

Chapter 1

Thinking Object Oriented

Since the invention of computer, many programming approaches have been tried. These techniques include modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable.

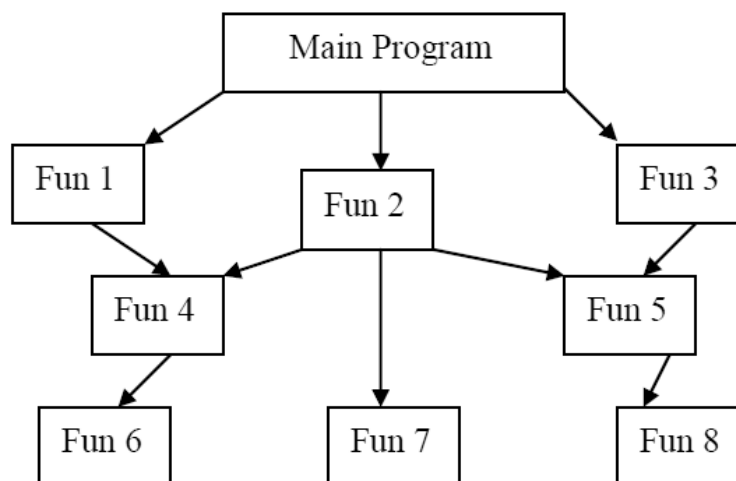
Procedural Oriented Language

A procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program. It contains a systematic order of statements, functions and commands to complete a computational task or program. The procedural language divides a program within variables, functions, statements and conditional operators.

In procedural approach,

- A program is a list of instruction.
- When program in PL become larger, they are divided into functions (subroutines, sub-programs, procedures).
- Functions are grouped into modules.

A typical program structure of Procedural programming is shown in figure below



Characteristics of POP

- Emphasis is on doing thing (algorithm).
- Large programs are divided into smaller programs known as functions.
- Data move openly around the system from function to function.
- Employee **top-down approach** in program design.

Disadvantages of POP

- It emphasis on doing things.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- It is difficult to create new data types with procedural languages.

- Most Procedural languages are not usually extensible and hence procedural programs are more complex to write and maintain.

Object-oriented Programming (OOP)

OOP is an approach to programming paradigm that attempts to eliminate some of the drawbacks of conventional programming methods with several powerful new concepts.

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

The general structure of OOP is shown in the figure below.

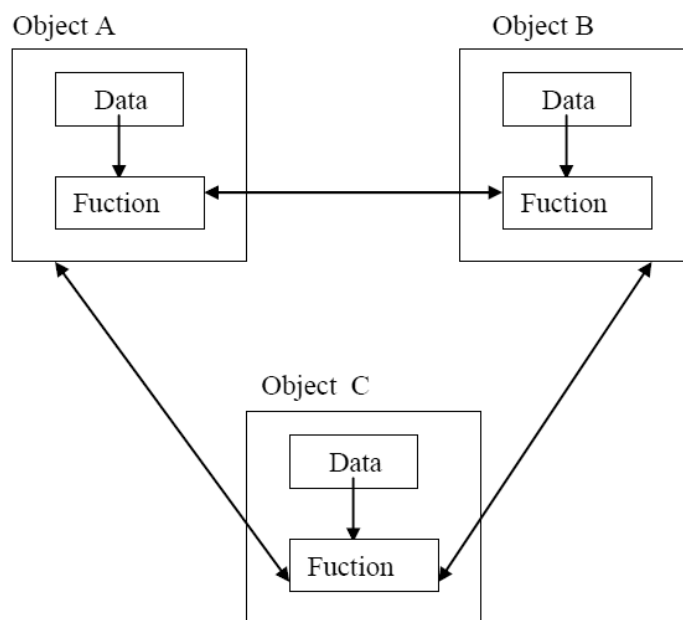


Fig: Organization of data and function in OOP

Characteristics of OOP

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Functions & data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
- Data is hidden & can't be accessed by external functions.
- Object can communicate with each other through function.
- New data & functions can be easily added.
- Follows Bottom up approach.

Benefits of OOP

- Making the use of inheritance, redundant code is eliminated and the existing class is extended.
- Through data hiding, programmer can build secure program.
- It is possible to have multiple instances of an object to co-exist without any interference.

- System can be easily upgraded from small to large systems.
- Software complexity can be easily managed.
- Message passing technique for communication between objects makes the interface description with external system much simpler.

Disadvantages of OOP

- Object oriented program required greater processing overhead – demands more resources.
- Requires the mastery in software engineering and programming methodology.
- Benefits only in long run while managing large software projects.
- The message passing between many objects in a complex application can be difficult to trace & debug.

Difference between OOP and POP

OOP	POP
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
The functions of the objects are linked via message passing.	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Example: C++ , JAVA	Example: C, FORTRAN

Basic concept / Principles of Object Oriented Programming

Basically there are 6 basic principles of OOP. They are as follows.

1. Object and Class
2. Data Abstraction and Encapsulation
3. Inheritance
4. Polymorphism
5. Dynamic Binding
6. Message Passing

Object and Class

Object is an instance of a class i.e. variable of class type. Objects are the basic runtime entities in an object oriented system. Objects contain data and code to manipulate the data. When a program is executed, the objects interact by sending message to one another. Generally, program objects are chosen such that they match closely with real world objects.

A class is a framework that specifies what data and what functions will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A Class is the collection of objects of similar type.

For example: mango, apple and orange are members of class Fruit. so if we create a class Fruit then mango, apple, orange can be the object of Fruit.

Object : Student	Object: Account
Data Name Date of birth Marks -----	Data Account number Account Type Name balance
Functions Total() Average() Display() -----	Functions deposit() withdraw() enquire()

Fig: Representation of Object

Data Abstraction and Encapsulation

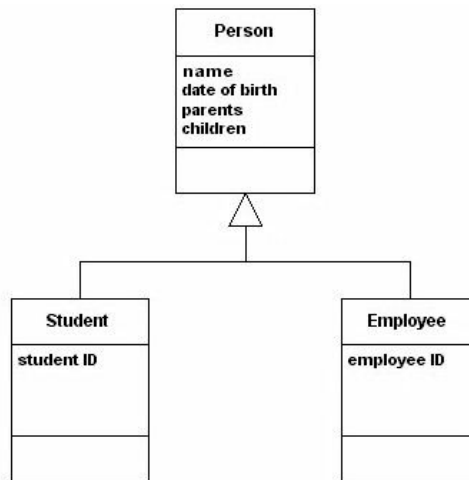
Encapsulation is the process of combining the data (called fields or attributes) and functions (called methods or behaviors) into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So **the term data hiding** is possible due to the concept of encapsulation, since the data are hidden from the outside world, so that it is safe from accidental alteration.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes uses the concept of abstraction and are defined as a list of abstract attributes and functions. Since classes uses the concept of data abstraction, they are known as abstract data types (ADT).

Inheritance

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. In OOP, the concept of inheritance provides the idea of reusability. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class). This process of deriving a new class from the existing base class is called inheritance.

It supports the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the code.



In the above figure, Person is base class while Student and Employee are derived form Person so they are referred to as derived class

Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However, the behavior depends upon the attribute the name holds at particular moment. Example of polymorphism in OOP is operator overloading, function overloading.

Example 1:

Operator symbol '+' is used for arithmetic operation between two numbers, however by overloading same operator '+' it can be used for different purpose like concatenation of strings. So the process of making an operator to exhibits different behaviors in different instances is called **operator overloading**.

Example 2:

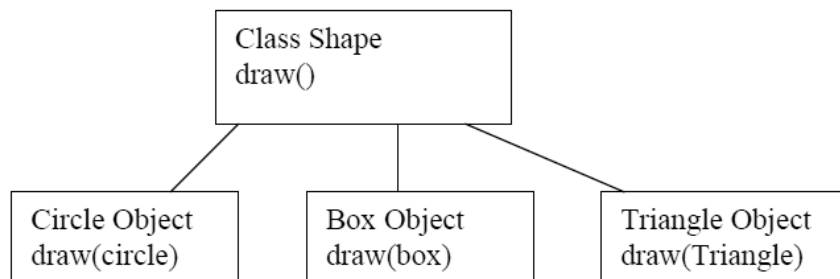


Fig: Polymorphism

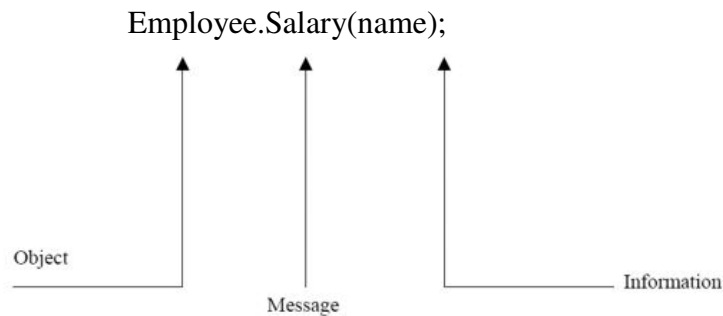
So in the above figure, a single function name is used to handle different number and different type of arguments. So using a single function name to perform different types of task is known as **function overloading**.

Message Passing

An object-oriented program consists of set of objects that communicate with each other. Object communicates with one another by sending and receiving information. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.

Example:



Dynamic Binding

Binding means link between procedure call and code to be execute. Dynamic binding means link exist between procedure call and code to be execute at run time when that procedure is call. It is also known late binding. It is generally use with polymorphism and inheritance. For example, complier comes to know at runtime that which function of sum will be call either with two arguments or with three arguments.

A New Paradigm

Q. Why new paradigm was needed instead of Procedural Approach?

Due to complex nature of software it was hard to understand the software and maintain the software. So software engineers need a new paradigm to cope this complexity. OOP was solution to this.

Object-oriented paradigm is a new method of implementation in which programs are organized as co-operative, each of which represents a instance of some class and whose classes are all members of a hierarchy of class united through inheritance relationships.

The important parts of OOP are

- Class
- Objects
- Object as an instance of class

Computation as Simulation

Object Orientation is a powerful technique when it comes to simulating real systems in a computer program. When it comes to designing our own programs using objects, as a general guide, it is a good idea to write a *class* and create *objects* for things are objects in real life.

Traditional model

- In traditional view computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various

memory, transforming them in some manner, and pushing the results back into other memory.

- Behavior of a computer executing a program is a process-state or pigeon-hole model
- By examining the values in the slots, we can determine the state of the machine or the results produced by a computation.
- This model may be a more or less accurate picture of what takes place inside a computer
- Real world problem solving is difficult in

Object Oriented Model

- Never mentioned memory addresses or variables or assignments or any of the conventional programming terms
- Instead, we spoke of objects, messages, and responsibility for some action
- This model is process of creating a host of helpers that forms a community and assists the programmer in the solution of a problem (Like in Flower example)
- This view of programming as creating a universe is in many ways similar to a style of computer simulation called ***“discrete event-driven simulation”***
- In a discrete event-driven simulation, the user creates computer models of the various elements (entities) of the simulation, describes how they will interact with one another, and sets them moving.
- Object oriented program is also similar to event driven simulation.
- Thinking about what we will be asking the objects to do or to tell you; these things can be implemented as their methods.

Coping with complexity

At early stages of computer programming development all programs are written in assembly language by single individual. As program become more and more complex, programmers have difficulties in remembering all information needed to develop and debug all software. As problem become more and more complex, even the best programmer cannot perform the task by himself. There will be group of programmer working together to solve complex problem.

Interconnections - the Bane of Complexity:

Many software systems are complex not because they are large, but because they have many interconnections. Interconnections make it difficult to understand pieces in isolation, or to carry them from one project to the next. The inability to cleanly separate out components makes it difficult to divide a task between several programmers. Complexity can only be managed by means of abstraction, by eliminating information that a programmer must know.

Nonlinear behavior of complexity

As programming task become larger, an interesting phenomenon was observed. A task that would take one programmer for two month to perform could not be completed by two programmers in one month. i.e. The work will be more complicated. This can be treated as nonlinear behavior, the reason behind this nonlinear behavior is complexity. In particular, the

interconnection between software components was complicated. Large quantities of information had to be communicated among the various members of the programming team. This behavior is called nonlinear behavior of complexity.

A Way of Viewing the World

To illustrate the major idea in OOP, let us consider the following real world situation.

Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who happens to be named Flora), tell her the variety and quantity of owners I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.

- **Agents and Communities**

In above example I solve my problem with help of agent (Object) flora to deliver the flower. There will community of agents to complete a task. In above example Flora will communicate with sally's florist, sally's florist will arrange flower, the arrangement of flower is hidden from me (data hiding/information hiding).

An object oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

- **Messages and Methods**

I will call flora for delivering flower to my friend sally. After that there will be chain of message passing and actions taken by various agents to deliver the flower as shown in figure above.

Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

- **Responsibilities**

My request for action indicates only the desired outcome (flowers for my friend). Flora is free to pursue any technique that achieves the desired objective i.e. to deliver flower to my friend and is not hampered by interference on my part.

A fundamental concept in object-oriented programming is to describe behaviour in terms of responsibilities.

- **Classes and Instances**

In above Scenario flora is florist we can use florist to represent the category (or class) of all florists. Which means Flora is instance (object) of class florist.

All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages

- **Class Hierarchies Inheritance**

Flora not necessarily because she is a florist but because she is a shopkeeper. Florist is a more specialized form of the category Shopkeeper. Flora is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so I know, for example, that Flora is probably bipedal. A Human is a Mammal and a Mammal is an Animal, and an Animal is a Material Object (therefore it has mass and weight).

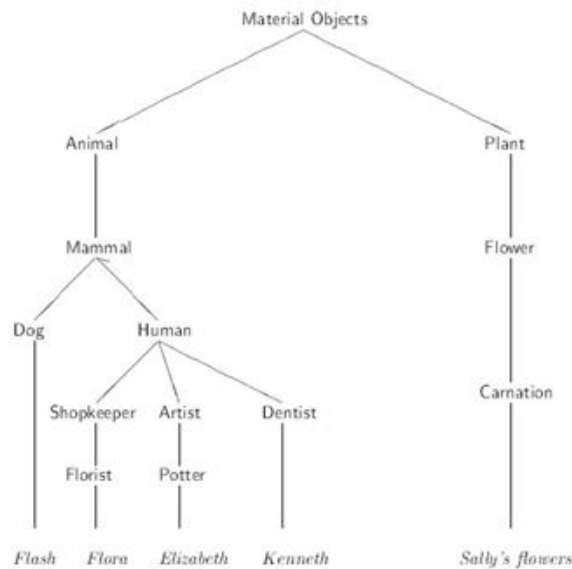


Figure : - A class hierarchy for various material objects.

Abstraction mechanism

The abstraction is the process of getting detail information according to the level of deep sight to the problem. If we open an atlas, we will open first see a map of world. This map will show only the most significant features only. It may show few details of mountains, oceans and large features of the earth. If you see the map of continent we may see more detail of country political boundaries, may be the city, rivers etc. similarly if we see the map of country, it might include main city, rivers, village, town and so on.

In OOP, Classes use theory of abstraction and defined list of abstract properties .Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.

Abstraction is related to both encapsulation and data hiding. Example of adding complex number, consider 3 object c1,c2,c3 c1 and c2 object is used to call input function, c3 is use for adding data in c1 and c2 , and c3 is also used for displaying output.

Varieties of class

- **Data managers classes** : Sometimes called data or state, are classes with the principal responsibility of maintaining data or state information. For example, in an abstraction of a problem, a major task for the class is simply to maintain. The data values that describe state information. Data managers are often recognizable as the nouns in a problem description and are usually the fundamental building blocks of a design.
- **Data sinks or data sources classes** : These are entities that generate data such as a random no. generator. Unlike a data manager, a data sink or data sources does not hold the data for any period, but generates it on demand.
- **View or observer class** : It an essential portion of most applications, display the information on a output devices such as terminal screen.
- **Facilitator or helper classes** : These are entities that maintain little or no state information themselves but assist with complex tasks. For example, in displaying an image we use the services of a facilitator class that handles the drawing of lines and text on the display device. In object-oriented languages like C++, Java etc. class describes a no. of possible run-time objects with same structure which are the instances of the class.

Chapter 2

Classes and Methods

Review of structures

Structure is a convenient tool for handling a group of logically related heterogeneous data types. Structure helps to organize data especially in large programs, because they provide group of variables of different data type to be treated as a single unit. It is most convenient way to keep related data under one roof.

- A structure is usually used when we need to store dissimilar or heterogeneous data together.
- The structure elements are stored in contiguous memory location as array.
- Structure elements can be accessed through a structure variable using a dot (.) operator.
- Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
- All the elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
- It is possible to pass a structure variable to a function either by value or by reference.
- It is possible to create an array of structure i.e. similar type of structure is placed in a common variable name. For example: we need to store the detail information of individual student in a class.

Example:

```
struct book
{
    char name[20];
    int pages;
    float price;
};
void main()
{
    struct book b1;
    b1.pages=500;
    b1.price=815.5;
    strcpy(b1.name, "C Programming");
    printf("\nName=%s, pages=%d and price=%f",b1.name,b1.pages,b1.price);
    getch();
    clrscr();
}
```

Limitation of Structure / Structure vs Class

- The standard C does not allow the struct data type to be treated like built-in data types. For example

```
struct complex
{
    float real, imag;
};
struct complex c1,c2,c3;
```

The complex numbers c1,c2,c3 can easily be assigned values using the dot operator. But we cannot add two complex numbers or subtract one from the other. That is $c3=c1 + c2$; is illegal in C.

Also data hiding is not permitted in C, that is structure members can be directly accessed by the structure variable by any function anywhere in their scope.

In c structure, it contains only variable or data as member but In C++, a class can have both variables and functions as members. It can also declare some of its members as private so that cannot be accessed directly by the external functions.

The data member in c++ structure are public by default which may lead to security issue but in Class the default access specifier is private.

Structure in C are only object based but Class support all features of OOP.

Class

A class is user-defined data type that includes different member data and associated member functions to access on those data. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into packages called classes. The data components of the class are called data members and the function components are called member functions.

Since class is a collection of logically related data items and the associated functions which operate and manipulate those data. This entire collection can be called a new **user defined data-type**. So, classes are user-defined data types and behave like the built-in types of a programming language.

Generally a class specification has two parts.

- i. Variable declaration
- ii. Function definition

The general form of a class declarations is

```
class class-name
{
    private:
        Variable declarations;
        Function declarations;
    public:
        Variable declarations;
        Function declarations;
};

#include<conio.h>
#include<iostream>
using namespace std;
class student      // class name student, starting of class
```

```

{
private:           // private data members name roll;
    char name[20];
    int roll;
public:           //public methods input() and output()
    void input()
    {
        cin>>name>>roll;
    }
    void output()
    {
        cout<<"Name : "<<name<<endl;
        cout<<" Roll No : "<<roll<<endl;
    }
};

int main()
{
    student s1;    // object (s1) declaration inside main()
    s1.input();    //calling methods syntax: object.method();
    s1.output();   //calling methods syntax: object.method();
    getch();
    return 0;
}

```

Difference between class and structure

Class	Structure
Class is a reference type and its object is created on the heap memory	Structure is a value type and its object is created on the stack memory.
Class can create a subclass that will inherit parent's properties and methods,	Structure does not support the inheritance.
A class has all members private by default.	A struct is a class where members are public by default.
Sizeof empty class is 1 Byte	Size of empty structure is 0 Bytes

Access Specifiers

Access specifiers are the keyword that controls access to data member and member functions within user-defined class. The access restriction to the class members is specified by the labeled public, private, and

protected sections within the class body. The keywords public, private, and protected are called access specifiers.

```
class Demo
{
public:
// public members go here
protected:
// protected members go here
private:
// private members go here
};
```

Public

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

Private

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

Protected

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class.

Creating object

Once a class has been specified (or declared), we can create variables of that type (by using the class name as datatype). General syntax for declaration of object is

Class_name Object_name

Example:

Student s1; //memory for s1 is allocated.

The above statement creates a variable s1 of type Student. The class variables are known as objects. So, s1 is called object of class Test.

Accessing Class Member

The private data members of a class can be accessed only through the member functions of that class.

However, the public members can be accessed from outside the class using object name and dot operator(.).

The following is the format for calling public members of a class.

Object_name.Function _name(actual argument)

Object_name.Data_name

Example:

If e1 is the object of Class Employee then we can access the public member function getname() with the following statement

E1.getname();

Defining a member Function

Member functions can be defined in two ways.

- Outside the class
- Inside the class

The code for the function body would be identical in both the cases. Irrespective of the place of definition, the function should perform the same task.

Outside the class

In this approach, the member functions are only declared inside the class, whereas its definition is written outside the class.

General form:

```
return-type class-name::function-name(argument-list)
{
-----
----- //function body
-----
}
```

Example:

```
void Employee::getdata()
{
-----
----- //function body
-----
}
```

Inside the class

Function body can be included in the class itself by replacing function declaration by function definition. If it is done, the function is treated as **an inline function**. Hence, all the restrictions that apply to inline function, will also apply here.

Example:

```

class Demo
{
    int a,b;
    public:
    void getdata()
    {
        cin>>a>>b;
    }
};

```

Here, getdata() is defined inside the class. So, it will act like an inline function.

A function defined outside the class can also be made 'inline' simply by using the qualifier 'inline' in the header line of a function definition.

Example:

```

class Demo
{
    -----
    -----
    public:
        void getdata(); // function declaration inside the class
};

void A::getdata()
{
    //function body
}

```

Function in C++

Dividing a program into function is one of the major principles of top-down structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Syntax of function:

```

void show( ) ; /* Function declaration */

main ( )
{
    -----
    -----
    show( ) ; /* Function call */
}

void show( ) /* Function definition */
{
    -----
}

```



```

----- /* Function body */
-----
}

```

Function Prototyping

The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.

Function prototype is a declaration statement in the calling program and is of the following form:

type function_name (argument-list) ;

Example:

```

float volume(int x, int y, float z) ; // legal
float volume (int x, int y, z) ; // illegal

```

Passing Arguments to Function

An argument is a data passed from a program to the function. In function, we can pass a variable by three ways:

1. Passing by value
2. Passing by reference
3. Passing by address or pointer

Passing by value:

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameter are not changed.

```

#include <iostream>
using namespace std;
void swap(int, int) ; // declaration prototype
int main ( )
{
    int x,y ;
    x=10 ;
    y=20 ;
    swap(x,y) ;
    cout <<x<<endl;
    cout<<y<<endl ;
    return 0;
}

```

```

void swap(int a, int b) // function definition
{
    int t ;
    t=a ;
    a=b ;
    b=t ;
}

```

Output:

x=10

y=20

Passing by reference:

Passing argument by reference uses a different approach. In this, the reference of original variable is passed to function. But in call by value, the value of variable is passed.

```

#include <iostream>
using namespace std;
void swap(int &, int &) ;
int main( )
{

    int x,y ;
    x=10 ;
    y=20 ;
    swap(x,y);
    cout<<x;
    cout <<y;
    return 0;
}
void swap(int &a, int &b)
{
    int t ;
    t=a ;
    a=b ;
    b=t ;
}

```

O/P:

20

10

Passing by Address or Pointer:

This is similar to passing by reference but only difference is in this case, we can pass the address of a variable.

```
#include <iostream>
using namespace std;
void swap(int *, int *) ;
int main( )
{
    int x, y ;
    x=10 ;
    y=20 ;
    swap(&x,&y);
    cout<<x;
    cout<<y;
    return 0;
}

void swap(int *a, int *b)
{
    int t;
    t=*a ;
    *a=*b ;
    *b=t ;
}
```

Inline Function:

The functions which are expanded inline by the compiler each time it's call is appeared instead of jumping to the called function as usual is called inline function. When a function is defined as inline, compiler copies it's body where the function call is made, instead of transferring control to that function. A function is made inline by using a keyword "inline" before the function definition.

Advantage: Increases execution speed

Disadvantages: More memory required because of copying

Example1:

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

```

#include<iostream>
using namespace std;
//While declaration no need for keyword inline
int Findcube(int x);
int main()
{
    cout<<Findcube(3);
    getch();
    return 0;
}
inline int Findcube(int x)
{
    return x*x*x;
}

```

Example 2:

WAP to display hello world by using class and inline member function

```

#include<iostream>
using namespace std;
class Demo
{
    public:
    inline void display();
};
int main()
{
    Demo d;
    d.display();
}
inline void Demo:: display()
{
    cout<<"hello world";
}

```

Example of C++ Program with Class

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. In main create two objects of a class Rectangle and for each object, call both of the function.

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. The two functions are defined outside the class. In main create two objects of a class Rectangle and for each object, call both of the function.

```

#include<iostream>

```

```

using namespace std;
class Rectangle
{
    float length,breadth;
    public:
        void readdata();
        void displaydata();
};
void Rectangle::readdata()
{
    cout<<"Enter lengtha and breadth of rectangle"<<endl;
    cin>>length>>breadth;
}
void Rectangle::displaydata()
{
    float area;
    area=length*breadth;
    cout<<"Area of rectangle= "<<area<<endl;
}
int main()
{
    Rectangle rect1,rect2;
    rect1.readdata();
    rect1.displaydata();
    rect2.readdata();
    rect2.displaydata();
}

```

Q. Define a class to represent a bank account. Include the following members

Data Member:

- i. Name of account holder
- i. Account Number
- ii. Account Type
- iii. Balance Amount

Member Functions:

- to take detail of the fields from user
- To deposit an amount
- To withdraw an amount
- To display name and balance

```

#include<iostream>
using namespace std;
class Account
{
    char name[50],acctype[20];

```

```

int accno;
float wdraw,bal,dep;
public:
    void setdata()
    {
        cout<<"Enter name of account holder"<<endl;
        cin>>name;
        cout<<"Enter the type of account"<<endl;
        cin>>acctype;
        cout<<"Enter the account number"<<endl;
        cin>>accno;
        cout<<"Enter the balance"<<endl;
        cin>>bal;
    }
    void deposit()
    {
        cout<<"Enter amount to be deposited"<<endl;
        cin>>dep;
        bal=bal+dep;
    }
    void withdraw()
    {
        cout<<"Enter amount to be withdraw"<<endl;
        cin>>wdraw;
        bal=bal-wdraw;
    }
    void display()
    {
        cout<<"Name= "<<name<<endl;
        cout<<"Balance= "<<bal;
    }
};
int main()
{
    Account a;
    a.setdata();
    a.deposit();
    a.withdraw();
    a.display();
}

```

Private Member Function

As we have seen, member functions are, in general, made public. But in some cases, we may need a private function to hide them from outside world. Private member functions can only be called by another function that is a member of its class. Object of the class cannot invoke private member functions using dot operator.

Example:

```

class Demo
{
    int a,b;

```

```

void func1(); //private member by default
public:
void func2();
};

void Demo::func2()
{
    Func1(); //func1() is private member so cant be accessed by using object of class Demo
}

```

Q. Define a class Student with the following specification

Private members:

Data member: Id, name, eng, math, science

Member function: float calculateTotal() to return the total obtained mark.

Public members:

Member functions:

getData() to take input(id, name, eng, math, science) from user showData() to display all the data member along with total on the screen.

Write a main program to test your class.

```

#include<iostream>
using namespace std;
class Student
{
    int id;
    char name[50];
    float eng,math,science;
    float calculateTotal()
    {
        return(eng+math+science);
    }
public:
    void getData()
    {
        cout<<"Enter id"<<endl;
        cin>>id;
        cout<<"Enter name"<<endl;
        cin>>name;
        cout<<"Enter marks in maths, science and english"<<endl;
        cin>>math>>science>>eng;
    }

    void showData()
    {
        float t;
        cout<<"ID= "<<id<<endl;
        cout<<"Name ="<<name<<endl;
        cout<<"Marks in maths, science and english ="<<math<<science<<eng<<endl;
    }
}

```

```

        t=calculateTotal(); // Within the class scope, we can call members directly
with their name
        cout<<"Total= "<<t;
    }

};
int main()
{
    Student s;
    s.getData();
    s.showData();
}

```

Function Overloading

Two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs – or both. When two more functions share the same name, they are said overloaded. Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

```

#include <iostream>
using namespace std;

int area(int) ; // function area with int return type
double area(double, int) ; // function area with double return type
long area(long, int, int) ; // function area with long return type

int main( )
{

    cout<<area(10);
    cout<<area(2.5,8);
    cout<<area(100L,75,15);
    return 0 ;

}

int area(int s) // square
{
    return(s*s) ;
}

double area(double r, int h) // Surface area of cylinder ;
{

```



```

return(2*3.14*r*h) ;
}
long area(long l, int b, int h) //area of parallelopiped
{
return(2*(l*b+b*h+l*h)) ;
}

```

Default Arguments

When declaring a function, we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function.

To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

```

#include <iostream>
using namespace std;
int main( )
{
int divide(int a, int b=2) ; //prototype and b=2 default value
cout<<divide(12);
cout<<endl;
cout<<divide(20,4);
return 0;
}
int divide(int x, int y)
{
int r ;
r = x / y ;
return(r) ;
}

```

O/P

```

6
5

```

this pointer

In C++, **this** pointer is used to represent the address of an object inside a member function. **For example**, consider an object *obj* calling one of its member function say *method()* as *obj.method()*. Then, **this** pointer will

hold the address of object *obj* inside the member function *method()*. The **this** pointer acts as an implicit argument to all the member functions.

```
#include<iostream>
using namespace std;
class Demo
{
private:
int num;
char ch;
public:
void setMyValues(int num, char ch)
{
    this->num = num;
    this->ch = ch;
}
void displayMyValues()
{
    cout<<num<<endl;
    cout<<ch;
}
};
int main()
{
    Demo obj;
    obj.setMyValues(100, 'A');
    obj.displayMyValues();
    return 0;
}
```

O/P:

100

A

Static data Members

When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. We can define class members static using static keyword.

Unlike normal member variables, static member variables are shared by all objects of the class.

```
class Something
```

```
{
public:
    static int s_value;
};
```

```
int Something :: s_value = 1;
```

```

int main()
{
    Something first;
    Something second;
    first.s_value = 2;
    cout << first.s_value << '\n';
    cout << second.s_value << '\n';
    return 0;
}

```

O/p

2

2

Static member function

Like static member variables, static member functions are not attached to any particular object. Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. Static member functions have no `*this` pointer.

We can access a static member function with class name, by using following syntax:

```
class_name:: function_name(parameter);
```

```

#include <iostream>
using namespace std;
class Demo
{
    private:
    //static data members
    static int X;
    static int Y;
    public:
    //static member function
    static void Print()
    {
        cout << "Value of X: " << X << endl;
        cout << "Value of Y: " << Y << endl;
    }
};
//static data members initializations
int Demo :: X = 10;
int Demo :: Y = 20;
int main()
{
    Demo OB;
}

```

```
//accessing class name with object name
cout<<"Printing through object name:"<<endl;
OB.Print();
//accessing class name with class name
cout<<"Printing through class name:"<<endl;
Demo::Print();
return0;
}
```

Output

```
Printing through object name:
Value of X: 10
Value of Y: 20
Printing through class name:
Value of X: 10
Value of Y: 20
```

Friend Function

One of the important concepts of is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The compiler knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Syntax:

```
class class_name
{
... ..
friend return_type function_name(arguments);
... ..
}
return_type functionName(argument/s)
{
... ..
// Private and protected data of class_Name can be accessed from
// this function because it is a friend function of className.
```

```
... ..  
}
```

How can we make two classes friendly?

i. Common friend function on both classes (see above examples)

ii. Friend class technique

Like friend function, a class can also be a friend of another class. A friend class can access all the private and protected members of other class. In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.

Example:

```
#include<iostream.h>  
  
class Rectangle  
{  
    int L,B;  
    public:  
    Rectangle()  
    {  
        L=10;  
        B=20;  
    }  
    friend class Square;    //Statement 1  
};  
  
class Square  
{  
    int S;  
    public:  
    Square()  
    {  
        S=5;  
    }  
    void Display(Rectangle Rect)  
    {
```

```

        cout<<"\n\n\tLength : "<<Rect.L;
        cout<<"\n\n\tBreadth : "<<Rect.B;
        cout<<"\n\n\tSide : "<<S;

    }

};

void main()
{
    Rectangle R;
    Square S;
    S.Display(R);    //Statement 2
}

```

Output :

```

Length : 10

Breadth : 20

Side : 5

```

Q. WAP to find the sum of two number using the concept of friend function

```

#include<iostream>
using namespace std;
class Demo
{
    int fn,sn;
public:
    void setdata()
    {
        cout<<"Enter two number"<<endl;
        cin>>fn>>sn;
    }
    friend void sum(Demo d);
};

void sum(Demo d)
{
    int sum;
    sum=d.fn+d.sn;
    cout<<"Sum= "<<sum;
}

int main()
{

```

```

        Demo a;
        a.setdata();
        sum(a);
    }

```

Q. Create classes called class1 and class2 with each of having one private member. Add member function to set a value (say setvalue) on each class. Add one more function max () that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class then set a value on them. Display the maximum number among them.

```

#include<iostream>
using namespace std;
class Class2;// forward declaration
class Class1
{
    int a;
public:
    void setdata()
    {
        cout<<"Enter a number"<<endl;
        cin>>a;
    }
    friend void max(Class1,Class2);
};
class Class2
{
    int a;
public:
    void setdata()
    {
        cout<<"Enter a number"<<endl;
        cin>>a;
    }
    friend void max(Class1,Class2);
};

void max(Class1 c1, Class2 c2)
{
    if(c1.a>c2.a)
    {
        cout<<"Larger= "<<c1.a;
    }
    else
    {
        cout<<"Larger= "<<c2.a;
    }
}

int main()
{
    Class1 x;
    Class2 y;

```

```
x.setdata();  
y.setdata();  
max(x,y);  
}
```

Reference Variable

Reference operator (&) is used to define referencing variable. Reference variable prepares an alternative name (alias name) for previously defined variable.

Syntax of referencing variable

Here same variable can be used with help of 2 names

<return-type>&reference-variable =variable;

Example

```
int y=9;  
int &x=y;
```

Now value of y is 9 and after initialization x also becomes 9

State and Behaviour of Object

State

An object's state is defined by the attributes (i.e. data members or variables) of the object.

In OOP state is defined as data members.

It is determined by the values of its attributes.

What the objects have, Example: Student has a first name, last name, age, etc...

Behaviour

An object's behaviour is defined by the methods or action (i.e. Member functions) of the object.

In OOP behaviours are defined as member functions.

It determines the actions of an object.

What the objects do, Example: Student attends a course "OOP", "C programming" etc...

Example: Consider Lamp as an Object

Its states are on/off and behaviours are turn on/ turn off

Chapter 3

Message, Instance and Initialization

Message Passing

Message Passing is sending and receiving of information by the objects same as people exchange information. So this helps in building systems that simulate real life. Following are the basic steps in message passing.

- Creating classes that define objects and its behavior.
- Creating objects from class definitions
- Establishing communication among objects

In OOPs, Message Passing involves specifying the name of objects, the name of the function, and the information to be sent.

Message passing syntax

object.methodname(arguments);

Where, object is a receiver.

Method name is a message

Arguments are information to be passed.

Object as function Argument

Unlike other argument, we can also pass object as function argument using following two ways.

Pass by value:

A copy of entire object is passed to the function. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

Pass by reference:

Only the address of the object is transferred to the function. Since the actual address of object is passed, any changes made to the object inside the function will reflect in the actual object used to call the function. This method is more efficient than previous one.

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments (Pass by value)

```
#include<iostream>
using namespace std;
class Time
{
```

```

        int hour,min;
    public:
        void gettime()
        {
            cout<<"Enter hour"<<endl;
            cin>>hour;
            cout<<"Enter minute"<<endl;
            cin>>min;
        }
        void puttime()
        {
            cout<<"Hour= "<<hour<<"\t";
            cout<<"Minute= "<<min<<endl;
        }
        void add(Time a, Time b)
        {
            min=a.min +b.min;
            hour=min/60;
            min=min%60;
            hour=hour +a.hour+b.hour;
        }
};
int main()
{
    Time t1,t2,t3;
    t1.gettime();
    t2.gettime();
    t3.add(t1,t2);
    t1.puttime();
    t2.puttime();
    t3.puttime();
}

```

Output:

```

Enter hour
1
Enter minute
30
Enter hour
1
Enter minute
30
Hour= 1 Minute= 30
Hour= 1 Minute= 30
Hour= 3 Minute= 0

```

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by reference)

```

#include<iostream>
using namespace std;
class Time
{
    int hour,min;

```

```

public:
    void gettime()
    {
        cout<<"Enter hour"<<endl;
        cin>>hour;
        cout<<"Enter minute"<<endl;
        cin>>min;
    }
    void puttime()
    {
        cout<<"Hour= "<<hour<<"\t";
        cout<<"Minute= "<<min<<endl;
    }
    void add(Time &a, Time &b) // object 'a' work on the same address that of object t1
    {
        min=a.min +b.min;
        hour=min/60;
        min=min%60;
        hour=hour +a.hour+b.hour;
    }
};

int main()
{
    Time t1,t2,t3;
    t1.gettime();
    t2.gettime();
    t3.add(t1,t2);
    t1.puttime();
    t2.puttime();
    t3.puttime();
}

```

Returning Objects

A function can take object as argument as well as return object.

Q. WAP to find the square of a given number using the concept of object as argument(call by reference) and returning object.

```
#include<iostream>
using namespace std;
class Demo
{
    int a;
    public:
        void setdata(int x)
        {
            a=x;
        }

        Demo square(Demo *p)
        {
            Demo x;
            x.a=p->a * p->a;
            return x;
        }

        void putdata()
        {
            cout<<a;
        }
};
int main()
{
    Demo o1,o2,o3;
    o1.setdata(10);
    o3=o2.square(&o1);
    o3.putdata();
}
```

Q. Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input(int ,int) to initialize x and y, Complex sum(Complex, Complex) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Complex
{
    int x,y;
```

```

public:
void input(int real,int img)
{
    x=real;
    y=img;
}

Complex sum (Complex c1, Complex c2)
{
    Complex c3;
    c3.x=c1.x+c2.x;
    c3.y=c1.y+c2.y;
    return c3;
}

void show()
{
    cout<<"Real Part= "<<x<<endl;
    cout<<"Imaginary part= "<<y<<endl;
}
};

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

    a.input(5,6);
    b.input(4,5);
    c=c.sum(a,b);
    c.show();
}

```

Initialization class Object

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as `setdata()` and `diplay()` to provide initial values to the private member variables.

An OOP also has a provision of initializing objects of a class during their definition itself. A class in C++ may contain two special member functions dealing with the internal working of a class. These functions are the constructors and the destructors. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it.

Constructor

A constructor is a special member function having the same name as that of the class which is used to automatically initialize the objects of the class type with legal initial values. The constructor is invoked automatically whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

Properties:

- They are also used to allocate memory for a class object.

- They execute automatically when an object of a class is created.
- Constructor's name is same as that of class name.
- They should be declared in the "public" section.
- They do not have return types, not even void and therefore, and they cannot return any values.
- Like C++ functions, they can have default arguments.
- Constructor is NOT called when a pointer of a class is created.

A constructor is declared and defined as below

```
class Demo
{
    private:
    -----
    -----
    public:
    Demo(); //constructor
};
Demo ::Demo() //note no return type required
{
    //Body of constructor if defined outside the class
}
```

Types of constructor

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor

A constructor that does not take any parameter is called default constructor. This constructor is always called by compiler if no user-defined constructor is provided.

The following Class definition shows a default constructor.

```
class Demo
{
    public:
    Demo() //Default Constructor
    {
    }
};
```

This constructor is also called as implicit constructor because if we do not provide any constructor with a class, the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

```

class A
{
    //no constructor
};

```

Also if we provide constructor with all default arguments, then that can also be considered as the default constructor or **constructor with default argument**.

```

class Demo
{
    public:
    Demo (int a=7)
    {

    }
};

```

Example:

Create a class Demo with two data member (id, name), a default constructor to initialize these fields and a member function display() to show student details. Write a main program to test your class.

```

#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
    int id;
    char name[10];
    public:
        Demo() // Default constructor
        {
            id=5;
            strcpy(name,"ram");
        }
        void display()
        {
            cout<<"ID= "<<id<<endl;
            cout<<"Name= "<<name<<endl;
        }
};
int main()
{
    Demo d;
    d.display();
}

```

Parameterized constructor

The constructors that can take arguments or parameters are called parameterized constructor. In this, we pass the initial value as arguments to the constructor function when the object is declared.

The following Class definition shows a default constructor.

```

class Demo
{

```

```

    public:
    Demo(int x, int y) //Parameterized Constructor
    {
        // constructor body
    }
};

```

Q. WAP to find the area of rectangle using the concept of parameterized constructor

```

#include<iostream>
using namespace std;
class Demo
{
    int l,b;
    public:
        Demo(int length, int breadth)
        {
            l=length;
            b=breadth;
        }
        void area()
        {
            int a;
            a=l*b;
            cout<<"Area of rectangle= "<<a<<endl;
        }
};
int main()
{
    Demo d(5,6);
    d.area();
}

```

Output:

```
Area of rectangle= 30
```

Q. Create a class Person with data member Name, age, address and citizenship_number. Write a constructor to initialize the value of a person. Assign citizenship number if the age of the person is greater than 16 otherwise assign values zero to citizenship number. Also, create a function to display the values. Write a main program to test your class.

```

#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
    char name[50],address[50];
    int citizenship_number,age;
    public:
        Demo(char n[],char ad[],int a, int cn)
        {

```



```

        strcpy(name,n);
        strcpy(address,ad);
        citizenship_number=cn;
        age=a;
    }
    void display()
    {
        cout<<"Name= "<<name<<endl;
        cout<<"Address= "<<address<<endl;
        cout<<"Age= "<<age<<endl;
        cout<<"CitizenshipNumber="<<citizenship_number<<endl;
    }
};
int main()
{
    char nm[50],ad[50];
    int cn,a;
    cout<<"Enter name"<<endl;
    cin>>nm;
    cout<<"Enter address"<<endl;
    cin>>ad;
    cout<<"Enter age"<<endl;
    cin>>a;
    cn=0;
    if(a>16)
    {
        cout<<"Enter Citizenship Number"<<endl;
        cin>>cn;
    }
    Demo d(nm,ad,a,cn);
    d.display();
}

```

Copy Constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. Parameterized constructors having object of the same class as parameter or argument is called copy constructor. The copy constructor is used to:

Initialize one object from another of the same type.

- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Q. A program to demonstrate the concept of copy constructor

```

#include<iostream>
using namespace std;
class Demo
{
    int a;
public:
    Demo()//default constructor

```

```

    {
    }
    Demo (int b)
    {
        a=b;
    }
    void display()
    {
        cout<<a<<endl;
    }
    //the below definition of copy constructor is optional
    Demo (Demo &x)
    {
        a=x.a;
    }
};
int main()
{
    Demo d(5);

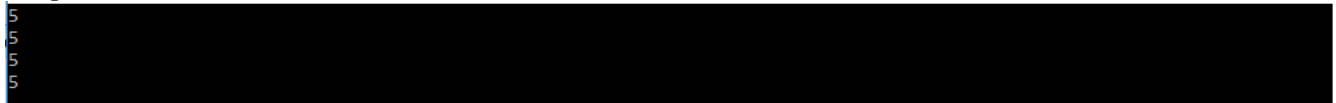
    Demo x(d); // copy constructor will called and if no copy constructor available the compiler will create one

    Demo y=d; //copy constructor will called and if no copy constructor available the compiler will create one

    Demo z;
    z=d; // It will not call copy constructor but assigns value member by member
    d.display();
    x.display();
    y.display();
    z.display();
}

```

Output:



Note:// we also can write statiemnt like Demo x(&d) and Define copy constructor Demo as

```

Demo(Demo *x)
{
    a= x.a;
}

```

Default vs parameterized constructor

Default Constructor	Parameterized Constructor
A constructor that has no parameter is called default constructor.	A constructor that has <u>parameter(s)</u> is called parameterized constructor.
Default constructor is used to initialize object with same default value like 0, null.	Parameterized constructor is used to initialize each object with different values.
When data is not passed at the time of creating an object, default constructor is called but not parameterized.	When data is passed at the time of creating an object, default constructor as well as parameterized constructor is called.

Constructor vs Method

Constructor	Methods
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Constructor Overloading

When more than one constructor functions are defined in the same class then we say the constructor is overloaded. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

Example

```
class Demo
{
    public:
        Demo() //Default Constructor
        {
        }
        Demo(parameter1) // Parameterized constructor with one parameter
        {
        }
        Demo(parameter1,parameter2)//Parameterized constructor with two parameters
        {
        }
}
```

```
}
```

In the above example, the constructor is overloaded because we have three different definition for same constructor name with different signature.

Q. WAP to find the area of circle and rectangle using the concept of constructor overloading

```
#include<iostream.>
using namespace std;
class Demo
{
    public:
        Demo(float radius)
        {
            cout<<"Area of circle= "<<(3.1416*radius*radius)<<endl;
        }
        Demo(float length, float breadth)
        {
            cout<<"Area of rectangle= "<<(length * breadth)<<endl;
        }
};
int main()
{
    Demo d(4); // invokes constructor with one parameter
    Demo d1(5,6); // invokes constructor with two parameter
}
```

Output:

```
Area of circle= 50.2656
Area of rectangle= 30
```

Constructor with default argument

We can define constructor with default argument unlike function with default argument. The following program demonstrates the concept of default constructor.

```
#include<iostream>
using namespace std;
class Demo
{
    int l,b;
    public:
        Demo(int length, int breadth=20)
        {
            l=length;
            b=breadth;
        }
        void area()
        {
            cout<<"Area of rectangle= "<<(l*b)<<endl;
        }
};
int main()
{
    Demo d(4);
```

```

        Demo d1(4,5);
        d.area();
        d1.area();
    }

```

Output:

```

Area of rectangle= 80
Area of rectangle= 20

```

Again consider the following program to find the square of number

```

#include<iostream.>
using namespace std;
class Demo
{
    int n;
public:

        Demo(int a=5) //Default constructor with default argument
        {
            n=a;
        }
        void square()
        {
            cout<<(n*n)<<endl;
        }
};
int main()
{
    Demo a;
    a.square();
    Demo a1(5);
    a1.square();
}

```

Output:

```

25
25

```

The default argument constructor can be called with either one argument or no argument as shown in above example. But if we use both default constructor as well as above default argument constructor then the declaration of statement

```
Demo a;
```

Will be ambiguous for whether to call Demo() or Demo(int=0).

Dynamic Constructor

Dynamic constructor is used to allocate the memory to the objects at the run time. Allocation of memory to objects at the time of their construction is called dynamic construction of objects. The memory is allocated with the help of new operator. Using new operator, we can allocate right amount of memory for each object when they are not of same size, thus resulting memory utilization.

Example:

```

#include<iostream>
using namespace std;

```

```

#include<string.h>
class Demo
{
    char *name;

    public:

        Demo(char *str)
        {
            int len;
            len=strlen(str);
            name=new char[len+1];
            strcpy(name,str);
        }
        void display()
        {
            cout<<"Name= "<<name<<endl;
        }
};
int main()
{
    Demo d("Ram"),d1("Hari");
    d.display();
    d1.display();

}

```

Dynamic initialization of Object

Class objects can be initialized dynamically (i.e. at the run time). The users provide the values at the run time. Dynamic initialization is that in which initialization value isn't known at compile-time. It's computed at runtime to initialize the variable.

Advantage: various initialization formats can be provided using constructor overloading.

Example:

```

#include<iostream.>
using namespace std;
class Demo
{
    int l,b;
    float r,area;
    public:
        Demo()
        {

        }
        Demo(int len, int bre)
        {
            l=len;
            b=bre;
            area=l*b;
        }
}

```

```

        Demo (float rad)
        {
            r=rad;
            area=3.1416 * r* r;
        }
        void display()
        {
            cout<<"Area= "<<area<<endl;
        }
    };
    int main()
    {
        Demo d1,d2;
        int l,b;
        float r;
        cout<<"Enter length and bradth"<<endl;
        cin>>l>>b;
        d1=Demo(l,b); // the value of object d1 will only be know at run time // simply Demo d1(l,b);
        d1.display();
        cout<<"Enter radius"<<endl;
        cin>>r;
        d2=Demo(r); // simply Demo d2(r);
        d2.display();
    }

```

In the above program we adopted dynamic initialization by manually calling the constructor and have not make use of object and dot operator to access members.

Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release dynamically allocated memory and to perform other “cleanup” activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

Example:

```

~Demo ()
{
}

```

Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function). They are also defined in the public section. Destructor never takes any argument, nor does it return any value. So, they cannot be overloaded.

Note: whenever ‘new’ is used to allocate memory in the constructors, we should use ‘delete’ to free the memory.

For example:

```

Demo::~~Demo()
{
    delete obj;
}

```

Also destructor can be defined inside class

```
#include<iostream>
using namespace std;
class Demo
{
    int *p;
public:
    Demo()
    {
        p=new int();
        *p=7;
    }
    ~Demo()
    {
        delete(p);
    }
};
```

For example:

Note: Destructor also can be invoked manually but basically it is useless because the destructor is automatically invoked when the scope of created object ends. So this lead to invoking of destructor twice.

```
#include<iostream>
using namespace std;
class Demo
{
};
int main()
{
    Demo d;
    d.~Demo();
}
```

Characteristic of Destructor

- It can't take any argument so can't be overloaded.
- It has same name as the class name but preceded by a tilde symbol.
- It can't be declared statics.
- It can't return a value.
- It should have public or protected access specifiers.
- Only one destructor exists in a class.

Example:

The below program demonstrates the concept of constructor and destructor

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



```

class Demo
{
static int count;
public:
    Demo()
    {
        count++;
        cout<<"Object created= "<<count<<endl;
    }
    ~Demo()
    {
        cout<<"Object Destroyed= "<<count<<endl;
        count--;
    }
};
int Demo::count;
int main()
{
    Demo a1,a2,a3;
    {
        Demo a4;
    }
}

```

Output:

```

Object created= 1
Object created= 2
Object created= 3
Object created= 4
Object Destroyed= 4
Object Destroyed= 3
Object Destroyed= 2
Object Destroyed= 1

```

Q. WAP to initialize a file named name with your name using dynamic constructor. Also write destructor to release the dynamically allocated memory.

```

#include<iostream>
using namespace std;
#include<string.h>
class Demo
{
    char *name;
public:
    Demo(char *str)
    {
        int len;
        len=strlen(str);
        name=new char[len+1];
        strcpy(name,str);
    }
    void display()
    {
        cout<<"Name= "<<name<<endl;
    }
}

```

```

    }
    ~Demo()
    {
        delete(name);
    }
};
int main()
{
    Demo d("Ram"),d1("Hari");
    d.display();
    d1.display();

}

```

Constructor vs Destructor

- Constructors **guarantee** that the member **variables** are **initialized** when an object is declared.
- Constructors automatically execute when a class object enters its scope.
- The **name of a constructor** is the **same as the name of the class**.
- A class can **have more than one constructor**.
- A constructor without parameters is called **the default constructor**.

- Destructor **automatically execute** when a class object goes out of scope.
- The name of a destructor is the tilde (~), followed by the class name (no spaces in between).
- A class can **have only one destructor**.
- The destructor has no parameters.

Q. "Can there be more than one destructor in the same class? If no, explain it with suitable example."

Stack and Heap memory

Stack

- It's a region of your computer's memory that stores temporary variables created by each function (including the main () function).
- The stack is a "Last in First Out" data structure and limited in size
- Every time a function declares a new variable, it is "pushed" (inserted) onto the stack.
- Every time a function exits, **all** of the variables pushed onto the stack by that function, are freed or popped (that is to say, they are deleted).
- Once a stack variable is freed, that region of memory becomes available for other stack variables.
- A key to understanding the stack is the notion that when a function exits, all of its variables are popped off (Removed) of the stack (and hence lost forever).

- There is no need to manage the memory yourself, variables are allocated and freed automatically

Advantages

- Memory is managed for you to store variables automatically. You don't have to allocate memory by hand, or free it once you don't need it any more.
- CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

□□ **Limitations**

Limit (varies with OS) on the size of variables that can be store on the stack

Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- □□ It is a more free-floating region of memory (and is larger).
- □□ To allocate memory on the heap, you must use **new** operator in C++.
- Once you have allocated memory on the heap, you are responsible for using **delete** to deallocate that memory once you don't need it any more.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Advantages and disadvantages of stack and heap based memory

Stack

- Very fast access
- Don't have to explicitly de-allocate variables
- Space is managed efficiently by CPU, memory will not become fragmented
- Local variables only
- Limit on stack size (OS-dependent)
- Variables cannot be resized

Heap

- Variables can be accessed globally
 - No limit on memory size
 - (Relatively) slower access
 - No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
 - You must manage memory (you're in charge of allocating (new) and freeing (delete) variables)
- Variables can be resized using new operator

Dynamic memory allocation in c++

In c++, Dynamic memory allocation can be achieved by using **new** and **delete** operator.

new operator

- Dynamic memory is allocated using operator **new**.
- ☐ **new** is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].
- ☐ It returns a pointer to the beginning of the new block of memory allocated.
- ☐ **Syntax is:**
data-type *ptr = new data-type;
data-type *ptr = new data-type [number_of_elements];
- ☐ The first expression is used to allocate memory to contain one single element of type data-type.
- ☐ The second one is used to allocate a block (an array) of elements of type data-type, where number_of_elements is an integer value representing the amount of these.
- ☐ **For example:**
int *ptr;
int *ptr = new int [5];

delete

- ☐ In most cases, memory allocated dynamically is only needed during specific periods of time within a program.
- ☐ Once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- ☐ Delete operator free the memory allocated to the variable i.e. it deletes the variables memory
- ☐ **Syntax is:**
delete pointer;
delete [] pointer;
- ☐ The first statement releases the memory of a single element allocated using new.
- The second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).
- ☐ **For example:**
delete ptr;
delete [] ptr;

Programming example of new and delete operator

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

```
void main()
{
```

```
    int size,i;
    int *ptr;
```

```

cout<<"\n\tEnter size of Array : ";
cin>>size;

ptr = new int[size];
//Creating memory at run-time and return first byte of address to ptr.

for(i=0;i<5;i++)    //Input array from user.
{
    cout<<"\n\tEnter any number : ";
    cin>>ptr[i];
}

for(i=0;i<5;i++)    //Output array to console.
    cout<<ptr[i]<<" ";

delete[] ptr;
//deallocating all the memory created by new operator

```

Memory recovery

Because in most languages objects are dynamically allocated, they must be recovered at run-time. There are two broad approaches to this:

- Force the programmer to explicitly say when a value is no longer being used:

```
delete pointername; // C++ example
```

- Use a **garbage collection** system that will automatically determine when values are no longer being used, and recover the memory.

Chapter 4

Object Inheritance and Re-usability

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class:

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

Super Class:

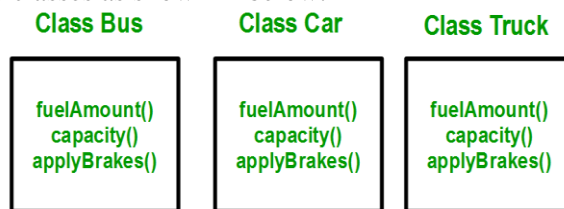
The class whose properties are inherited by sub class is called Base Class or Super class.

The derived class inherits some or all the traits from base class. The base class is unchanged by this. Most important advantage of inheritance is re-usability. Once a base class is written and debugged, it need not be touched again and we can use this class for deriving another class if we need. Reusing existing code saves time and money. By re-usability a programmer can use a class created by another person or company and without modifying it derive other class from it.

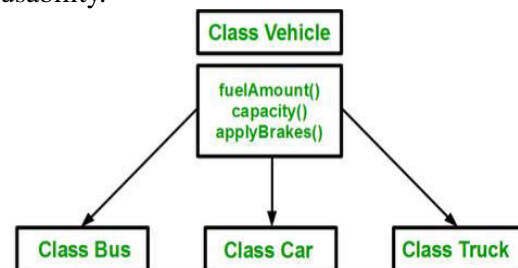
Why and when to use inheritance?

Consider a group of vehicles.

We need to create classes for Bus, Car and Truck. The methods `fuelAmount()`, `capacity()`, `applyBrakes()` will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below:



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



Syntax:

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

Purpose of Inheritance

- 1 Code Reusability
- 2 Method Overriding (Hence, Runtime Polymorphism.)
- 3 Use of Virtual Keyword

Advantages of Inheritance

1. Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
2. Reusability enhanced reliability. The base class code will be already tested and debugged.
3. As the existing code is reused, it leads to less development and maintenance costs.
4. Inheritance makes the sub classes follow a standard interface.
5. Inheritance helps to reduce code redundancy and supports code extensibility.
6. Inheritance facilitates creation of class libraries

Disadvantages of Inheritance

1. Inherited functions work slower than normal function as there is indirection.
2. Improper use of inheritance may lead to wrong solutions.
3. Often, data members in the base class are left unused which may lead to memory wastage.
4. Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes

Example of Inheritance

// C++ program to demonstrate implementation of Inheritance

```
#include <iostream>
using namespace std;
//Base class
class Parent
{
    public:
    int id_p=9;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
    int id_c=8;
};
//main function
int main()
{
    Child obj1;
    cout << "Child id is " << obj1.id_c << endl;
    cout << "Parent id is " << obj1.id_p << endl;
    return 0;
}
```

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public mode:

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode:

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode:

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

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

// C++ Implementation to show that a derived class

// doesn't inherit access to private data members.

// However, it does inherit a full parent object

```
class A
{
    public:
    int x;
    protected:
    int y;
    private:
```

```

    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};

```

Forms of Inheritance (Sub Class/Sub Types)

Inheritance is used in variety ways according to user's requirements. The Following are forms of inheritance.

- **Sub classing for specialization (subtype):** The derived child class is a specialized form of the parent class, in other words, the child class is subtype/subclass of the parent
- **Sub classing for specification:** The parent class defines behaviour that will be implemented in the child class. The inheritance for specification can be recognized when a parent class does not implement actual behaviour but it defines how the behaviour will be implemented in the child classes
- **Sub classing for construction:** The child class can be constructed from parent classes by implementing the behaviours of parent classes
- **Sub classing for generalization:** Sub classing for generalization is the opposite to sub classing for specifications. The base class holds the common properties that will inherit to the derived class.
- **Sub classing for extension:** The subclass for extension adds new functionality in child class from base class while designing new child class. It is done simply add new function other than parent class have.
- **Sub classing for limitation:** The subclass for limitation occurs when the behaviour of subclass is similar or more dependent to the behaviour of parent class.
- **Sub classing for variance:** The child class and parent class are varied when the level of inheritance increased, and the class and subclass relationship is imaginary.
- **Sub classing for combination:** The child class inherits features from more than one classes

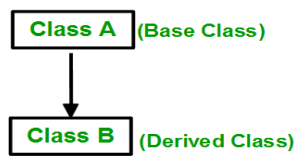
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

- a. Single Inheritance
- b. Multiple Inheritance
- c. Hierarchical Inheritance
- d. Multilevel Inheritance
- e. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

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

Example:

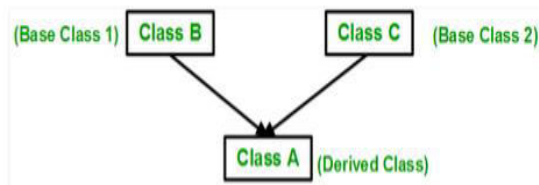
A class **Room** consists of two fields length and breadth and method int area() to find the area of room. A new class BedRoom is derived from class Room and consist of additional field height and two methods setData (int,int,int) to set the value for three fields and int volume() to find the volume. Now write the c++ program to input the length ,breadth and height and find the area and volume.

```
#include<iostream>
using namespace std;
class Room
{
    protected:
        float length, breadth;
    public:
        int area()
        {
            return(length*breadth);
        }
};
class BedRoom : public Room
{
    private:
        float height;
    public:
        void setData(int l,int b, int h)
        {
            length=l;
            breadth=b;
            height=h;
        }

        int volume()
        {
            return(length * breadth * height);
        }
};
int main()
{
    BedRoom b;
    b.setData(3,4,5);
    cout<<"Area of bedroom= "<<b.area()<<endl;
    cout<<"Volume of bedroom="<<b.volume();
}
```

2. Multiple Inheritance:

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



Syntax:

```

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

```

Example:

Q. Create two classes class1 and class2 each having data member for storing a number, a method to initialize it. Create a new class class3 that is derived from both class class1 and class2 and consisting of a method that displays the sum of two numbers from class1 and class2.

```

#include<iostream>
using namespace std;
class class1
{
    protected:
        int n;
    public:
        void getn(int p)
        {
            n=p;
        }
};
class class2
{
    protected:
        int m;
    public:
        void getm(int q)
        {
            m=q;
        }
};
class class3: public class1, public class2
{
    public:
        void displaytotal()
        {
            int tot;
            tot=n+m;
            cout<<"Sum ="<<tot;
        }
};
int main()
{
    class3 a;
    a.getm(4);
    a.getn(5);
    a.displaytotal();
}

```

Ambiguity/Problem with multiple inheritance

Sometime we have to face an ambiguity problem in using multiple inheritance when a function with the same name appears in more than one base class. Consider the following example

```

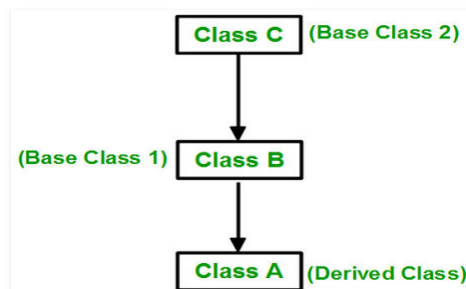
#include<iostream>
using namespace std;
class class1
{
    protected:
        int a;
    public:
        void get(int x)
        {
            a=x;
        }
};
class class2
{
    protected:
        int b;
    public:
        void get(int x)
        {
            b=x;
        }
};
class class3: public class1, public class2
{
    public:
        void displaytotal()
        {
            cout<<"Total= "<<(a+b);
        }
};
int main()
{
    class3 a;
    //a.get();// the request for get() is ambiguous as get() is defined in both class1 and class2
    a.class1::get(5);
    a.class2::get(6);
    a.displaytotal();
}

```

So we see that, When compilers of programming languages that support this type of multiple inheritance encounter, super classes that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. We can solve this problem by defining a named instance within the derived class using the class resolution operator with the function as shown in above example.

3. Multilevel inheritance

In multilevel inheritance, a class is derived from another subclass.



The general form is

```

class A
{
    //member of A
}
class B :public/private/protected A
{

```

```

        //own member of B
    }
class C :public/private/protected B
{
    //own member of C
}

```

In above case, class B is derived from Class A while class C is derived from derived class B. So, the class A serves as base class for B and B serves as base class for C therefore B is also called as **intermediate class** because it provides a link between class A and C.

Example:

A class Student consists of field roll, a method to assigns roll number. A new class Test is derived from class Student and consists of two new fields sub1 and sub2, a method to initialize these fields with obtained mark. Again, a new class Result is derived from Test and consists of a field total and a method to display entire details along with total obtained marks. WAP to input roll number, marks in two different subject and display total.

```

#include<iostream>
using namespace std;
class Student
{
    protected:
        int roll;
    public:
        void setroll(int r)
        {
            roll=r;
        }
};
class Test: public Student
{
    protected:
        float sub1, sub2;
    public:
        void setmark(float m1, float m2)
        {
            sub1=m1;
            sub2=m2;
        }
};
class Result : public Test
{
    private:
        float total;
    public:
        void display()
        {
            total=sub1+sub2;
            cout<<"Roll number= "<<roll<<endl;
            cout<<"Mark in first subject= "<<sub1<<endl;
            cout<<"Mark in second subject= "<<sub2<<endl;
            cout<<"Total= "<<total;
        }
};
int main()
{
    int r;
    float s1,s2;
    cout<<"Enter roll number"<<endl;
    cin>>r;
    cout<<"Enter marks in two subject"<<endl;
    cin>>s1>>s2;
    Result res;
    res.setroll(r);
}

```

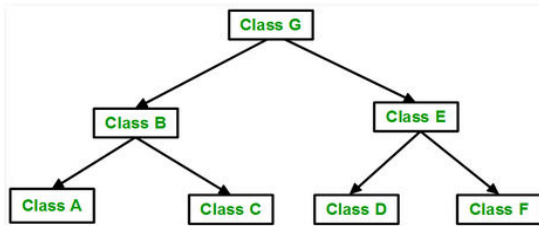
```

    res.setmark(s1,s2);
    res.display();
}

```

4. Hierarchical Inheritance

In hierarchical inheritance, two or more classes inherits the properties of one existing class.



The general form is:

```

class A
{
    //member of A
}
class B : public/private/protected A
{
    //own member of B
}
class C : pubic / private / protected A
{
    //own member of C
}

```

In above example, A is base class which includes all the features that are common to sub classes. Class B and Class C are the sub classes that shares the common property of class A and also can define their own property. Note we also can define a subclass that serve as base class for the lower level classes and so on.

Example:

A company needs to keep record of its following employees:

i) Manager ii) Supervisor

The record requires name and salary of both employees. In addition, it also requires section_name (i.e. name of section, example Accounts, Marketing, etc.) for the Manager and group_id (Group identification number, e.g. 205, 112, etc.) for the Supervisor. Design classes for the above requirement. Each of the classes should have a function called set() to assign data to the fields and a function called get() to return the value of the fields. Write a main program to test your classes. What form of inheritance will the classes hold in this case?

```

#include<iostream>
#include<string.h>
using namespace std;
class Employee
{
    private:
        char name[30];
        float salary;
    public:
        void setName(char *n)
        {
            strcpy(name,n);
        }
        void setSalary(float s)
        {
            salary=s;
        }
        char * getName()
        {
            return name;
        }
}

```

```

    }
    float getSalary()
    {
        return salary;
    }
};
class Manager: public Employee
{
    private:
        char section_name[50];
    public:
        void setSection_name(char *sn)
        {
            strcpy(section_name,sn);
        }
        char * getSection_name()
        {
            return section_name;
        }
};
class Supervisor: public Employee
{
    private:
        int group_id;
    public:
        void setGroup_id(int gid)
        {
            group_id=gid;
        }
        int getGroup_id()
        {
            return group_id;
        }
};

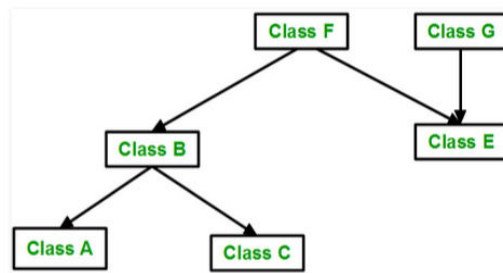
int main()
{
    Manager m;
    m.setName("Bhesh Bahadur Thapa");
    m.setSalary(50000);
    m.setSection_name("Accounts");
    cout<<"Name= "<<m.getName()<<endl;
    cout<<"Salary= "<<m.getSalary()<<endl;
    cout<<"Section= "<<m.getSection_name()<<endl;

    Supervisor s;
    s.setName("Sagar Kunwar");
    s.setSalary(40000);
    s.setGroup_id(5);
    cout<<"Name= "<<s.getName()<<endl;
    cout<<"Salary= "<<s.getSalary()<<endl;
    cout<<"Group ID= "<<s.getGroup_id()<<endl;
}

```

5. Hybrid inheritance

In hybrid inheritance , it can be the combination of single and multiple inheritance or any other combination.



One form can be as below

```

Class X
{
    //member of X
}
Class A : public/private/protected X
{
    // own member of A
}
Class B: public/private/protected A
{
    //own member of C
}
Class C: public/private/protected A
{
    //own member of C
}
Class D: public/private/protected A
{
    //own member of D
}
  
```

Above example consists both single and hierarchical inheritance hence called hybrid inheritance

Example:

The below example shows the hybrid inheritance i.e. combination of multilevel and multiple inheritance

Q. Create a class Student with data member roll_no and two functions to initialize and display it. Derive a new class Test which has two methods to assign and display marks in two subjects. Create a new class Sport with two functions that assign and display the score in sports. Now create another class Result that is derived from both class Test and Sport, having a function that displays the total of marks and score. Write a main program to test your class.

```

#include<iostream>
using namespace std;
class Student
{
    private:
        int roll;
    public:
        void setroll()
        {
            cout<<"Enter roll number"<<endl;
            cin>>roll;
        }
}
  
```

```

    }
    void showroll()
    {
        cout<<"Roll= "<<roll<<endl;
    }
};
class Test: public Student
{
    protected:
        float com,eng;
    public:
        void setmark()
        {
            cout<<"Enter marks of computer and English "<<endl;
            cin>>com>>eng;
        }
        void showmark()
        {
            cout<<"Computer= "<<com<<endl;
            cout<<"English= "<<eng<<endl;
        }
};
class Sport
{
    protected:
        float score;
    public:
        void setscore()
        {
            cout<<"Enter score in sports "<<endl;
            cin>>score;
        }
        void showscore()
        {
            cout<<"Score in sports= "<<score<<endl;
        }
};
class Result: public Test, public Sport
{
    private:
        float tot;
    public:
        void showtotal()
        {
            tot=com+eng+score;
            cout<<"Total obtained marks= "<<tot<<endl;
        }
};
int main()
{
    Result res;
    res.setroll();

```



```

        res.setmark();
        res.setscore();
        res.showroll();
        res.showmark();
        res.showscore();
        res.showtotal();
    }

```

Programming Examples

1. Define a shape class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.

- i. Derive triangle and rectangle classes from shape class adding necessary attributes.
- ii. Use these classes in main function and display the area of triangle and rectangle.

```

#include<iostream.h>
#include<conio.h>
class shape
{
    protected:
    float breadth, height, area;
    public:
    void getshapedata()
    {
        cout<<"Enter breadth:"<<endl;
        cin>>breadth;
        cout<<"Enter height:"<<endl;
        cin>>height;
    }
};

class triangle: public shape
{
    public:
    void calarea()
    {
        area=(breadth * height)/2;
    }
    void display()
    {
        cout<<"The area of triangle is"<<area<<endl;
    }
};

class rectangle: public shape
{
    public:
    void calarea()
    {
        area=breadth * height;
    }
    void display()
    {
        cout<<"Area of rectangle is"<<area<<endl;
    }
};

int main()
{
    triangle T;
    rectangle R;
    cout<<"Enter triangle data:"<<endl;
    T.getshapedata();
}

```

```

    cout<<"Enter rectangle data:"<<endl;
    R.getshapedata();
    T.calarea();
    R.calarea();
    T.display();
    R.display();
}

```

2. Define a *student* class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.

- i. Derive a *Computer Science and Mathematics* class from *student* class adding necessary attributes (at least three subjects).
- ii. Use these classes in a main function and display the average marks of computer science and mathematics students.

```

#include<iostream.h>
#include<conio.h>
class student
{
    protected:
        float english, sum, avg;
    public:
        void getstudentdata()
        {
            cout<<"Enter english marks:"<<endl;
            cin>>english;
        }
};

class computer : public student
{
    float IT, cprog, networks;
    public:
        void getcomputerdata()
        {
            cout<<"Enter marks in IT:"<<endl;
            cin>>IT;
            cout<<"Enter marks in cprog:"<<endl;
            cin>>Cprog;
            cout<<"Enter marks in networks:"<<endl;
            cin>>Networks;
        }
        void average()
        {
            sum=english+IT+cprog+networks;
            avg=sum/4;
            cout<<"Average marks is"<<avg;
        }
};

class mathematics : public student
{
    float calculus, stat, algebra;
    public:
        void getmathdata()
        {
            cout<<"Enter marks in calculus:"<<endl;
            cin>>calculus;
            cout<<"Enter marks in statistics:"<<endl;
            cin>>stat;
            cout<<"Enter marks in Linear Algebra:"<<endl;
            cin>>algebra;
        }
};

```

```

    }
    void average()
    {
        sum=english+calculus+stat+algebra;
        avg=sum/4;
        cout<<"Average marks is"<<avg;
    }
};

int main()
{
    computer C;
    mathematics M;
    cout<<"Enter marks of computer students:"<<endl;
    C.getstudentdata();
    C.getcomputerdata();
    cout<<"Enter marks of mathematics student:"<<endl;
    M.getstudentdata();
    M.getmathdata();
    C.average();
    M.average();
}

```

3. Define a *Clock* class (with necessary constructor and member functions) in OOP (abstract necessary attributes and their types). Write a complete code in C++ programming language.

- i. Derive *Wall_Clock* class from *Clock* class adding necessary attributes.
- ii. Create two objects of *Wall_Clock* class with all initial state to 0 or NULL.

```

#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class clock
{
protected:
    char model_no[10];
    float price;
    char manufacturer[50];
public:
    void getclockdata()
    {
        cout<<"Enter clock manufacturer:"<<endl;
        cin>>manufacturer;
        cout<<"Enter model number:"<<endl;
        cin>>model_no;
        cout<<"Enter price:"<<endl;
        cin>>price;
    }

    void clockdisplay()
    {
        cout<<"Model number="<<model_no<<endl;
        cout<<"Manufacturer="<<manufacturer<<endl;
        cout<<"Price="<<price<<endl;
    }
};

```

```

class wall_clock: public clock
{
    int hr, min, sec;
public:
    wall_clock()
    {
        strcpy(model_no,NULL);
        strcpy(manufacturer,NULL);
        price=0.0;
        hr=0;
        min=0;
        sec=0;
    }
    void getwallclockdata()
    {
        cout<<"Enter hour, minute and seconds:"<<endl;
        cin>>hr>>min>>sec;
    }
    void wallclockdisplay()
    {
        cout<<"Time="<<hr<<":"<<min<<":"<<sec<<endl;
    }
};

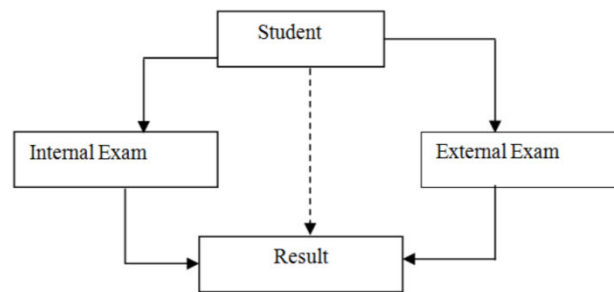
int main()
{
    wall_clock W1, W2;
    cout<<"Enter data for W1:"<<endl;
    W1.getclockdata();
    W1.getwallclockdata();
    cout<<"Value of W1:"<<endl;
    W1.clockdisplay();
    W1.wallclockdisplay();

    cout<<"Enter data for W2:"<<endl;
    W2.getclockdata();
    W2.getwallclockdata();
    cout<<"Value of W2:"<<endl;
    W2.clockdisplay();
    W2.wallclockdisplay();
}

```

Multi Path Inheritance

When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in Figure below.



Here, all three kinds of inheritance exist i.e. multilevel, multiple and hierarchical. The class Result has two direct base class Internal Exam and External exam which themselves have a common base class Student. So, the class Result inherits the class Student via two separate path called as multi path inheritance. The class student is sometime called as indirect base class. It also can be inherited directly as shown by the broken line.

Virtual Base Class

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

Problem with multipath inheritance and virtual base class

Since all the public and protected members of indirect base class i.e. Student are inherited into final derived class i.e. Result via two paths, first via Internal Exam and second via External exam. This means the class Result would have duplicate sets of members inherited from Student. This introduces ambiguity and should be avoided.

This ambiguity can be avoided by making a common base class as virtual base class while declaring the direct or intermediate base classes as shown below.

```

class Student //grand parent
{
};
Class Internal_Exam: virtual public Student //parent1
{
};
Class External_Exam: virtual public A//parent2
{
}
Class Result: public Internal_Exam, public External_Exam//child
{
    //only one copy of A will be inherited
}
  
```

So, when a class is made virtual base class, c++ takes care to see that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

Note: the use of keyword virtual and public can be done in any order

Programming example:

Create a class Student with data member roll_no and two functions to initialize and display it. Derive two new classes Theory and Practical from Student. Define suitable functions to assign and display theory and practical marks for two

different subjects. Again, derive a new class Result from both class Theory and Practical and add a new function to display the final total marks of student. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
    private:
        int roll;
    public:
        void setroll()
        {
            cout<<"Enter roll number"<<endl;
            cin>>roll;
        }
        void showroll()
        {
            cout<<"Roll= "<<roll<<endl;
        }
};
class Theory: public virtual Student
{
    protected:
        float comth,ength;
    public:
        void setdatatheory()
        {
            cout<<"Enter Theory marks of computer and English "<<endl;
            cin>>comth>>ength;
        }
        void showmarkstheory()
        {
            cout<<"Computer(Theory)= "<<comth<<endl;
            cout<<"English(Theory)= "<<ength<<endl;
        }
};
class Practical: public virtual Student
{
    protected:
        float compr,engpr;
    public:
        void setdatapractical()
        {
            cout<<"Enter Practical marks of computer and English "<<endl;
            cin>>compr>>engpr;
        }
        void showmarkspractical()
        {
            cout<<"Computer(Practical)= "<<compr<<endl;
            cout<<"English(Practical)= "<<engpr<<endl;
        }
};
class Result: public Theory, public Practical
{
    public:
        void showtotal()
        {
            float tot;
            tot=comth+ength+compr+compr;
            cout<<"Total obtained marks= "<<tot<<endl;
        }
};
```

```

int main()
{
    Result res;
    res.setroll(); //ambiguous because multipath exist to reach setroll() from derived class so must use virtual base class to overcome this
    res.setdatatheory();
    res.setdatapractical();
    res.showroll();
    res.showmarkstheory();
    res.showmarkspractical();
    res.showtotal();
}

```

```

Enter roll number
5
Enter Theory marks of computer and English
70
75
Enter Practical marks of computer and English
20
25
Roll= 5
Computer(Theory)= 70
English(Theory)= 75
Computer(Practical)= 20
English(Practical)= 25
Total obtained marks= 185

```

Constructor and Destructor in Inheritance

- Constructor is used to initialize variables and allocation of memory of object
- Destructor is used to destroy objects
- Compiler automatically calls constructor of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance constructor of base class is executed first and then constructor of derived class
- If derived class object goes out of scope or deleted by programmer the derived class destructor is executed first and then base class destructor

Default Constructor (No arguments) in inheritance

- In this case it is not compulsory to have derived class constructor if base class have a constructor
- Example Default Constructor in inheritance

```

class A { //Base Class

public:

A() {                                     //Base Class A Constructor

cout<<"Constructor Class A"<<endl;

}

                                     //Base Class A Destructor

~A()

{

cout<<"Destructor Class A"<<endl;

}

};

class B:public A                       //Derived class

```

```

{
public:
B()                                //Derived Class B Constructor
{
cout<<"Constructor Class B"<<endl;
}
~B() {                             //Derived Class B Destructor
cout<<"Destructor Class B"<<endl;
}
};
int main()
{
B obj;                             //Derived class object obj
}

```

In Above Program Class A is base class with one constructor and destructor, Class B is derived from class A having a constructor and destructor. In main() Object of derived class i.e. class B is declared. When class B's object is declared constructor of base class is executed followed by derived class constructor. At end of program destructor of derived class is executed first followed by base class destructor.

Output:

Constructor Class A

Constructor Class B

Destructor Class B

Destructor Class A

Parameterized constructor (with arguments) in inheritance

- In parameterized constructor it is compulsory to have derived constructor if there base class constructor.
- Derived class constructor is used to pass arguments to the base class.
- If derived class constructor is not available, it is not possible to pass arguments from derived class object to base class constructor

Example of parameterized constructor passing different argument for base class and derived class

```

class A //Base class
{
protected:
int a;
A(int x) //Base class constructor with one argument(x)
{
a=x;
cout<<"Constructor : Class A : value : "<<a<<endl;
}
}

```



```

};

class B:public A //Derived class B from base class A
{
protected:
int b;
public:
//derived class constructor passing arguments(y, z)
//y is used in derived class and z is passed base class A
B(int y,int z):A(z)
{
b=y;
cout<<"Constructor : Class B value : "<<b<<endl;
}
};

int main()
{
//derived class object passing arguments to derived & base class
B obj(5,3);
}

```

Principal of Substitutability

Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program.

Subtype

The term “subtype” is used to describe the relationship between different types that follow the principle of “substitution”. If we consider two types (classes or interfaces) A and B, type B is called as a subtype of A, if the following two conditions are satisfied:

- a. The instance (object) of type B can be legally assigned to the variable of type A.
- b. The instance (object) of type B can be used by the variable of type A without any observable change in its behavior.

Subclass Vs Subtype

- ✓ To say that A is a subclass of B, declares that A is formed using inheritance.
- ✓ To say that A is a subtype of B, declares that A preserves the meaning of all the operations in B.

Object Composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a “has-a” relationship between two objects. A car “has-a” transmission. Your computer “has-a” CPU. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail. Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body cannot be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person's body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a "death relationship".

IS-A and HAS-A Relation

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).

IS-A Rule or Relationship:

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing".

For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

HAS-A Rule (Container Class/Containership/Composition)

HAS rule means division into parts.

Consider the term "A car has an engine", "A bus has an engine". We cannot use engine the super class properties to all child classes because the engine has different for all transportation means. The engine is independent properties that cannot be directly called to the car, bus classes. We can include the class engine or class brake in class car not derive them

It is also called container class or containership because object of one class will be used inside other class which means member of one class is available in other class.

Example for Container Class/composition/has-a rule

```
class A
{
    int x;
    public:
    A()
    {
        x=1;
    }
    void display1()
    {
        cout<<x;
    }
};
class B //Class B
{
    int y;
    A a; //Object of Class A declared in class B
```

```

public:
B()    //Constructor Class B
{
    y=9;
}
void display2()
{
    a.display1 (); //Calling member of Class A from member of B
    cout<<y;
}
};

int main()
{
    B b; //object of class B
    b.display2();
}

```

In Above program there are 2 class A & B, Class A have a data member integer x and one default constructor and one member function display1 (), Class B have 2 data member one is integer y and another is object (a) of class A and one constructor and one member function i.e. display2() and inside display2 we call member function of class A i.e. display1().

Here class A's object is declared inside class B, so class B contains members of class A or class B HAS A object of class A (Class B is composite class).

Composition Vs Inheritance

Composition	Inheritance
Composition indicates the operation of an existing structure	Inheritance is a super set of existing structure
Cannot reuse code directly but provide greater functionality	Inheritance can be directly reused the code and function provided by parent class
Code become shorter than inheritance	Code become longer than composition
Very easy to re-implement the behaviors and functions	Difficult to re-implement the behaviors
Example:	Example:

Chapter 5

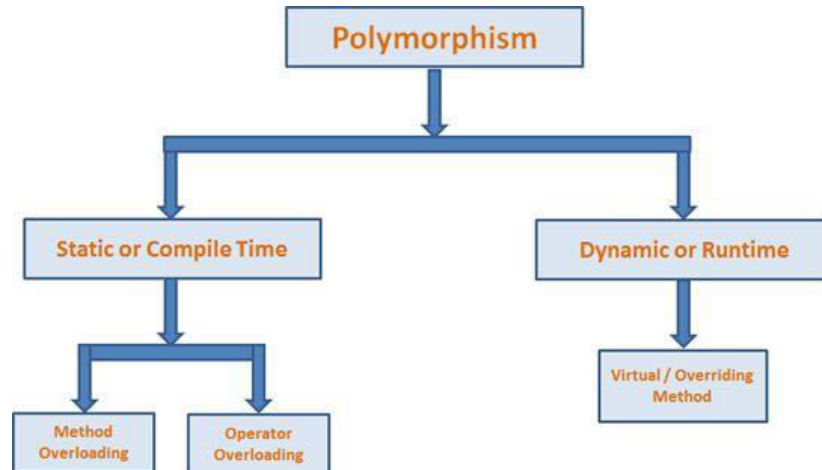
Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism is considered as one of the important features of Object Oriented Programming.

Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee. So a same person possess have different behavior in different situations.



In C++ polymorphism is mainly divided into two types:

- ✓ Compile time Polymorphism
- ✓ Runtime Polymorphism

1. Static or Parametric or Compile time polymorphism

Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is also called early binding. In this the compiler selects the appropriate function during the compile time.

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

➤ **Function Overloading:**

When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.

Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

```
// C++ program for function overloading
using namespace std;
class Geeks
{
    public:
    // function with one int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
    // function with same name but one double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }
}
```

```

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

};

int main()
{
    Geeks obj1;
    obj1.func(7);           // The first 'func' is called
    obj1.func(9.132);       // The second 'func' is called
    obj1.func(85,64);       // The third 'func' is called
    return 0;
}

```

Output:

```

value of x is 7
value of x is 9.132
value of x and y is 85, 64

```

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

➤ **Operator Overloading:**

The concept by which we can give special meaning to an operator of C++ language is known as operator overloading.

An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading. Operator overloading allows us to assign multiple meanings to the operators. Compiler generates the appropriate code based on the manner in which the operator is used.

When an operator is overloaded, that operator does not lose its original meaning. Instead, it gains additional meaning relative to the class for which it is defined. We can overload the C++ operators except the following:

- ✓ Scope Resolution Operator (: :)
- ✓ Membership Operator (.)
- ✓ Size of operator (size of)
- ✓ Conditional Operator (? :)
- ✓ Pointer to Member Operation (.*)

Syntax for operator overloading:

The operator overloading is done by using a special function called operator function, called operator function. The general syntax for operator function is:

```

return_type operator_op (arg list)
{
    Function body
}

```

For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

Example:

a. Overloading Unary Operator

```
#include <iostream>

using namespace std;

class index
{
    public:
    int count;
    void getdata(int i)
    {
        count=i;
    }
    void showdata ()
    {
        cout<< "count=" <<count<<endl;
    }
    void operator++()
    {
        ++count;
    }
};

int main()
{
    index a1;
    a1.getdata(3);
    a1.showdata();
    ++a1;
```

```

        a1.showdata();

    return 0;

}

```

b. Overloading Binary Operator

```

// CPP program to illustrate Operator Overloading
#include<iostream>
using namespace std;
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {
        real = r;
        imag = i;
    }
    // This is automatically called when '+' is used with between twoComplex
objects
    Complex operator + (Complex &obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print()
    {
        cout << real << " + i" << imag << endl;
    }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
Output:
12 + i9

```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

2. Dynamic or Subtype or Runtime Polymorphism

If a member function is selected while the program is running, then it is called run time polymorphism. This feature makes the program more flexible as a function can be called, depending on the context. This is also called late binding. The example of run time polymorphism is virtual function.

This type of polymorphism is achieved by Function Overriding. Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example:

```
#include <iostream>
using namespace std;
// Base class
class Parent
{
    public:
    void print()
    {
        cout << "The Parent print function was called" << endl;
    }
};
// Derived class
class Child : public Parent
{
    public:
    // definition of a member function already present in Parent
    void print()
    {
        cout << "The child print function was called" << endl;
    }
};
int main()
{
    Parent obj1; //object of parent class
    Child obj2 = Child(); //object of child class
    obj1.print(); // obj1 will call the print function in Parent
    obj2.print(); // obj2 will override the print function in Parent and call the print function in Child
    return 0;
}
```

```
The Parent print function was called
The child print function was called
```

Q. Difference between function overloading and function overriding

Basis for Comparison	Overloading	Overriding
Prototype	Prototype differs as number or type of parameter may differ.	All aspect of prototype must be same.
Keyword	No keyword applied during overloading.	Function which is to be overridden is preceded by keyword 'virtual', in the base class.
Distinguishing factor	Number or type of parameter differs which determines the version of function is being called.	Which class's function is being called by the pointer, is determined by, address of which class's object is assigned to that pointer.

Basis for Comparison	Overloading	Overriding
Defining pattern	Function are redefined with same name, but different number and type of parameter.	Function is defined, preceded by a keyword 'virtual' in main class and redefined by derived class without keyword.
Time	Compile time.	Run time.
Constructor/Virtual function	Constructors can be overloaded.	Virtual function can be overridden.
Destructor	Destructor cannot be overloaded.	Destructor can be overridden.
Binding	Overloading achieves early binding.	Overriding refers to late binding.
Example	<pre>void area(int a); void area(int a, int b);</pre>	<pre>Class a { public: virtual void display() { cout << "hello"; } }; Class b:public a { public: void display() { cout << "bye"; }; };</pre>

Difference between early and late binding

	Early Binding	Late Binding
1	It is also known as compile time polymorphism because compiler selects the appropriate member function for the particular function call at the compile time.	It is also called run time polymorphism because the appropriate member functions are selected while the program is executing or running.
2	The information regarding which function to invoke that matches a particular call is known in advance during compilation. Hence it is also called early binding.	The compiler does not know which function to bind with particular function call until the program is executed.
3	This type of binding is achieved using function and operator overloading.	This type of binding is achieved using virtual function.
4	The function call is linked with particular function at compile time statically. So, it is also called static binding.	The selection of appropriate function is done dynamically at run time. So, it is also called dynamic binding.

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.

- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. They must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Example:

```
#include<iostream>
using namespace std;
class base
{
public:
    virtual void print ()
    {
        cout<< "print base class" <<endl;
    }

    void show ()
    {
        cout<< "show base class" <<endl;
    }
};
class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
```

```

        bptr->show();
    }

```

Pure Virtual Functions (or deferred method or abstract method):

It is the special case of overriding. It's possible that we'd want to include a virtual function in a base class so that it may be redefined in a derived class, but there is no meaningful definition we could give for the function in the base class. It can be defined in the base class but not implemented. The child class provides its implementation.

```

class Shape
{
    protected:
        int width, height;
    public:
        Shape( int a=0, int b=0)
        {
            width = a; height = b;
        }
        // pure virtual function virtual
        virtual int area() = 0;
};

```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Example:

```

#include <iostream>
using namespace std;
class b
{
    public:
        virtual void show()=0;
};

class Derived: public b
{
    public:
        void show()
        {
            cout<< "inside derived class" ;
        }
};

int main()
{
    Derived d;
    b *p;
    p = &d;
    p ->show();
}

```

A class that contains pure virtual function is known as pure abstract class.

Difference between run time and compile time polymorphism

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the compiler .	In Run time Polymorphism, call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers .
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

Coercion Polymorphism (Casting) or Type conversion

Coercion happens when an object or a primitive is cast into another object type or primitive type.

For example,

```
float b = 6; // int gets promoted (cast) to float implicitly
```

```
int a = 9.99 // float gets demoted to int implicitly
```

Explicit casting happens when we use C's type-casting expressions

Three types of situations might arise for data conversion between different types:

- (i) Conversion from basic type to class type.
- (ii) Conversion from class type to basic type.
- (iii) Conversion from one class type to another class type.

Conversion from basic type to class type

In this type of conversion the source type is basic type and the destination type is class type. Means basic data type is converted into the class type.

```
class_object = basic_type_variable;
```

- For example we have class *employee* and one object of employee '*emp*' and suppose we want to assign the employee code of employee '*emp*' by any integer variable say '*Ecode*' then the statement below is the example of the conversion from basic to class type.

```
emp = Ecode ;
```

Here the assignment will be done by converting “*Ecode*” which is of basic or primary data type into the class type

The conversion from basic type to the class type can be performed by two ways:

1. Using constructor

```
/* Program to convert basic type to class type using constructor */

#include "iostream.h"
#include "conio.h"
class Time
{
    int hrs,min;
    public:
        Time(int);
        void display();
};

Time :: Time(int t)
{
    cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
    hrs=t/60;
    min=t%60;
}

void Time::display()
{
    cout<<hrs<< ": Hours(s)" <<endl;
    cout<<min<< " Minutes" <<endl;
}

void main()
{
    clrscr();

    int duration;
    cout<<"Enter time duration in minutes";
    cin>>duration;

    Time t1=duration;

    t1.display();
    getch();
}
```

Here, we have created an object “*t1*” of class “*Time*” and during the creation we have assigned integer variable “*duration*”. It will pass time duration to the constructor function and assign to the “*hrs*” and “*min*” members of the class “*Time*”

2. Using Operator Overloading

```
/* Program to convert from basic type to class type using operator overloading */

#include "iostream.h"
#include "conio.h"
class Time
```

```

{
    int hrs,min;
    public:
        void display();
        void operator=(int); // overloading function
};
void Time::display()
{
    cout<<hrs<< ": Hour(s) "<<endl ;
    cout<<min<<": Minutes"<<endl ;
}

void Time::operator=(int t)
{
    cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
    hrs=t/60;
    min=t%60;
}

void main()
{
    clrscr();
    Time t1;
    int duration;
    cout<<"Enter time duration in minutes";
    cin>>duration;
    cout<<"object t1 overloaded assignment..."<<endl;
    t1=duration;
    t1.display();
    cout<<"object t1 assignment operator 2nd method..."<<endl;
    t1.operator=(duration);
    t1.display();
    getch();
}

```

Conversion from class type to basic type

In this type of conversion the source type is class type and the destination type is basic type. Means class data type is converted into the basic type.

basic_type_variable = class_object;

For example we have class *Time* and one object of *Time* class '*t*' and suppose we want to assign the total time of object '*t*' to any integer variable say '*duration*' then the statement below is the example of the conversion from class to basic type.

duration= t ; // where, t is object and duration is of basic data type

Example:

/ Program to demonstrate Class type to Basic type conversion. */*

```

#include "iostream.h"
#include "conio.h"
#include "iomanip.h"

```

```

class Time
{
    int hrs,min;

```

```

        public:
            Time(int ,int); // constructor
            operator int(); // casting operator function
            ~Time()         // destructor
            {
                cout<<"Destructor called..."<<endl;
            }
};

Time::Time(int a,int b)
{
    cout<<"Constructor called with two parameters..."<<endl;
    hrs=a;
    min=b;
}

Time :: operator int()
{
    cout<<"Class Type to Basic Type Conversion..."<<endl;
    return(hrs*60+min);
}

void main()
{
    clrscr();
    int h,m,duration;
    cout<<"Enter Hours ";
    cin>>h;
    cout<<"Enter Minutes ";
    cin>>m;
    Time t(h,m);    // construct object
    duration = t;    // casting conversion OR duration = (int)t
    cout<<"Total Minutes are "<<duration;
    cout<<"2nd method operator overloading "<<endl;
    duration = t.operator int();
    cout<<"Total Minutes are "<<duration;

    getch();
}

```

Conversion from one class type to another class type

In this type of conversion both the type that is source type and the destination type are of class type. Means the source type is of class type and the destination type is also of the class type. In other words, one class data type is converted into another class type.

Object_of_destination_class = Object_of_source_type

For example we have two classes one for “*computer*” and another for “*mobile*”. Suppose if we wish to assign “*price*” of *computer* to *mobile* then it can be achieved by the statement below which is the example of the conversion from one class to another class type.

mob = comp ; // where mob and comp are the objects of mobile and computer classes respectively.

There are two ways to convert from one class type to another class type conversion

1) By using constructors

Here source class should have constructor . And destination class have a conversion routine. Conversion routine (constructor) would have source class object as an argument.

Example: //conversion from polar to rectangular coordinate:

```

#include<iostream>
#include<math.h>
using namespace std;
class polar
{
public:
float r,th;
polar(){}
polar(int a,int b)
{
r=a;
th=b;
}
void show()
{
cout<<"In polar form:\nr="<<r<<" and theta="<<th;

}
};
class rectangular
{
float x,y;
public:
rectangular(){}
rectangular(polar p)
{
x=p.r*cos(p.th);
y=p.r*sin(p.th);
}
void show()
{
cout<<"\nIn Rectangular form:\nx="<<x<<"and y="<<y;

}
};
int main()
{
    polar p(5.5,3.14/2);
    p.show();
    rectangular r;
    r=p;
    r.show();

}

```

II) by using casting operator (i.e. operator function)

Here source class contains conversion routine(i.e. operator function) and destination class have constructor.


```
// conversion from polar to rectangular coordinate  
include<iostream.h>
```

```
#include<math.h>
```

```
const double pi=3.141592654;
```

```
class rectangular
```

```
{
```

```
double x,y;
```

```
public:
```

```
rectangular()
```

```
{
```

```
x=0;
```

```
y=0;
```

```
}
```

```
rectangular(double a,double b)
```

```
{
```

```
x=a;
```

```
y=b;
```

```
}
```

```
void output()
```

```
{
```

```
cout<<"("<<x<<" "<<y<<"")";
```

```
}
```

```
};
```

```

class polar
{
double theta,r;

public:
polar ()
{
theta=0;
r=0;
}

operator rectangular()
{
double x,y;

//float atheta=theta*pi/180;

x=r*cos(theta);
y=r*sin(theta);

return rectangular(x,y);
}

void output()
{
cout<<"\nr="<<r;

cout<<"\ntheta="<<theta;

}

};

```

```

int main()

{

rectangular r1,r2;

polar p1(3,45);
r1=p; // polar to rectangular conversion
r1.display();
}

```

Example:

/* Program to convert class Time to another class Minute.

```

#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
class Time
{
    int hrs,min;
public:
    Time(int h,int m)
    {
        hrs=h;
        min=m;
    }
    Time()
    {
        cout<<"\n Time's Object Created";
    }
    int getMinutes()
    {
        int tot_min = ( hrs * 60 ) + min ;
        return tot_min;
    }
    void display()
    {
        cout<<"Hours: "<<hrs<<endl ;
        cout<<" Minutes : "<<min <<endl ;
    }
};
class Minute
{
    int min;
public:

    Minute()
    {
        min = 0;
    }
};

```

```

    }
    void operator=(Time T)
    {
        min=T.getMinutes();
    }
    void display()
    {
        cout<<"\n Total Minutes : " <<min<<endl;
    }
};
void main()
{
    clrscr();
    Time t1(2,30);
    t1.display();
    Minute m1;
    m1.display();

    m1 = t1;    // conversion from Time to Minute

    t1.display();
    m1.display();
    getch();
}

```

Polymorphic Variable (Assignment Polymorphism):

The variable that can hold different types of values during the course of execution is called polymorphic variable. It can also be defined as the variable that is declared as one class and can hold values from subclass. In C++, it works with pointers and references. **Pure polymorphism** means a function with polymorphic variable as arguments (or parameters).

This Pointer

If only one copy of each member function exists and is used by multiple objects, then for proper data members accessing and updating, Compiler supplies an implicit pointer along with the functions names as 'this'. **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. Friend functions do not have **this** pointer, because friends are not members of a class. Only member functions have **this** pointer.

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

```

#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
    }
}

```

```

        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}

```

2) To return reference to the calling object

```

#include<iostream>
using namespace std;
class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test&setX(int a) { x = a; return *this; }
    Test&setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

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

    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}

```

Method overriding

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

```

#include <iostream>

using namespace std;

class BaseClass {

```

```
public:

void disp()
{
    cout<<"Function of Parent Class";
}

};

class DerivedClass: public BaseClass{

public:

void disp() // it overrides the method of class Baseclasss
{
    cout<<"Function of Child Class";
}

};
```

Chapter 6

Template and generic programming

Template

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any type.
- A template is a blueprint or formula for creating a generic class or a function.
- A template is one of the recently added feature in C++. It supports the generic data types and generic programming.
- Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types.

Example:

- i. A class template for an **array class** would enable us to create arrays of various data types such as int array and float array.
 - ii. A function template say mul() can be used for multiplying int, float and double type values.
- A Template can be considered as Macro. When an object of specific type is defined for actual use, the template definition for that class is substituted with the required data type.
 - Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the template is also called as parametrized class or functions.

Features of Template

- i. Templates are easier to write. We can create only one generic version of our class or function instead of manually creating specializations.
- ii. Templates are easier to understand, since they provide a straightforward way of abstracting type information.
- iii. Templates are type safe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Class Template

The general form of a generic class declaration is shown here:

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

Advantage of class Template

- i. One C++ Class Template can handle different types of parameters.
- ii. Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the C++ template class.

- iii. Templates reduce the effort on coding for different data types to a single set of code.
- iv. Testing and debugging efforts are reduced.

Disadvantage of Template

- i. Many compilers historically have very poor support for templates, so the use of templates can make code somewhat less portable.
- ii. Almost all compilers produce confusing, unhelpful error messages when errors are detected in template code. This can make templates difficult to develop
- iii. Each use of a template may cause the compiler to generate extra code (an instantiation of the template), so the indiscriminate use of templates can lead to code bloat, resulting in excessively large executable.

Example:

1. WAP to demonstrate the concept of generic class.

```
#include<iostream>
using namespace std;
template<class T>
class Demo
{
    private:
        T x;
    public:
        Demo( T p)
        {
            x=p;
        }
        void show()
        {
            cout<<"\n"<<x;
        }
};
int main()
{
    Demo <char *>p1("Ram");
    Demo <char> p2 ('A');
    Demo <int> p3(5);
    Demo <float>p4 (5.5);
    p1.show ();
    p2.show ();
    p3.show ();
    p4.show ();
}
```



```
}
```

Output:

Ram

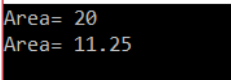
A

5

5.5

2. WAP to find the area of circle using the concept of generic class.

```
#include <iostream>
using namespace std;
template<class T> // here T is generic data type
class Rectangle
{
    private:
        T len,bre;
    public:
        Rectangle(T l, T b)
        {
            len=l;
            bre=b;
        }
        T area()
        {
            return(len*bre);
        }
};
int main()
{
    Rectangle<int>r1(4,5);
    cout<<"Area= "<<r1.area()<<endl;
    Rectangle <float>r2(4.5, 2.5);
    cout<<"Area= "<<r2.area()<<endl;
}
```



```
Area= 20
Area= 11.25
```

In the above program, to define a class template Rectangle, we must prefix its definition by template<class T>. This prefix tells the compiler that we are going to declare a template and use T as a type name in the class definition. Thus, Rectangle has become parameterized class with T as its parameter. So, T can be substituted by any data type like int, float or any user defined data type.

Here in the example the statement Rectangle<int>r1(4,5); initialize T as int type and hence the class constructor receive integer type argument and method T area() also returns integer type value. Similarly, the statement Rectangle <float>r2(4.5, 2.5); initialize T as float type and hence the class constructor receive floating type argument and method T area() also returns Floating type value

Or , we can write the same program as below:

```
#include <iostream>
using namespace std;
template<class T> // here T is generic data type
class Rectangle
{
    private:
        T len,bre;
```

```

        public:
            setdata(T x, T y)
            {
                len=x;
                bre=y;
            }
            T area()
            {
                return(len*bre);
            }
    };
    int main()
    {
        Rectangle<int> r1;
        r1.setdata(4,5);
        cout<<"Area= "<<r1.area()<<endl;
        Rectangle <float> r2;
        r2.setdata(4.5,2.5);
        cout<<"Area= "<<r2.area()<<endl;
    }

```

3. WAP to find the sum of two complex number using the concept of generic class.

```

#include <iostream>
using namespace std;
template<class T>
class complex
{
    T real;
    T img;
public:
    complex()
    {
        real=0;
        img=0;
    }
    complex(T f1, T f2)
    {
        real=f1;
        img=f2;
    }

    complex sum(complex tmp)
    {
        complex result;
        result.real = tmp.real+real;
        result.img = tmp.img+img;
        return result;
    }

    void show()
    {
        cout<<"Real= "<<real<<endl;
        cout<<"Imaginary= "<<img<<endl;
    }
};

int main()
{
    //finding the sum of two integer complex number

```

```

    complex<int> c1(3,6);
    complex<int> c2(2,-2);
    complex<int> c3;//will invoke default constructor
    c3=c1.sum(c2);
    c3.show();

    //finding the sum of two float complex number
    complex<float> c4(3.5,6.0);
    complex<float> c5(2.5,-2.5);
    complex<float> c6;//will invoke default constructor
    c6=c4.sum(c5);
    c6.show();
}

```

```

1 Real= 5
2 Imaginary= 4
3 Real= 6
4 Imaginary= 3.5

```

4. WAP to find the sum of two complex number using the concept of generic class and operator overloading.

```

#include <iostream>
using namespace std;
template<class T>
class complex
{
    T real;
    T img;
public:
    complex()
    {
        real=0;
        img=0;
    }

    complex(T f1, T f2)
    {
        real=f1;
        img=f2;
    }

    complex operator +(complex tmp)
    {
        complex result;
        result.real = tmp.real+real;
        result.img = tmp.img+img;
        return result;
    }
    void show()
    {
        cout<<"Real= "<<real<<endl;
        cout<<"Imaginary= "<<img<<endl;
    }
};

int main()
{
    //finding the sum of two integer complex number
    complex<int> c1(3,6);
    complex<int> c2(2,-2);
    complex<int> c3;//will invoke default constructor
    c3=c1+c2; // equivalent to c3=c1.operator +(c2);
    c3.show();
}

```

```

//finding the sum of two float complex number
complex<float> c4(3.5,6.0);
complex<float> c5(2.5,-2.5);
complex<float> c6;//will invoke default constructor
c6=c4+c5; // equivalent to c6=c4.operator+(c5);
c6.show();
}

```

```

Real= 5
Imaginary= 4
Real= 6
Imaginary= 3.5

```

Class Template with multiple parameters

When we have to use more than one generic data type in a class template, it can be achieved by using a comma-separated list within the template specification as below.

```

template<class T1, class T2.....class TN>
Class class_name
{

};

```

Example:

```

#include <iostream>
using namespace std;
template<class T1, class T2 >
class Demo
{
    private:
        T1 a;
        T2 b;
    public:
        Demo(T1 x, T2 y)
        {
            a=x;
            b=y;
        }
        void show()
        {
            cout<<"A= "<<a<<endl;
            cout<<"B= "<<b<<endl;
        }
};

int main()
{
    Demo<int,float> d1(1,4.5);
    d1.show();

    Demo<char,int> d2('B',6);
    d2.show();

    Demo <char *, float>d3("bhesh",6);
    d3.show();
}

```

```
A= 1
B= 4.5
A= B
B= 6
A= bhash
B= 6
```

Function Template

Like class, we can also define function templates that can be used to create a family of functions with different arguments types. The general format for function template is given below

```
template<class T>
returnType of type T function_name (argument of type T)
{
    //body of function with type T whenever appropriate
}
```

Note: Main function can't be declared as template

Example:

1. WAP to declare template function that can be used to find the area of rectangle.

```
#include<iostream>
using namespace std;
template<class t>
t area(t len, t bre)
{
    return(len*bre);
}
int main()
{
    int l1=6,b1=4;
    cout<<"Area= "<<area(l1,b1)<<endl;
    float l2=2.5,b2=2.0;
    cout<<"Area= "<<area(l2,b2);
}
```

2. WAP to declare a function template that can be used to swap two values of a given type of data.

```
//using concept of pointer
#include <iostream>
using namespace std;
template<class T >
void swap( T *fn, T *sn )
{
    T tmp;
    tmp=*fn;
    *fn=*sn;
    *sn=tmp;
}

int main()
{
    int m=5,n=6;
    cout<<"Before swapping"<<endl;
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
    swap(&m,&n);
    cout<<"After swapping"<<endl;
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

Or

```
//Using concept of reference variable
#include <iostream>
using namespace std;
template<class T >
void swap( T &fn, T &sn )
{
    T tmp;
    tmp=fn;
    fn=_sn;
    fn=tmp;
}

int main()
{
    int m=5,n=6;
    cout<<"Before swapping";
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
    swap(m,n);
    cout<<"After swapping";
    cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

3. Create a function template that can be used to find the maximum numbers among 10 numbers of given type of data(int, float etc)

```
#include <iostream>
using namespace std;
template<class T >
T Max( T data[] )
{
    T great=data[0];
    for(inti=0;i<10;i++)
    {
        if(data[i]>great)
            great=data[i];
    }
    return great;
}

int main()
{
    int a[10]={4,3,7,9,8,5,88,34,23,11};
    cout<<"Largest number= "<<Max(a)<<endl;

    float b[10]={4.5,3.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
    cout<<"Largest number= "<<Max(b);
}
```

```
Largest number= 88
Largest number= 99.8
```

4. Create a function templates that can be used to find the minimum and maximum numbers among 10 numbers of given type of data(int, float etc)

```
#include <iostream>
using namespace std;
template<class T >
T Max( T data[] )
{
    T great=data[0];
    for(inti=0;i<10;i++)
    {
        if(data[i]>great)
```

```

        great=data[i];
    }
    return great;
}

template<class T >
T Min( T data[] )
{
    T small=data[0];
    for(inti=0;i<10;i++)
    {
        if(data[i]<small)
            small=data[i];
    }
    return small;
}

int main()
{
    int a[10]={4,3,7,9,8,5,88,34,23,11};
    cout<<"Largest number= "<<Max(a)<<endl;
    cout<<"Smallest number= "<<Min(a)<<endl;

    float b[10]={4.5,2.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
    cout<<"Largest number= "<<Max(b)<<endl;
    cout<<"Smallest number= "<<Min(b)<<endl;
}

```

```

Largest number= 88
Smallest number= 3
Largest number= 99.8
Smallest number= 2.6

```

5. Create a function template to add two matrices.

```

#include<iostream>
using namespace std;
template<class t>
void add(t x [][][10], t y [][][10], t row, t col)
{
    tsm[10][10];
    for(inti=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            sm[i][j]=x[i][j]+y[i][j];
        }
    }
    for(inti=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            cout<<sm[i][j]<<" ";
        }
        cout<<endl;
    }
}

int main()
{
    int m[10][10],n[10][10],row,col;
    cout<<"Enter the number of rows and column"<<endl;
    cin>>row>>col;
    cout<<"Enter the elements of first matrix"<<endl;
    for(inti=0;i<row;i++)

```

```

    {
        for(int j=0;j<col;j++)
        {
            cin>>m[i][j];
        }
    }
    cout<<"Enter the elements of second matrix"<<endl;
    for(inti=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            cin>>n[i][j];
        }
    }
    add(m,n,row,col);
}

```

Function template with multiple parameters

Like template class, we can use more than one generic data type in the template statement, using a comma separated list as shown below.

```

template<class t2, class t2.....>
Return_typefunction_name(arguments of types T1, T2.....)
{

}

```

Example:

```

#include <iostream>
using namespace std;
template<class T1, class T2 >
void show(T1 x, T2 y)
{
    cout<<"First Parameter= "<<x<<endl;
    cout<<"Second Paramter= "<<y<<endl;
}
int main()
{
    show(1,2);
    show(1.4,3.5);
    show('r',"Ram");
}

```

```

First Parameter= 1
Second Paramter= 2
First Parameter= 1.4
Second Paramter= 3.5
First Parameter= r
Second Paramter= Ram

```

Overloading of Template Function

A template function can be overloaded either by template functions or ordinary functions of its name. In such case, overloading resolution is accomplished as follows

- i. Calling an ordinary function that has an exact match.
- ii. Calling a template function that could be created with an exact match.
- iii. Trying normal overloading resolution to ordinary functions and call the one that matches.

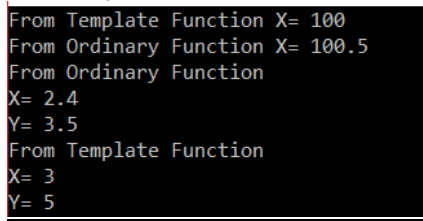
An error is generated if no match is found. Also no automatic conversion are applied to arguments on the template function.

Example:

```
#include<iostream>
using namespace std;
template<class t>
void display(t x)
{
    cout<<"From Template Function X= "<<x<<endl;
}

template<class t1, class t2>
void display(t1 x, t2 y)
{
    cout<<"From Template Function"<<endl<<"X= "<<x<<endl<<"Y= "<<y<<endl;
}
void display(float x)
{
    cout<<"From Ordinary Function X= "<<x<<endl;
}
void display(float x, float y)
{
    cout<<"From Ordinary Function"<<endl<<"X= "<<x<<endl<<"Y= "<<y<<endl;
}

int main()
{
    display(100);//will invoke template function with one argument as no exact matching ordinary function exist.
    display(100.5f);// will invoke ordinary function as exact matching ordinary function exist
    display(2.4f,3.5f);//will invoke ordinary function as exact matching ordinary function exist
    display(3,5);//will invoke template function with two argument because non ordinary function matches this signature
}
```



```
From Template Function X= 100
From Ordinary Function X= 100.5
From Ordinary Function
X= 2.4
Y= 3.5
From Template Function
X= 3
Y= 5
```

Example 2:

```
#include<iostream>
using namespace std;
void display(float x)
{
    cout<<"From Ordinary Function1 X= "<<x<<endl;
}

template<class t1, class t2>
void display (t1 x, t2 y)
{
    cout<<"From template Function X= "<<x<<" Y= "<<y<<endl;
}

int main()
{
    display(100); //will invoke ordinary function1 with one argument of type float(i.e. int casted to float)
    display("bhesh",5);
}
```

```
}
```

```
From Ordinary Function1 X= 100  
From template Function X= bhash Y= 5
```

Exception Handling in C++

Introduction to error

In computing, an error in a program is due to the code that does not conform to the order expected by the programming language. An error may produce an incorrect output or may terminate the execution of program abruptly or even may cause the system to crash.

Types of Errors

- i. Compile Time Error
- ii. Runtime Error

Compile Time Error

At compile time, when the code does not comply with the C++ syntactic and semantics rules, compile-time errors will occur. The goal of the compiler is to ensure the code is compliant with these rules. Any rule-violations detected at this stage are reported as compilation errors. These errors are checked.

The following are some common compile time errors:

- Writing any statement with incorrect syntax
- Using keyword as variable name
- Attempt to refer to a variable that is not in the scope of the current block
- A class tries to reference a private member of another class
- Trying to change the value of an already initialized constant (final member)
- etc.....

Runtime Error

When the code compiles without any error, there is still chance that the code will fail at run time. The errors only occur at run time are called run time errors. Run time errors are those that passed compiler's checking, but fail when the code gets executed. These errors are unchecked.

The following are some common runtime errors:

- Divide by zero exception
- Array index out of range exception
- StackOverflow exception
- Dereferencing of an invalid pointer
- etc.....

So, runtime errors are those which are generally can't be handled and usually refers catastrophic failure.

Exception

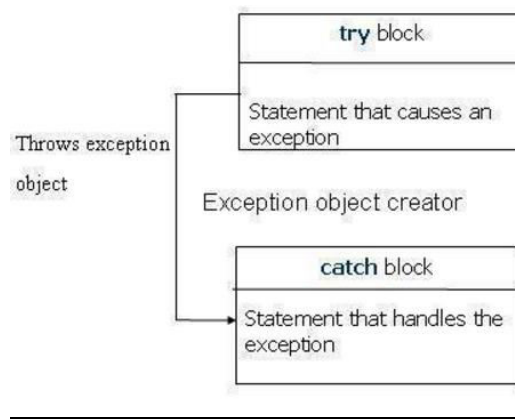
An exception is a run-time error. Proper handling of exceptions is an important programming issue. This is because exceptions can and do happen in practice and programs are generally expected to behave gracefully in face of such exceptions.

Unless an exception is properly handled, it is likely to result in abnormal program termination and potential loss of work. For example, an undetected division by zero or dereferencing of an invalid pointer will almost certainly terminate the program abruptly.

Types

1. **Synchronous:** Exception that are caused by events that can be control of program is called synchronous exception. Example, array index out of range exception.
2. **Asynchronous:** Exception that are caused by events beyond the control of program is called asynchronous exception. Example, Exception generated by hardware malfunction.

Basic steps in exception handling



The **purpose/Use** of exception handling mechanism is to provide means to detect and report an exception so that appropriate actions can be taken. Exception handling mechanism in C++ consists of four things:

- i. Detecting of a run-time error (Hit the exception)
- ii. Raising an exception in response to the error (Throw the exception)
- iii. Receive the exception information (Catch the exception)
- iv. Taking corrective action. (Handle the exception)

C++ provides a language facility for the uniform handling of exceptions. Under this scheme, a section of code whose execution may lead to run-time errors is labeled as a **try block**. Any fragment of code activated during the execution of a try block can raise an exception using a **throw clause**. All exceptions are typed (i.e., each exception is denoted by an object of a specific type). A try block is followed by one or **more catch clauses**. Each catch clause is responsible for the handling of exceptions of a particular type. When an exception is raised, its type is compared against the catch clauses following it. If a matching clause is found, then its handler is executed. Otherwise, the exception is propagated up, to an immediately enclosing try block (if any). The process is repeated until either the exception is handled by a matching catch clause or it is handled by a default handler.

The general form is as below:

```
.....
.....
try
{
    .....
    .....
    throw exception; //block of statements which detect and throws an exception
}
catch(type arg) //catches exception
{
    .....
    .... //block of statements that handle the exception
    .....
}
.....
.....
```

Example:

WAP to input two number and divide first number by second. The program must handle the divide by zero exception.

```

#include<iostream>
using namespace std;
int main()
{
    int nu, de, res;
    cout<<"Enter numerator and denominator";
    cin>>nu>>de;
    try
    {
        if(de==0)
        {
            throw(de); //throw int object
        }
        else
        {
            res=nu/de;
            cout<<"Result= "<<res;
        }
    }
    catch(inti)    //catches the int type exception
    {
        cout<<"Divide by zero exception occurred: de= "<<i;
    }
}

```

```

Enter numerator and denominator4
2
Result= 2

```

```

Enter numerator and denominator4
0
Divide by zero exception occurred: de= 0

```

So, in the above program, when no exception is thrown, the catch block is skipped and outputs correct result. But when the user input zero for denominator, the exception is thrown using throw statement with int type object as argument. Since the exception object type is int, the exception handler i.e. catch statement containing int type argument catches the exception and handles it by displaying necessary message or performing necessary steps further.

Note:

- i. During execution, when throw statement is encountered, then it immediately transfers the control suitable exception handler i.e. catch block. Hence all the statement after throw in try block are skipped.
- ii. If there are multiple catch block with integer type argument, then the first one immediately after try block gets executed.
- iii. When an exception object is thrown and non of the catch block matches, the program is aborted with the help of abort() function which is invoked automatically.

Invoking function that generates Exception

Exceptions can also be thrown from function that are invoked from within the try block. The point at which throw is executed is called **throw point**. Once an exception is thrown to the catch block, control can't return to the throw point.

The general form is:

```

typedef function_name(arg list) // Function with exception
{
    Throw (object); // throws an exception
}
.....
.....
try
{
    .....
    ..... // function is invoked from here
    .....
}
catch(type org) //catches exception

```

```

{
    .....
    ..... //handles the exception here
    .....
}

```

Example:

```

#include<iostream>
using namespace std;
int divide(int,int);
int main()
{
    int nu,de,res;
    cout<<"Enter numerator and denominator";
    cin>>nu>>de;
    try
    {
        res=divide(nu,de);
        cout<<"Result= "<<res;
    }

    catch(inti)    //catches the int type exception
    {
        cout<<"Divide by zero exception occurred: de= "<<i;
    }
}
int divide(intfn, intsn)
{
    if(sn==0)
    {
        throw(sn);
    }
    else
    {
        return(fn/sn);
    }
}

```

```

Enter numerator and denominator4
2
Result= 2

```

```

Enter numerator and denominator4
0
Divide by zero exception occurred: de= 0

```

Nesting try statement

A try block can be placed inside the block of another try, called as nested try.

Example:

WAP to input two numbers and divide first number by second. The result must be stored in the index entered by user.

```

#include<iostream>
using namespace std;
int divide(int,int);

```

```

int main()
{

```

```

inta,b,ind,res;
int c[10];
cout<<"Enter numerator and denominator"<<endl;
cin>>a>>b;
try
{
    if(b==0)
    {
        throw(b);
    }
    cout<<"Enter the index to store the result";
    cin>>ind;
    try
    {
        if(ind>=10)
        {
            throw(ind);
        }
        res=a/b;
        c[ind]=res;
        cout<<"Result "<<res<<" successfully stored at index "<<ind;
    }
    catch(int e)
    {
        cout<<"Index out of range exception occurred: index= "<<ind;
    }
}
catch(int e )
{
    cout<<"Divide by zero exception occured: b="<<b;
}
}

```

```

Enter numerator and denominator
4
2
Enter the index to store the result8
Result 2 successfully stored at index 8

```

```

Enter numerator and denominator
4
0
Divide by zero exception occured: b=0

```

```

Enter numerator and denominator
4
2
Enter the index to store the result88
Index out of range exceptoin occured: index= 88

```

Multiple catch statement

It is also possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try as shown below.

```

try
{
    catch(type1object);
    catch(type2object);
    .....
    catch(typeNobject);
}
catch(type1 arg)
{
}

```

```

catch(type2 arg)
{

}
catch(typeNarg)
{

}

```

Example:

```

#include<iostream>
using namespace std;
int divide(int,int);
int main()
{
    try
    {
        doublenu,de,res[10],c;
        intind;
        cout<<"Enter numerator and denominator"<<endl;
        cin>>nu>>de;
        if(de==0)
        {
            throw(de);
        }
        cout<<"Enter the index to store the result"<<endl;
        cin>>ind;
        if(ind>=10)
        {
            throw(ind);
        }
        c=nu/de;
        res[ind]=c;
        cout<<"Result= "<<c<<" Successfully stored at "<<ind<<" posiion";
    }
    catch(int a)
    {
        cout<<"Index out or range exception: Index= "<<a;
    }
    catch(double d)
    {
        cout<<"Divide by zero exception: Denominator= "<<d;
    }
}

```

```

Enter numerator and denominator
4
2
Enter the index to store the result
6
Result= 2 Successfully stored at 6 position

```

```

Enter numerator and denominator
4
0
Divide by zero exception: Denominator= 0
-----

```

```

Enter numerator and denominator
4
3
Enter the index to store the result
88
Index out or range exception: Index= 88

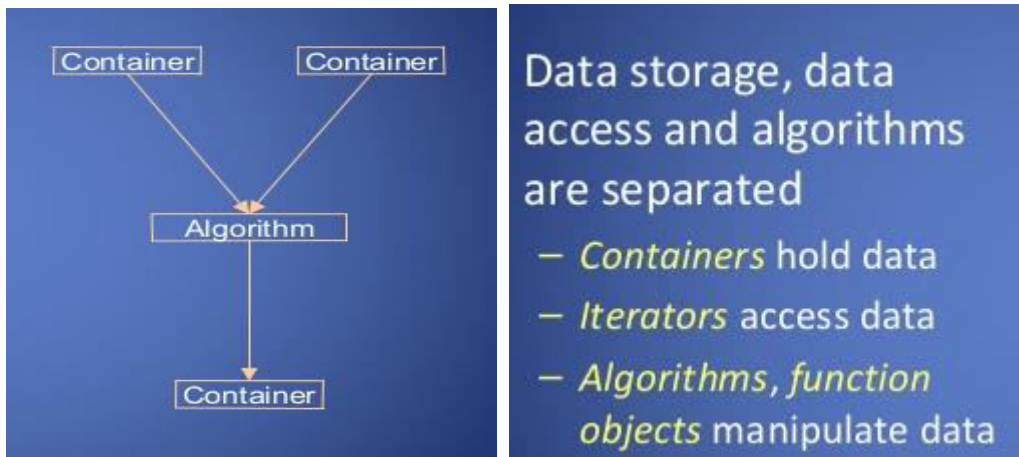
```

Standard Template Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized.

STL has three components

- Algorithms
- Containers
- Iterators



Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
 - Sorting
 - Searching
 - Important STL Algorithms
 - Useful Array algorithms
 - Partition Operations

Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
 - vector
 - list
 - deque
 - arrays
 - forward_list
- Container Adaptors : provide a different interface for sequential containers.
 - queue
 - priority_queue
 - stack
- Associative Containers : implement sorted data structures that can be quickly searched
 - set
 - multiset
 - map
 - multimap

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

Chapter 7

Object Oriented Design

Reusability Implies Non-interference

- Conventional programming proceeds largely by doing something to something else, modifying a record or updating an array, for example. Thus, one portion of code in a software system is often intimately tied, by control and data connections, to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values, or simply through inappropriate use of and dependence on implementation details of other portions of code. A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible.
- One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behaviour to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express.

Programming in small

Programming in the small characterizes projects with the following attributes:

- Code is developed by a single programmer, or perhaps by a very small collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.
- The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.

Programming in large

Programming in the large characterizes software projects with features such as the following:

- The software system is developed by a large team, often consisting of people with many different skills. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.
- With programming in the large, coding managers place emphasis on partitioning work into modules with precisely-specified interactions. This requires careful planning and careful documentation.
- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

Role of behaviors in OOP

- Earlier software development methodologies looked at the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application. But structural elements of the application can be identified only after a considerable amount of problem analysis.
- A formal specification often ended up as a document understood by neither programmer nor client. But behaviour is something that can be described almost from the moment an idea is

conceived, and (often unlike a formal specification) can be described in terms meaningful to both the programmers and the client.

Responsibility Driven Design

Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behaviors at all levels of development.

Responsibility-Driven-Design is a way of designing complex software systems using objects and component technology. Responsibility-Driven Design was conceived in 1990 as a shift from thinking about objects as data + algorithms, to thinking about objects as roles + responsibilities. Responsibility-Driven Design catalyzes object technology with practical techniques and thinking tools.

A case study in RDD

- The Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that will replace the index-card system of recipes found in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.

CRC cards (class responsibility collaborator)

- As the design team walks through the various scenarios they have created, they identify the components that will be performing certain tasks. Every activity that must take place is identified and assigned to some component as a responsibility.

i) Give Components a Physical Representation:

While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the “surrogate” for the software during the scenario simulation.

The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling

ii) The What/Who Cycle

Design often this proceeds as a cycle of what/who questions. First, the design team identifies what activity needs to be performed next. This is immediately followed by answering the question of who performs the action. In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component

The secret to good object-oriented design is to first establish an agent for each action.

iii) Documentation

Two documents should be essential parts of any software system: the user manual and the system design documentation. Work on both of these can commence even before the first line of code has been written. CRC cards are one aspect of the design documentation, but many other important decisions are not rejected in them. Arguments for and against any

major design alternatives should be recorded, as well as factors that influenced the final decisions.

Classname	
Responsibilities	Collaborators

Fig: CRC card format

Greeter	
Display informative initial message Offer user choice of options Pass control to either <ul style="list-style-type: none"> • Recipe database manager • Plan manager for processing 	Database manager Recipe manager

Fig: Greeter class in IIKH scenario

ATM CRC card example

Card reader	
Tell ATM when card is inserted Read information from card Eject card Retain card	ATM Card

Cash dispenser	
<ul style="list-style-type: none"> ✓ Keep track of cash on hand, starting with initial amount ✓ Report whether enough cash is available ✓ Dispense cash 	Log

Advantages

- Uses index cards.
- Cheap, readily available, amenable to group use.

- Forces you to be concise and clear.
- Puts your attention on the what, not how.
- Simple, easy methodology
- Intuitive
- Portable

Sequence Diagram

It is used to show dynamic behavior of the components.

- In the diagram, time moves forward from the top to the bottom. Each component is represented by a labelled vertical line. A component sending a message to another component is represented by a horizontal arrow from one line to another. Similarly, a component returning control and perhaps a result value back to the caller is represented. The commentary on the right side of the figure explains more fully the interaction taking place.

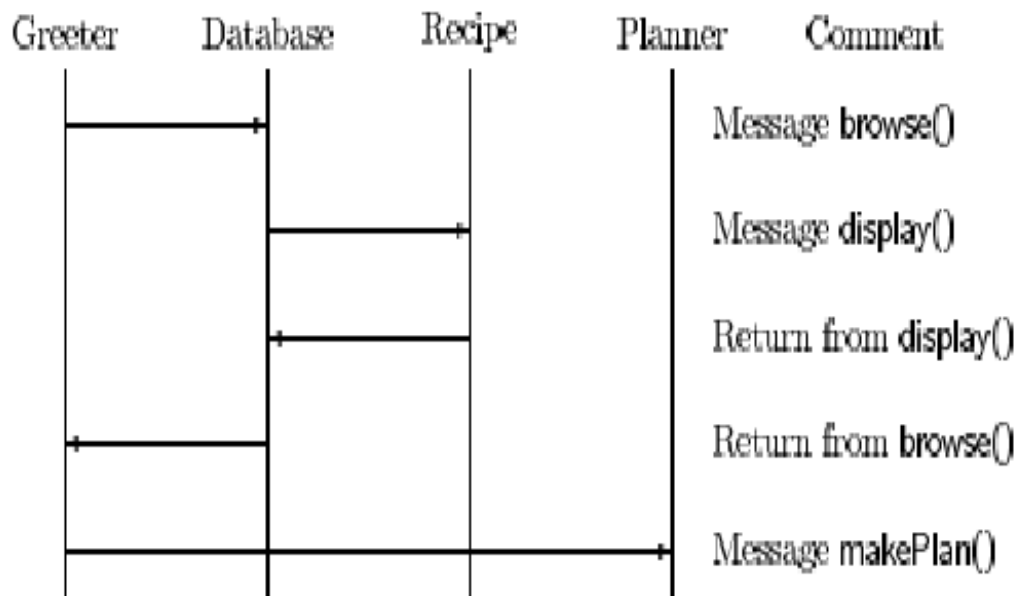


Fig: a sequence diagram for IIKH

Software components

- In programming and engineering disciplines, a component is an identifiable part of a larger program or construction. Usually, a component provides a particular function or group of related functions. In programming design, a *system* is divided into *components* that in turn are made up of *modules*. *Component test* means testing all related modules that form a component as a group to make sure they work together.

- In object-oriented programming , a component is a reusable program building block that can be combined with other components to form an application. Examples of a component include: a single button in a graphical user interface, a small interest calculator, an interface to a database manager.

i) Behavior and State: One way to view a component is as a pair consisting of behaviour and state:

- The **behavior** of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the protocol. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.
- The **state** of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

ii) Coupling and Cohesion

- Two important concepts in the design of software components are coupling and cohesion. Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.
- Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse

iii) Interface and Implementation

It is very important to know the difference between interface and implementation. For example, when a driver drives the car, he uses the steering to turn the car. The purpose of the steering is known very well to the driver, but the driver need not to know the internal mechanisms of different joints and links of various components connected to the steering.

An interface is the user's view of what can be done with an entity. It tells the user what can be performed. Implementation takes care of the internal operations of an interface that need not be known to

the user. The implementation concentrates on how an entity works internally. Their comparison is shown in Table

Comparison of interface and implementation

Interface	Implementation
<ul style="list-style-type: none"> • It is user's viewpoint. (What part) • It is used to interact with the outside world. • User is permitted to access the interfaces only. • It encapsulates the knowledge about the object. 	<ul style="list-style-type: none"> • It is developer's viewpoint. (How part) • It describes how the delegated responsibility is carried out. • It describes how the delegated responsibility is carried out. • It provides the restriction of access to data by the user.

Formalizing the Interface

- The first step in this process is to formalize the patterns and channels of communication.
- A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto method names. Along with the names, the types of any arguments to be passed to the function are identified.
- Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument or global value, or maintained internally by the component, must be clearly identified.

Coming up with names

- Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem.

The following general guidelines have been suggested:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as “CardReader” or “Card reader”, rather than the less readable “cardreader”.
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a “TermProcess” a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations.

Design and representation of components

- The task now is to transform the description of a component into a software system implementation major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfil the assigned responsibilities
- It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they have been chosen, the code used by a component in the fulfilment of a responsibility is often almost self-

evident. But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

- It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

Implementation of components

- The next step is to implement each component's desired behavior. If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language.
- An important part of analysis and coding at this point is characterizing and documenting the necessary **preconditions** a software component requires to complete a task, and verifying that the software component will perform correctly when presented with legal input values.

Integration of components

- Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using stubs: simple dummy routines with no behavior or with very limited behavior: for the as yet unimplemented parts.
- Testing of an individual component is often referred to as **unit testing**.
- **Integration testing** can be performed until it appears that the system is working as desired.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as **regression** testing.
- Give example of car making with different components bumper, gear, engine etc.