Unit 1 Introduction to C++

1.1 Basic concepts, features, advantages and applications of OOP

1.1.1 Introduction to OOp's:

- Now a day's programming methodologies have changed dramatically since the invention of the computer. As increasing complexity of programs high-level languages were introduced that gave the programmer more tools with which he can handle complexity.
- OOP's is suitable for the huge applications which maps directly to a real world system/problem.
- In OOP there are the ideas of classes and objects. Using this powerful technique, large programs (like thousands of lines of codes) can be written, maintained and reused very efficiently. That is why OOP has the superiority over procedural programming.
- In OOP data of a class can be made private so that only member functions of the class can access the data. Due to this data hiding principle programmer can build a secure program.
- With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated and the use of existing class is extended. This saves time and cost of program.
- Polymorphism means many forms with this the same function or operator can be used for different purposes. It helps to manage software complexity easily.

1.1.2 Concepts of OOp's:

1. **Class:**

- A class is a user defined data type which contains Data Members and Member Functions.
- When defining a class new Abstract Data type is created.
- The variables declared within the class are called "Data Members".
- The Functions declared within the class are called "Member Functions".
- In class data and its associated functions bind together. It also called encapsulation.
- We can define class members as private, public or protected by default they are private.

- Once a class is created then we can create any number of objects.

2. Object :

- Objects are the basic run- time entities in an object oriented system i.e. an instance of a class is called as Object.
- An object in nothing but variable, whose data type is class
- They may represent a Person, Place , Bank etc.

3. Data Encapsulation :

- The method of combining the Data and Functions into single unit is called "Data Encapsulation".
- It is a hiding the details of internal data members of an object or to restrict direct data access to the class members is called "Data Hiding" or "Information Hiding".
- It helps in reducing program complexity.

a) Data Abstraction :

Abstraction concept is used in class, hence class is also known as Abstract Data Type. In abstraction only those properties and methods are accessed which are require to user.

b) Data Hiding :

The members of the class are either private or public when the members are private those members can be used with in the class. This is called as Data Hiding.

4. Inheritance :

- The mechanism of deriving new class from existing one is called Inheritance.
- Inheritance is a process by which object of one class acquire the properties of object of other class
- The OOP's inheritance stands for reusability, this means that additional features can be added
- to an existing class without modifying it.

Types of Inheritance:

1) Single Inheritance

2) Multilevel Inheritance

3) Multiple Inheritance

4) Hierarchical Inheritance

5) Hybrid Inheritance\

5. Polymorphism :

- Polymorphism means ability to take more than one form
- Polymorphism is extensively used in implementing Inheritance
- In OOP's Polymorphism refer to fact that a single operation can have a different behavior in different object.
- In other words ,different object reacts differently to the same message

Types of Polymorphism :

1) **Compile-Time Polymorphism (Static/Early Binding)**

a) Function Overloading

b)Operator Overloading

2) **Run-Time Polymorphism (Dynamic/Late Binding)**

a)Virtual function Overloading

1.1.3 Features of OOp's :

- Emphasis on data rather than procedure.
- Programs are subdivided into Objects.
- Data is hidden and is restricted by direct access from external functions.
- Object may communicate through each other with functions.
- New data and functions are easily added.
- It facilitates reusable code that can save lot of time.

1.1.4 Advantages of OOp's :

- Introduces concept of Inheritance which is useful in extending the properties of base class into child class along with its existing methods which increases code reusability, helps in reduction of code and minimizes debugging effort within a code.
- Data Encapsulation which is a data hiding property within a class. Encapsulation protects an object from external access by clients and also it provides security of the data.
- In polymorphism same function name and operator can be used for different purposes. This helps to manage software complexity easily.
- Larger problems can be divided into small parts. It is easy to partition the work in a project based on objects.
- Multiple objects can be co-exist in class without any interference i.e. every object has its own separate data members and member functions.
- OOP provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has a clearly defined interface.

1.2 Introduction, applications and features of C++

1.2.1 Introduction to C++ :

- C++ is an object oriented programming language.
- C++ was developed by “Bjarne Stroustrup” at AT & T’s (American Telecom and Telegraph) Bell Laboratories in USA 1980.
- C++ is an extension of C with major feature addition of the class construct feature.
- The most important feature that C++ adds to C are Classes, Inheritance, Function Overloading and Operator Overloading
- These features enable to create Abstract Data Type. Inherits properties from existing Data Type & support Polymorphism

1.2.2 Applications of C++

- It is a Superset of C. C++ is an incremented version C.
- C++ added with the facilities are classes, inheritance, function overloading and operator overloading

- It allows to create abstract class and to inherits properties of existing class
- Any real life application system such as editor, compiler, database communication systems can be built by C++.
- C++ programs can be easily implemented, maintained and expanded.
- C++ used in to develop operating systems, video games, compilers.
- C++ is used extensively in embedded and real-time systems.

1.2.3 Features of C++

- C++ uses simple English language which helps to developer to easily understand and write program.
- C++ provides the features of Object-oriented programming
- Its strength is variety of built-in-functions provided by C library which helps to develop any complex program easily.
- C++ supports dynamic memory allocation and also it releases the allocated memory any time.
- C++ program is multi-paradigm means it follows three paradigms Generic, Imperative, Object-Oriented.
- C++ have a Exception handling mechanism to handle run time errors

1.3 Input and Output operator in C++

1.3.1 Input Operator (Extraction Operator ">>") :

e.g. cin>>a>>b;

- cin is also an object that is associated with the standard input stream, that is the keyboard
- The Right Shift Operator ">>" is called the extraction operator or get from operator.
- It works similar to %s speacifier used with scanf() function .
- It is used to send input from the keyboard to the variable.

1.3.2 Output Operator (Insertion Operator "<<") :

e.g. cout<< "This is my first c++ Program"

- It display above string on screen.

- The Identifier cout is a predefined object that represents the standard output stream in c++.
- The Left Shift Operator "<<" is called the insertion operator or put to operator.
- It inserts the contents of the variable on it's right to the object on it's left.

1.4 A sample C++ program

Documentation Section	/* Program developed by.....*/
Link Section	#include<iostream.h> #include<conio.h>

Class Declaration Section
(Data Members & Member Functions)
	class rectangle
	{
	int area,l,b;
	public:
	void getadata()
	{
	cout<<"Enter Length and Breadth";
	cin>>l>>b;
	}
	void display()
	{
	area=l*b;
	cout<<"Area of Rectangle"<<area;
	} };

main() function section
{
Declaration Section
---------- 	
Executable Section
--------- 	
}
	void main()
	{
	rectangle x;
	x.getdata();
	x.display();
	getch();
	}

Unit 2 Beginning with C++

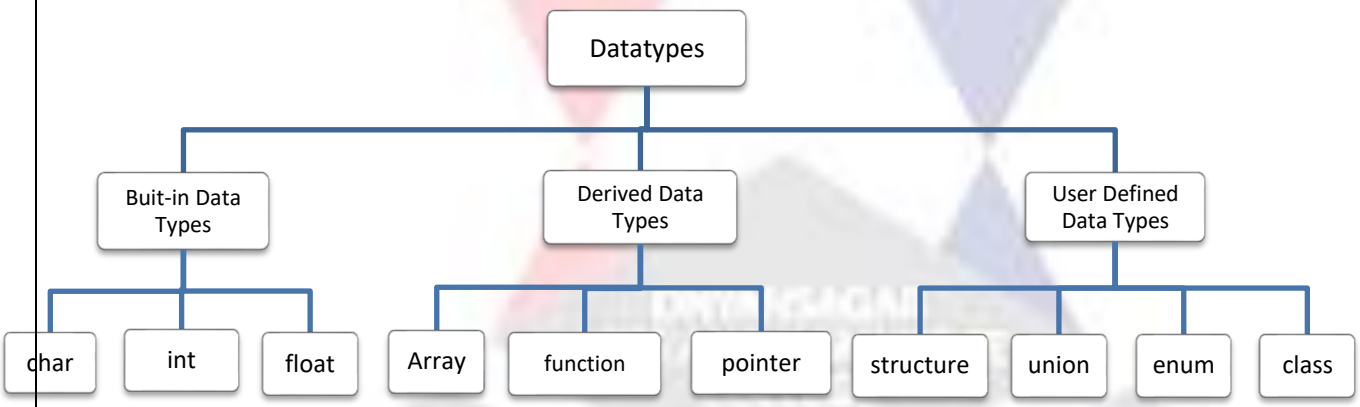
2.1 Data type and Keywords :

2.1.1 Data Types :

Every Program works on Data. Programming language provides way to store data with its type. When variable is declared you have to specify that what type of data it can contain.

The Data types specify that size and which type of value stored in that variable.

C language is rich in its Data type.



A. Built-in Data types :

C++ compiler supports all the buit-in data types

1. Character Data type (char) :

It stores a single character. A single character can be defined as char type data.

Data type	Range	Byte
char	-128 to +127	1
unsigned char	0 to 255	1

2. Data type int :

It stores integer or whole numbers.

Data type	Range	Byte
int	-32768 to 32767	2
unsigned int	0 to65535	2
short int	-32768 to 32767	2
unsigned short int	0 to65535	2
long int	-2147483648 to2147483647	4
unsigned long int	O to 4294967295	4

3. Data type float :

It stores floating point or real numbers.

Data type	Range	Byte
float	3.4e-38 to 3.4e+38	4
double	1.7e-308 to 1.7e+308	8
long double	3.4e-4932 to 1.1e+4932	10

B. Derived Data types :

1. Array :

An array is collection of data items of the same data type. An array is fixed-size sequenced collection of elements of the same data type.

Syntax :

data type arrayname[size];

Example :

int rollno[3];

float marks[5];

char name[30];

2. Structure :

Structure is a derived data type. A structure contains a number of data types grouped together. We make use of structure to represent a collection of data items of different types.

Example :

The employee data of people like name, address, salary. etc.

The data of books like bookno ,name,author and price etc.

The general form of a structure definition is

struct structname

```
{
        Datatype member1;
        Datatype member2;
        ------------
        ------------
        ------------
};
```

3. Data type enum :

A user defined data type enumerated data type is along with it's set of identifiers can be created by following declaration

Example :

enum datatype name(const1,const2,.....,constn);

enum datatype daysofweek(sun,mon,tue,wed,thu,fri,sat);

Data type void :

The void data type has no values. It is usually to specify the type of function. The type of function is void means it doesn't return any value to calling function.

Example :

void message ( );

2.2 Keywords :

Keywords are reserved words. They have fixed and predefined meaning and these meaning cannot be changed.

The keywords cannot be used as variable name because which is not allowed in C++.

Keywords help in building blocks of program.

There are only 62 keywords available in C++.

A list of 32 Keywords in C++ Language which are available in C language are given below.					
auto	break	case	char	const	continue
double	else	enum	extern	float	for
int	long	register	return	short	signed
struct	switch	typedef	union	unsigned	default
do	if	Static	while	void	volatile
goto	Sizeof				
A list of 30 Keywords in C++ Language which are not available in C language are given below.					
asm	dynamic_cast	namespace	reinterpret_cast	bool	delete
explicit	new	static_cast	false	catch	mutable
operator	template	friend	private	class	protected
this	inline	public	throw	const_cast	true
try	typeid	typename	using	virtual	wchar_t

Unit 3 :Classes and Objects

3.1 Structure and class, Class, Object

6. CLASS:

- A class is a user defined datatype which contains Data Members and Member Functions. It encapsulates Data Members and Member Functions into the class.
- When defining a class new Abstract Data type is created that is treated like built-in function.
- The variables which are declared within the class are called “Data Members”.
- The Function which are declared within the class are called “Member Functions”
- Class is a way to bind data and its associated functions together.
- The Members of a class can be private, public or protected.
- The by default class members are private.
- Once a class is created then we can create any number of objects.
- Class can be created with the keyword class. Its name is user defined type name.
- As per the OOP’s principal Data Members should be private and Member Functions should be public. Only the Member Functions have to access the private data of class.

Syntax for class definition:

```
classclass name
        {
private:
Data Member;
Member Function;
Public:
        Data Member;
          Member Function;
        };
```

Example:

```
class employee
        {
        intempid ;
        charename [30];
```

```
    public :
    getdata();
        display();
};
```

7. OBJECT:

- An object in nothing but variable,whose data type is class.
- Objects are the basic run- time entitiesin an object oriented system i.e. an instance of a class is called as Object.
- They may represent a Person, Place , Bank etc.

Syntax :

class name object name;

Example : student x;

Accessing members of class using object like.

x.getdata();

x.display();

Here the dot (.) operator is called as member access operator.

3.2 Access specifiers, defining data member

ACCESS SPECIFIERS/ MODIFIERS:

- The body and class contain Data Members and Member Functions.
- They are grouped under different sections namely private, public and protected which are known as "Visibility Modes" or "Access Specifiers ".
- Access Specifiers restricts the access to class members. This mechanism is used to implement in"Encapsulation'' or "Data Hiding" in C++.

1) private:

When the members of a class i.e. "Data Members'' and "Member Functions" are declared in private section of a class then those members can be only be accessed by member function and friends of that class.
The private Data Members are not accessible outside the class.
All the Members of a class are by default private.

2) **public:**

When the members of a class i.e. "Data Members" and "Member functions" are declared in public section of a class then those members can be accessed by outside the class or everyone.

3) **protected:**

Members declared in a protected section of a class is very similar to private member and they can be accessed in child classes which are called derived classes in Inheritance.

When the Data Members needs to be used in derived class then they are declared as protected

3.3 Defining member functions inside and outside class definition.

DEFINING MEMBER FUNCTIONS:

- A function which is declared as member of a class is called "Member Function''.
- Member function are mostly specified as public because they have to called from Outside the class either in program or function
- It can access all the data members of a class which are declared as private
- Member function are designed to implement the operations allowed on the "Data Members" of a class.

Members function can be defined in two ways

1) Inside the class definition
2) Outside the class definition

1) Inside the class definition :

- It is possible to define a function inside a class.
- It is treated as like “Inline Function” normally only small function defined insidethe class.

Example:

```
#include<iostream.h>
#include<conio.h>
class swap
{
        int a,b,c;
        public:
        void getdata()
        {
                cout<<"Enter any two numbers";
                cin>>a>>b;
        }
        void display()
        {
                c=a;
                a=b;
                b=c;
                cout<<"After Swapping Value of First No.="<<a<<endl;
                cout<<"After Swapping Value of Second No.="<<c;
        }
};
void main()
{
        swap x;
        clrscr();
        x.getdata();
        x.display();
        getch();
}
```

2) Outside the class definition :

Scope Resolution Operator [::]

- Member functions can also be defined outside the classboundaries using Scope Resolution operator (::)

- The definition offunction is very much similar to the normal function.

Example:

```
#include<iostream.h>
#include<conio.h>
class result
{
        ints1,s2,s3,total;
        float per;
        public:
        void getdata();
        void display();
}
void result :: getdata()
        {
                cout<<"Enter marks of Three Subjects";
                cin>>s1>>s2>>s3;
        }
voidresult::display()
        {
                total=s1+s2+s3;
                per=total/3;
                cout<<"Total Marks="<<total<<endl;
                cout<<"Percentage="<<per;

        }
};
void main()
{
        result x;
        clrscr();
        x.getdata();
        x.display();
        getch();
}
```

3.4 SIMPLE C++ PROGRAM USING CLASS

A SAMPLE C++ PROGRAM WITH CLASS

```
#include<iostream.h>
#include<conio.h>
class add
{
        int a,b,c;
        public:
        void getdata()
        {
                cout<<"Enter any two numbers";
                cin>>a>>b;
        }
        void display()
        {
                c=a+b;
                cout<<"addition="<<c;
        }
};
void main()
{
        add x;
        clrscr();
        x.getdata();
        x.display();
        getch();
}
```

3.5 Memory allocation for objects

MEMORY ALLOCATION FOR OBJECTS

- When a class is declared no storage is allocated for Data Members.
- Memory is allocated for objects when they are declared.
- For the member function of class memory is allocated only once when they are defined in the class.

- All objects of class share the same member functions.But when an object is created then separate memory space is allocated for the data members of that object because data members of each object has different values.

3.6 STATIC DATA MEMBERS AND STATIC MEMBER FUNCTIONS

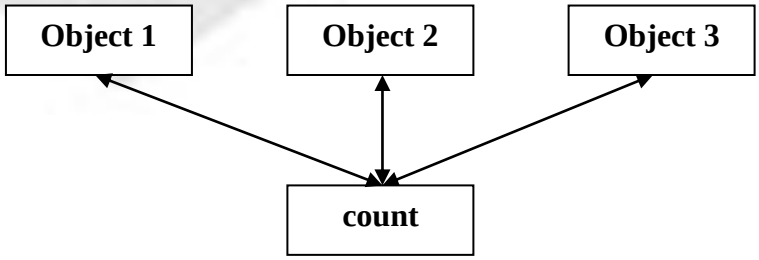
- In object oriented programming for each object a separate storage will be allocated for the data members and common storage for all the member functions.
- In some cases it is necessary to share the data member for all the objects at that time we use static data members.
- In C++ you can declared Data Member and Member Function as a static.
- These members belongs to class hence it is called as class members.
- All the objects of class can share same copy of static member.

STATIC DATA MEMBERS

- Static data members are common for the entire class and not for each object of class.
- Static data members can be declared by using reserved word static.
- Static data members must have a global declaration.
- The default initial value of the static data member is zero and it can't allow other initializations.
- Static members belongs to class hence it is called class variable.
- In the case of static data members only one copy of the data is created and it will be shared by all objects.
- We can access static members through scope resolution operator(::) to the class name.
- Static data member will have same value for all the objects. Hence if one object alters the value of static data members. It's altered value will be available for all other objects.

Syntax :

```
Class classname
{
------------
staticdatatypedatamember;
------------
}
```



STATIC MEMBER FUNCTION

- Like static data member we can also have static member function.
- A static member function can only access static members of same class.
- A static member function cannot access non static members.
- A static member function can be called using the class name and scope resolution operator.
- We can call static member function through scope resolution operator (::) to the class name. There is no need of class object to call static member function
- Static member function call like

  Class name :: function name(argument list);

```
#include<iostream.h>
#include<conio.h>
class patient
{
        int pid;
        char name[30],desease[20];
        staticchar doctor[30];
        public:
        void getdata()
        {
                cout<<"Enter Patient ID, Name and Desease";
                cin>>pid>>name>>desease;
        }
        void display()
        {
                cout<<"Patient Id="<<pid<<endl;
                cout<<"Name="<<name<<endl;
                cout<<"Desease="<<desease<<endl;
        }
        staticvoid showdoctor()
        {
                cout<<"Doctor Name="<<doctor<<endl;
        }
};
char patient::doctor[30]="Dr. Sandip Mahamuni";
void main()
{
```

```
        patient x;
        int n,i;
        clrscr();
        x.getdata();
        x.display();
        patient::showdoctor();
        getch();
}
```

UNIT 4 : Constructor and Destructor

CONSTRUCTOR

- A constructor is a special member function which is use to initialize the data members of a class.
- It is special because it's name is same as the class name which it's belongs.
- It is called constructor because it construct the values of data members of the class.
- It should be declare in public section.
- Constructor function automatically called whenever an object of that class is created.
- It doesn't have a return type, it is not even void type, therefore they can't return value.
- They can't be inherited. Through a derived class can call the base class constructor.
- A constructor cannot be static or virtual.
- Any number of constructor can be defined in a class and such a set of constructor implements a concept called constructor overloading

A Constructor is declared as follows:

```
class value
        {
                int a,b,c;
                public:
                  value(void);        //Constructor Declare
                  display();
           };
        void value :: value(void)   // Constructor Defined
{
                  a=0;
                  b=0;
           }
```

Syntax :

```
class  classname
{
private:
private members;
public:
classname()
{
          //constructor body definition
}
}
```

TYPES OF CONSTRUCTOR

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Dynamic Constructor**

1. DEFAULT CONSTRUCTOR :-

A constructor that accept no parameters is called as the Default constructor.

```
#include<iostream.h>
#include<conio.h>
class add
        {
                int a,b,c;
                public:
                add()
{
                  a=5;
                  b=7;
         }
         void display()
                {
                  c=a+b;
                  cout<<“Addition=“<<c;
                }
};
void main()
```

```
{
        add x;
        x.display();
        getch();
}
```

2. PARAMETERIZED CONSTRUCTOR :

A constructor that accept parameters is called as the parameterized constructor.Thus the parameterized constructor has no argument list written in the brackets associated with function definition header line.

```
#include<iostream.h>
#include<conio.h>
class multi
{
int a,b,m;
public:
multi(int x,int y)
{
```

```
a=x;
b=y;
}
void display()
{
m=a*b;
cout<<"multiplication="<<m;
}
};
void main()
{
multi x(5,7);
clrscr();
x.display();
getch();
}
```

8. COPY CONSTRUCTOR :

- A Copy constructor is used to declare and initialize an object with the content of another object. There are two methods to declare the copy constructor.
- The copy constructor uses object of the same class to initialize another object of the class.

Method 1

class name object1(object2); E.g. add y(x);

Method 2

class name object1=object2; E.g. add y=x;

```
#include<iostream.h>
#include<conio.h>
class add
        {
            int a,b,c;
            public:
add (int x , int y)
         {
            a=x;
            b=y;
         }
void display()
        {
          c=a+b;
          cout<<"Addition="<<c;
         }
};
void main()
            {
            add x(10,20), y(x);
                x.display();
                y.display();
                getch();
            }
```

9. DYNAMIC CONSTRUCTOR :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class sample
{
char *s;
int length;
public:
sample(char * name)
{
length=strlen(name);
s=new char[length+1];
strcpy(s,name);
}
void display()
{
cout<<s;
}
};
void main()
{
samplex("govind");
clrscr();
x.display();
getch();
```

```
}
```

DESTRUCTOR

- A destructor is a function which is use to destroy the object.
- It's name is same as the class name, but it is preceded by ~ sign
- It is called destructor because it destroys the object and releases the memory for further use.
- Destructor is automatically called there is no need to call it.
- It doesn't have a return type, it is not even void type.

A destructor is declared as follows:

```
#include<iostream.h>
#include<conio.h>
class value
        {
                inta,b,c;
                public:
        void value()   // Constructor Defined
{
                  a=5;
                  b=7;
           }
void display()
          {
```

Syntax :

```
Class calssname
{
private:
        ……….
public:
classname( )
        {
                Constructor body definition
        }

~classname( )
        {
                Destructor body definition
        }
};
```

```
        c=a+b;
    cout<<"Addition="<<c;
      }
~ value( )
        {
Cout<<"Destructor is called";
        }

};
void main()
            {
             add x(10,20);
             x.display();
             getch();
             }
```

INLINEFUNCTION:

- To eliminate the cost of calls to small function, C++ proposes a new function called as inline function.
- A inline function is a function, that is expanded in a line when it is invoked.
- When we prefix the keyword inline, the function becomes an inline function.
- All inline functions must be defined before they are called.
- Inline keyword just sends a request, not a command to the compiler. The compiler ma ignore this request if the function definition is too long too complicated and compiles the function as a normal function.
- Inline expansion makes a program to run faster.

Inline expansion may not work with :

- loop, a switch, or goto exists.
- For functions not returning values, if a statement exists.
- If function contains static variables.
- If inline function are recursive.

Syntax :

```
inlineReturntypeFunctionName(parameters)
        {
                // body of main function
         }
```

Example:

```
#include<iostream.h>
#include<conio.h>
classstudent
{
        ints1,s3,s3,total;
        float per;
        public:
        void getdata();
        void display();
}
inline void student :: getdata()
        {
                cout<<"Enter Marks of three subjects";
                cin>>s1>>s2>>s3;
        }
inline voidstudent::display()
        {
                total=s1+s2+s3;
                per=total/3;
                cout<<total<<endl<<per;
        }
};
void main()
{
        student x;
        clrscr();
        x.getdata();
        x.display();
        getch();
}
```

UNIT 5 : Inheritance

INHERITANCE

- The mechanism of deriving new class from existing one is called Inheritance.
- The old class is referred as Base Class and new class is referred as Derived Class.
- Inheritance is a process by which object of one class acquire the properties of object of other class
- C++ strongly supports the concept of reusability. This means that additional features can be added to an existing class without modifying it.
- Once a class has been written and tested, it can be adopted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of existing once.
- The reuse of class that has already been tested , debugged and use many times , can save the efforts of developing and testing the same again

TYPES OF INHERITANCE :

1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

Defining derived classes

```
class Derived Class Name : Visibility Mode Base Class Name
 {
        data members;
        member functions;
 }
```

Example

```
class add
{
        int a,b,c;
        public:
        getdata()
{
            cout<<"Enter any two numbers";
```

```
                    cin>>a>>b;
                  }
};
class result : public add
{
                Public:
                display()
                    {
void getdata();
                      c=a+b;
        cout<<"Addition="<<c;
                    }
};
```

1. SINGLE INHERITANCE:

Deriving the properties of one class into another class is called as single Inheritance.

In Single Inheritance we have only one Base class and one Derived Class.

Example.

```
        class add
{
                int a,b,c;
                public:
                getdata()
{
                    cout<<"Enter any two numbers";
                    cin>>a>>b;
                  }
};
class result : public add
 {
                  Public:
                  display()
        {
void getdata();
                     c=a+b;
                     cout<<"Addition="<<c;
                    }
};
        void main()
            {
```

```
        result x;
        x.display();
        getch();
}
```

2. MULTILEVEL INHERITANCE:

Multilevel Inheritance has one base class and more than one derived classes. A derived class which is derived from another derived class is called as multilevel Inheritance.

Example

```
class  student
{
        int rollno;
        char name[30];
        protected:
        getdata()
{
cout<<"Enter roll no and name of student;
cin>>rollno>>name;
        }
};
```

```
        class  marks : public student
 {
                int sub1,sub2,sub3;
                public:
                getmark()
{
                  getdata();
                  cout<<"Enter marks of three subjects;
                  cin>>sub1>>sub2>>sub3;
        }
};

class result : public marks
              {
                int total;
                Public:
                 display()
                   {
void getmark();
        total=sub1+sub2+sub3;
        cout<<"Total Marks="<<total;
                    }
};
        void main()
              {
        result x;
        x.display();
        getch();
               }
```

3. MULTIPLE INHERITANCE:

A derived class which is derived from more than one classes is called as multiple Inheritance. Multiple Inheritance has more than one base class and one derived class. This derived class has the properties of all the base classes as well as derived class.

Example

```
        class  rectangle
```

```
{
                int length,breadth;
                public:
                getdata()
{
        cout<<"Enter  a length and breadth of rectangle;
        cin>>length>>breadth;
                    }
};
        class  triangle
{
                 int base,height;
                 public:
                 readdata()
{
                     getdata();
                     cout<<"Enter  a base and height of triangle;
                     cin>>base>>height;
                   }
};

class area : public rectangle,public triangle
              {
                 int arect,atri
                 Public:
                  display()
                    {
                       void getdata();
                       void readdata();
                       arect=length*breadth;
                       cout<<"Area of Rectangle="<<arect;
                       atri=0.5*base*height;
                       cout<<"Area of triangle="<<atri;
                    }
                  };
        void main()
              {
                area x;
                x.display();
                getch();
              }
```

4. HIERARCHICAL (TREE) INHERITANCE:

A Tree structure of inheritance is called as Hierarchical Inheritance.

Example

```
class  number
{
        int a,b;
        public:
        getdata()
{
          cout<<"Enter  any two numbers;
          cin>>a>>b;
         }
};
class  add : public number
{
         int c;
         public:
         display()
{
            getdata();
            c=a+b;
           cout<<"Addition ="<<c;
          }
};

class  sub : public number
          {
   int s;
         Public:
          show()
            {
              void getdata();

              s=a-b;
              cout<<"Substraction="<<s;

            }
```

```
        };
void main()
      {
        add x;
        sub y;
        x.display();
        y.show();
       getch();
      }
```

10.HYBRID INHERITANCE:

It is a combination of Multilevel and Multiple Iheritance.

```
#include<iostream.h>
#include<conio.h>
class theory
{
protected:
int t1,t2;
public:
void gettheory()
{
cout<<"enter a marks of two theory subject";
cin>>t1>>t2;
}
};
class practicle:public theory
{
protected:
int p1,p2;
public:
void getpracticle()
{
gettheory();
cout<<"enter a marks of two practicle subject";
cin>>p1>>p2;
}
};
```

```
class sports
{
protected:
int s1,s2;
public:
void getsports()
{
cout<<"enter a marks of two sports subject";
cin>>s1>>s2;
}
};
class result:public sports, practicle
{
int t;
float per;
public:
void display()
{
getpracticle();
getsports();
t=t1+t2+p1+p2+s1+s2;
cout<<"total marks="<<t;
per=t/6;
cout<<"percentage="<<per;
}
};
void main()
{
result x;
clrscr();
x.display();
getch();
}
```

Polymorphism

FUNCTION OVERLOADING:

- Function overloading refers to creating multiple functions with the same name but different parameter list.
- Different parameter list can be with reference to the number of parameters or the type parameters.
- Since, for overloaded functions the function have same name,a function can be called with common name but the function that will be invoked is based on the parameter type and member of parameters.

```
#include<iostream.h>
#include<conio.h>
class overload
{
public:
void volume(int s)
{
asquare=s*s;
cout<”Area of Square”<<asquare;

}
void volume(double pi,int r)
{
acircle=pi*r*r;
cout<”area of circle”<<acircle;

}
void volume(long l,intb,int h)
{
vcyl=l*b*h;
cout<”volume of Cylinder”<<vcyl;
}
};
void main()
{
overload x;
clrscr();
x. volume (5);
x. volume(3.14,7);
x.volume(9.2,3,4);
```

```
getch();
}
```

OPERATOR OVERLOADING:

The mechanism of assigning a new meaning to an already existing operator is called operator overloading.

Necessary rules for operator loading :

- Only existing operators can be given a new meaning i.e. only existing operators can be overloaded.
- Even after overloading the basic meaning of the operator remains the same.
- Overloaded operators follow the same syntax as that of original operator.
- Unary operators overloaded by means of member function can't accept parameter.
- Unary operators overloaded by means friend function can accept upto one parameter.
- Binary operators overloaded by means of friend function can accept upto one or two parameter.
  - **Some of the operators cannot be overloaded.**
  - Sizeof ( )
  - Member Operator (.)
  - Pointer to Member Operator(.*)
  - Scope Resolution Operator(: :)
  - Conditional Operator (? :)
  - **Some of operators cannot be overloaded by friend function**
  - Assignment Operator (=)
  - Function call Operator(())
  - Subscripting Operator ([ ])
  - Class member Access Operator (→)

==========

Q. PROGRAM TO OVERLOAD OPERATOR (*)

```
#include<iostream.h>
#include<conio.h>
class multi
{
int a;
```

```
public:
void getdata()
{
        cout<<"enter a no";
        cin>>a;
}
void display()
{
        cout<<"answer"<<a;
}
friend multi operator *(multi x,multi y)
};
multi operator *(multi x,multi y)
{
        multi z;
        z.a=x.a*y.a;
        return(z);
}
void main()
{
        multi x,y,z;
        clrscr();
        x.getdata();
        y.getdata();
        z=x*y;
        z.display();
        getch();
}
```

==========

Q. PROGRAM TO OVERLOAD OPERATOR (+) TO ADD TWO NUMBERS

```
#include<iostream.h>
#include<conio.h>
class add
{
int a;
public:
void getdata()
{
        cout<<"enter a no";
```

```
        cin>>a;
}
void display()
{
        cout<<"answer"<<a;
}
friend add operator+(add,add);
};
add operator+(add x,add y)
{
        add z;
        z.a=x.a+y.a;
        return(z);
}
void main()
{
        add x,y,z;
        clrscr();
        x.getdata();
        y.getdata();
        z=x+y;
        z.display();
        getch();
}
```

==========

Q. PROGRAM TO OVERLOAD OPERATOR (==)

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string
{
char str[20];
public:
void getdata()
{
        cout<<"enter a strig";
        cin>>str;
}
friend void operator==(string,string)
};
```

```
void operator==(string x,string y)
{
if(strcmp(x.str,y.str)==0)
{
        cout<<"both strings are equal";
}
else
{
        cout<<"both strings are not equal";
}
}
void main()
{
        string x,y;
        clrscr();
        x.getdata();
        y.getdata();
        x==y;
        getch();
}
```

==========

Q. PROGRAM TO OVERLOAD OPERATOR (++)

```
#include<iostream.h>
#include<conio.h>
class increment
{
int a,b;
public:
void getdata()
{
        cout<<"enter two no";
        cin>>a>>b;
}
void display()
{
        cout<<a<<b;
}
void operator++()
{
        a++;
        b++;
```

```
}
};
void main()
{
        increment x;
        clrscr();
        x.getdata();
        ++x;
        x.display();
        getch();
}
```

Working with files

Input/Output with files

C++ provides the following classes to perform output and input of characters to/from files:

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therfore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file
```

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

But let's go step by step:

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function open():

open (filename, mode);

Where filename is a null-terminated character sequence of type const char * (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open():

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectivelly assumed, even if a mode that does not include them is passed as second argument to the open() member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the open() member function and has the exact same parameters as this member. Therefor, we could also have declared the previous myfile object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is_open() with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

Text files

Text file streams are those where we do not include the ios::binary flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with cout:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Data input from a file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
```

```
This is a line.
This is another line.
```

```
 if (myfile.is_open())
 {
   while (! myfile.eof() )
   {
     getline (myfile,line);
     cout << line << endl;
   }
   myfile.close();
 }

 else cout << "Unable to open file";

 return 0;
}
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called eof() that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed myfile.eof() becomes true (i.e., the end of the file has been reached).

Template

Class Templates

Static Members

A templated class can have static members

Each type has an associated set of static members

Declarations

static T my Function (args) { myFunction body}

static int myValue // must be defined outside of the class

Definition

template <class T> int MyClass<T>::myValue = aValue;

Class Templates

Forward Declarations…..

template <class T> class MyClass // forward reference

….

….and Friends

```
template <class T> class MyClass
{
    friend class YourClass<T>;
    public:
```

Declaring an Instance

template <class T> class MyClass

{

public:

……

}

MyClass <char> myInstance;

Observe the use of <type> as part of the class name

Function and Class Templates

- Specialization

May need to a special version of a template to handle types not easily included in an existing template.

Function

returnType functionName (args) { function body }

Member Function

className < specialized type> functionMember (args)

{ function body }

Class

class className < specialized type> { class body };

- Explicit Instantiation

May wish to create a specific version of a function or class

Function

returnType functionName (args);

Class

class className < explicit type>