
cout<<"\nAddress="<<p;
cout<<"\tvalue="<<*p;
delete p;
}
```

Output:

```
Address=65677  value=10
```

Here p is pointer to integer that holds *new* allocated address of integer and initialize by 10. After end need it is removed using *delete.*

Allocating Arrays

We can create dynamic array by allocating arrays using ***new*** **operator.**

syntax:

```
p = new array_type [size];
```

Here, *size* is maximum limit of array.

To free an array, use this form of **delete:**

```
delete [ ] p;
```

Example: ***Program:u2p10.cpp***

```
#include<iostream.h>
void main()
{
     int *p, n i;
     cout<<"\nInput array size:";
     cin>>size;
     p=new int[n]; // allocate  integer array
     cout<<"Input data:";
     for(i=0; i<n; i++ )
          cin>>p[i];
     cout<<"\nArray data:"
     for(i=0; i<n; i++)
          cout<<"\t"<<p[i];
```

```
        delete [] p; // release the array
    }
```

Output:

```
Input array size:3
Input data: 10 20   30(Enter)
Array data: 10 20   30
```

Here "p=new int[n];" allocate array at run time and after used "delete [] p;" released array space.

Allocating object

We can also create object at run time using ***new*** **operator.**

syntax:

```
p = new class_type;
```

To free object**:**

```
delete p;
```

Example: ***Program:u2p11.cpp***

```
#include<iostream.h>
class A
{
    public:
        A()
        {
            cout<<"\nObject created";
        }
        ~A()
        {
            cout<<"\nObject removed";
        }
        void show()
        {
            cout<<"\nwww.LRsir.net";
        }

};
void main()
{
    A *p;
    p=new A;
    p→show();
    delete p;
}
```

Output:

```
Object created
www.LRsir.net
object removed
```

here →(arrow) operator is used to access members by pointer to object.

Constructor and destructor:

Meaning of constructor: At runtime constructor creates objects of the class. When object is creating, all data members of its class are also creates in the memory. We can execute codes of specified function automatically during object creation, such function called constructor function.

Features of constructor function in C++:

1. The name of constructor function is similar to the class name.
2. It does not specify any function type. (even not void)
3. It always defines in public section.
4. It can be default or parameterized.
5. More than one constructor functions can be defined in one class i.e. can be overloads.

General Syntax of constructor:

```
class cls_name
{
      Data member section
      public:
            //default constructor
            cls_name()
            {
                  …
            }
            // parameterized constructor
            cls_name(type par1, type par2,….)
            {
                  …
            }
};
```

Type of constructor function:

On the basis of parameters defined in constructor function, it can be classified into following categories.

1. Default constructor
2. Parameterize constructor
3. Overloaded constructor (multiple constructors in a class)
4. Copy constructor
5. Constructor with default argument

1. Default constructor: A constructor function that does not specified any parameters and used to initialize default values of object's data member.

Example: ***Program:u2p12.cpp***

```
#include<iostream.h>
class A
{
      int a,b;
      public:
      // default constructor
```

```
        A()
        {
                a=10;
                b=20;
        }
        void showdata()
        {
                cout<<"\n"<<a<<"\t"<<b;
        }
};
void main()
{
        A ob1,ob2;
        Ob1.showdata();
        Ob2.showdata();
}
Output:
10    20
10    20
```

In this program A() is a default constructor that invoked automatically when object of class A is declared without passing values. For each object of A, its default constructor will set 10 & 20 to the data members a & b respectively.

2. Parameterized constructor: A constructor that specified one or more parameters and used to initialize object's data member by passing values along with object declaration.

Example: ***Program:u2p13.cpp***

```
#include<iostream.h>
class A
{
        int a,b;
        public:
        // parameterized constructor
        A(int p1,int p2)
        {
                a=p1;
                b=p2;
        }
        void showdata()
        {
                cout<<"\n"<<a<<"\t"<<b;
        }
};
void main()
{
        A ob1(10,20),ob2(11,22);
        Ob1.showdata();
        Ob2.showdata();
}
```

Output:

```
10    20
11    22
```

In this program A(int p1, int p2) is a parameterized constructor that invoked automatically when object of class A is declared with two integer values. These two values will be store to the data members of object.

3. Overloaded constructor: When more than one constructor is defined in a class, then such constructors are called overloaded constructor. Every constructor must be distinct on the basis of parameters.

Example: ***Program:u2p14.cpp***

```
#include<iostream.h>
class A
{
      int a,b;
      public:
      // default constructor
      A()
      {
            a=10;
            b=20;
      }
      // constructor with one parameter
      A(int p1)
      {
            a=p1;
            b=22;
      }
      // constructor with two parameters
      A(int p1,int p2)
      {
            a=p1;
            b=p2;
      }
      void showdata()
      {
            cout<<"\n"<<a<<"\t"<<b;
      }
};
void main()
{
      A ob1, ob2(11),ob2(5,7);
      Ob1.showdata();
      Ob2.showdata();
      ob3.showdata();
}
Output:
10    20
11    22
5     7
```

In this program, three constructors are defined with different number of parameters. Object ob1 invoked A(), ob2(11) invoked A(int) and ob3(5,7) invoked A(int, int).

4. Copy constructor: A constructor that is capable to copy old object into new object at the time of object creation.

Example: ***Program:u2p15.cpp***

```
#include<iostream.h>
class A
{
      int a,b;
      public:
      // copy constructor
      A(A &ob)
      {
            a=ob.a;
            b=ob.b;
      }

      // constructor with two parameters
      A(int p1,int p2)
      {
            a=p1;
            b=p2;
      }
      void showdata()
      {
            cout<<"\n"<<a<<"\t"<<b;
      }
};
void main()
{
      A ob1(10,20);
      A ob2=ob1; // or A ob2(ob1);
      Ob1.showdata();
      Ob2.showdata();
}
Output:
10    20
10    20
```

In this program, A(A &oc) is a copy constructor that can copy object ob1 into ob2.

5. Constructor with default argument: A constructor that defines parameter with default values called constructor with default arguments. is capable to copy old object into new object at the time of object creation.

Example: ***Program:u2p16.cpp***

```
#include<iostream.h>
class A
{
      int a,b;
```

```
        public:
        // constructor with default arguments
        A(int p1=0,int p2=0)
        {
              a=p1;
              b=p2;
        }
        void showdata()
        {
              cout<<"\n"<<a<<"\t"<<b;
        }
};
void main()
{
        A ob1,ob2(11), ob3(5,7);
        ob1.showdata();
        ob2.showdata();
        ob3.showdata();
}
Output:
0     0
11    0
5     7
```

In this program, A(int p1=0, int p2=0) is constructor with default argument and it invoked for ob1 or for ob2(11) or for ob3(5,7).

Meaning of destructor: At runtime destructor removes unused object from memory. When object is removing, all data members are also remove. We can execute codes of specified function automatically during object removal, such function called destructor function.

Features of destructor function in C++:

1. The name of destructor function is similar to the class name with prefix ~ sign.
2. It does not specify any function type. (not even void)
3. It always defines in public section.
4. It has no parameters.
5. Destructor never overloads.

General Syntax of constructor:

```
class cls_name
{
        Data member section
        public:
              //destructor
              ~cls_name()
              {
                    …
                    …
              }
};
```

Example: ***Program:u2p17.cpp***

```
#include<iostream.h>
class A
{
     int a,b;
     public:
     // constructor
     A()
     {
          a=10;
          b=20;
          cout<<"\nObject created";
     }
     // destructor
     ~A()
     {
          cout<<"\nObject removed";
     }
     void showdata()
     {
          cout<<"\n"<<a<<"\t"<<b;
     }
};
void main()
{
     A ob1;
     Ob1.showdata();
}
```

Output:

```
10   20
Object created
Object removed
```

In this program A() is a constructor that invoked when object is created whereas ~A() is destructor that invoked when object is remove from memory.

Unit-3

Function overloading:

Function overloading is one example of compile time polymorphism. When more than one function are defined with the same name but mismatch on the basis of parameters then such functions are called overloaded function and this mechanism called function overloading.

Features:

1. At least two functions have same name.
2. All overloaded functions must be distinct on the basis of parameters.
3. All overloaded functions can have same return type.
4. Overloaded functions are linked to its function call at compile time.
5. Two functions of same name and same parameters are not permitted.

Example: ***Program:u3p1.cpp***

```
#include<iostream.h>
class A
{
     int a;
     float b;
     public:
     void getdata()
     {
          cout<<"\nInput integer then real value:";
          cin>>a>>b;
     }
     void getdata(int p)
     {
          a=p;
          cout<<"\nInput real value:";
          cin>>b;
     }
     void getdata(float p)
     {
          cout<<"\nInput integer value:";
          cin>>a;
          b=p;
     }
     void getdata(int p1, float p2)
     {
          a=p1;
          b=p2;
     }
     void showdata()
     {
          cout<<"\nInteger value is "<<a;
          cout<<"\nReal value is "<<b;

     }

};
```

```
void main()
{
     A ob1,ob2,ob3,ob4;
     ob1.getdata();
     ob2.getdata(10);
     ob3.getdata(11.2);
     ob4.getdata(5,7.5);

     ob1.showdata();
     ob2.showdata();
     ob3.showdata();
     ob4.showdata();
}
```

Output:

```
Input integer then real value:     22   33.4 (Enter)
Input real value: 44.2 (Enter)
Input integer value: 55 (Enter)

Integer value is 22
Real value is 33.4

Integer value is 10
Real value is 44.2

Integer value is 55
Real value is 11.2

Integer value is 5
Real value is 7.5
```

In this program getdata() is overloaded function that overload 4 times.

1. void getdata(){}. It is invoked by ob1.getdata()
2. void getdata(int p){}. It is invoked by ob1.getdata(10)
3. void getdata(float p){}. It is invoked by ob1.getdata(11.2)
4. void getdata(int p1, intp2){}. It is invoked by ob1.getdata(5,7.5)

All these functions have same name but different on the basis of parameters. In this way defining functions are called function overloading. Using function overloading we do not remember to name of all functions having similar task.

Operator overloading:

Operator overloading is one more example of compile time polymorphism. Operator operates on numeric data. If operator operates on objects then we say that operator is overloaded with objects and this mechanism called operator overloading.

For example:

Binary + operator can operate numeric data (2+3) as well as operate two objects of same class (ob1+ob2).

All operators can be overloaded with objects except following operators.

1. Conditional operator (? :)
2. Scope resolution operator (::)
3. Size operator (sizeof)
4. Class member access operator (. and .*)

Defining operator overloading:

Operator overloading is possible only if that operator is defined using *operator* function and pass objects in the form of arguments. In c++ operator function can be defined in following two ways.

1. operator overloading using member function of class
2. operator overloading using friend function of class

Operator overloading using member function of class:

Using *Operator* keyword we can define a member function in the class that perform operator overloading. Such function defined using following syntax:

```
class cls_name
{
      Data members
      public:
      ret_type operator±(type arg1,type arg2,…)
      {
            Apply ± operator on each data members
      }
};
```

± may be any operator like + - / * ++ --.

If operator function is defined as a member function then first object call operator function and second object pass as argument.
Ex: ob3=ob1+ob2 will call *operator*+ in the form of ob3=ob1.operator+(ob2).

Unary operator overloading:

1. Overloading unary minus operator:

–obj (convert sign of all values of data members)

Example: ***Program:u3p2.cpp***

```
#include<iostream.h>
class A
{
      int a, b;
      public:
```

```
        void getdata(int p1, intp2)
        {
            a=p1;
            b=p2;
        }
        void showdata()
        {
            cout<<"\na="<<a;
            cout<<"\tb="<<b;
        }
        //unary - operator overloading function
        void operator-()
        {
            a=-a;
            b=-b;
        }
    };
    void main()
    {
        A ob;
        ob.getdata(10,-20);
        ob.showdata();
        -ob; // unary - operator overloading
        ob.showdata();
    }
```

Output:

```
a=10        b=-20
a=-10       b=20
```

In this example unary minus(-) operator is apply on object (-ob). At compile time it is converted into *ob.operator-()* statement, because *operator-()* is a member function and this function reverse the sign of every data members of object ob.

2. Overloading increment (++) operator:

Obj++ (Increment values of each data members by one)

Example: ***Program:u3p3.cpp***

```
#include<iostream.h>
class A
{
    int a, b;
    public:
    void getdata(int p1, intp2)
    {
        a=p1;
        b=p2;
    }
    void showdata()
    {
        cout<<"\na="<<a;
        cout<<"\tb="<<b;
    }
```

```
        //   ++ operator overloading function
        void operator++()
        {
            a++;
            b++;
        }
    };

    void main()
    {
        A ob;
        ob.getdata(10,20);
        ob.showdata();
        ob++;      // ++ operator overloading
        ob.showdata();
    }
```

Output:

```
a=10        b=20
a=11        b=21
```

In this example ++ operator is apply on object (ob++). At compile time it is converted into *ob.operator++()* statement, because *operator++()* is a member function and this function increment values of every data members by one of object ob.

3. Overloading decrement (--) operator:

Obj-- (Decrement values of each data members by one)

Example: ***Program:u3p4.cpp***

```
#include<iostream.h>
class A
{
    int a, b;
    public:
    void getdata(int p1, intp2)
    {
        a=p1;
        b=p2;
    }
    void showdata()
    {
        cout<<"\na="<<a;
        cout<<"\tb="<<b;

    }
    //   -- operator overloading function
    void operator--()
    {
        a--;
        b--;
    }
};
```

```
void main()
{
     A ob;
     ob.getdata(10,20);
     ob.showdata();
     ob--;      // -- operator overloading
     ob.showdata();
}
```

Output:

```
a=10       b=20
a=9        b=19
```

In this example -- operator is apply on object (ob--). At compile time it is converted into *ob.operator--()* statement, because *operator--()* is a member function and this function decrement values of every data members by one of object ob.

Binary operator overloading: (Arithmetic and shorthand operators)

1. **Overloading Binary Arithmetic operator: (+,-,*,/, %)**
   ob3=ob1+ob2
   Each data members of objects ob1 and ob2 are added then assign to object ob3.

Example: ***Program:u3p5.cpp***

```
#include<iostream.h>
class A
{
     int a, b;
     public:
     void getdata(int p1, intp2)
     {
          a=p1;
          b=p2;
     }
     void showdata()
     {
          cout<<"\na="<<a;
          cout<<"\tb="<<b;

     }
     //Binary + operator overloading function
     A operator+(A ob)
     {
          A ot;
          Ot.a=a+ob.a;
          Ot.b=b+ob.b;
          return ot;
     }
};
void main()
{
     A ob1,ob2,ob3;
     Ob1.getdata(10,20);
```

```
        Ob2.getdata(1,2);
        ob3=ob1+ob2;    // Binary + operator overloading
        ob3.showdata();
    }
```

Output:

```
a=11        b=22
```

In this example binary + operator is apply on object (ob1) and (ob2). At compile time it is converted in the form of *ob3= ob1.operator+(ob2)* statement, because *operator+()* is a member function. This function call by (ob1) and (ob2) is pass into (ob), therefore (a, b) are data members of (ob1) and (ob.a, ob.b) are members of (ob2). In this function we apply binary + operator on each data member(like ot.a=a+ob.b). Finally resulting object (ot) returns to object (ob3).

2. **Overloading Binary shorthand operator: (+=,-=,*=,/=, %=)**
   ob1+=ob2
   Every data members of object ob1 are increases by data members of object ob2.

Example: ***Program:u3p6.cpp***

```
#include<iostream.h>
class A
{
    int a, b;
    public:
    void getdata(int p1, intp2)
    {
        a=p1;
        b=p2;
    }
    void showdata()
    {
        cout<<"\na="<<a;
        cout<<"\tb="<<b;
    }
    //Binary += operator overloading function
    A operator+=(A ob)
    {
        a+=ob.a;
        b+=ob.b;
    }
};
void main()
{
    A ob1,ob2;
    Ob1.getdata(10,20);
    Ob2.getdata(5,7);
    ob1+=ob2; // Binary += operator overloading
    ob1.showdata();
}
```

Output:

```
a=15        b=27
```

In this example shorthand += operator is apply on object (ob1) and (ob2). At compile time *ob1+=ob2* is converted in the form of *ob1.operator+=(ob2)* statement, because *operator+=()* is a member function. This function call by (ob1) and (ob2) is pass into (ob), therefore (a, b) are data members of (ob1) and (ob.a, ob.b) are members of (ob2). In this function we apply shorthand += operator on each data member(a+=ob.a).

Operator overloading using friend function of class:

Using *Operator* keyword we can define a friend function of the class that can also perform operator overloading. Such function defined using following syntax:

```
class cls_name
{
      Data members
      public:
      friend ret_type operator± (type, type,…);
};
ret_type operator± (type arg1,type arg2,…)
      {
            Apply ± on each data members
      }
```

± may be any operator like +, -, /, *, ++, --, += etc.

If operator function is defined as a friend function of the class then every object will passes into operator function in the form of arguments.

Ex: ob3=ob1+ob2 will be convert into ob3=operator+(ob1,ob2).

Overloading Binary plus(+) operator using friend function:

ob3=ob1+ob2
Each data members of objects ob1 and ob2 are added then assign to object ob3.

Example: ***Program:u3p7.cpp***

```
#include<iostream.h>
class A
{
      int a, b;
      public:
      void getdata(int p1, intp2)
      {
            a=p1;
            b=p2;
      }
      void showdata()
      {
            cout<<"\na="<<a;
            cout<<"\tb="<<b;

      }
      friend A operator+(A,A);
};
```

```
//Binary + operator overloading using friend function
          A operator+(A oa,A ob)
          {
               A ot;
               Ot.a=oa.a+ob.a;
               Ot.b=oa.b+ob.b;
               return ot;
          }
          void main()
          {
               A ob1,ob2,ob3;
               Ob1.getdata(10,20);
               Ob2.getdata(1,2);
               ob3=ob1+ob2;//Binary+operator overloading
               ob3.showdata();
          }
```

Output:

```
a=11       b=22
```

In this example binary + operator is apply on object (ob1) and (ob2). At compile time it is converted in the form of *ob3= operator+(ob1,ob2)* statement, because *operator+()* is a friend function of class A. In this function object (ob1) and (ob2) are passed into (oa, ob) therefore (oa.a, oa.b) are data members of (ob1) and (ob.a, ob.b) are members of (ob2). In this function we apply binary + operator on each data member(like ot.a=oa.a+ob.b). Finally resulting object (ot) returns to object (ob3).

Overloading new and delete:

It is possible to overload **new** and **delete**. You might choose to do this if you want to use some special allocation method like malloc for new and free for delete.

For Example: ***Program:u3p8.cpp***

```
#include <iostream.h>
#include<alloc.h>
class A
{
     int a,b;
     public:
          A()
          {
               a=0;
               b=0;
          }
          A(int p1,int p2)
          {
                a=p1;
                b=p2;
          }
          void showf()
          {
               cout<<"\na="<<a<<"\tb="<<b
          }
```

```
                // new overloaded
                void *operator new(size_t size)
                {
                      cout << "In overloaded new.\n";
                      void *p;
                      p = malloc(size);
                      return p;
                }
                // delete overloaded
                void operator delete(void *p)
                {
                      cout << "In overloaded delete.\n";
                      free(p);
                }
        };
        //----
        void main()
        {
              A *p1, *p2;
              p1 = new loc (10, 20);
              p2 = new loc (-10, -20);
              p1->showf();
              p2->showf();
              delete p1;
              delete p2;
        }
```

Output:

```
In overloaded new
In overloaded new
10 20
-10 -20
In overloaded delete
In overloaded delete
```

when **new** or **delete** are encountered and If these have been overloaded, the overloaded versions are used.

Overloading Some Special Operators

Array subscripting [ ], function calling ( ), and class member access ->

Overloading [ ]

In C++, the **[ ]** is considered a binary operator when you are overloading it. Therefore, the general form of a member "*operator[ ]()*" function is as shown here:

```
type CName::operator[](int i)
{
      // . . .
}
```

If **ob** is an object then Ob[3]translates into *Ob.operator[](3).* That is, the value of the expression within the subscripting operators is passed to the ***operator[ ]()*** function in its explicit parameter.

For example, following program has overloaded **operator[ ]()** function returns the value of the array as indexed by the value of its parameter.

Example:***Program:u3p9.cpp***

```
#include <iostream.h>
class A
{
      int a[3];
      public:
      A(int i, int j, int k)
      {
            a[0] = i;
            a[1] = j;
            a[2] = k;
      }
      //Overloading []
      int operator[](int i)
      {
            return a[i]; //Return indexing array data
      }
};
//------
void main()
{
      A ob(1, 2, 3);
      cout << ob[1]; // displays 2
}
```

Output:

```
2
```

In this program ob[1] translate into ob.operator[1] and this function return ob[1] 's value.

Overloading ( )

When you overload the ( ) function call operator, then you are creating an operator function that can be passed an arbitrary number of parameters.

For example: Statement *Ob(10, 23.34, "www.LRsir.net")* translates into

Ob.operator()(10, 23.34, " www.LRsir.net ");

Example: ***Program:u3p10.cpp***

```
#include <iostream.h>
class A
{
      int a,b;
      public:
            // Overload ( ) for loc.
            Void operator()(int p1, int p2)
            {
                  a = p1;
                  b = p2;
            }
            void showf()
            {
```

```
                    cout<<"\na="<<a<<"\tb="<<b
                }
        };
        //------
        void main()
        {
             A ob;
             ob(10, 20); //Call Overloaded ()
             ob.showf();
        }
Output:
a=10 b=20
```

Thus ob(10,20) will call overloaded ( ) function. And this will assign values.

Overloading →

The → pointer operator, also called the *class member access* operator, is considered a unary operator when overloading. When → operator is overloaded then *ob→ClassMember translate* **into** *ob.operator→() .*

The following program illustrates overloading the **–>** by showing the equivalence between **ob.i** and **ob–>i** when **operator–>()** returns the **this** pointer:

Example: ***Program:u3p11.cpp***

```
        #include <iostream.h>
        class A
        {
             public:
                  int a;
                  //Overloaded →
                  A *operator→()
                  {
                       return this;
                  }
        };
        //-----
        void main()
        {
             A ob;
             Ob→a = 10; // same as ob.a
             cout << ob.a << " " << ob→a;
        }
Output:
10   10
```

An **operator→()** function must be a member of the class upon which it works.

Overloading the , Comma Operator

You can overload C++'s comma operator. The comma is a binary operator. When **,** (comma) operator is overloaded then *(ob1,ob2) translate into ob1.operator,(ob2) .*

Here is a program that illustrates the effect of overloading the comma operator.

Example: ***Program:u3p12.cpp***

```
#include <iostream.h>
class A
{
     public:
          int a;
          //Overloaded ,
          A operator,(A ob)
          {
               A temp;
               temp.a=ob.a;
               cout<<"\na="<<a;
               return temp;
          }
};
void main()
{
     A ob1,ob2;
     ob1.a = 10;
     ob2.a = 20;
     cout << "\n"<<ob1.a; // display 10
     cout << "\n"<<ob2.a; // display 20

     ob1=(ob1,ob2); //Overloading , operator

     cout << "\n"<<ob1.a; // display 20 of ob2
     cout << "\n"<<ob2.a; // display 20
}
```

```
Output:
10
20
20
20
```

In this program ob1=(ob1,ob2) translate into ob1=ob1.operator,(ob2). This function return object ob2 that assign ob1.

Overloading <<(Insertion operator) and >>(Extraction operator):

The functions that overload the insertion(<<) and extraction(>>) operators are generally called *inserters* and *extractors*, respectively.

Example: ***Program:u3p13.cpp***

```
#include<iostream.h>
class A
{
     int a, b;
     public:
     friend istream &operator>>(istream &stream, A &o);
     friend ostream &operator<<(ostream &stream, A o);
};
```

```
istream &operator>>(istream &stream, A &o)
{
     cout<<"\nInput two integers:";
     stream>>o.a>>o.b;
     return stream;
}
ostream &operator<<(ostream &stream, A o)
{
     stream<<"\na="<<o.a<<"\tb="<<o.b;
     return stream;
}
void main()
{
     A ob1,ob2;
     cout<<Input data for two object";
     cin>>ob1>>ob2;
     cout<<"Data of two objects are:";
     cout<<ob1<<ob2;
}
```

Output:

```
Input data for two object
Input two integers:10    20(enter)
Input two integers:5     7(enter)
Data of two objects are:
a=10 b=20
a=5  b=7
```

In this example >> and << are overload with objects.

Unit-4

Inheritance:

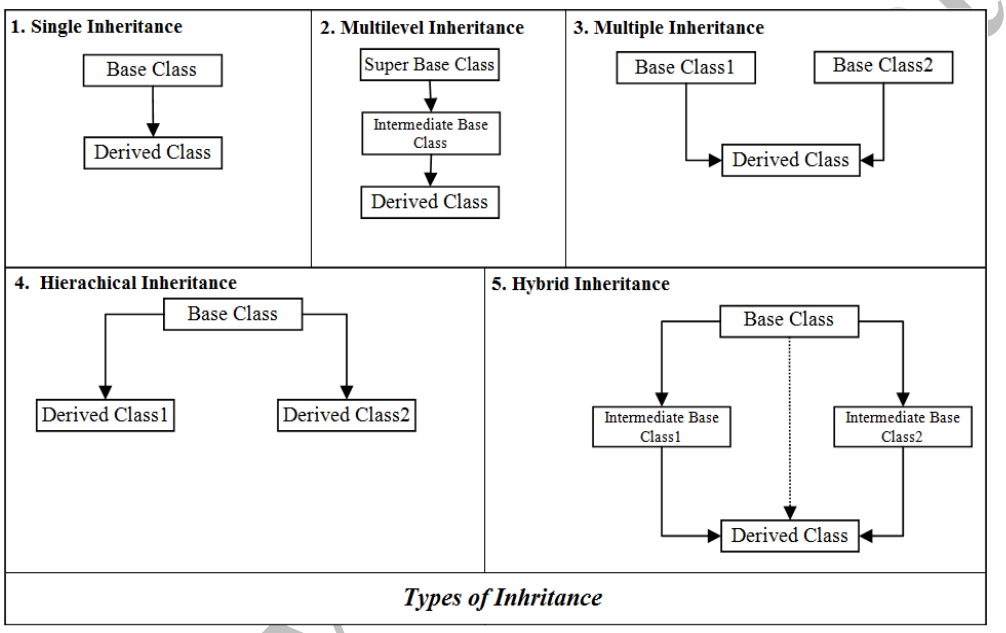
The ability to access members of one class by another class called inheritance. At least two classes are required for inheritance.

1. Base class (Old class / Super class): A class that provides members to another class called base class.
2. Derived class (New class / sub class): A class that can access members of base class called derived class.

Derived class can access only protected and public members of base class.

Types of inheritance:

In C++, inheritance can be implemented in following ways.



Types of Inhritance

Syntax of derived class:

```
class derived_class: mode base_class
{
      Members of derived class
};
```

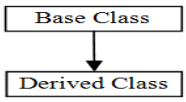
Here-
: is used to connect derived class to the base class.

Mode/ Base class access control(Access protected member of base class): may be private or protected or public. Default is private. This mode change protected and public members of base into another mode for derived class. It can be explain using following table.

	private mode with derived	**protected mode with derived**	**public mode with derived**
private members of base	Not inherited	Not inherited	Not inherited
protected members of base	Becomes private	Remain protected	Remain protected
public members of base	Becomes private	Becomes protected	Remains public

When object of derived class is creates then all data members of base class will be creates then data members of derived class. Member functions of derived class can access only protected and public members of base class.

Example: **Single Inheritance** (One base and one derived class)



Program:u4p1.cpp

```
#include<iostream.h>
class base
{
     int a;    // only usable by base
     protected:
          int b;    //usable by base or derived only
     public:
          void bgetdata(int p)
          {
               a=p;
          }
          void bshowdata()
          {
               cout<<"\nIn base a="<<a;
          }
};
class derived: public base
{
     public:
          void dgetdata(int p)
          {
               b=p;      //uses protected of base
          }
          void dshowdata()
          {
               cout<<"\nIn derived b="<<b;
          }

};
void main()
{
     derived od;
     od.dgetdata(10);
     od.dshowdata();

     od.bgetdata(20);
     od.bshowdata();
}
```

Output:

```
In derived b=10
In base a=20
```

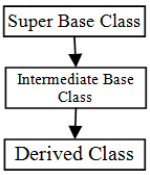
In this example derived class is publically connected to base class therefore ***protected and public members of base class can be directly used by member functions of derived class***, that is protected data b is uses by dgetdata() and dshowdata() of derived class. In this class private data a of base class is not permitted.

When object of derived class is created then first all data members of base class are created then all data members of derived class are created, but private data of base class are only visible within member functions of base class.

Object of derived class can call all public member functions of derived class as well as public member function of base class.(due to public mode).

Multilevel Inheritance:

If it is possible to access protected and public members of base class (grand) by its drive's(parent) derived class(child) then it is called multilevel inheritance.



Example: ***Program:u4p2.cpp***

```
#include<iostream.h>
class grand
{
      protected:
            int g;
};
class parent: public grand
{
      protected:
            int p;
};
class child: public parent
{
      int c;
      public:
            void getdata(int p1, int p2, int p3)
            {
                  g=p1;
                  p=p2;
                  c=p3;
            }
            void shodata()
            {
             cout<<"\n data g is of super base (grand) "<<g;
             cout<<"\n data p is of intermediate base (parent) "<<p;
             cout<<"\n data c is of derived (child) "<<c;
            }
};
void main()
{
      child od;
      od.getdata(10,20,30);
      od.showdata();
}
```

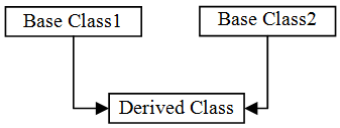
Output :

```
data g is of super base (grand) 10
data p is of intermediate base (parent) 20
data c is of derived (child) 30
```

In this example, child is a derived class that can access protected or public members of its intermediate base class (parent) as well of super base class (grand).

Multiple Inheritances:

When one derived class has more than one base class called multiple inheritances. It means one derived class can access protected and public members of more base classes.



Example: ***Program:u4p3.cpp***

```
#include<iostream.h>
class base1
{
      protected:
            int b1;
};
class base2
{
      protected:
            int b2;
};
class derived: public base1, public base2
{
      int d;
      public:
            void getdata(int p1, int p2, int p3)
            {
                  b1=p1;
                  b2=p2;
                  d=p3;
            }
            void shodata()
            {
             cout<<"\n data b1 is of base1 "<<b1;
             cout<<"\n data b2 is of base2 "<<b2;
             cout<<"\n data d is of derived "<<d;
            }
};
void main()
{
      derived od;
      od.getdata(10,20,30);
      od.showdata();
}
```

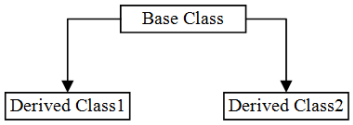
Output :

```
data g is of base1 10
data p is of base2 20
data c is of derived 30
```

In this example, derived class can access protected data b1 of base1 class and protected data b2 of base2 class, therefore it is called multiple inheritance.

Hierarchical Inheritances:

When one base class has more than one derived class, called hierarchical inheritances. It means more than one derived class can also access protected and public members of one base classes. (base is server and all derived classes are client)



Example: ***Program:u4p4.cpp***

```
#include<iostream.h>
class base
{
      protected:
            int b;
};
class derived1: public base
{
      int d1;
      public:
            void getdata(int p1, int p2)
            {
                  b=p1;
                  d1=p2
            }
            void shodata()
            {
             cout<<"\n data b is of base "<<b;
             cout<<"\n data d1 is of derived1 "<<d1;
            }

};
class derived2: public base
{
      int d2;
      public:
            void getdata(int p1, int p2)
            {
                  b=p1;
                  d2=p2
            }
            void shodata()
            {
             cout<<"\n data b is of base "<<b;
             cout<<"\n data d2 is of derived2 "<<d2;
            }
};

void main()
{
      derived1 od1;
      od1.getdata(10,20);
      od1.showdata();
      derived2 od2;
```

```
        od2.getdata(5,7);
        od2.showdata();
}
```

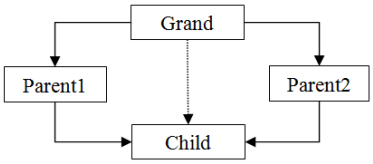
Output:

```
data b is of base 10
data d1 is of derived1 20
data b is of base 5
data d2 is of derived2 7
```

In this example, derived1 class and derived2 class both can access protected data b of base class, therefore it is called hierarchical inheritance.

Hybrid inheritance (virtual base class)

A structure of inheritance having more than one type of inheritance called hybrid inheritance.



This structure has one hierarchical inheritance "*Grand →(Parent1 and Parent2)*", one multiple inheritance *"(Parent1, Parent2)→Child "* and two multilevel inheritance "*Grand→Parent1→Child*", "*Grand→Parent2→Child.*

Problem with hybrid inheritance:

In above type of inheritance, when object of "*Child"* class is created then data members of *"Grand"* class are crated two times. One set via *"Parent1"* class and other set via *"Parent2"* class. This causes ambiguity errors if object of *"child"* class uses data members of *"Grand"* class. This should be avoided.

Solution of ambiguity problem: "Virtual base class"

All multiple paths between Grand and Child class (Grand→Parent1→Child, Grand→Parent2→Child) can be avoided by making a common base class(Grand) as virtual base class while declaring intermediate base classes (parent1 and parent2). Due to virtual base class, only one copy of Grand members are created for Child class.

Implementing virtual base class: *Program:u4p5.cpp*

```
#include<iostream.h>
class grand
{
      protected:
            int g;
};
class parent1: virtual public grand
{
      protected:
            int p1;
};
class parent2: virtual public grand
{
      protected:
            int p2;
};
```

```
class child: public parent1, public parent2
{
     private:
          int c;
     void getdata(int w, int x, int y, int z)
     {
          g=w;
          p1=x;
          p2=y;
          c=z;
     }
     void showdata()
     {
          cout<<"\ngrand data is "<<g;
          cout<<"\nparent1 data is "<<p2;
          cout<<"\nparent2 data is "<<p1;
          cout<<"\nchild data is "<<c;
     }
};
void main()
{
     child od;
     od.getdata(10,20,30,40);
     od.showdata();
}
```

Output:

```
grand data is 10
parent1 data is 20
parent2 data is 30
child data is 40
```

In this program grand class is declared virtual base class for child class via all intermediate base classes parent1 and parent2, therefore when object od of child class is created then data g of grand class is created only times due to virtual base class and ambiguity problem does not arises.

Constructor and destructor used in inheritance:

Constructor:

When object of derived class is created then order of creating data members and invoking constructor functions are following.

1. **In Single inheritance:** First base class then derived class.
2. **In multilevel inheritance:** First super base (grand) then intermediate (parent) and finally derived.
3. **In multiple inheritance:** First from left base class towards right most base class and finally derived.

Destructor:

When object of derived removed then order of removing data members and invoking destructor functions are reversing of constructors. It means, that last created will first remove.

Example:

Program:u4p6.cpp

```
#include<iostream.h>
class base
{
     public:
          base()
          {    cout<<"\n base constructor";    }
          ~base()
          {    cout<<"\n base destructor";     }
};
class derived: public base
{
     public:
          derived()
          {    cout<<"\n derived constructor";}
          ~derived()
          {    cout<<"\n derived destructor"; }
};
void main()
{
     derived od;
}
```

Output:

```
base constructor
derived constructor
derived destructor
base destructor
```

When object of derived class create then first constructor of base class calls then of derived. When object remove then first destructor of derived calls then of base.

Passing parameters to the base class constructor:

When base class contain parameterized constructor then we can pass parameters to base constructor at the time of object creation of derived.

Example: ***Program:u4p7.cpp***

```
#include<iostream.h>
class base
{
     protected:
          int a;
     public:
          base(int p)    //base class constructor
          { a=p; }
};
class derived: public base
{
     int b;
     public:
          derived(int p1,int p2): base(p1)
          {
               b=p2;
          }
```

```
                void showdata()
                {
                     cout<<"\nbase a="<<a;
                     cout<<"\nderived b="<<b;
                }
};
void main()
{
     derived od(10,20);
     od.showdata();
}
```

Output:

```
base a=10
derived b=20
```

It is clear that that using

```
derived(int p1, int p2):base(p1)
```

we can pass parameters to base constructor. Constructor header of derived receive all parameters and we can pass them to the base class.

Granting Access

When a base class is inherited as **private**, all public and protected members of that class become private members of the derived class. However, we can use one or more inherited members to their original access specification. It is possible using following syntax in derived class.

```
base::member
```

Example: ***Program:u4p8.cpp***

```
#include<iostream.h>
class base
{
     public:
          int b;
};
class derived: private base
{
    public:
          base::b;  //make b public again
};
void main()
{
     derived od;
     od.b=10;
     cout<<"\n base b="<<b;
}
```

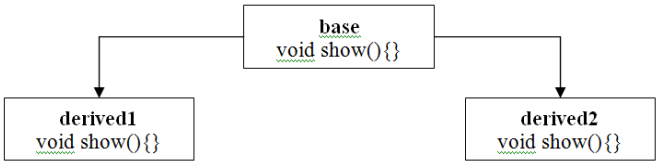
Output:

```
base b=10
```

In this example, base is derived private but we have to give grant public data b of base remains public in derived using *"base::b"*.

Virtual Function: (Run time polymorphism)

Run time polymorphism is possible using virtual function. In hierarchical inheritance, it may be happened that same name function headers are defined in base class as well as in all derived classes. This situations causes ambiguity when object of derived class invoked such function.



Dynamic binding:(Pointer to derived class)

Ambiguity can be avoided by dynamic binding of function. To create dynamic binding, address of derived object is assigned to base pointer. At runtime when base pointer call a function that defined in both classes (base as well as derived) then function of base class is bind up.

```
base *p;
derived od;
p=&od;
p→show();
```

If function is defined using *virtual* keyword in base class and same function is also defined in derived class then base pointer will call function of derived.

Example: ***Program:u4p9.cpp***

```
#include<iostream.h>
class base
{
     public:
          virtual void show()
          {
               cout<<"\nI m defined in base";
          }
};

class derived1: public base
{
     public:
          void show()
          {
               cout<<"\nI m defined in derived1";
          }
};
class derived2: public base
{
     public:
          void show()
          {
               cout<<"\nI m defined in derived2";
          }
};
```

```
void main()
{
     base *p;
     derived1 od1;
     p=&od1;
     p→show(); //I m defined in derived1
     derived2 od2;
     p=&od2;
     p→show();//I m defined in derived2
}
```

Output:

```
I m defined in derived1
I m defined in derived2
```

If *virtual* keyword removes then output will be:

```
I m defined in base
I m defined in base
```

In this program, if base pointer holds address of od1 then function of derived1 class will be invoked and if base pointer has address of od2 then function of derived2 class will be invoke. If *virtual* keyword removes then function of base class will be invoked whatever addresses hold by base pointer.

Pure virtual function: (abstract base class)

- **Pure virtual function:** If virtual function in base class does not defined with code and assign by 0 called pure virtual function.
- **Abstract base class:** A base class that contains pure virtual functions can't be used for creating objects, such class called abstract class. The purpose of creating abstract base class is to provide members to the derived classes.

Example: ***Program:u4p10.cpp***

```
#include<iostream.h>
class abstract
{
     public:
          virtual void show()=0; //pure virtual
};
class derived1: public abstract
{
     public:
          void show()
          {
               cout<<"\nI m defined in derived1";
          }
};
class derived2: public abstract
{
     public:
          void show()
          {
               cout<<"\nI m defined in derived2";
          }
```

```
};
void main()
{
      // abstract ob; error
      base *p;
      derived1 od1;
      p=&od1;
      p→show(); //I m defined in derived1

      derived2 od2;
      p=&od2;
      p→show();//I m defined in derived2
}
```

Output:

```
I m defined in derived1
I m defined in derived2
```

In this example, "*virtual void show()=0*;" is called pure virtual function and "*abstract*" called abstract base class because its object class can't be created and their functions are defined by derived classes.

Early vs. Late Binding

Early binding: (static or compile time binding)

Early binding occur at compile time. When we define any function in a program then that function is bind up to its all function call statements during compilation. This event occurs before running the program therefore it is called early binding.

Examples : normal function, function overloading and operator overloading functions.

Late binding: (dynamic or run time binding)

Late binding occur at run time of program. When we define any function in a program then that function is bind up to its all function call statements during execution of program. This event occurs after compiling the program therefore it is called late binding.

Examples of late bindings are virtual functions. It is possible using base pointer.

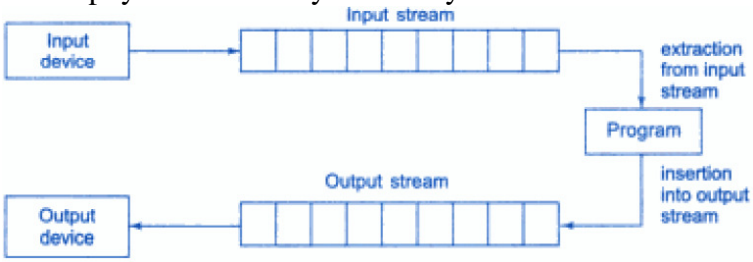
Unit-5

The C++ Input / Output system basics:

Every program takes some data as input and generates processed data as output. In C++ I/O operations is done using predefined streams and stream class.

C++ streams:

A stream is a sequence of bytes. C++ I/O system operates through streams. A *stream* acts as a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system.



It is clear that, a C++ program extracts byte from input stream and inserts bytes into input streams. Input stream reads data from input device (keyboard) and output stream sends output to the output device (monitor). Thus streams are used to create an interface between C++ program and I/O devices.

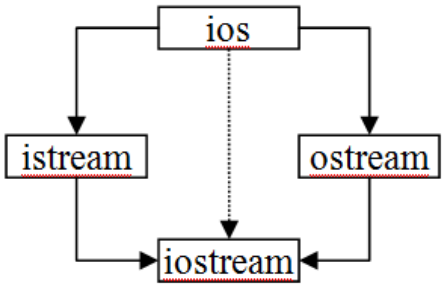
C++ predefined streams:

When a C++ program begins execution, four built-in streams are automatically opened. By default, the standard streams are used to communicate with the console. They are:

1. ***cin***: for standard input from keyboard
2. ***cout***: for standard output to the screen

The basic stream classes:

All C++ I/O operations are performed using console and file streams. All stream classes are declared in *iostream.h* header file. The hierarchy of I/O stream classes in iostream.h header file is following.



ios class is base of *istream* and *ostream* classes and these two class are base of *iostream* class. i*os* is declared virtual base class so that *iostream* uses one copy of *ios* members.

ios provides the basic supports for formatted and unformatted I/O operations. *istream* class provides facilities for input data and *ostream* provides facility for output. *iostream* provides facilities for handling both input and outputs.

For example *cout* stream used for output because it is defined using *ostream* class and cin is used for input because defined using *istream* class.

Unformatted I/O operations:

Overloaded operators >> and <<

>> operator is overloaded with *cin* object for input whereas << is overloaded with *cout* object for output operations. >> is overloaded in the *istream* class and << is overloaded in *ostream* class.

Reading data from keyboard: *cin>>var1>>var2>>.....>>varN;*
Display data on screen: *cout<<item1<<item2<<.....<<itemN;*

Example: ***Program:u5p1.cpp***

```
#include<iostream.h>
void main()
{
     int a,b;
     cout<<"\nInput two values:";
     cin>>a>>b;
     cout<<"\na="<<a<<"\tb="<<b;
}
```

Output:

```
Input two values: 10     20(Enter)
a=10 b=20
```

get() and put() function:

get() define in *istream* class used to read one character from keyboard by *cin* whereas put() define in *ostream* class used to show one character on screen by *cout*.

Example: ***Program:u5p2.cpp***

```
#include<iostream.h>
void main()
{
     char ch;
     cout<<"\nPress any key:";
     cin.get(ch);   //read character
     cout.put(ch);  //show on screen
}
```

Output:

```
Press any key: a(enter)
a
```

getline() and write() function:

getline() define in *istream* class used to read whole line of words that end with newline character (enter key) by *cin* whereas **write()** define in *ostream* class used to show characters up to given limit on screen by *cout*.

Example: ***Program:u5p3.cpp***

```
#include<iostream.h>
void main()
{
     char line[30];
     cout<<"\nInput a line of text:";
     cin.getline(line,30);     //read line
     cout.write(line,30);      //show 30 character
}
```

Output:

```
Input a line of text: www.LRsir.net(enter)
www.LRsir.net
```

Formatted console I/O operations

C++ supports a number of member functions for formatting outputs. These functions are classified into following categories.

1. Defined in *ios* class.
2. using manipulators
3. creating our own manipulators.

***ios* Formatted functions:** *ios* class contain a number of member functions used to format output. These functions can be invoked using *cout* stream because *ostream* is derived class of *ios* class.

1) ***cout.width(w):***

This member function of *ios* set the number of columns(w) for the output of only one item.

Example: ***Program:u5p4.cpp***

```
#include<iostream.h>
void main()
{
      cout.width(20);//set 10 columns
      cout<<"www.LRsir.net";
      cout.width(15);
      cout<<"open it";
}
output:
```

							w	w	w	.	L	R	s	i	r	.	n	e	t								o	p	e	n		i	t

The value www.LRsir.net is printed right justified in first 20 columns, Next item print in 15 columns.

2) ***cout.precision(d):***

This member function of ios set the number of digits(d) after decimal point of floating number.It is in effect until reset.(default precision digits are 6)

Example: ***Program:u5p5.cpp***

```
#include<iostream.h>
void main()
{
      cout.precision(3);
      cout<<2.12178<<"\n";
      cout<<3.5004<<"\n";
}
output:
2.122 (rounded)
3.5 (not show 00)
```

here precision shows only 3 digits afer decimal point.

3) ***cout.fill('*'):***

This member function of ios fills unused spaces by any character like *.It is in effect until reset.

Example: ***Program:u5p6.cpp***

```
#include<iostream.h>
void main()
{
      cout.fills('*');
      cout.width(20);//set 10 columns
      cout<<"www.LRsir.net";
      cout.width(15);
      cout<<"open it";
}
```

output:

*	*	*	*	*	*	*	w	w	w	.	L	R	s	i	r	.	n	e	t	*	*	*	*	*	*	*	*	o	p	e	n		i	t

Blank spaces are filled by *.

4) ***cout.setf():*** This is a member function of *ios* class that can be used for multiple formatted output. In this function we pass one or two flags as arguments.

- *cout.set(flag):* pass only one argument

flag	use
ios::showpos	Print + before positive number(+22)
ios::showpoint	Show . and zero(2.00)

- *cout.set(flag,group):* It is overloaded form where we pass two arguments.

flag	Group / bit flag	use
ios::left	ios::adjustfield	For left justification
ios::right	ios::adjustfield	For right justification
ios::scientific	ios::floatfield	Represent floating number like 5E-3

Example: ***Program:u5p7.cpp***

```
#include<iostream.h>
void main()
{
      cout.width(20);//set 10 columns
      cout.fills('*');
      cout.precision(2);
      cout.setf(ios::showpos);
      cout.setf(ios::showpoint);
      cout.setf(ios::left, ios::adjustfield);
      cout<<23;
}
```

output:

+	23	.	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

5) ***cout.unsetf(flag):*** this *ios* member is used to clear flag that set on previous statement.

Manipulators:

A set of functions defined in *iomanip.h* called manipulators used to formatted output. These functions can be used as in following form.

```
cout<<manip1<<manip2<<manip3<<item;

cout<<manip1<<item1<<manip2<<item2;
```

There are following some important manipulators-

1) ***setw(w)***: used to set number of columns(w) of formatted output.
2) ***setprecision(d)***: used to set number of digits(d) after decimal point.
3) ***Setfill('*')*:** used to fill blank space by a character like *.
4) ***endl:***used to set curser at new line.

Example: ***Program:u5p8.cpp***

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
      cout<<setprecision(2)<<12.136;
      cout<<endl<<setw(20)<<setfill('*')<<"www.LRsir.net";
}
```

Output:

```
12.14 (rounded)
*******www.LRsir.net
```

It is clear that manipulators can be used in *cout* statement to formatted output.

Creating Your Own Manipulator Functions

If one type of formatted sequence is required many times then every time we have to rewrite same sequence before showing items. This is very time expensive. We can also create our own manipulators using sequence of all required formats. It will be used in *cout* statement, similar to predefined manipulators.

Syntax:

```
ostream &manip-name(ostream &stream)
{
    // your code here
    return stream;
}
```

No argument is used when the manipulator is inserted in *cout* statement.

Example1: Set newline +10 columns + Fill blank by *

Program:u5p9.cpp

```
#include <iostream.h>
#include <iomanip.h>
ostream &userm(ostream &stream)
{
      stream<<"\n";
      stream.width(20);
      stream.fill('*');
      return stream;
}
```

```
void main()
{
     cout << userm << www.LRsir.net;
     cout << userm << "my website";
}
```

Output:

```
*  *  *  *  *  *  *  w  w  w  .  L  R  s  i  r  .  n  e  t
*  *  *  *  *  *  *  *  *  *  m  y  *  w  e  b  s  i  t  e
```

In this example, ***userm*** is user created manipulator that define three formats.

Miscellaneous

Templates

Using template, we can create only one class for any type of data members and define functions for any type of parameters. A template can be considered as a macro. We use template in place of all those data type that have to replace by other data types. For example, following program has a class A that defines integer data members, but we want work on float data members then we would be replace every integer type by float type.

For example: ***Program:temp1.cpp***

```
#include<iostream.h>
class A
{
     int a;
     int b;
     public:
          void getdata(int p1, int p2)
          {
               a=p1;
               b=p2;
          }
          void showdata()
          {
               cout<<"\na="<<a;
               cout<<"\tb="<<b;
          }
          int sum()
          {
               return a+b;
          }
};
 void main()
{
     int x=10, y=20,s;
     A ob;
     ob.getdata(x,y);
     ob.showdata();
     s=ob.sum();
     cout<<"\nSum="<<s;
}
```

Output:

```
a=10 b=20
Sum=30
```

Problem in this program:

In this program class has ***int*** data members and member function getdata() defines ***int*** parameters, sum() retrun ***int*** data. We have to redefine this class for any other type like ***float*.**

Solution: class templates and function templates.

Class template: A class created from a class template is called *template class*.

Class template for one parameter type: *template* keyword is used as a prefix before the class and T defines replacing data type in class.

- Syntax for defining template class:

```
template <class T>
class clsname
{
   Use T in place of data type
};
```

- Syntax of defining object of template class is

```
classname<datatype> objectname;
   A<int> ob1; or
   A<float> ob2;
```

Example of class template for one type: ***Program:temp2.cpp***

```
#include<iostream.h>
template <class T>
class A
{
     T a;
     T b;
     public:
          void getdata(T p1, T p2)
          {
               a=p1;
               b=p2;
          }
          void showdata()
          {
               cout<<"\na="<<a;
               cout<<"\tb="<<b;
          }
          T sum()
          {
               return a+b;
          }
};
void main()
{
     A<int> ob1;
     A<float> ob2;
     cout<<"\nworking with integer data:";
     ob1.getdata(10,20);
     ob1.showdata();
     int s1=ob1.sum();
     cout<<"\nSum="<<s1;
     cout<<"\nworking with float data:";
     Ob2.getdata(5.2,6.3);
```

```
        Ob2.showdata();
        float s2=ob2.sum();
        cout<<"\nSum="<<s2;
    }
```

Output:

```
working with integer data:
a=10 b=20
Sum=30

working with float data:
a=5.2     b=6.3
Sum=11.5
```

Now in this class A can be used for any data type where we have to define template T.

Class template for multiple parameter types:

T1, T2 etc can be used for multiple types.

- Syntax for defining template class for multiple types:

```
template <class T1,class T2,..>
class clsname
{
   Use T1,T2,..in place of data types
};
```

- Syntax of defining object of template class is

```
classname<type1,type2> objectname;
```

First int and second float: `A<int, float> ob1;`

Example of class template for two type:

Program:temp3.cpp

```
#include<iostream.h>
template <class T1, class T2>
class A
{
     T1 a;
     T2 b;
     public:
          void getdata(T1 p1, T2 p2)
          {
               a=p1;
               b=p2;
          }
          void showdata()
          {
               cout<<"\na="<<a;
               cout<<"\tb="<<b;
          }
          T2 sum()
          {
               return a+b;
          }
};
```

```
void main()
{
     A<int,float> ob1;
     cout<<"\nworking with multiple types:";
     ob1.getdata(10,20.2);
     ob1.showdata();
     float s=ob1.sum();
     cout<<"\nSum="<<s;
}
```

Output:

```
working with multiple types:
a=10 b=20.2
Sum=30.2
```

class A can be used for any two data types using template T1 and T2.

Function template: Like class template, we can also define function templates that could be used to create a family of functions with different argument types.

- Syntax for defining function template:

```
template <class T>
ftype fname(arguments of T type)
{
   Body of function used arguments of T type
};
```

Example of swapping of any two numbers of any types.

Program:temp4.cpp

```
#include<iostream.h>
template <class T>
void swap(T &x,T &y)
{
     T temp=x;
     x=y;
     y=temp;
}
void main()
{
     int a=10,b=20;
     float p=2.2,q=3.4;
     swap(a,b);//pass integers
     swap(p,q);//pass floats
     cout<<"\nInteger:a="<<a<<"\tb="<<b;
     cout<<"\nFloat:p="<<p<<"\tq="<<q;
}
```

Output:

```
Integer:a=20    b=10
Float:p=3.4     b=2.2
```

In this way using function template we can pass different types of parameters to one function.

File handling I/O operation in C++:

File is a collection of characters saved in secondary memory (hard disk) with some properties (attributes).

File attributes:

1. File name (Ex: myfile)
2. File extension (Ex:.txt)
3. Date and time of file creation & Modification(Ex: 1/24/16 7:22 pm)
4. Size of file(Ex: 2KB)
5. File access permission (Ex: Read / Write / Execute) etc.

File Operations:

1. Create and Open new file(loading)
2. Open existing file
3. Writing data to the file
4. Reading data from file
5. Searching data on file
6. Closing file

File types: (Binary file and Text file)

For the same content size of these two type of files are different because various character translation occurs in text mode file.

File stream classes and header file:

- *ifstream* for reading data from file,
- *ofstream* for writing data to the file
- *fstream* for read or write operation to the file

All these file stream classes are declared in ***fstream.h*** header file.

Opening and closing file: Before opening a file, we must first obtain streams.

1) ifstream fr; //to read file data
2) ofstream fw; //to write data in file
3) fstream frw; //for read and write data of file

fr, fw and frw are file stream used to open and read / write operation on file.

Syntax: `stream.open("filename",mode);`

Mode set file pointer position and type of file.

Mode	applicable_stream_class	use_and_ pointer_position
ios::in	istream/fstream	read and beginning
ios::out	ostream/fstream	write and beginning
ios::app	ostream/fstream	append and end
ios::ate	all	append and end but can move beg
ios::binary	all	open in binary mode (default is text)
ios::trunc	ostream	remove content of exist file

closing the file: After read / write operation on file, file should be unload from memory using close() function of file stream class.

Syntax: `stream.close();`

Example1: Open file only in read mode

```
ifstream fr;
fr.open("file1.txt");
//or
fr.open("file1.txt", ios::in);
if(!fr)
{
     cout<<"\n unable to open file";
     //handle error
}
---read operation-------
fr.close();
```

Example2: Open file only in write mode

```
ofstream fw;
fw.open("file1.txt");
//or
fw.open("file1.txt", ios::out);
if(!fw)
{
     cout<<"\n unable to open file";
     //handle error
}
---write operation-------
fw.close();
```

Example3: Open file only in append mode

```
ofstream fa;
fa.open("file1.txt",ios::app);
if(!fa)
{
     cout<<"\n unable to open file";
     //handle error
}
---append operation-------
fa.close();
```

Example4: Open file in binary and write mode

```
ofstream fw;
fw.open("file1.txt",ios::binary|ios::out);
if(!fw)
{
     cout<<"\n unable to open file";
     //handle error
}
---binary write operation-------
fw.close();
```

Using << and >>

Writing text to the file using <<:

Program:file1.cpp

```
#include<iostream.h>
#include<fstream.h>
void main()
{
     ofstream fw;
     fw.open("file1.txt");
     if(!fw)
     {
          cout<<"\n unable to open file";
          return;
     }
     fw<<"www.LRsir.net\n"; //write text to the file
     fw.close();
}
```

When this program executes then *file1.txt* creates in current location and content **www.LRsir.net** will be write into file using << insertion operator.

Reading text from the file>>:

Program:file2.cpp

```
#include<iostream.h>
#include<fstream.h>
void main()
{
     ifstream fr;
     fr.open("file1.txt");
     if(!fr)
     {
          cout<<"\n unable to open file";
          return;
     }
     char txt[50];
     fr>>txt; //read text from file
     cout<<txt;
     fw.close();
}
```

Output: www.LRsir.net (Read from file1.txt)

Using put() and get()

Writing character to the file in binary mode using put():

Program:file3.cpp

```
#include<iostream.h>
#include<fstream.h>
void main()
{
     ofstream fw;
     fw.open("file1.txt",ios::out|ios::binary);
```

```
        if(!fw)
        {
             cout<<"\n unable to open file";
             return;
        }
        char ch;
        cout<<"Type text: (Press ! to stop)";
        while(1)
        {
             cin.get(ch);
             if(ch=='!')break;
             fw.put(ch);
        }
        fw.close();
}
```

Contents are write to the file1 until ! mark.

Reading character from file in binary mode using get():

Program:file4.cpp

```
#include<iostream.h>
#include<fstream.h>
void main()
{
        ifstream fr;
        fr.open("file1.txt",ios::in|ios::binary);
        if(!fr)
        {
             cout<<"\n unable to open file";
             return;
        }
        char ch;
        while(fr) //false when eof
        {
             fr.get(ch);
             cout.put(ch);
        }
        fr.close();
}
```

Contents are write to the file1 until ! mark.

Using write() and read()

Writing different type of data to the file in binary mode using write():

Program:file5.cpp

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>

struct employee
{
        int id;
        char name[20];
        float sal;
};
```

```
void main()
{
     employee emp;
     emp.id=101;
     strcpy(emp.name,"Lokesh rathore";
     emp.sal=30000.34;
     ofstream fw;
     fw.open("file1.txt",ios::out|ios::binary);
     if(!fw)
     {
          cout<<"\n unable to open file";
          return;
     }
     fw.write((char*) &emp,sizeof(emp));
     fw.close();
}
```

Reading different type of data from file in binary mode using read():

Program:file6.cpp

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
struct employee
{
     int id;
     char name[20];
     float sal;
};
void main()
{
     employee emp;
     ifstream fr;
     fr.open("file1.txt",ios::in|ios::binary);
     if(!fr)
     {
          cout<<"\n unable to open file";
          return;
     }
     fr.read((char*) &emp,sizeof(emp));
     cout<<emp.id<<"\n";
     cout<<emp.name<<"\n";
     cout<<emp.sal<<"\n";

     fw.close();
}
```

Using getline()

Reading a line upto \n character from file in binary mode using getline():

Program:file7.cpp

```
#include<iostream.h>
#include<fstream.h>
void main()
{
```

```
        ifstream fr;
        fr.open("file1.txt");
        if(!fr)
        {
             cout<<"\n unable to open file";
             return;
        }
        char str[255];
        str=fr.getline();
        cout<<str;
        fr.close();
}
```

Contents of one line are read from the file1.

eof() function(detecting end of file): EOF is termination character of every file. When pointer reaches to EOF then eof() function gives true value.

```
if(fr.eof())
     cout<<"File ends";
```

ignore() function: ignore reading up to given number of character or given character. Following code ignores character until space is encounter or 10 characters have been read.

```
fr.ignore(10,' ');
```

peek() function: gives next character of read stream.

```
char ch=fr.peek();
```

ch has next character(may be EOF)

putback() function: gives last read character from a read stream.

```
char ch=fr.putback();
```

ch has previous character.

flush() function: force write stream data to the file before closing file.

```
fw.flush();
```

seekg() function: set read stream at given offset addess for next read operation.

```
fr.seekg(offset,origin);
origin may be
ios::beg  beginning of file
ios::cur  current location
ios::end  end of file
```

seekp() function: set write stream at given offset addess for next write operation.

```
fstream frw;
frw.open("file1.txt",ios::in|ios::out|ios::binary)
frw.seekp(offset,origin); //fstream
```

tellg() and tellp(): return current file position.

Exception handling:

At run time program can gives error due to some statement like m/n and n may be 0, opening file then program can be terminated from that point and remaining statements does not executed. It is called exception.

We can handle such exception in C++ program using *try-catch* block.

Syntax:

```
try
{
 //code creates exception
}
catch(type arg1)
{
     //error message with value
}
```

All the statements that create exception should be defined in *try* block. If exception occurred then it is thrown to the *catch* block with exception value. In this block we trace type of exception and show message to user. After then remaining statements continued for execution.

Example: ***Program:excep1.cpp***

```
#include<iostream.h>
void main()
{
     int c, a, b;
     cout<<"input two number:";
     cin>>a>>b;
     try
     {
          c=a/b;
          cout<<c;
     }
     catch(int i)
     {
          cout<<"\nException occurs";
          cout<<"\n Exception value is "<<i;
     }
}
```

Output1:

```
input two number: 10     0(enter)
Exception occurs
Exception value is 1
```

Output2:

```
input two number: 10     2(enter)
5
```

Remark: Not supported in TurboC++ Compiler.