A Survey of Programming Techniques

Yagya Rimal
Pokhara University (PU)

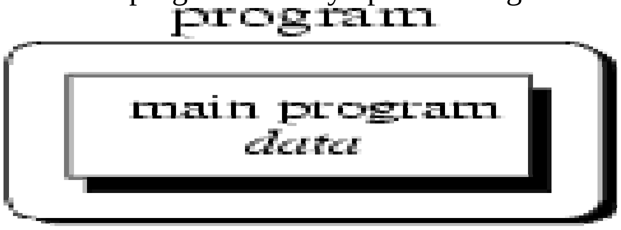
This chapter is a short survey of programming techniques. We use a simple example to illustrate the particular properties and to point out their main ideas and problems.

Roughly speaking, we can distinguish the following learning curve of someone who learns to program:

- Unstructured programming,
- Procedural programming,
- Modular programming and
- Object-oriented programming.

1.1 Unstructured Programming

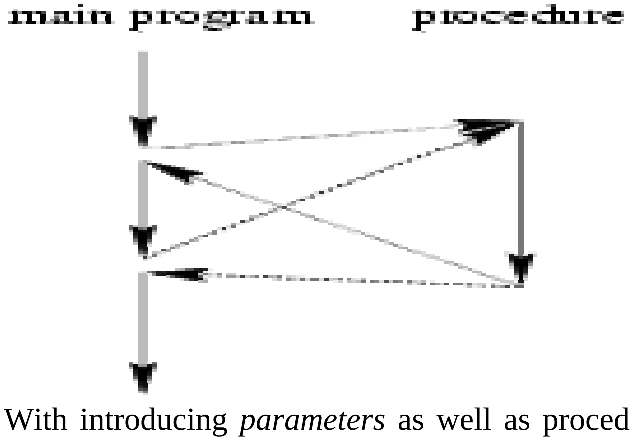
Usually, people start learning programming by writing small and simple programs consisting only of one main program. Here “main program" stands for a list of commands or *statements* which modify data throughout the whole program. We can illustrate this as shown in **Figure 1.1:** Unstructured programming. The main program directly operates on global data.



As know, these programming techniques provide tremendous disadvantages once the program gets sufficiently large. For example, if the same statement is needed at different locations within the program, the statement must be copied. This has lead to the idea to *get* these lists, name them and offering a technique to call and return from these *procedures*.

1.2 Procedural Programming

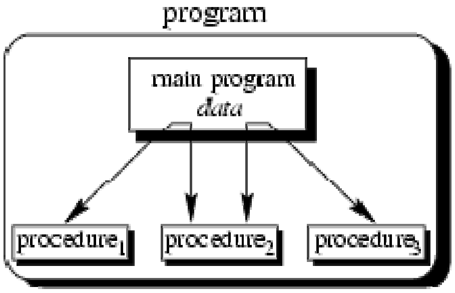
With procedural programming we are able to combine returning statements into one single place. A *procedure call* is used to invoke the procedure. After the statement is processed, flow of control proceeds right after the position where the call was made **Figure 1.2:** After processing flow of controls proceed where the call was made.



With introducing *parameters* as well as procedures of procedures (*sub procedures*) programs can now be written more structured and error free. For example, if a procedure is correct, every time it is used it produces correct results. Consequently, in cases of errors you can narrow your search to those places which are not proven to be correct.

Now a program can be viewed as a list of procedure calls. The **main program** is responsible to pass data to the individual calls, the data is processed by the procedures and, once the program has finished, the resulting data is presented. Thus, the *flow of data* can be illustrated **as a hierarchical** graph, a *tree*, as shown in **Figure 1.2:** Procedural programming.

The main program coordinates calls to procedures and hands over appropriate data as parameters.



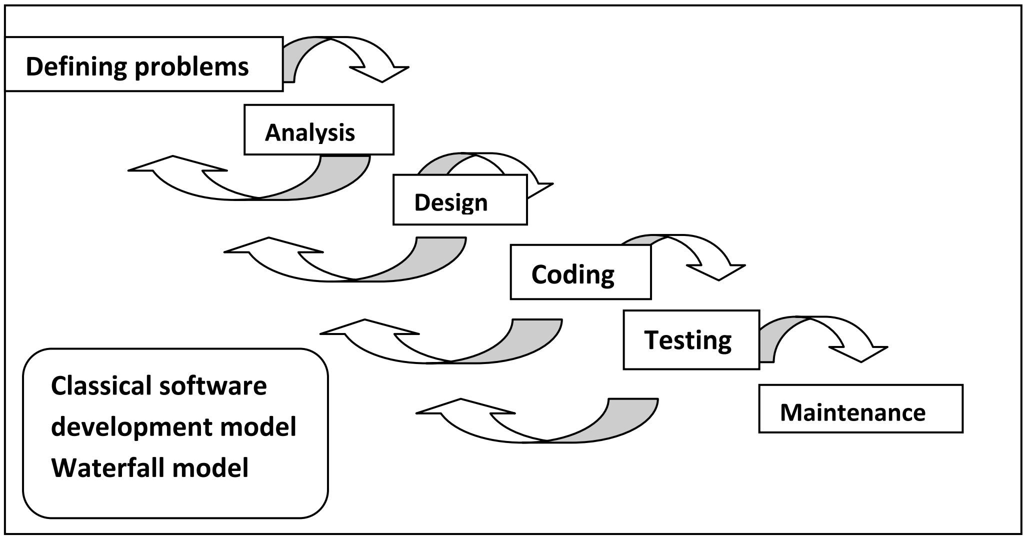
To sum up: Now we have a single program which is divided into small pieces called procedures. To enable usage of general procedures or groups of procedures also in other programs, they must be separately available. For that reason, modular programming allows grouping of procedures into modules. **Therefore, a program in a procedural language is a list of instructions. When the programmer crates the list of instructions that computer carries them out. When the program become large enough those instructions become difficult to understand hence function and subroutine need to overcome those difficulties so the program must to divide into more functions this process is called developing structural programming to solve the definite problem. The procedural oriented program has given more emphasis on function and data are given secondary priority. The procedural programming may only access some function some data within those function. The arrangement of data cannot change by function call.**

Some characteristics of procedural programming language

- Emphasis is one doing things (algorithm).
- Large programs are divided into smaller program known as function.
- Most of the function share global variable.
- One function transfer data to another function.
- Functions are designed in top down approaches in program.
- Examples: C, FORTRAN Cobalt etc.

Procedure Oriented Paradigm:

Software development is usually characterized by series of stages depends on the various tasks involved in the development process. The classical software lifecycle that is most widely used for the procedure oriented development. The classical life cycle is based on an underlying model, commonly referred to as the waterfall model. This model attempts to break up the identifiable activities into series of actions, each of which must be completed before the next begins.



Problem definition: *This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only development but also the user to understand the problem better.*

Analysis: *This covers the details of study of the requirements of both the user and software. This activity is basically concerned with what of the system such as.*
What are the inputs to the system?
What are processes required?
What are the outputs expected?
What are the constraints?

Design: *The design phase deals with various concepts of system design such as data structure, system architecture, and algorithms. This phase translate the requirements into a representation of the software.*

Coding: *Coding refers to the translation of the design into machine readable form. The more detailed the design, the easier is the coding and better its reliability.*

Testing: *Once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole system. It requires a detailed plans as to what, when and how to test.*

Maintenance: *After the software has been installed, it may undergo some changes. This may occur due to change in users' requirement, a change in the operating system the software that has not been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.*

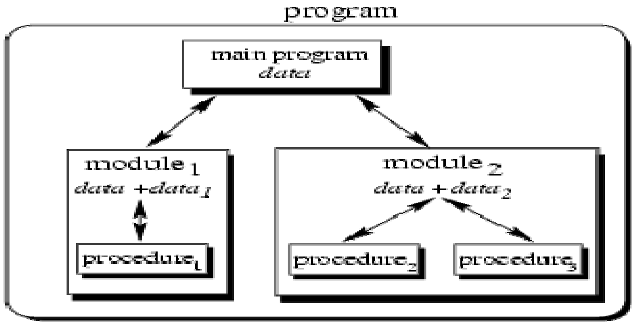
Each phase of the life cycle has its own goals and outputs. The output of one phase acts as an input to the next phase.

The software development technique in procedural oriented programming use functional de-compositation techniques is known as top down model. The functions are decomposed into the solution space as an interdependent set of function. The functions are decomposed into sequences of progressively simpler functions. The final system is seen as a set of functions that are organized in a top down hierarchal structure.

There are several drawbacks in top down decomposition approach:
It does not allow evolutionary changes in the software.
The system is characterized by single function at the top which is not always true. In fact many systems have no top.
Data is not given important that it describes.
It does not encourage reusability of the code.

1.3 Modular Programming

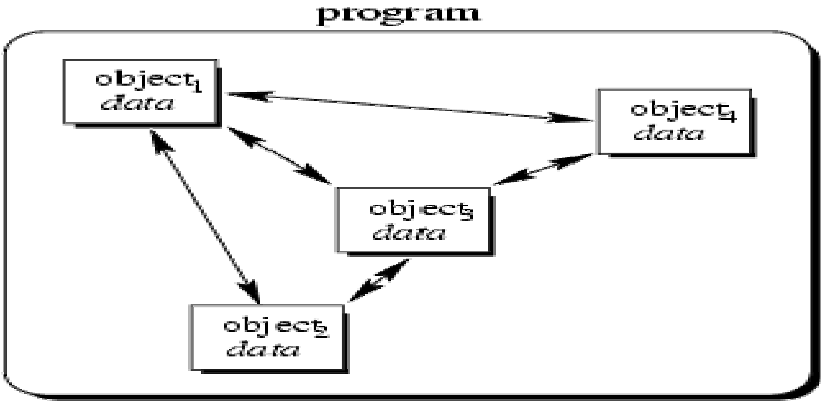
With modular programming procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program **Figure 1.3:** The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.



Each module can have its own data. This allows each module to manage an internal *state* which is modified by procedure call module. However, there is only one main module and each module exists within it.

1.4 Object-Oriented Programming

Object-oriented programming solves some of the problems. In contrast to the other techniques, it supports a network of interacting *objects its methods and functions* **Figure 1.4:** Objects of the program interact by sending messages to each other.



Consider the multiple lists example above. The problem here with modular programming is that you must explicitly create and destroy your list. Then you use the procedures of the module to modify each steps/objects.

In contrast to that, object-oriented programming would have as many objects. Instead of calling a procedure which must provide the correct method calling, we could directly send a message to the list of objects. Roughly speaking, each object implements its own module allowing for many lists to coexist. Consequently, there is no longer the need to explicitly call a creation or termination procedure. The fundamental idea behind OOPs is to combine both data and function are into single unit. Such data and functions are called object. OOPs is the software development technique that has being gained popularity and productivity over traditional methodology.

Some of the striking features of object-oriented programming are:

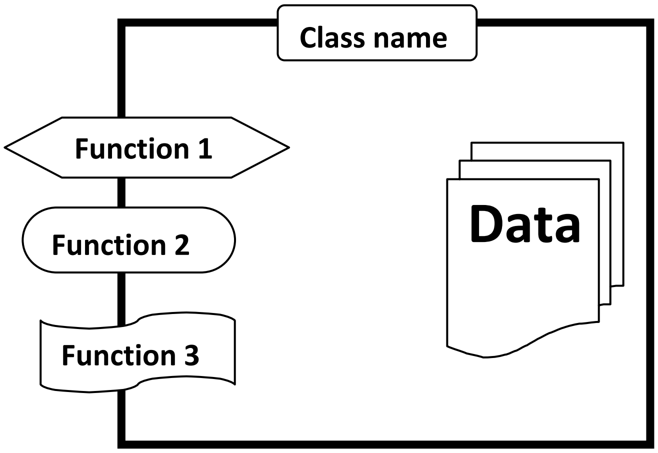
- Emphasis is on data rather than procedure.
- Programs are divided into what are known as object.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in data structure.
- Data is hidden and cannot be access by external function.(higher security)
- Objects may communicate with each other through functions
- New data and functions can be easily added whenever necessary
- Follows button-up approach in program development.

Object Oriented Paradigm:

An object oriented paradigm draw general conceptual systems theory to address of real world activities. A system can be viewed as a collection of entities that interacts together to accomplish certain objectives. Entities may represent physical objects such as equipment, furniture and people. In object oriented analysis, the entities are called the objects. As the name indicates, the object-oriented paradigm gives greater emphasis on the objects that encapsulate data and procedures. They play the central role in all the stages of the software development. Therefore OOP has given high degree of object overlap between the stages. An object oriented version of software development life cycle must therefore take into account there two aspects. Object oriented analysis refers to the method of specifying requirements of the software in terms of real world objects, their behaviors, and interactions. Object oriented design on the other hand finds out the software requirement into specifications for objects and classes hierarchies form which objects can be created. All the phase in aspect oriented model works together closely because of the object model. In one phase, the problem domain objects are identified while in the next phase. The design phase required for a particular solution is specified according to the object behaviors and characterizes. The object oriented approach should have to give emphasis to the top down a button down approaches. The top down approaches decomposition techniques can be applied to the design of individual classes while the final system can be constructed with the help of class modules using the bottom up approach.

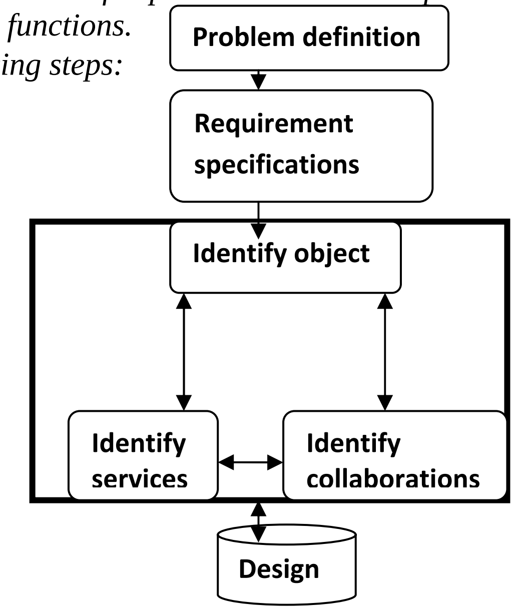
- ***Object oriented notion:*** *There are essential parts of any design and development process is called notion. Object oriented design concept gives more emphasis to classes, objects, functions, subclasses and their interrelationship. There are some special properties of OOP:*

- ***Classes and objects***
- ***Instance of objects***
- ***Message passing and communication***
- ***Inheritance relationship***
- ***Composition relationship***
- ***Hierarchical chart***
- ***Client server relationship***



Steps in object oriented analysis

- *The analysis is basically concerned with the decomposition of a problem into its components parts and establishing a logical model to describe the system functions.*
- *Object oriented analysis approach consist of the following steps:*
- *Understanding the problem*
- *Draw the specifications of requirements of the user*
- *Identifying the objects and their attributes*
- *Identifying the services*
- *Collaboration between objects in terms of services*



Basic Concepts of Object Oriented Programming /Characterizes of Object Oriented Programming

Object oriented programming is a term which is interpreted differently by different people. It is therefore necessary to understand some of the concepts used extensively in OOPs.

Object: Objects are basic primary entities they are represent as a person, place, bank, table or any items that the program must communicate with. The problems are analyzed in terms of object its natures of communication so that they match closely with real world objects. When a program is executed, the objects interact by sending messages to one another. An object is a unit /module/code where all data and procedures related to that object are defined with in it. All functions are tied within data. Data are given primary priority.

Advantages: Increase productivity, decrease maintenance cost, code reusability.

Classes: Objects are the member of class which describes data and function. Suppose Animal is a specific class and cat is one object of animal class. The basic features of animal has 2 eyes, 4 legs, head are common to all animals. Thus cat is the object of class animal, cat inherits those properties and cat may also have some distinguished/specific/features than other animals. Once a class has been defined, we can create any number of objects belonging to that class. Similarly account is the class within it credit, debit, withdraw may be the objects of that account class.

Inheritance: The classes are divided into sub class. Animal class can be divided into sub class like water and land animal. Land animal can inherit some or all properties of higher class animal and can put more characters of its own class. The class that represent whole is called base class or super class. The lower then the base class is called child or derived class. The next derived class may be the base of another derived class or subsequent class. Thus inheritance is the process by which objects of one class acquire the properties of another class. It supports the concept of hierarchal classification.

Reusability: The concept of inheritance provides the great idea for reusability. Once a class has been written and debugged, it can be distributed and used to other programmer. The new class will inherit the capabilities of the old class but free to add new features of its own class.

Polymorphism: Polymorphism means the ability to take more than one form. The operators may behave different behavior in different instances. The behavior depends upon data types used in operation. The + operator is used for to add two integers and we can add two string too. This means the operation may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Data Abstraction and Encapsulation:

Data encapsulation is most striking features of object oriented programming that made up of data and function into a single unit. The data can not access outside where they are wrapped. This insulation of the data form direct access by the program is called data hiding. Data abstraction is the process of obtaining all details information of particular unit/module. The class uses the concept of data abstraction, they are known as Abstract Data Types (ADT).

Dynamic binding: Binding refers to the linking of procedure call to the code to be executed in response to the function call. Dynamic binding means that code associated with a given procedure call is not known until the time of procedure call.

Message communication: We know that an object oriented programming consists of a set of objects that communicate with each other. The process of programming in an object oriented programming therefore involves the following basic steps:

Creating classes that define objects and behaviors.

Crating objects form class definitions.

Establishing communication among objects.

A message for an object is request for execution of procedure, and therefore the requested action invokes a function (procedure) in the receiving object that generates the desired result.

Object Based Programming: *The programming language that support objects but they don't support inheritance and dynamic binding of data.*

Object Based features=Inheritance + Dynamic binding

Benefits of OOPs

Through inheritance, we can eliminate redundant code

We can build program modules that can used in later time.

Data hiding helps the programmer to build secure program

It is possible to have a multiple instance of an object

It is essay to map out various objects to solve the real world problem

The program can be easily upgraded form small to large

Software complexity can be easily managed

The memory management under **C++** would be easier and more transparent.

Programs would be less bug-prone/error free.

Application of OOPs:

- User interface design such as window
- Real business applications because it supports inheritance and polymorphism
- Real time system
- Object oriented data based
- Neutral network and parallel programming
- Decision support and office automation system. CAD System

Thus object oriented technology is certainly going to change the way software engineers, analyze and design and implementation.

Exercises

1. What are the main conceptual differences between object-oriented programming and the other programming techniques?
2. What is object oriented programming? How it is different form the procedure-oriented programming?
3. Distinguish between the following terms
a) Objects and classes
b) Inheritance and polymorphism
c) Dynamic binding and message passing

References:

E BALAGURSAMY, 1999 Object oriented programming with C++, Delhi, Tata Mc Geaw Hill

Timothy Budd, 2009: An Introduction to Object Oriented Programming, Delhi, Anand Sons

Bjerne Stroustrop, 2007: The C++ Programming Language, Third edition, AT&T Lab India, Anubha Printers

Walter Savitch, 2001: Problem Solving with C++ the object of programming, second edition, India Saurbh, India

Fred Richards, C/C++ Programming Style, Guidelines Internet

EBook Download any books from internet on PDF format

Write a program to print out your First name and roll no

```
#include<iostream.h>
#include<conio.h>
class info{ // class declaration
int a;    // member data(properties)
char s1[34];
public:   // public access specifies
void getdata(){// function having void return type
cout<<"Enter your First name";
cin>>s1;
cout<<"Enter your rollno";
cin>>a; }            // function close
void display(){// another function void return type
cout<<"Your name is:::"<<s1<<endl;
 cout<< "Your Rollno is:::"<<a;
 } };// function close and class close
 void main(){// main method
 info p; // object creation
p.getdata();/object. function name
p.display();
 getch(); }//end main method
```

What is C++

C++ is a superset of c, which support OOPs. It was developed by Bjarne Strostrup in 1986 at AT&T Bell Labs. The name is a combination of C and double increment operator indicate more development of C language. Version 1.2 was initially released then 2.1, 4.1 etc versions were released. Multiple inheritances, abstract class pointers, overloading are the key important features of C++.

Starting C++ program

Open TC in c: or D: drive then go to bin directory then open TC file(or) Type C:\TC\BIN\TC.PIF on the Dos prompt. If you are running Borland C++ go to start then got accessories then select Borland C++ then select Borland C++ then go to file new.

Compiling and execution

Save your code with extension “filename.cpp” then compile it that formed filename.obj then linking process makes it filename.exe then run by pressing CTRL+F9 or go to run menu.

A Way of Viewing the World Agents

The real world problem can be easily solved with the help of object oriented programming in which the whole problem is divided into various objects and those objects are classified according to their nature of data type.

To illustrate the above situation, suppose I want to send some rose to my fiend who lives in very far (Germany). It is not possible to give the red rose to his hand on the occasion of his birth day. But this problem can be easily solved with the help of using many agents (ie objects). Then I should go to Florist center and give him some message of wishes (with specification of rose) then the florist in my town has responsibility to deliver my message to my friend in Germany. He might use some methods and algorithm to deliver that information without any changes. Therefore the first florist established the communication to the florist in the Germany. That florist must have to mange the arrangement of flowers in Germany. He must arrange other agents like wholesaler, retailers, delivery person, gardener, grower etc. If only he can manage all concern mediators in a channel, he sends the message to my friend on time. Therefore it is concluded that the big problem can be easily solved with the help of various agents in object oriented manner. An object oriented program is structured as a community of interacting agents called objects.

Another example:

Suppose I want send some flowers to my grand-mother, but she is not in this world right now.

- *I went to Florist Center and give specification of flowers, numbers and messages.*
- *Then florist has a responsibility to send those flowers according to my specification.*
- *The florist might have to consult many agents like wholesaler, arranger, and gardeners, deliver persons (these are objects again).*
- *Thus we can conclude that the problems in the real world can be solved by using many agent/intermediate.*

In conclusion

- Object oriented problem can be solved with the help of many other individuals/objects.
- With out others helps the problems could not be solved.
- Object oriented program is structured as a community of interacting agents called objects.
- Each object plays significant role and action to accomplish the problem.

Computation as simulation

The above problem of sending flowers to my friend can be easily solved with the help of traditional method pigeon- whole model, where both of us have to access on that **pigeon hole.** We can communicate whenever we need some activities to other. But, computer is an electronic device that processes the data/ instructions given to the system through memory address. But this model may be less accurate picture of what takes place inside computer, it is certainly not the way most people go about solving problems.

In contrast, the object oriented framework less mention memory address, variables, and assignments. Instead, we talk about object, messages and responsibility for some action to solve the real world problems

Simulation (The operation by which the real situation is represented)

The way of programming is to create the universe (**main method)** is the many ways similar to the computer simulation called **event driven**. We know computer takes data and instruction through keyboard and those data and instruction processed. By this manner we can design the methods inside the class to fulfill the requirement. In this way the designer has full rights to design the various elements/ objects, describes and designs the various items in the program and finally established the relationship to one another. Thus object oriented programming style is similar to computation and simulation.

There are some procedures

- Analysis the real world problems with various dimension/elements.
- Analysis the problem with its related objects
- Analysis the objects according to their characteristics
- Develop the relational model
- Developed main method
- Develop the algorithm
- Design coding
- Develop project/software.

Coping with complexity

The needs and requirements human beings are never ends that’s makes world better and easier than the past. Thus the system designers have given more priority to satisfied many human requirements/ problems. The today's real situation might not do exactly apply in the future demand therefore system designers have only rights to change those previous structure without disturbing the last code. In this situation the big problem must have to design into smaller and smaller units according to their behaviors. The strategies used by decision makers to manage uncertainty in complex system were solved by **J. von Neumann.** His contribution helps to solve various algorithms and methods. This situation needs the techniques to organize in a specific manner to control the overall problems of human beings. When we decomposed the main task into various units the uncertainty arises at all levels. That leads nonlinear dynamic patterns among newly formed units. The software designers have to take various decisions while designing the programs to meet real human requirement correctly.

Decision-support techniques are almost universally designed to assist decision makers to maximize their decision and minimize errors. Decision-support techniques are used when the complexity and uncertainty arises in the situation so that software become complete, error-prone, and changing over time. Additionally, decision makers are likely to pay for certainty; they do so by selecting decision choices, less risk, to overcome uncertainty in each level.

Uncertainty in decision making takes several forms:

(a) The decision makers **are compelled to guess which of the outcomes will actually take place.**

(b) The decision **but does not know the problems associated with each outcome.**

(c) The decision maker has an **incomplete knowledge of the possible outcomes**.

(d) The decision maker has **incorrectly identified the problem** options thus become meaningless if they were known.

The complexities of situations obstruct decision makers to apply good decision at various levels in programming. Decision may take in a simple manner or a linear regression equation, or a complex computer program such as some of the expert systems. There is the linkage between complexity of the problems and the solution of the control of uncertainty.

When large computational problems divides into single processor computer architectures, users often must consider the use of multiprocessor systems. This decision is often made with multiprocessors have a well-managed to overcome. For large codes, the cost of parallel software development can easily manage the hardware on which the code is intended to run. Programmers often have sense of the complexity of a language. However, because programmers typically have experience with only a subset of the parallel languages and are subject to personal bias, a comprehensive comparison cannot be based on an opinion poll. The relationship between complexity and performance is also discussed in the context of message-passing.

For Reference Reading only

Our purpose here is to develop theoretical prospective in linear programming, starting from the various viewpoint of computational comp1exity. The standard form of the linear programming problem is to maximize a linear function main tool for solving the linear programming problem in practice is the class of simplex algorithms proposed and developed by Danzig. However, applications of nonlinear programming methods, inspired by Karmarkar's work may also become practical tools for certain classes of linear programming problems. Complexity-based questions about linear programming and related parameters have been raised since the 1950s, before the field of computational complexity started to develop. The practical performance of the simplex algorithms has always seemed good. In particular, the number of iterations seemed polynomial and even linear in the dimensions of problems being solved. Exponential examples were constructed only in the early 1970s, starting with the work of Klee and Minty. The field of computational complexity developed rapidly during the 1970s. The question of the complexity of linear programming was formalized in a new and more precise sense. The result was widely because of popular the simplex algorithm. While a multi-dimensional algorithm for the same problem was very inefficient. It was only natural that interest in the field started to increase in two directions: (i) analyzing the behavior of the simplex method from a different viewpoint, and (ii) searching for other methods.

Khachiyan's result relies on the so-called logarithmic-cost model. It is still an open question whether a system of linear inequalities can be solved in a number of arithmetic operations that is polynomials bounded by the dimensions of the system, independently of the magnitudes of the coefficients. In such an algorithm is *given for systems with at most two variables per inequality (whereas the general case can be reduced to at most three variables per inequality). The general linear programming algorithm whose number of elementary operations is independent of the magnitudes of coefficients in the objective-function and the right-hand-side vectors, but depends on the coefficients in the matrix A.*

Theoretical research on algorithms in recent years focused on the direction of estimating the asymptotic worst-case complexity of problems in. For instance, knowing that a certain problem can be solved in polynomial time, it is of interest to find exact (asymptotic) upper and lower bounds on the time it should take any algorithm to solve the problem in the worst case. There has been much research done in this direction in the related field of computational geometry. It improves the upper bound on the complexity of linear programming, again under the logarithmic-cost model.

Abstraction Mechanism

If we open an atlas, we will often 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 on the earth. If you see the map of a continent we may see more detail of country political boundaries, may be city and rivers etc. Similarly if we see the map of a country, it might include main city, rivers, village, town, and roads etc. If we see the map of a town we find more detail information of road, land, cluster, settlement, road and many more.

In this manner, we find in all level certain information has been included and certain information has been omitted. So there is no way to represent all the details at higher abstraction.

The abstraction is the process of getting detail information according to the level of deep sight to the problem. If you want to get more detail information about the topic, you should go deeper to the problem.

Layers of abstraction

The typical program written in object oriented style give significant features level of abstraction. The higher levels of abstraction provide the detail parts of the program. This can be analyzed by searching more related object its nature of interaction to one another. The highest level of abstraction can be view as a community of associate objects. If we need more detail information about the problem, we should give detail study of interaction between objects. This process requires the two main agents. First, the **designers** who design the object into code should have to communicate according to objects behavior. Secondly, the **community of agents** they interact with each other. Similarly the level of abstraction is not found all objects oriented program not all objects used to solve problems. The third level of interaction may be analyzed **between the two individual objects**. This type of interaction can be view as client and server. First, the client give request to the server then the server should have to satisfy to client’s request. Thus it is concluded that the more the level you analyzed the more the detail information you can get.

Division into parts

The most common techniques use to solve the complex problem with a division into components parts and those components are analyzed individually as a single unit.

Encapsulation and interchangeability

Encapsulation is the process of combining all related items into one single unit so that the real problems can be easily solved. The most benefit of encapsulation is that it permits us to consider the possible interchangeability.

Interface and implementation

An interface describes what a system is designed to do. The interface says nothing about how the designed task is being performed. So implementation is the way of transfer your logic to the system.

The service view

The service view and relationship between objects give the clear picture to design the task. Each community member/ agent provides a service to solve the problem.

Composition

Composition is another techniques used to create complex structure into single parts.

Pattern

Pattern is the new approach to solve problem with the help of previous solution/software. The new requirements are analyzed and add to solve present demand.

Object Oriented Design: Responsibility Implies Noninterference

We know that object oriented program supports inheritance, method passing, creation of many agents and classes etc. The modularization is the process of decomposing the main task into various modules through which we can solve the whole problem easily. In such situation we just transfer some task to the object so that those objects have responsibility to satisfy its objective. Similarly in our flower example when I give the request of sending flower to my friend. The florist becomes responsible person to deliver the flowers. I don't care about what method and algorithm he will follow. Thus one person of code in a software system is tied and controlled to many other sections. Thus responsibility driven design attempts to cut links among all but make few responsible agents only.

Programming in the small and in the large:

The great difference between small and large programming is its scale of creation and its application.

There are some characteristics of small program:

Mostly, the code was developed by single person so that single person can understand all aspects of the project. Small program has small coverage of social demand.

On the other hand large programs are:

The software system is developed by large team, with different person. The individual experts are involved in the specific design. The major disadvantage of large programming is the management of detail and communication among them.

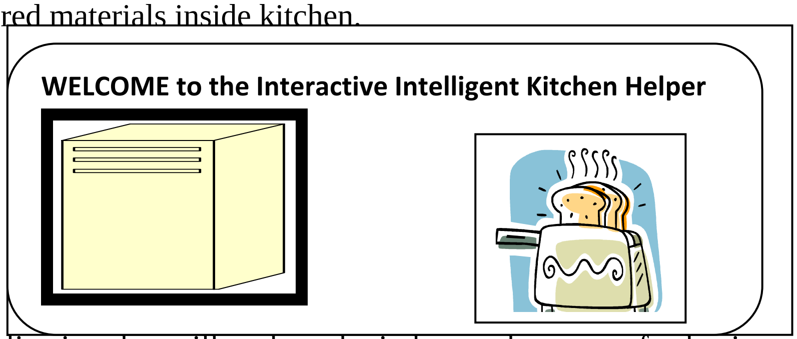
Role of behavior in OOPs

The traditional software development techniques are concern about data structure and procedure only so that software have limited used and application. That software could not satisfied human requirements. But the object oriented programming has given more concern to the object and its behavior. So the study of behavior its linkages and relationship among objects should have to analyzed before designing the software.

Responsibility Driven Design (RDD) *developed by Rebecca Wirts-Brock. RDD technique has given more emphasis on behavior at all level of software development.*

A Case Study in RDD

Imagine you are software manager in a company. One day your boss walks into your office with an idea that he want to develop kitchen helper ie. (**Interacting Intelligent Kitchen Helper**). So being the manager you should to design all the requirements software on Interactive Intelligent Kitchen Helper (IIKH) that helps to prepare foods and required materials inside kitchen.



IIKH is a PC based application that will replace the index card system of selecting menu items. The kitchen helper assists in preparing meals for an extended period – say a week. The requirement of software is that will works automatically, prepare the list of cooking utensils menu and recipe for entire period. The software development needs several programmers working together with him. This is complex situation it should be handled by following points.

To clarify ambiguities

The initial goal of the design team must be clarify the unclear statements. The overall outline algorithm of requirement should have to fulfill the demand of your boss.

Behaviors of items

The entire development team should have to design those items clearly in terms of behaviors and nature.

Database Management

A database management system is a collection of program that enables user to organized data according to their nature/behaviors. The database is general purpose software system that provides facilitates the process of categorizing; manage data from huge data items.

Abstraction: The higher level of abstraction need before coding start.

Graphic: Some designer may proposed that graphics tools with picture should have to display while selecting menu.

Automation:/dynamism Software automation is the process through which user can easily adds new features according the requirement increases (new add, edit, update, save).

Components design: The components are simply an abstract entity that can perform task and fulfills some responsibilities. But it is not compulsory represent for component how a component will perform a task. A component must have a small well defined set of responsibility. A component should interact with other components.

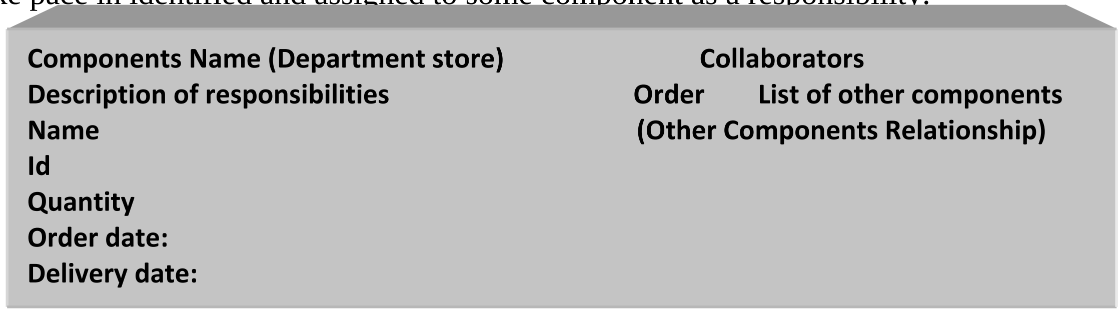
After analyzing all situations and the components, we can design the software that manages inside kitchen.

CRC Cards (Components, Responsibility Collaborator)

Components: *A component must have a small well defined set of responsibility.*

Collaborator: *Two classes that depend on each other for the execution of their behaviors are said to be collaborators.*

An index card is the documents that describe many related information of an individuals (name, address, id, organization name issue date etc.). Similarly CRC card is another type of card that describes name (components), responsibilities and collaborators of a particular items, is used during the process of system analysis and design. While designing the software, we should have to analyze various parts/ components to accomplish the certain task. The identified components that will be perform certain tasks. Every activity that must take pace in identified and assigned to some component as a responsibility.



Components Name (Department store) **Collaborators**
Description of responsibilities **Order** **List of other components**
Name **(Other Components Relationship)**
Id
Quantity
Order date:
Delivery date:

While writing on the face of card first name of software components is written at the top of card, the responsibility of component is written on the below of it, then the collaborator is written at the another side of it. Those components may have various relationships with other components. Such cards are known as CRC card.

The software development system needs the combination of various parts work together message passing, control flow and calling objects are regular phenomena among various components. An advantage of CRC card is that they are widely used and provides alternative design to the system designer. The physical separation of the cards encourages understanding the logical components from the various components.

Components and behavior

The team further decided to add some interactive process when the system begins, on the interactive window before released the software. The responsibility for displaying those components inside the main methods is called composition. The selective tools may used to communicate inner system. They again realized the following action to be put down on that program.

It can be browse the database without any particular meal plan.

Add a new item to the database.

Edit an existing menu.
Review an existing plan for several meals.
Create new plan of meals.
The final outlooks of the software become.

Welcome to Greeter	**Collaborators**
Display information	Database Manager
User choice	Plan Manager
Data Base Manager	
Plan Manager for processing	

Software components

A set of instructions written for computer to solve human requirements is known as software.

Software engineers have been trying various methods, tools and procedures to control the process of software development in order to build high quality software .they are used in all the stages of software development process, planning, analysis, design development and maintenance. There are various paradigms or set of methods and tools. The selection of particular paradigm depends on the nature of application programming language used and requirement. Good successful software must:

Satisfied the user requirements
Easy to understand by the user
Be easy to operate
Have good user interface
Be easy update
Have adequate security controls
Handle errors and exceptions

Behavior and state:

We know that object oriented programming has given more importance to the formation of components; these components certainly hold some sort of information. One way to view such a component is as a behavior and state. The behavior of a component is the set of actions it can do. The complete description of all the behavior for a component is sometimes called the protocol. Similarly, the state of a component represents all the information held within it at a given point of time.

Instances and classes:

The term class is used to describe a set of objects with similar behavior. An individual representation of a class is known as an instance. All instance of a class will respond to the same instructions and perform in a similar manner.

Coupling and cohesion:

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 task that is similar. In software design, the component has to achieve high cohesion.

Coupling describes the relationship between software components. In software development component the amount of coupling should reduced because if the relationship between component is more. There will be very difficult to edit or modify.

The coupling will increase if same data or same state is access by more components. To minimize coupling, the responsibility should have to divide into different various components. So that the components should not have same value and nature.

Interface and implementation:

In software design, large program is divided into small components and components are assigned to different programmer a component may be related with each other therefore the programmer have possibility to know how to use another component developed by other programmer, but, no need to know how the component is implemented.

Let’s have an e.g. of A/c management software of a college which is divided into no of components suppose there is fee data base components which keeps the record of all fee structure of all faculties and another component have civil engineering which manage all fees of civil engineering department. Here the programmer of civil eng. Component interface with fee data base but no need to know detail description of fee database components. The separation of interface & implementation is most important concept in software engineering. It also mentions the concept of information hiding in software design.

Interface and implementation leads to two different views of the system software. The interface view is the face by other programmer and it describes what a software component can perform. An implementation view is the face seen by the programmer working or particular component. It describes how components is completing task.

Another most important feature of object oriented programming is reuse of software component. To re-use the software component, there must be well inter-connection between various components of software.

Formalize the interface:

Formalize the interface is description of components in which the patterns and channels of communication is formalized. The decision should make a general structure so that it will be used to implement each other component. A component with only one behavior and no internal state is made into function. For e.g. a component that simply take string text & convert to uppercase but components with more task or more behavior is implemented as class. The components of software must be named & components have responsibilities that are mapped by procedure of function. Then types of arguments are passed to function. It component requires data then that data are passed through argument or global variables and data are maintained by component. The data must be identified clearly which is used in component.

Integration of components:

Once software subsystem have been individually designed and tested, they can be integrated into the final product. This is often not a single step but of large process. Starting from simple adding components slowly added to the system. During integration it is common to be an error in software system. Those errors and mistakes are solved gradually and formed large unit software.

Maintenance and evolution:

After completing the development of software system delivering the product in to the market, there may be some requirements will be added like changing the system, making update according the hardware etc. So software maintenance describes the activities that must be performed in future, to maintain the software according to user. The varieties of activities fall into software maintenance and evolution that are listed below:

- Errors or bug can be discovered in delivered product. These must be corrected and upgraded.
- Requirement may change according the change in rules and regulation of government.
- Hardware may change, the system may more to different platform or input deceives or any other change in devices etc.
- User demand may change. The user may demand greater functionality, lower cost easier use etc.
- Better documentation may be required by user.

Function in C++

We know that object oriented programming is based on the principle of dividing the problem into various functions. The typical way of getting something done in C++ program is to call a function to do it. Defining a function is the way you specify how an operation to be done. While designing the function, the top down approach is used. Another advantage of using the functions is that it is possible to reduce the size of a program and using them in different places in the main program. Function continue to building the blocks of C++program. In fact C++ has added many functions to make more reliable and flexible. Sometimes function may be overloaded to make different task depending on the arguments passed to it. Most of these modifications are aimed to meet object oriented requirements.

The main function

The main function is the starting point of the program execution. Each program requires the main function. The syntax of the main function is like.

```
 void main(){
…..;//statements
……; }
```

The functions that have to return value should use the return statement for termination. Since the return type of function is int by default. If there is not return type the compilers generate error messages. Many operating systems test return value to determine if there is any problem. The use of return (0) statement will indicate that the program was successfully executed.

```
int main(){
……; //statements;
…….;
return (0);  }
```

Function prototyping

Function prototyping is the rules to declare function. When a function is called the compiler uses the template to ensure that proper arguments are passed and the return value is treated correctly. Any violation in matching the arguments or return type will give the error message. The declaration syntax of function is: ***type function-name (arguments-list);***

Each argument variable must declare independently inside parentheses:

float doadd (int a, int b); // It says that doadd is the function that return floating values after getting a and b in integer type**.**

The function can be called in a program as: cube=doadd (3, 5); // here cube is new data type

We can declare function with empty parameters: ***void display (); // with out arguments***

Write an example to demonstrate pre-defied function

The function whose statement already defined in some header file is called pre-defined function

```
#include<iostream.h> //pre processor
#include<conio.h>
void square();// function declaration
void main() {  clrscr(); // main method
square(); //function calling in main method
getch(); }// close main method
void square(){// function design start here
int p;
cout<<"enter the values";
cin>>p;
int a=p*p;
cout<<"the square of "<<p <<"is"<<a;    }//function close
```

Write an example to demonstrate function with arguments

The function that does not return any value to other function is called function without return type.

```
#include<iostream.h> //pre processor
#include<conio.h>
void swap(int, int);// function with two arguments
void main(){// main method
int a,b; // data items
cout<<"Enter two number";
cin>>a>>b;
swap(a, b);// function calling inside main method
getch();     }//main method closed
void swap(int a, int b){// function design
int k;  k=a; a=b; b=k;
cout<<"the value of a a&b after exchange are "<<a<<"&"<<b; } //function end
```

Inline function

As we know function saves memory space because same code can call from different parts of program, the function need not duplicate the same task in memory again and again. When the compiler detects a function call, it normally generates a jump to the function. At the end of function it jumps back to the calling function. This sequence of events save memory space but it needs extra time. To save execution time in short function the code in function body directly inline with the code in calling program. When the function is called, the actual code forms execution. The inline function is only applicable for very short function. The function is made online by using keyword inline.

WAP using inline function to calculate average of two numbers.

```
#include<iostream.h> //preprocessor
#include<conio.h>
Inline float average(float a, float b){// inline function declaration
return (a+b)/2;  }//end function
void main(){// main method
float avg;
avg=average(10.5,20.6);// function calling and passing values to float type data
cout<<"The average is"<<avg;
getch();} //main method close
```

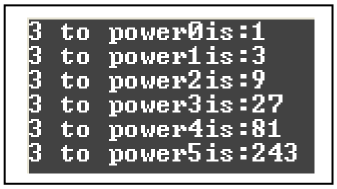
Recursive function:

We know various types of control structures may execute automatically till the user defined condition become true. The function is the block of statements that has some functionality which helps the programmer to solve the problem easily. In sometime a function definition may contain a call to itself such case the function is said to be recursive.

Recursion is a process by which a function calls itself repeatedly until some specified condition is true

WAP to demonstrate recursive function of calculating power function

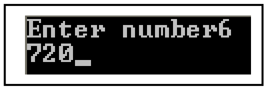
```
#include<iostream.h>
#include<conio.h>
int power(int x, int n);// function declaration
void  main() {//main
clrscr();
for(int n=0;n<=5;n++)//for
cout<<"3 to power"<<n<<"is:"<<power(3,n)<<endl;//function power
getch(); }//main closed
int power(int x, int n){//function power of int type
if(n<0) cout<<"Impossible";
if(n>0)
return (power(x, n-1)*x);
return (1); }
```



```
3 to power0is:1
3 to power1is:3
3 to power2is:9
3 to power3is:27
3 to power4is:81
3 to power5is:243
```

WAP program to demonstrate factorial of user given number by suing recursive function

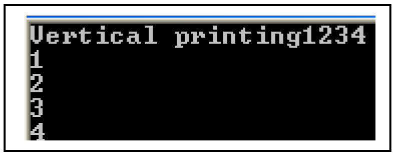
```
#include<iostream.h>
#include<conio.h>
int fact(int a);//function declaration
void main(){
clrscr(); int k;
cout<<"Enter number";
cin>>k;
cout<<fact(k);//function in main method
getch(); }
int fact(int a){// function design
if(a<=1) return 1;
else return (a*fact(a-1)); }
```



```
Enter number6
720_
```

Write a program to print out horizontal input of any number but gives vertical output by using recursive function

```
#include<iostream.h>
#include<conio.h>
void vertical(int n);
void main(){
int p;
cout<<"Vertical printing";
cin>>p;
vertical(p);
```



```
Vertical printing1234
1
2
3
4
```

```
  getch(); }
  void vertical(int p){
  if(p<10) cout<<p<<endl;
  else {
  vertical(p/10);
  cout<<(p%10)<<endl;}}
```

ENCAPSULATION

Warping data and functions into a single unit is called encapsulation. Data and function are tied together logically. The term encapsulation describes the data hiding features of oops. Class is abstract idea that supports abstraction in various levels. Data abstraction means hiding and searching information from outside world. In class member variables and function can be declared as private so that they cannot access outside the class. Therefore data becomes more secured and protected from others uses.

Encapsulation is most important features of object oriented programming language. The variables declared inside the class are known as data members and the functions are known as member functions. The variables which are declared inside class as private are also called instance variables because these variables can only changed by methods of same class or at same instance.

Access specifies for visibility

There are three types of access specifies available in C++ programming language. They are public, private, and protected. The key feature of OOP is data hiding (i.e data is enclosed with in a class). If the data items are declared as public, they can be accessed form outside the class with out any restriction. Class methods are usually public by default. Public members can be accessed by members and objects of class. **Syntax:**

public: data member;
member function;

On the other hand, the members of class which declared as private are only be accessed only from within the class. If both of the keywords are missing, then by default all the members are private. Data is most often defined as private. Such a class is completely hidden form the outside access and does not serve any purpose.

Items declared as protected are private within a class and are available for private access in the derived class.

Access Specifies	Accessible form own class	Accessible form derived class	Accessible objects outside class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Example:

```
#include<iostream.h>// pre processor
#include<conio.h>
class obj{// class creation here obj is class name
private: int sum;//accessible in own class
public: void setdata(int d){sum=d;} // accessible in protected own and derived
void show(){ cout<<"\nData is"<<sum;} };// class closed
void main(){// main method
clrscr();
obj o;// object of class
o.setdata(2);//object .function
o.show();
getch();    } //main method close
```

Structure

Structure is a user defined data type with temple that used to define its data properties. The structure is a convenient tool for handling a group of logically related data items. Once the structure type has been defined, we can create variables of that type using declarations, which are similar to the built-in type declarations. Syntax:

struct student {
char name [25];

int roll;
int marks; };

The struct declares student as a new data type that can hold three fields of different data types. These fields are known as elements. The identifier student is the name of struct it holds homogonous data type.

struct student a // here a is declaration as object of that struct. The member variable can be accessed using the **"."** operator. obj.name, obj. roll, obj. marks.

```
#include<iostream.h> // preprocessor
#include<conio.h>
struct info {// structure declaration
char name[30];
char address[25];
char occupation[20]; };//structure first end
struct info1 {// second structure declaration
long int  phone;
int age;  };//second structure end
void main() {// main method
struct info1 i ; //first structure obj
struct info  i1;// second structure obj
cout<<"Enter your phone:"; cin>>i.phone;//passing first structure item
cout<<"Enter the age::"; cin>>i.age;
cout<<"Enter the name:"; cin>>i1.name;//passing second structure item
cout<<"\Enter u address:"; cin>>i1.address;
cout<<"Enter occupation:"; cin>>i1.occupation;
cout<<"\nThe name is:"<<i1.name<<endl;
cout<<"\nThe address:"<<i1.address<<endl;
cout<<"\nThe occupation is:"<<i1.occupation<<endl;
cout<<"\nThe phone is::"<<i.phone<<endl;
cout<<"\nThe age is:"<<i.age;
getch();  }
```

Classes

A class is an abstract idea that represent with data structure and functions. The group of object that can share common properties and relationship among them. In C++ the class is a new data type that contains member variables and member functions which operate on variables. An object is said to be an instance of a class. Class allows data hiding, if necessary for external use. Functions associated with a class are called methods. When we design a class we use new abstract data type that can be treated built-in data type. Generally a class specification has two parts:

class declaration
function definition

Class declaration describes type and scope of its members. The class function describes how the class functions and data members are implemented.

Syntax

```
class item { int num; float cost;
public:
void getdata ( int a, float b);
void putdata();   };
```

The class declaration is similar to a struct declaration but class keyword is used to design class name having its structure. The body of a class is enclosed within brace and terminated by a semicolon (';'). The class body contains the declaration of variables and functions. These functions and variables are called data variables and member function of particular class.

We usually give a class meaningful name, such as item, student, furniture etc. This name becomes a new type identifier that can be used to declare instance of class. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function getdata () can be used to assign values to the member variables and putdata () for displaying their values. These functions only access to the data members from outside the class. This means that data cannot be accessed by any function if they are not is not the member of that class.

Creating Objects

Once a class has been declared, we can create object of that class name. The class representative is known as object. Therefore x is called object of that class. We may also declare more than one object in one statement.

Definition (Object) *An **object** is an instance of a class. It can be uniquely identified by its **name** and it defines a **state** which is represented by the values of its attributes at a particular time.*

The state of the object changes according to the methods which are applied to it. We refer to these possible sequences of state changes as the behavior of the object:

*The **behavior** of an object is defined by the set of methods which can be applied on it.*

The concept of object-orientation programming gives more importance to the class, object and function. Object-oriented programming is therefore the implementation of abstract data type. At runtime instances of these classes, the objects achieve the goal of the program by changing their states. Consequently, find running program as a collection of objects. The question arises of how these objects *interact and to satisfy your demand.*

Message passing

A class is made up with objects, function and data items where objects are created, destroyed and *interacted*. This interaction is based on *messages* which are sent from one object to another asking the recipient to apply a method on itself.

```
  int i;     /* Define a new integer object */
  i.setvalue(1); /* Set its value to 1 */
```

Here integer objects *i* should set its value to 1. This is the message apply method *setvalue* with argument 1 is send to object *i*. We can notate that the sending a message with the help of "." operator is used.

Sending a message asking an object to apply method is similar to the procedure call in traditional programming languages. However, in object-oriented programming is a view of autonomous objects which communicate with each other by exchanging messages. Objects react when they receive messages by applying methods on themselves. They also may deny the execution of a method, for example if the calling object is not allowed to execute the requested method.

In our example, the message and the method which should be applied once the message is received have the same name: We send setvalue with argument 1 to object *i* which applies setvalue (1).

*A **message** is a request to an object to invoke one of its methods. A message therefore contains*

- *the **name** of the method and*
- *the **arguments** of the method.*

Consequently, invocation of a method is just a reaction caused by receipt of a message. This is only possible, if the method is actually known to the object.

The three different parts of message passing are:

Receiver: The object to which the message is send.

Message selector: It indicates the particular message.

Argument: It is information to respond to message it is not most essential component.

To invoke member function we use period (dot) after receiver or object.

Definition (Method) *A **method** is associated with a class. An object invokes a method as a reaction to receipt of a message.*

Accessing class members

The private data of a class can be accessed only through the member functions of that class. The main () cannot contain statements that access number and cost directly.

object name. function (arguments);

x. getdata (45);

Example of class and function

```
    #include<iostream.h>//preprocessor
    #include<conio.h>
    class test{//declaration of class test
    char name[20];
    int roll, marks;//data items
```

```
public : void getdata() { //function to getdata
cout <<"Enter name :";   cin>> name;
cout <<"Enter the roll";       cin>> roll;
cout<< "Enter the  marks";     cin>>marks; }// function close
void putdata(){//function to display data
cout <<"The name is"<<name<<endl;
cout<<"Roll is:"<<roll<<endl;
cout<<"The marks is:"<<marks;  } };// function close class close
void main(){// main method
clrscr();
test t;//object creation
t.getdata(); // calling method
t.putdata();
getch(); }
```

Outside the class definition (Scope resolution operator)

In sometime member functions that are declared inside classes have to be defined separately outside the class. If the programmer wants to use the member function outside the class then scope resolution (::) operator is used to do so. Scope resolution operator can understand that the member function of a particular class is calling from outside the class.

Syntax: void class name :: function name(){ }.

Create a class employee with three data members emp no, name and address, a function created read data to take the details of employee information from the user and another function called display data to display the detail information of employee. In main () create two objects of a class employee and for each object call the read data and display data function.

```
#include<iostream.h>// preprocessor
#include<conio.h>
class employee{ // declaring class name employee
int empno; char name[20]; char add [25]; //data items
public:// publicly accessibility specifies
void read();//one function
void display(); };// another function/ class closed
void test::getdata(){//scope resolution operator for first function
cout<<"Enter empno, name, address";
cin>>empno;
cin.getline(name,20);
cin.getline(add,45); }//end first function
void test::putdata(){// second function start
cout<<"Empno:"<<empno<<endl;
cout<<"Name:"<<name<<endl;
cout<<"address:"<<add<<endl;}//end second function
void main(){// main method
clrscr();
employee t;// object creation
t.getdata();//obj. First member function
t.putdata();
getch();}// main method function
```

Varieties of classes

In general object oriented programming like C++ supports various types of classes. They are super class, derived class, sub class, inherit, anonymous etc.

The varieties of classes are categorized according the task performed by classes are:

1. Data Manager
2. Data source or data sink
3. View or observer class
4. Facilitator or helper class

1. Data manager: we know that object oriented programming supports largely in objects and classes. When designing the class of any items all the data items and member function are to be placed inside the class. It is main class which has responsibility to maintain the data or information it may use by another class. Data manager class is known as data or state class that supports in software system development. Data manager class maintain database and arrange data values which facilitate to easy access of data base classes.

2. Data Source or data sinks: A class which supports two way data production is known as data source class. Data source class accepts some data and processor some data for further used is known as data sink class. This type of class does not hold data for certain period of time but it generates data & supply to others. Data sink means **accept** data and data source means **supply** data therefore this type of class can supply data and accept supplied data and further process. For e.g. a class generates random number supplied by user's range, produce two way generator and accepter. Here the first class that produces random number is fall into data source and another class which accepts the data fall into data sinks categories.

3. View or observer class: This type of class concerned with displaying information on an output devices such monitor. To display information an attractive form may needs the code may be complex and frequently modified therefore, this type of class only handles display behavior.

4. Facilitator and helper class: This class is helper class of data manager class and other classes which facilities the proper maintaining data proper display of information etc.

Constructor and Destructor

Creation of constructor is another way to declared member function in the class. Constructor executes automatically when the member function is declared. **An object initialization is carried out using a special member function is called constructor.** It is a convenient way to initialize object when it is first created without to make separate call to member function. Thus a constructor is a member function that is executed automatically whenever an object is created. Constructor may or may not take arguments.

Rules of constructor

The class name and constructor name must be the same.

Constructor does not have return type (int, float).

Constructor may be declared as without parameters or with parameter.

Constant key word cannot be writing before constructor.

Constructor is executed when the object of the class is created; no special thing programmer has to execute the constructor.

Rules of destructor

The class name and the destructor name must be same.

- Destructor does not have any return type.
- Destructor must be declared as without parameter.
- Constant key word cannot write before destructor.
- Destructor is executed when the object of the class is created.
- Tilled ~ sign is used for denoting the destructor.

Syntax:

```
class test {
data members;
public:
test();// constructor
void display();
~test();//destructor
};// class closed
```

Example:

```
#include<iostream.h>//preprocessor
#include<conio.h>
class test{  //Class declaration
char name[20] ;// class properties
char add[25];
int roll;
public : test(); // constructor method must be public and without return type
void display(); //for display method
~ test();    };//destructor and encapsulation
```

```
void test::test(){//accessing constructor method outside class
cout<<"Enter name address and roll";
cin.getline(name,34);
cin.getline(add,34);
cin>>roll;  } // constructor close
void test:: display(){//accessing display method form outside class
cout<<"Name is:"<<name<<endl<<"Address is:"<<add<<endl<<"Roll:"<<roll; }
test::~test() {}//destructor calling
void main(){// main method
clrscr();
test t; //class object
t.display();//but constructor is not called because it execute automatically
getch();  }
```

Stack versus heap storage allocation

The memory space is used to store value of variable and object allocated in the program. When your program terminates these memories area should have to release. This concept is similar with stack vs. heap storage allocation. There are mainly two ways to allocate and release memory they are automatic and dynamic.

The memory space for static or automatic variable is created when the procedure block containing variables declaration is started. They are released automatically when the procedure is exited/ terminate. The name of variable and memory space is available and cannot change during variable value and their present is existing. The memory allocation of automatic variable is like a stack. Stack means First in Last out. The user cannot release memory of automatic variable.

Heap storage allocation is used by dynamic variables. Dynamic memory allocation is performed by using **new operator** in C++. In dynamic memory allocation, newly created memory is allocated for variables and value can be changed if memory space specifies no longer pointing to it. Hence memory can create when the programmer send request.

Variables created with new operator are called dynamic variables because they are created and destroyed while the program running. When compared with dynamic variables, ordinary variables are in static type. If a variable is local to a function, then the variables is created in C++ system as static. The ordinary variables that we have been using are called automatic variables. They are automatically destroyed when the function call ends.

There is one another category of variables namely global variables. Global variables are variables that are declared outside of any function defined.

Memory Recovery:

If the automated memory is no longer used, heap based allocation techniques can recover it. When the allocated memory is not used then, C++ provides library routine to detect free memory. Other language like Java automatically detects that unused space and automatically make free. In large program large no of variables are used when use large memory space but all the variables may not use always in the program. Therefore unused memory save can recover and save memory.

Mechanism for creation and initialization:

Two types of variables which play very important role in memory allocation are automatic and dynamic variables. An automatic variable is assigned zero automatically when it is declared. The memory space is allocated when block containing declaration. Memory becomes free when control exit form that block. Automatic initialization of variable can be done by using constructor in C++. In constructor, the variables are initialized when the object is created.

It allocates automatic memory.

It automatically initialize variable when object is created.

The memory allocated through constructor can automatically destroyed through destructor.

Write a program to initialize constructors with one, two arguments, that to calculate the area and do square of one side rectangle.

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

```
class area{    // class declaration
int length;    // member data
int breath;
int ss;            //to store square value
public:area();     //default constructor
area(int p);       //constructor with one argument
area(int a,int b); //constructor with two argument
void display1();   //display one method
void display2();};     //display method and encapsulation  class closed
area::area(int a,int b){ // designing outside constructor
length=a;  breath =b;             }
area::area(int p){    // designing outside constructor
ss=p;}
void area:: display2(){  //display first method
int c=ss*ss;             //new int holds square
cout<<"\nThe square of"<<c; }
void area::display1(){ //display second
int p=length*breath;
cout<<"\nArea of "<<p; }
void main(){           //main method
clrscr();
area a(6,7); //constructor with 2 argument
a.display1();
area aa(4);  // constructor with 1 argument
aa.display2();
getch(); }
```

In above example different types of constructor are used. The area () is a constructor having no parameter, is known as default constructor, area (int p) is constructor having one parameter and another constructor area (int a,int b) is constructor having two parameters. The constructor having same name but different parameter is called constructor overloading. Constructor overloading is formed when a classed formed with more than two constructors are used but having different parameter.

In above example initialization of variable is automatic memory allocation for variables are also automatic. To release or make memory free destructor is used.

Static data member and member function

In C++ static member function and member variables are declared by using keyword static. The main rule of static variable is to maintain values common to entire class. Following are some important features of static variable and function.

1. Static data members are initialized to zero when the object of that class is created.
2. Only one copy of static data member created and shared by all class.
3. It is only visible within class.
4. Static function can have access to only other static member of same class.
5. Static member function can be called using class name.

As mentioned above, a method may be declared as static, meaning that it acts at the class level. Therefore, a static method cannot refer to a specific instance of the class (i.e. it cannot refer to `this`, `Me`, etc.), unless such references are made through a parameter referencing an instance of the class, although in such cases they must be accessed through the parameter's identifier instead of `this`.

Give example of static member function and data

```
#include<iostream.h>// preprocessor
#include<conio.h>
class try{//class declaration
int id;
static int tot;//static data member
public :
try(){tot++;id=tot;} constructor and  interchange values tot and id
void print(){// print id
cout<<"ID is:"<<id<<endl; }
static void printcount(){//static member function to print tot
cout<<"\n number of instance are "<<tot; }
  };// class close
int try::tot=0;// some compiler needs to initialized static variable
void main(){// main method
clrscr();
try t,tt; //objects of classes
t.print();//print id
tt.print();//print id second time
try::printcount ();//directly calling static member function
getch(); }
```

Nested class

Nested class is that class in with more than one class are interrelated each other in a main class then we can call them inside the class or outside the class. Both classes should closed with} ;};. We can further divided into inner class or outer class.

Example:

```
#include<iostream.h >
#include<conio.h>
class test{
char name[20];
public :
void get1(); void put1();
class test1{
        int roll;
        public:
        void get2();      void put2();
        };};
void test::get1(){
cout<<"Enter the name";
cin>>name;}
void test::put1(){
cout<<"Name is:"<<name<<endl; }
void test::test1::get2(){cout<<"Enter roll ";
cin>>roll; }
  void test::test1::put2(){
  cout<<"Roll:"<<roll<<endl;}
void main(){
clrscr();
test t;
test::test1 t1;
t.get1();
t1.get2();
t.put1();
t1.put2();
getch(); }
```

Pointer

A pointer is most efficient way of declaring dynamic memory allocation that does not take any value directly, but it can point to some other variable indirectly. It is used to access the address or the values of another variable. The pointer used two operators they are “&” and “*”. One is address operator (&) is used for getting address of another normal variable similarly, the indirection operator (*) which is used for getting the value of another normal variable.

```
#include<iostream.h>
#include<conio.h>
void main(){
clrscr();
int a=10;  int *p; int **p1; int ***p2;
p=&a; p1=&p;  p2=&p1;
cout<<"Address of a:"<<p<<endl;
cout<<"Values of a?"<<*p<<endl;
cout<<"Address of p:"<<p1<<endl;
cout<<"Values of p:"<<*p1<<endl;
cout<<"Address of a:"<<p<<endl;
cout<<"Values of a:"<<*p1<<endl;
cout<<"Address of p1:"<<p2<<endl;
cout<<"values of p1:"<<*p2<<endl;
cout<<"values of a:"<<**P1<<endl;
cout<<"values of p"<<***p2<<endl;
getch();}
```

Dynamic memory allocation (the new and delete operator)

C++ supports dynamic memory allocation by using unary operators (new and delete).The new key word is used to create dynamic memory and the delete keyword is used to release the memory. We can also create object by using “new” keyword and release by using “delete” keyword. The allocated memory will not release inside block until the use of delete operator. The lifetime of an object or data is directly under the control of user and unrelated with block structure.

General syntax: int *a=new int;

Here pointer “*a” is pointer variable is used to pointing new location of variable. The new operator sufficiently allocates memory to hold data.

WAP to calculate length of string by allocating memory for string variable dynamically.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main(){
char *s=new char[45];
int len;
cout<<"Enter name ";
cin.getline(s,45); //cin>>s;
len=strlen(s);
cout<<"length is"<<len<<"\t& text is:"<<s;
delete s;
getch();  }
```

Dynamic initialization of object

Initialization of object at runtime is known as dynamic initialization of object. At the time of execution, user can give input to initialize the object. The advantage dynamic initialization is that we can provide different value as user’s need at the time of execution. Dynamic initialization makes the program more flexible. Similarly;

We can also initialize memory dynamically int *a=new int (10);

We can release memory by using delete operator. These operators enable the system to allocate the enough memory for each object and date. Allocation of memory for objects at the time of their construction is known as dynamic constructor of object.

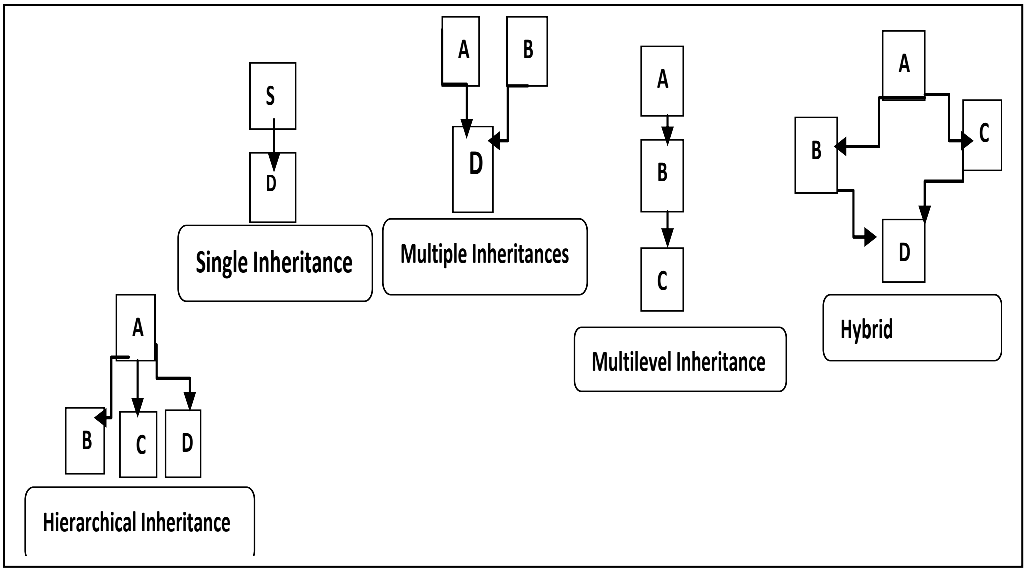
Describe dynamic allocation of memory and give an example of new and delete operator with class

```
  #include<iostream.h>
 #include<conio.h>
 #include<string.h> //to control string
    class add{  //class declaration
    char *str;  // pointer type variable
    public:
    add(char* ptr){  //constructor having one argument
    int i= strlen(ptr);//length of string,
    str= new char[i+1]; //string pointer having length //1 for \0
    strcpy(str,ptr); }  //string copy
    ~add(){ delete str; } //destructor delete operator
     void display(){
    cout<<"The string will be:\n"<<str;}};// display, class close
void main(){   clrscr();
  add a("Goodmorning"); //passing values
  a. display();// calling display method
  getch(); }
```

Inheritance

In programming languages, inheritance means that the behavior, function and data items related to child class are always accessible from the parent class. It is possible to access all the properties of the parent class and adds more properties as well. Reusability is another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also increase productivity. While doing so the derived class must have debugged and tested before it is used, it can be adopted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. ***The mechanism of deriving a new class from an old class is called inheritance (or derivation)***. The old class is known as base class and new one is called as derived class. The derived class inherits some or all properties form the base class. A class also inherits properties form more than one class or form more than one level. For example, if a class Dog is a sub class of Mammal, and class Mammal is a subclass of Animal, then class Dog will inherit attributes both form Mammal and from Animal. They are five types:

1) Single Inheritance
2) Multiple Inheritances
3) Multi-level Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance



Defining Derived Classes

A derived class is defined by specifying its relationship with the base class in addition to its own details. The syntax of defining a derived class is:

class derived-class name: visibility mode base class-name { statements;
statements; };

The colon indicates that the derived class name is derived form the base class name. The visibility mode is optional, if present, may be either private or public. The default visibility mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived. When a base class is privately inherited by a derived class, public members of the base class become private members of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class.

On the other hand, when the base class is publicly inherited, the public members of the base class become public members, of the derived class and therefore they are accessible to the objects of the derived class. In both case the private members of base class are not inherited and therefore, the private members of the base class will never become the members of its derived class.

WAP to demonstrate public, private and protected

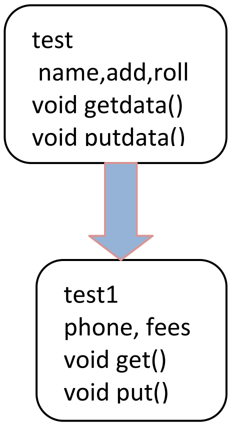
```
#include<iostream.h>
#include<conio.h>
class Parent{     // Parent class
private:    int three;// private data item
protected:  int two;//protected data item
public:     int one;//Public data item and public specifies for all items
Parent(){   //constructor to set data
one=1;two=2;three=3;}
void pdisplay(){//parent display method
cout<<one<<two<<three<<endl;}                   };//parent class close
      class child: public Parent{//new child derived class publicly
                                    //derived form Parent
      public:void cdisplay(){// public specifies child display method
      cout<<one<<two<<endl;} }; //here <<three is not accessible because it
// is private member of Parent class child class closed
void main(){ clrscr();//main method
Parent p;// object of parent class
child c;// object of child class
p.pdisplay();// parent method call
c.cdisplay();// child method call
cout<<c.one<<endl;// cout<<p.two;c.two<<p.three are not accessible
getch();  }
```

Single Inheritance

Inheritance is a property of OOP in which the characteristics of one class comes into another class. In such case a derived class and only one base class are used to share some properties of one another classes.

Example:

```
      #include<iostream.h>
      #include<conio.h>
      class test {      // Super class
      char name[20];char add [25]; int roll;//data members
      public: void getdata(){ //public function
      cout<<"name is add& roll";
      cin>>name>>add>>roll; } //getdata close
      void putdata(){   // for display function
      cout<<"\nName:"<<name<<"\nAdd:"<<add<<"\nRoll:"<<roll;}
                  };
class test1 : public  test {//derivation of super class
long int ph;  float fees;     //derived data members
public :  void get(){   //sub class function for give data
test::getdata();{ // calling super class member function
cout<<"Enter the phone:"<<endl; cin>>ph;
cout<<"Enter the fees";
cin>>fees;  }  }  //givedata closed
void put(){       //derived class function for display data
test::putdata();{ // base class function into derived class for display
cout<<"\nPhone:"<<ph<<"\nFees:"<<fees;}} };//both class closed
      void main(){  clrscr();
      test1 t1;   //object of derived class
      t1.get();
      t1.put();
      getch();    }
```

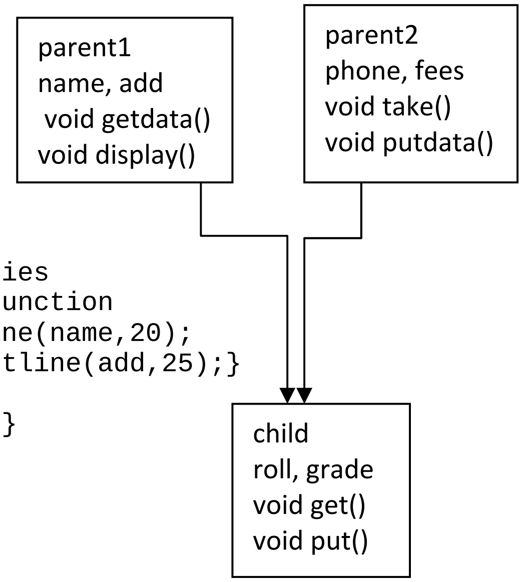


In the above class test is the base class having some properties name, address and roll. All the activities inside base class are handled by two functions getdata() and putdata(). The getdata() function takes data items from the user and putdata() function displays those information to the user. Similarly in the derived class text1 is designed to set out some properties like fees and phone numbers are the own data items. To handled these activities get() and put() are two functions used to manage give data and display data items. While declaring the derived class**” class test1: public test {”** the public visibility mode is used to get access all the data items and function present in the base class into derived class test1. The derived class function gets() is designed in such a way that the base class function getdata() is nesting inside it (public : void get(){ ***test::getdata();*** { **).** Therefore it is possible to get base class properties inside the derived class text1 methods. Similarly, in the main method we just create the object of derived class then the respective functions are called in single inheritance mechanism.

In this way we can access the data items of super class to the subsequent derived class.

Multiple Inheritances

A derived child class can inherit the properties from more than one existing classes as shown figures. Multiple inheritances allow us to get the common features from several existing classes. It is like a child inheritance of physical characteristic of their parents.



The syntax:

class D: visibility A, visibility B {
//statements;
//statements; } Example:

```
         #include<iostream.h>
         #include<conio.h>
         class parent1 {//first super class
         char name[20];char add[25];//parent properties
         public:  void getdata(){//public givedata function
         cout<<"\nEnter Your Name:"<<endl; cin.getline(name,20);
         cout<<"\nEnter Your Address:"<<endl; cin.getline(add,25);}
         void display(){//display function
         cout<<"\nName is:"<<name<<"\nAdd is:"<<add;}
                        };// parent1 closed
class parent2{//another parent2  super class
int ph;  int fees;//data items
public :    void take();//function to givedata
void putdata();//function to display data
            };//parent2 class closed
void parent2::take(){//parent2 function scope resolution to give data
cout<<"\nEnter Your fees:";   cin>>fees;
cout<<"\nEnter Your phone:"; cin>>ph;}
void parent2::putdata(){//parent2 function scope resolution to display
cout<<"\nFees is:"<<fees<<"\nPhone is:"<<ph<<endl;}
   class child: public parent1,parent2{//derived child class from both parents
   int roll;char grade;
   public:void get() {//child class function to give data
   parent1::getdata();parent2::take();//calling both parent functions give data
   cout<<"\Roll"; cin>>roll;
   cout<<"\nGrade:";cin>>grade; }
   void put(){// child class function to display data
   parent1::display();parent2::putdata();//calling both parent functions display
   cout<<"\nRoll:"<<roll<<endl<<"\nGrade:"<<grade<<endl;}
   };//closed derived
      void main(){clrscr();
      child c;//object of derived class
      c.get();c.put();//obj.derived class function name
      getch();}
```

In the above program parent1 is independent super class having getdata () and display () member functions are used to give data to the system and to display parent1 properties. Similarly parent2 is another independent super class having take () and putdata () member functions are used to give data and display the parent2 class properties. The derived child class prepared ***“class child: public parent1, parent2 {“*in this way.** The child class gets all the properties of its super classes and roll and grade are two properties set out of its own. In the derived class get and put functions calls the respective member functions of parent1 and parent2. Therefore the child class becomes the derived class from parent1 and parent2 classes.

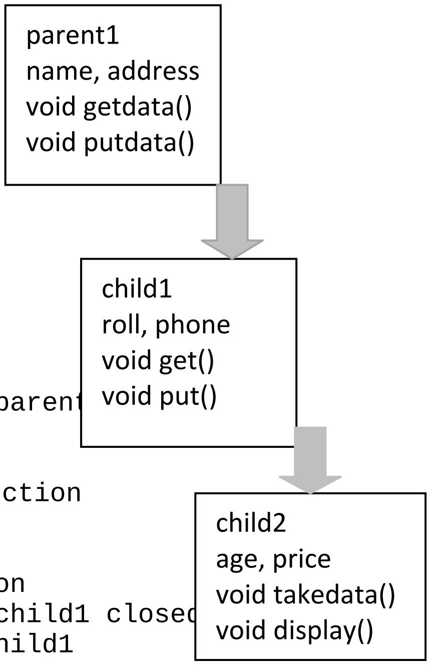
Multilevel Inheritance

Multilevel inheritances can be achieved when we place super class A serves as a base class for the derived class B which again behaves as the base class for the derived class C. The B is known as intermediate base class since it provides a link for the inheritance between A and B. The chain ABC is known as inheritance path. This process can be extended to any number of levels depending upon the requirement.

A derived class with multilevel level inheritances is declared as follows:

class A{……};// Base class

class B: public A{…..};//B derived from A
class C: public B{……….};//C derived from B
Example:



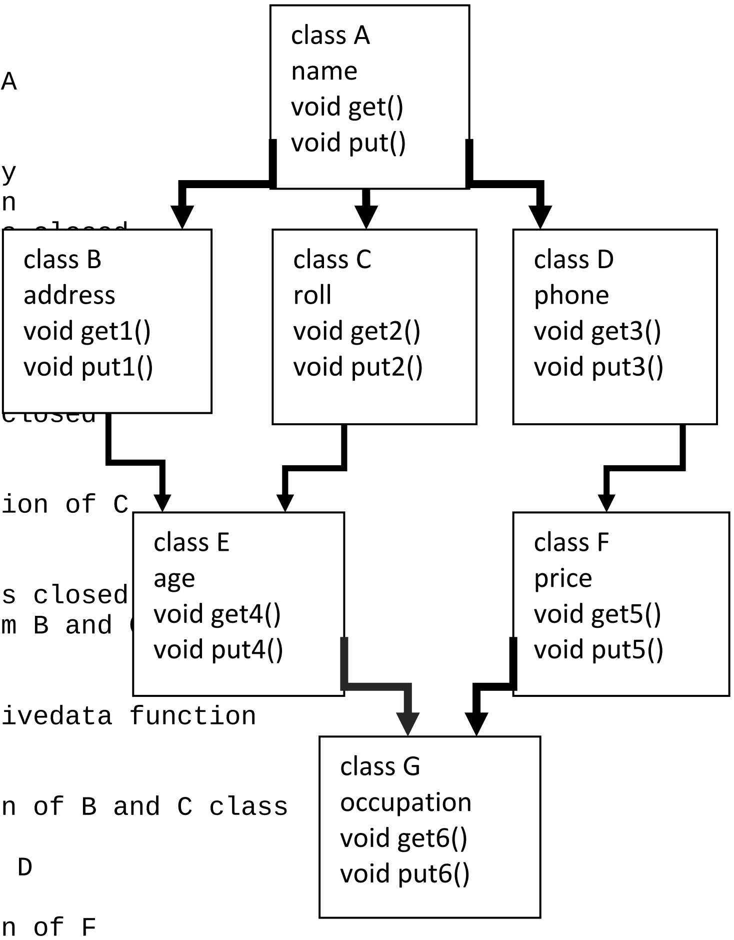
```
      #include<iostream.h>
      #include<conio.h>
      class parent1 {//parent class
      char name[30];  char add[25];//data items
      public: void getdata(){//methods to give data
      cout<<"\n Enter Name and Address:";cin.getline(name,30);
      cin.getline(add,25);  }//method closed
      void putdata(){// method display data
      cout<<"\n Name is:"<<name<<"\n Address is:"<<add;}
         };//parent class closed
            class child1:public parent1{//child1 derived from parent
            int roll; long int ph;//data items of child
            public: void get(){//child member function
           parent1 :: getdata();//accessing parent getdata function
            cout<<"\nEnter roll and phone"; cin>>ph>>roll; }
            void put(){//child1 display function
            parent1::putdata();//calling parent display function
            cout<<"\nRoll is:"<<roll<<"\nPhone is:"<<ph;}};// child1 closed
      class child2:public child1{//child2 derived class form child1
      int age; float price;//member items
      public: void takedata(){//child2 member function to getdata
      child1::get();// calling second derived member function to givedata
      cout<<"\nEnter Age:"; cin>>age;
      cout<<"\nEnter Price";cin>>price;}
      void display(){//child2 member function for display
      child1::put();//calling child1 member function for display
      cout<<"\nAge is:"<<age;
      cout<<"\nPrice is:"<<price; }
      };//child 2 class closed
void main(){ clrscr();
child2 c2; //obj of child2 last derived class
c2.takedata();//obj.child2
c2.display();//obj.child2
getch();  }
```

In the above code parent1 is base class having some properties name, address, this can be operated with the help of two methods getdata and putdata member functions. Similarly new child1 is secondary level derived class from parent1 having roll and phone properties which is operated with the help of get and put methods. Likewise new child2 is again new derived class from just above child1 class only. Child2 class also has some properties age and price can be operated take data and display member function of the child2. This situation maintains different level of inheritance form just senior class.

Hybrid Inheritance

In the hybrid inheritance it possible to access two or more types of inheritance is used to design a program. It is the combination of single as well as multilevel inheritance when grouped together is known as hybrid inheritance. The following figure shows clear picture of this inheritance.

```
     #include<iostream.h>
    #include<conio.h>
    class A {//super class
    char name[30];//data items
    public: void get(){//giving data function
    cout<<"\nEnter Your name:"<<endl;
    cin.getline(name,30); }
    void put(){//display data function
    cout<<"\nYour Name is:"<<name; }
            };//class A closed
      class B: public A {//B derived from A
      char add[30];//data items
      public:  void get1() {//givedata
      A::get();//nesting givedata function of A
      cout<<"\nEnter Your address:"<<endl;
      cin.getline(add,30);}
      void put1(){//member function for display
      A::put();//nesting super display function
      cout<<"\nThe address is:"<<add;}};//class closed
class C: public A {//derived class C from A
int roll;//data items
public: void get2(){//givedata function of C c
cout<<"\nEnter Your roll"; cin>>roll; }
void put2(){//display data function
cout<<"\nYour Roll is:"<<roll; }  };//class C closed
      class D: public A{//D derived class
      long int ph;//data items
      public:   void get3() {//give data function of C
      cout<<"\nEnter Your phone"; cin>>ph;}
      void put3(){//display data function
      cout<<"\nYour phoneno is:"<<ph;}};//class closed
class E: public B,public C{//derived class from B and C
int age;//data items of E class
public: void get4(){//givedata function of E
B::get1(); C::get2();//calling B and C class givedata function
cout<<"\nEnter your age";cin>>age;}
void put4(){//display data function of E
B:: put1();C::put2();//calling display function of B and C class
cout<<"\nYour ageis:"<<age;} };//class closed
      class F: public D{//F derived class form D
      float price;//data items
      public: void get5() {//give data function of F
      D::get3();//give data function of D
      cout<<"\nEnter the price ";cin>>price; }
      void put5() { D::put3() ;//display data of F and calling function from D
      cout<<"\nThe price is:"<<price;}   };//class closed
      class G: public E, public F{//derived class G from E and F
      char occ[34];//data items of G
      public: void get6(){//givedata function of G
      E::get4();F::get5();//calling givedata function of E and F
      cout<<"\nEnter your occupation:"<<endl;cin.getline(occ,34);}
      void put6(){//display function of G
      E::put4();F::put5();//calling display function of E and F
      cout<<"\nYour occu is:"<<occ;}  };//class G closed
            void main(){    clrscr(); //main method
            G g1;//object of G class
            g1.get6();//obj.G class givedata function
            g1.put6();//obj.G class duisplay data function
            getch();  }
```

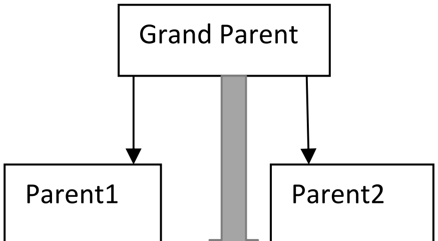


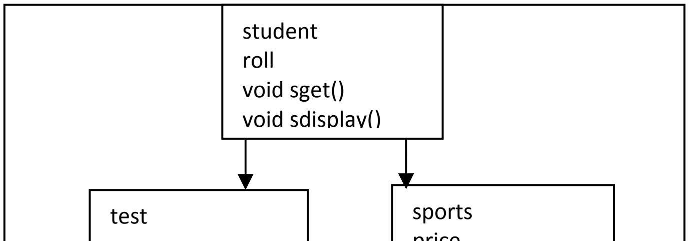
Virtual base classes

We have discussed the situation when we used both multiple and multilevel inheritance. The child class has two direct base classes' parent 1 and parent 2 which themselves have a common base class. The child inherits two times of the grandparent properties. This creates ambiguity and should be avoided.

The duplication of inherited members due to these multiple path can be avoided by making the common base class (Ancestor class) is known as virtual base class.

When a class is made virtual base class, c++ takes necessary care to see that only one copy of that class is inherited, regardless of many inheritance path exists between the virtual base class and the keywords virtual and public may be3 used in either order. In the below figure class student is virtual base class having some





```
   #include<iostream.h>
   #include<conio.h>
            class student{
            int roll;
            public:
            void sget(){cout<<"Enter roll"; cin>>roll;}
            void sdisplay(){cout<<"\nYour roll is:"<<roll; }};
   class  test:  virtual public student{
   char name[34];
   public:
   void tget(){ cout<<"Enter name";  cin>>name; }
   void tdisplay(){ cout<<"\nYour name is:"<<name; }};
      class sports : virtual  public student{
      int price;
      public:
      void ssget(){cout<<"Enter your price";  cin>>price;}
      void ssdisplay(){cout<<"\nYour price is:"<<price;}};
   class result:  public test, public sports{
   int fee;  public:
   void rget(){  tget();   sget();   ssget();
   cout<<"Enter fees";cin>>fee;}
   void rdisplay(){
   student::sdisplay(); test::tdisplay(); sports::ssdisplay();
    cout<<"\nYour fee is:"<<fee; }};
   void main(){
   result r;
   r.rget();
   r.rdisplay(); getch(); }
```

Copy Constructor

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of reconstructing and using a copy constructor

```
#include<iostream.h>
#include<conio.h>
class code{
      int id;
      public:
      code();
      code(int a){id=a;}
      code(code & x){
      id=x.id; }
      void display(){
      cout<<id;} };
            void main(){
            code A(100);
            code B(A);
            code C=A;
            code D;
            D=A;
            cout<<"";A.display();
            cout<<"";B.display();
            cout<<"";C.display();
            cout<<"";D.display();
            getch(); }
```

Give an example of constructor overloading

```
#include<iostream.h>
#include<conio.h>
class complex{
float x,y;
public: complex();
complex(float a){
x=y;y=a;}
complex(float p,float q){x=p,y=q;}
void show(complex);};
void show(complex c){cout<<c.x<<"j"<<c.y<<"\n";}
void main(){
complex a(27,3.5);
complex(1.5);
cout<<"a"<<show(a);
cout<<"b"<<show(b);
getch();}
```

Forms of Inheritance

Inheritance is used in variety of ways to solve users' requirements. In some case it is used to establish relationship between single parent and child. In some cases it become large mesh relationship among parent, child1, child2 etc. as the requirements increases the dependency on parents and child automatically increased. There are some forms of inheritance

a) Subclassing/inheritance 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 class. For example a class Window provides the general windows properties like move, resize, iconification, maximum, minimum and so on. The same window properties are reused to design Notepad, WordPad, Word, and Excel. This is only possible through the use of inheritance.

b) Subclassing for specification

The parent class defines behavior that will be implemented in the child class. The inheritance for specification can be recognized when a parent class does not implement actual behavior but it defines the behavior that will be implemented in the child classes.

c) Subclassing for construction

The child class can be constrructe3d form parent classes by implementing the behaviors of parent classes, the desired behavior can be collected and constructed the child class. Some arguments and behaviors provided by the parent class may redesign inside the sub class construction.

d) Subclassing for generalization

The child class modifies some of the methods of the parent class. It is applicable when we build on a base of existing classes that we do not wish to modify. The child class modifies or override (replace) some method of the parent class. A child class can be extent the behavior of the parent class to create a more general kind of object. Sub classing for generalization is the opposite to sub classing for specifications.

e) Subclassing for extension

The subclass for extension adds new functionality in child class form base class while designing new child class. It is done simply add new method to those of parent class. The child class adds new functionality to the parent class but does not change any inherited behaviors.

f) Subclassing for limitation

The subclass for limitation occurs when the behavior of subclass is similar or more dependent to the behavior of parent class. Subclass for limitation occurs when it is define in the restricted area. In sometime the child class restricts the use of some of the behaviors inherited form the parent class.

g) Subclassing for variance

The child class and parent class are varied when the level of inheritance increased, and the class and subclass relationship is imaginary.

h) Subclassing for combination

The child class inherits features form more than one class. There is a multiple inheritance in which the child class inherits features form more than one classes.

Benefits of Inheritance

a) Software reusability

Due to inheritance some code should not be rewrite, we just reused predefined behaviors of super class to the derived class while designing the program. The behaviors of inheritance increase the productivity of the software designer, because the programmers spend much time to rewrite the same code, that has been written before.

b) Code sharing

Code sharing can be done in several levels of object oriented design; many users can use the same parent class properties while designing the derived class.

c) Consistency in interface

In case of inheritance the behavior use in child class is some in all cases when two or more classes inherit form the same super class. We are assured that the behavior they inherit will be the same in all cases.

d) Software components

An inheritance provides programmers with the ability to construct reusable software components. We can construct the reusable software component by collecting or reusing the behaviors.

e) Rapid prototyping

Since we can reuse the code and we can construct new components, the development time is reduce this features is known as rapid prototyping. With the help of inheritance software systems can be generated more quickly and easily is known as software prototype.

f) Polymorphism

Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be fit in different applications.

g) Information hiding

A programmer who reuses a software component needs only to understand the nature of the components that react to other components. Through the use of private, protected visibility specifies the program become more secured.

Disadvantage of inheritance

1) **Execution speed:** the execution speed may be slow due to interface of different behaviors to reduce execution time. The programmer should spend more time while developing same output.
2) **Program size:** due to inheritance the programming size become large if the developer only concern with reducing the size of program .it is very difficult to produce high quality and error free system.
3) **Program complexity:** The program becomes complexity while using inheritance will be very difficult to understand.
4) **Message passing overloading:** Due to message passing the exception speed becomes less.

The is a rule and the Has a rules of Abstraction

The idea of division into parts and division into specializations represents two most important forms of abstraction used in OOP. They are commonly known as is-a and has a abstractions.

Division into parts is has an abstraction, means the term "**A car has an engine", "A bus has an engine. In such abstraction engine is common type they have specialized in terms of engine's variation.** In programming language similar type of inheritance concept is used that manage data and function associated with child classes are always an extension of the properties associated with parent classes. This rules that is commonly used to test whether two concepts should be linked by an inheritance relationship.

Is a relationship says that the first component is specifies the instance of 2nd component. The data and behavior related to mammals can be inherited to the sub class. The e.g of this relationship is A **dog is a mammal**. There relations derive from a simple rule that test the relationship between two concepts. In such, a case mammal behaves as super class and a dog is object that can inherit all properties of its parent class.

There are two relationships between the components of the system by knowing the relationship, we can easily understand and apply object oriented software reuse techniques.

Has a relationship occurred when second component of the first but both components are not same for example. The **car has a engine**. **The dog has a tail.** By knowing the relationship we can extends the components as many as our requirements.

Is a relationship:

```
class A{
public : void getdata(){
}};
class B: public: A{ };
```

//**"A dog is a mammal"** Here mammal is super its property can be taken to the dog class.

Has a relationship:

```
class A{ }
class B{ public: a obj;};
```

//**here "A car has engine"** the engine is independent properties that can be directly called to the car class without extends and derived from methods.

Abstract class

An abstract class is that class which is not used to create objects. The abstract class is always base class whose members are derived and used by objects of other child class. Due to abstract class no more copy of objects should be created in a program.

```
#include<iostream.h>
#include<conio.h>
class A{
public: int a;int b;};
class B: public A{
int tot,a,b;
public:
 void give(int x,int y){  a=x; b=y;  }
void display(){
tot= a+b;
cout<<"Total:"<<tot; }};
void main(){  clrscr();
B obj;
obj.give(90,80);
obj.display();
getch();}
```

Abstract classes (C++ only)

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (`= 0`) in the declaration of a virtual member function in the class declaration.The following is an example of an abstract class:

```
class AB {
public:
  virtual void f() = 0;};
```

Function `AB::f` is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
  virtual void g() { } = 0;};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
struct A {
  virtual void f() = 0;};
struct B : A {
  virtual void f() { }};
// Error:
```

```
// Class A is an abstract class
// A g();
// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);
int main() {
// Error:
// Class A is an abstract class
//   A a;
   A* pa;
   B b;
// Error:
// Class A is an abstract class
//   static_cast<A>(b);}
```

Class A is an abstract class. The compiler would not allow the function declarations `A g()` or `void h(A)`, declaration of object a, nor the static cast of b to type A.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
  virtual void f() = 0;};
class D2 : public AB {
  void g();};
int main() {
  D2 d;}
```

The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function `f()`from AB. The compiler will allow the declaration of object d if you define function `D2::g()`.

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
```

Pure Abstract Classes

An abstract class is one in which there is a declaration but no definition for a member function. The way this concept is expressed in C++ is to have the member function declaration assigned to zero.

Example

```
class PureAbstractClass{
public:
  virtual void AbstractMemberFunction() = 0; };
```

A pure Abstract class has only abstract member functions and no data or concrete member functions. In general, an abstract class is used to define an interface and is intended to be inherited by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. The users of this class must declare a matching member function for the class to compile.

```
#include<iostream.h>
#include<conio.h>
class area {
double dim1, dim2;
public:
void setarea(double d1, double d2){
dim1 = d1; dim2 = d2;}
void getdim(double &d1, double &d2){
d1 = dim1; d2 = dim2; }
virtual double getarea() = 0; };  // pure virtual function
```

```
class rectangle : public area {
public:
double getarea(){
double d1, d2;
getdim(d1, d2);
return d1 * d2;} };
 class triangle : public area {
public:
double getarea(){
double d1, d2; getdim(d1, d2);
return 0.5 * d1 * d2; } };
  void main() { area *p; rectangle r; triangle t;
r.setarea(3.3, 4.5); t.setarea(4.0, 5.0);
p = &r;
cout << "Rectangle has area: " << p->getarea() << '\n';
p = &t;
cout << "Triangle has area: " << p->getarea() << '\n';
  getch();}
```

Subclass subtype and substitutability

A subclass which inherits the features of upper class is known as subclass. Subclass provides a way of constructing new components using existing components. The particular behavior which is inherited in the base class is known as subtype. Subtype is defined in terms of behaviors not in terms of structure if we have two classes A and B then, A is super class and B is sub class. The behavior of A can be shared by sub class (child class). The relations of the sub class and subtype are clearly defined in terms of the relationship of data types associated with parent class to the data type associated with derived class. One of the most important features object oriented language is that the type associated with a value hold by a variable may not exactly match the type associated with the declaration of that variable. A variable declared as an Integer for example can never hold a value of string. A variable declared as a parent class can hold a variable that is an instance of a child class.

Some relations of subclass and subtype are given below

1.) Instance of sub class must implement through inheritance. It can be access all functions and properties defined in the parent class. Similarly subclass also can be defined new functionality
2.) Instance of subclass influence all data associated with parent class.

Substitutability

The principle of substitutability referred to the relation between variable declared as one class and the variable of class of variable is same as the class if class if variable us sane as the class of variable or sub class variable. The concept of substitutability is provided by inheritance.

Substitutability is a feature of programming in which certain behaviors can substitutes in other of parts of the program. The principle of substitutability says that if we have two classes A and B. Then B is subclass of A that can shared possible substitute on the derived class B for instance of class.

Composition and inheritance

We know that object is simply a concept of substitution of data and behaviors when concept of composition is applied to reduce existing data values in the development of new data type. The portion of state of new data structure is simply the instance of existing data structure. The composition clearly indicates what operations can be called on a particular data structure.

Comparison between composition and Inheritance

Composition	Inheritance

1.Composition indicates the operation of existing structure.
2. In composition cannot reuse code directly but provide greater functionality.
3. It is code become shorter then inheritance
4. In comparison it is very easy to re-implement the behaviors and functions.
Syntax: derived class name(arguments): base class(argument), base class(argument);
`#include<iostream.h>`
`#include<conio.h>`
`class A{`
`public:`
`int p; int q; int r;`
`void getdata(){ cout<<"Enter  first values`
`of pqr";`
` cin>>p>>q>>r;`
`cout<<"\nThe first values are:\n"<<"`
`\np:"<<p<<"\nq:"<<q<<"\nr:"<<r<<endl;}};`
`class B{`
`int q,r;`
`public: A a;// object of a declare here`
`void getdata(){`
`a.getdata();`
`cout<<"Enter second  q and r";`
`cin>>q>>r;`
`cout<<"\nq:"<<q<<"\nr:"<<r; }};`
`void main(){`
`B obj;`
`obj.getdata();`
` getch();}`
	1.Inheritance is a super set of existing structure when object is located.
	2.Inheritance can be directly reused the code and function provided by parent class.
	3.I Its code become longer than composition
	4.It is difficult to re-implement the behaviors rather than composition.
	syntax: child class name: visibility parent class
	`#include<iostream.h>`
	`#include<conio.h>`
	`class A{`
	`public:`
	`int p; int q; int r;`
	`void getdata(){ cout<<"Enter  first`
	`values of pqr";`
	` cin>>p>>q>>r;`
	`cout<<"\nThe first values`
	`are:\n"<<"\np:"<<p<<"\nq:"<<q<<"\nr:"<<r<`
	`<endl;}};`
	`class B: public A{`
	`int q,r;`
	`public:`
	`void getdata(){`
	`A::getdata();`
	`cout<<"Enter second  q and r";`
	`cin>>q>>r;`
	`cout<<"\nq:"<<q<<"\nr:"<<r; }};`
	`void main(){`
	`B obj;`
	`obj.getdata();`
	` getch();}`

Create a class bank account including following data members Name of depositor, account number, account type, Bank amount and member function to assign initial values to deposit amount

To withdraw amount

To display values and balance amount.

```
#include<iostream.h>
#include<conio.h>
class bbalance{
char name[20];
long int accno;
char type[90];
float amount;
public: void  create(){
cout<<"Enter your name"; cin>>name;
cout<<"Enter your accno"; cin>>accno;
cout<<"Enter your type"; cin>>type;
cout<<"Enter your name balance"; cin>>amount;}
void deposi(float amt){
 amount+=amt;}
void withdraw(float amt){
if (amount< amt) cout<<"Invalid";
else amount-=amt; }
void display(){
cout<<"Name:"<<name;
cout<<"accoutno:"<<accno;
cout<<"Type:"<<type;
cout<<"Balance is"<<amount;
cout<<"Thanks for visit to Kamana Bank";}};
```

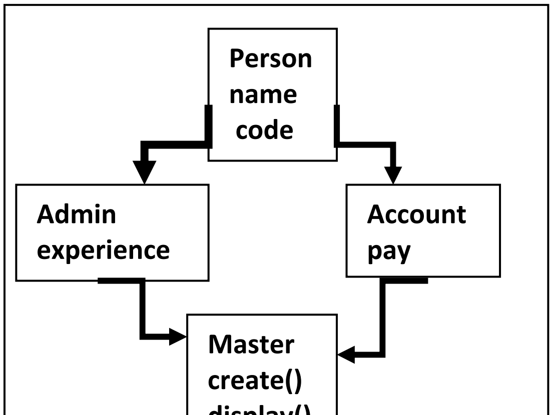
```
void main(){
clrscr();
bbalance a1;
a1.create();
a1.deposi(10000);
a1.display();
cout<<"\nThanks for depositing\n";
a1.withdraw(2000);
cout<<"Thanks for withdraw";
a1.display();
cout<<"Final thanks";
getch();}
```

Create a class called city that will have two member variables city name char [20], distance form in float, add member functions to set and retrieve the city name and distance form Kath separately. Add new member function add distance that takes two arguments in the main function initialized the three objects set first and second city to the Pokhara and Dhandnadi. Display the distance form Kathmandu to Pokhara and distance Kathmandu to Dhandnadi calling add distance of third city object

```
#include<iostream.h>
#include<conio.h>
class city{
char name[30];
float dfktm;
public:
void setdata(){
cout<<"Enter Cityname";
cin>>name;
cout<<"Enter distance form";
cin>>dfktm;  }
void retrive(){
cout<<"Cityname is:"<<name;
cout<<"Distanceform"<<dfktm; }
float adddistance(city c1, city c2){
 float d;
d=c1.dfktm+c2.dfktm;
return d;}};
void main(){
city c1;
city c2;
city c3;
c1.setdata();
c2.setdata();
c1.retrive();cout<<endl;
c2.retrive();cout<<endl;
cout<<"The total distance form Kathmandu: "<<c3.adddistance(c1,c2);
getch();}
```

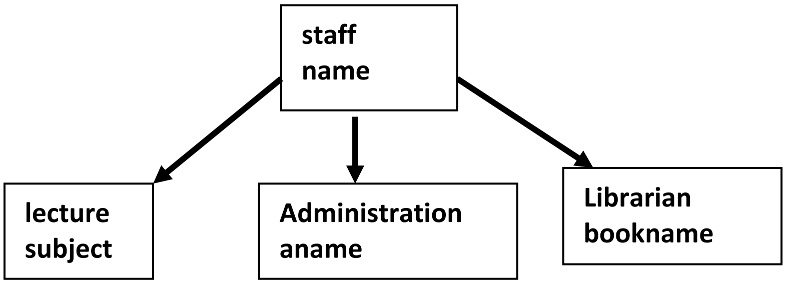
Refer the figure given below the class master derived information form both amount and admin class which inform derived information form the class person. Define for classes and write a program to create and display that information

```
#include<iostream.h>
#include<conio.h>
class person{
protected:
char name[30];
int code; };
class admin: virtual public person{
protected: int exp;};
class account:  virtual public person{
protected: float pay;};
```



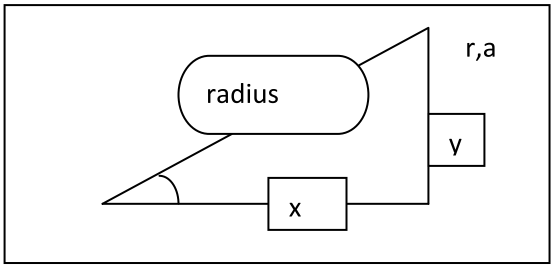
```
class master: public admin, public account{
public:
void create(){
cout<<"Enter your name:";cin.getline(name,45);
cout<<"Enter code:";cin>>code;
cout<<"Enter experience:";cin>>exp;
cout<<"Enter pay:";cin>>pay;}
void display(){
cout<<"\nName is:"<<name;
cout<<"\nCode:"<<code;
cout<<"\nExprienc:"<<exp;
cout<<"\nYour pay:"<<pay;} };
void main(){
master m;
m.create();
m.display();
getch();}
```

Develop a complete program for an institution which wishes to maintain a database of its staffs the database is derided in numbers of classes whose relationship is shown in figure specifies all the classes and display constructor and function top create database and retrieve the individual information as pre the requirements.



```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class staff{
int id;
char name[34];
public:
staff(int i,char n[]){
id=i;
strcpy(name,n);}
void display(){
cout<<"\nYour id is:"<<id;
cout<<"\nYour nameis:"<<name;}};
class lecture: public staff{
char sub[20];
public:
lecture(int i,char n[],char s[]):staff(i,n){
strcpy(sub,s);}
void display(){
staff::display();
 cout<<"\nYour subject is:"<<sub;}};
class admin: public lecture{
char aname[34];
public:
admin(int i,char n[],char s[],char as[]):lecture(i,n,s){
strcpy(aname,as);}
void display(){
lecture:: display();
cout<<"\nYour admname is:"<<aname;} };
class lib: public admin{
char bname[34];
public:
lib(int i,char n[],char s[],char as[],char bn[]):admin(i,n,s,as){
strcpy(bname,bn);}
void display(){
admin:: display();
cout<<"\nYou book name is:"<<bname;} };
void main(){
clrscr();
lib l(1,"Yagya","Computer","ChandParkah","Object Oriented in C++");
l.display();
getch();}
```

Design a class polar which describes a point in the plane using polar coordinate radius and angle. A point in polar coordinates.
Use + operator overloading
Use the following trigonometry functions
x=r* cos(a)
y=r*rsin(a)
a=atan(x/y)
r=sqrt(x*x+y*y)



```
 #include<iostream.h>
 #include<conio.h>
 #include<math.h>
 class polar{
 public:
 double radius,angle;
 double getx(){return radius*cos(angle);}
 double gety(){return radius*sin(angle);}
 polar(){ radius =0.0;   angle=0.0;}
 polar(float r,float a){radius=r;angle=a;}
  void display(){ cout<<"Radius"<<radius;   cout<<"Angle"<<angle; }
  polar operator+(polar p2){ double x=getx()+p2.getx();
  double y=gety()+p2.gety();  double r=sqrt(x*x+y*y);
  double a=atan(x/y); return polar(r,a);}};
      void main(){
      polar p1(10.0,0.0);
      polar p2(10.0,1.57);
      polar p3;
      p3=p1+p2;
      p1.display();
      p2.display();
      p3.display();
      getch();      }
```

Create a class time represent with a member to take represent second, minute, hour, read the value of time in seconds form the user as basic data type and display the result in hh::mm:: ss?

```
#include<iostream.h>
#include<conio.h>
class time{
int ss,mm,hh;
public: time(){}
time(long int s){
 hh=(int)s/3600;
 s=s%3600; mm=s/60; ss=s%60;}
void display(){cout<<"hh"<<hh<<"mm"<<mm<<"ss"<<ss;}};
void main(){ clrscr();
time t=3701;
 t.display();
getch();}
```

Polymorphism

In programming languages, *polymorphism* means that some code or operations or objects that behave differently in different contexts.

For example, the + (plus) operator in C++:

```
4 + 5       <-- integer addition
3.14 + 2.0  <-- floating point addition
s1 + "bar"  <-- string concatenation!
```

In C++, that type of polymorphism is called *overloading*.

Definition:

Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings or functions to the operators or functions. Poly, referring to many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Polymorphism is a powerful feature of the object oriented programming language C++. A single operator + behaves differently in different contexts such as integer, float or strings referring the concept of *polymorphism*. The above concept leads to operator *overloading*. The concept of overloading is also a branch of *polymorphism*. When the exiting operator or function operates on new data type it is *overloaded*. This feature of polymorphism leads to the concept of *virtual methods*.

Polymorphism refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code of different shapes such as circles, squares, rectangles, etc. One way to define each of these classes is to have a member function for each that makes each shape. Another convenient approach the programmer can take is to define a base class named Shape and then create an instance of that class. The programmer can have array that hold pointers to all different objects of the followed by a simple loop structure to make each instance, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for *Virtual function* implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

- All different classes must be derived from a single base class. In the above example, the shapes of (circle, triangle, and rectangle) are from the single base class called Shape.
- The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

Features and Advantages of the concept of Polymorphism:

Applications are easily extendable:

Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- Helps in reusability of code.
- Provides easier maintenance of applications.
- Helps in achieving robustness in applications.

Types of Polymorphism:

C++ provides three different types of polymorphism.

- Operator overloading
- Virtual functions
- Function name overloading

In addition to the above three types of polymorphism, there exist other kinds of polymorphism:

- run-time

- compile-time
- ad-hoc polymorphism
- parametric polymorphism

Other types of polymorphism defined:
Run-time:
The *run-time* polymorphism is implemented with inheritance and virtual functions. At runtime when it is known what class object are under consideration the appropriate versions of the function is invoked. Since the function is linked with the particular class much later after compilation, this process is term as late binding. It is also known as dynamic binding because the selection of the appropriate function is done dynamically at runtime.
Compile-time:
The *compile-time* polymorphism is implemented with templates. The operation of doing same name and different forms of action is done by the compiler, so compiler is able to select the appropriate function for a particular call at compile time. This is called early binding (static binding). This is also known as compiling time polymorphism. This compile time polymorphism means that an object is bound to its function call at compile time.
Ad-hoc polymorphism:
If the range of actual types that can be used is finite and the combinations must be individually specified prior to use, this is called *ad-hoc* polymorphism.
Parametric polymorphism:
If all code is written without mention of any specific type and thus can be used transparently with any number of new types it is called *parametric* polymorphism.

Operator overloading: In computer programming, operator overloading is a specific case of polymorphism in which some or all of operators like +, =, or == have different implementations depending on the types of their arguments. Sometimes the overloadings are defined by the language; sometimes the programmer can implement support for new types. Operator overloading is claimed to be useful because it allows the developer to program using notation "closer to the target call and allows user-defined types a similar level of syntactic support as types built into the language. It can easily be use function calls.
Operator overloading means giving addition meaning of operator when they applied to user defined data-types. The concept of operator overloading is not only applicable for user defined data type but also used for normal application. Some examples are operators are used to multiply to members as well as it is used to assign the pointer variable, + operator is used to add the two numbers as well as it is used to concatenate two strings. By overloading operator, it original meaning will not lost.
To define an additional task to an operator, we have to define the relation of operator with class and OOP provides a special function called operator function. The operator function is defined as follows.
reurtn type class:: operator(arguments list){
….. ;;;;;; }
We can overloaded all C++ operator except.*, ::, sizefo and ?:

Unary Operator Overloading

```
#include<iostream.h>
#include<conio.h>
class unary{
int p; int q; int r;
public:
   void getdata(int a,int b,int c);
   void display();
   void operator-();  };
 void unary:: getdata(int a,int b,int c){
 p=a;q=b;r=c;}
 void unary::display(){
 cout<<"p is:"<<p<<"\nq is:"<<q<<"\nris:"<<r;}
 void unary::operator-(){
 p=-p;q=-q;r=-r; }
```

```
void main(){ clrscr();
unary u;
u.getdata(30,-50,40);
cout<<"before oberloaded\n";
u.display();
cout<<"\nAfter oberloaded";
-u;//overloading -operator
u.display();
getch(); }
```

In the above program –ve operator is overloaded there operator is used to make –ve values of built in data type of p, q and r. The – operator is used to change the sign of data members of the object (-u) since this function is a member function of the same class, it can directly access the member of the object).

Another example:

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class polar{
public:
float radius, angle;
float getx(){ return (radius*cos(angle));}
float gety(){ return (radius*sin(angle));}
public:
polar(){ radius=0.0; angle=0.0;}
polar(float r, float a){ radius=r; angle=a;}
void display(){ cout<<"(:"<<radius<<"):"<<angle;}
polar operator+(polar p2){
double x=getx()+p2.getx();  //cout<<"X:"<<x;
double y=gety()+p2.gety();//cout<<"Y:"<<y;
float r=sqrt(x*x+y*y);
float a=atan(y/x);
return  polar(x,y);  }};
void main(){ clrscr();
polar p1(10.0,0.0);
polar p2(10.0,1.507);
polar p3=p1+p2;
cout<<"\nfirst display\n";
p1.display();
cout<<"\nsedond display\n";
p2.display();
cout<<"\nThird display\n";
p3.display();
getch();}
```

Write a program to add two complex numbers by using Binary operator overloading

```
#include<iostream.h>
#include<conio.h>
class complex{
float x,y;
public:
complex(){}
complex(float r,float i){ x=r;y=i;}
complex operator+(complex);
void display(); };
complex complex::operator+(complex c){
complex temp;
temp.x=x+c.x;
temp.y=y+c.y;
return (temp); }
void complex:: display(){
cout<<x <<"/t"<<y<<endl;}
void main(){ clrscr();
complex c1,c2,c3;
c1=complex(4.1,3.5);
c2=complex(1.6,2.4);
c3=c1+c2;
cout<<"c1";c1.display();
cout<<"c2";c2.display();
cout<<"c3";c3.display();
getch(); }
```

In above example + operator is overloaded. + Operator is applying to add the user defined variable c1, c2,c3. C3 is sum of c1 and c2. The operator function have only on e parameter. The data member of c1 is access directly and c2 is access by using dot operator.

++ operator loading

```
#include <iostream>
#include<conio.h> #include
class ThreeD {
  int x, y, z;
public:
  ThreeD() { x = y = z = 0; }
  ThreeD(int i, int j, int k) {x = i; y = j; z = k; }
  ThreeD operator++(); // prefix version of ++
  void show() ; } ;  // Overload the prefix version of ++.
ThreeD ThreeD::operator++(){
  x++;    y++;      z++; // increment x, y, and z
  return *this;}  // Show X, Y, Z coordinates.
void ThreeD::show() {
  cout << x << ", ";     cout << y << ", ";    cout << z << "\n"; }
void  main(){
  ThreeD a(1, 2, 3);
  cout << "Original value of a: ";
  a.show();
  ++a;  // increment a
  cout << "Value after ++a: ";
  a.show();    getch(); }
```

Type casting (type conversion)

```
int m;
float x=3.1414;
m=(int)x;
```

The programming language simply assignment operator does automatic type conversion. The data type of data to right of an assignment operator is automatically converted to the type of variable on the left. In this case the fractional part is truncated. But in the case to add two object and assigns the result to the third object. When we try out to calculate the values of object to be passed on the third variable in v3= v1 +v2, this will create great problem to hold those objects in v3. In such case the compiler does not make any compliant. In case the object the value of all the data members of right hand object are simply copied into the correspondent members of object.

Since the user defined data types are designed by us to suit our requirements, the compiler does into support automatic type conversion. If an expression contains different data like integer and float the left and right hand side then the compiler automatically convert form from type to another type by applying the type conversion rule provide by compiler. The automatically type conversion means that the right side of data is automatically converted to left side type. This type of conversion rule can not apply in case of user defined data type. In user defined data type, we must define conversion routines ourselves. The possible type conversion is as

1. Conversion from basic type to class type/User defined type
2. Conversion from class type to basic type
3. Conversion from one type to another type.

Conversion between Basic to user defined/Class type

This conversion form basic type to class type is easily accomplished. It may be recalled that the user of constructor. The compiler is built to time object. Note: that the constructor used for the type conversion takes a single argument takes a single argument whose type is to be converted.

```
#include<iostream.h>
#include<conio.h>
class time{
int h,m;
public:
time();
time(int t){//constructor
h=t/60;
m=t%60;}
void display(){
cout<<"Hour:"<<h;
cout<<"Minutes"<<m;}};
void main(){
time t1
int a=4566;
t1=a;//type conversion
t1.display();
getch();}
```

In above the constructor time (int t) converts a basic type int to user defined data type time. This is called when obj is created with single argument.

Make a class called memory with member data to represent bytes: Kilobytes and megabytes. Read the value of memory in bytes form user as basic data type and display the result in user defined memory type **(Hint use basic to user defined type conversion)**

```
#include<iostream.h>
#include<conio.h>
class memory{
int byte, mb,kb;
public:
memory(int b){
mb=b/1048576;//(1024*1024)
int fkb=b%1048576;
kb=fkb/1024;
byte=fkb%1024;}
void display(){
cout<<"\nThe mb:"<<mb;
cout<<"\nThe kb:"<<kb;
cout<<"\nthe byte:"<<byte;}};
void main(){
memory m=243242323;
m.display();
getch();}
```

Conversion form one user defined data type to basic type

The constructor function does not support this operation so C++ allows us to define an overloaded casting operator that could be used to convert user defined to basic type. When compiler encounter a statement that requires that conversion of a class type, it is calls the casting operator function to do the job, casting operator function should satisfied: It must be a class member

It must not specifying a return type

It must not have any arguments.

Syntax:

Operator type name (){………

;;;;;;;;;;;;;}

WAP to convert time into minute

```
#include<iostream.h>
#include<conio.h>
class time{
```

```
int h,m;
public:
time( int a,int b ){ h=a;m=b;}
operator int(){ int mm;
             mm=h*60+m;
             return mm;}};
void main(){
time t1(2,59);
int p=t1;
cout<<p;
getch();}
```

Let us take an example which converts distance in meter to feet and inch and feet and inches. Two types of conversions techniques are applied in which converts basic type to user defined and user defined to basic type

```
#include<iostream.h>
#include<conio.h>
const float m=3.280833;//constant
class distance{ //class
int feet;
float inches;
public:
distance(){//constructor
 feet=0; inches =0;}
distance(float meter){//constructor with one argument
float ft= m*meter; //convert meter to feet
feet=int(ft);
inches= 12*(ft-feet); }
distance(int f, float i){//constructor two argu
feet=f, inches =i;}
void show(){
cout<< feet<<"\t"<<inches;}
operator float(){//conversion function to meter
float fractfeet=inches/12;//convert inches
fractfeet+=float(feet); // add to feet
return fractfeet/m; }}; // convert to meter
void main(){
distance d=2.35;
cout<<"Distanceis :\n";
d.show(); d=1.00;
cout<<"\ndistancwe";
d.show();
distance d2(10,10.25);
float met=float(d2);
cout<<"\ndistance";
cout<<met;
met=d2;
cout<<met;
getch();}
```

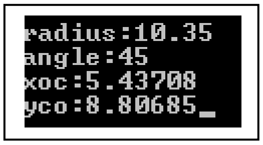
In main section of above program, conversion of float value to user defined data type distance occurs. distance d=2.35; in this line 2,35 float value is converted to distance d and finally convert to feet and inches by calling one argument constructor. Another function operator float () function convert feet and inches to meter in this line mtrs= float (d2); this converts the distance object to its equivalent float value in meters.

Conversion between object of different classes

Define a class name as polar and rectangle to represent rectangle form. Use conversion routine to convert for m one type to another (Polar to REC)

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class Rec{
float yco;       float xco;
```

```
 public: Rec(){
 xco=0.0;   yco=0.00;}
 Rec(float x,float y){
 xco=x;  yco=y; }
 void display(){
 cout<<"xoc:"<<xco;
 cout<<"\nyco:"<<yco;} };
 class pol {
 float radius;  float angle;
 public:
 pol(){ radius=0.0;angle=0.0;}
 pol(float r, float a){
 radius=r;  angle= a; }
void display(){
cout<<"radius:"<<radius<<"and angle:"<<angle;}
 operator Rec(){
  double x=radius*cos(angle);
  double  y=radius*sin(angle);
 return Rec(x,y); } };
void main(){
clrscr();
Rec r;
pol p(10.35,45);
r=p;
cout<<"\npol";  p.display();
cout<<"\nrec";  r.display();
getch();  }
```



```
radius:10.35
angle:45
xoc:5.43708
yco:8.80685_
```

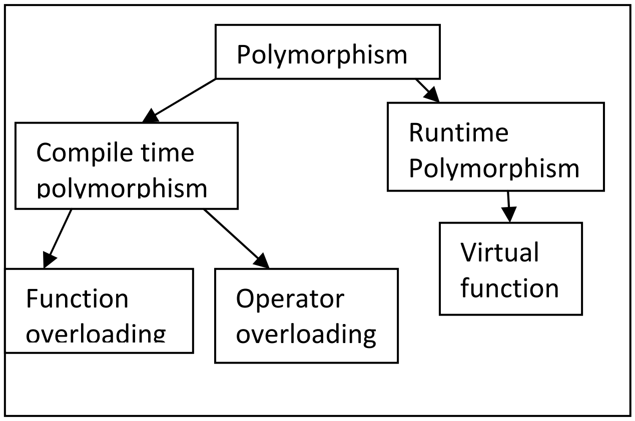
The conversion between objects of different classes can be done in the same way may as conversion between basic to user defined type.

Pointer

It simply means one name, multiple form, we have already seen   how the concept of polymorphism is implemented using the overloaded function and operators. The overloaded members’ functions are selected for invoking by making arguments both type and number. This information is known to the compile at the time compile time. And therefore compiler is able to select the appropriate function for a particular call at the compiler time itself. This is called early binding or static binding, it also known as compiler time polymorphism. Early binding simply means that the objects to be bound to its function call at compile time.

class A{
int x;
public: void show(){}};
class b: public A{
int y;
public: void show(){};

In this case void show () function doesn’t overloaded. In fact the compiler does not understand what to do and which function calling by the compiler. It would be very nice if we could select appropriate function while the program running. The C++ supports a mechanism known what objects are under consideration, the appropriate function called. It links the particular class much later after the compilation process is known as compilation binding/ late binding.



The dynamic binding is one of the powerful techniques it requires the use of pointers objects,

Pointer to derived class:

Pointers can be used to point the object of derived class which is type –compatible with pointers to base object. Consider there are two classes, be is super class and d is derived class.

B*ptr;//pointer to class b type

B b;//base object

D d;// derived object

Ptr=&b;//ptr point to object

If we want to point object d by ptr then ptr=&d;

However there is a problem of the derived class D. Using ptr, we can access only those members inherited form B and not the members that originally belong to D. In case member of D has the same name as one of the members of B then any reference to that member by ptr will always the base class member

```
#include<iostream.h>
#include<conio.h>
class B{
public: int b;
void show(){ cout<<"b"<<b<<endl;}};
class D: public B{
public: int d;
void show(){cout<<"b="<<b<<endl<<"d="<<d;}};
void main(){
B *ptr; //pointer object
B b;  //normal object
ptr=&b;//passing normal to pointer object
ptr->b=100;
ptr->show();
D d;ptr=&d;ptr->b=300;
//ptr->d=300; not works
ptr->show();
D* dptr;dptr=&d;
dptr->d=300;
dptr->show();
getch(); }
```

Polymorphic Variable:

Polymorphism is not only possible through overloading (Operator and functions) but also though polymorphic variables. Polymorphic variable are those variables which have same name but can hold different type of values. For example a variable can hold integer as well as float values polymorphic variables are used in subtype. In c++ the polymorphic variable should occur if declare class of a variable and the class which use the declared variable is different. In c++ polymorphic variable are possible to use only by using pointer or references.

int a=10;

float b=20.4; int *p=&b;

```
#include<iostream.h>
#include<conio.h>
class item{
int code;
float price;
public:
void getdata(){
cout<<"Enter code and price";cin>>code>>price;}
void display(){
cout<<"Code"<<code<<"price"<<price;}};
void main(){
item x;
item *ptr;
ptr=&x;
ptr->getdata();
ptr->display();
getch(); }
```

Function overloading

Function overloading is a process in which more than one function can have the same name but with different arguments. Overloaded functions are selected by matching both member and types of argument when they are called. There information is known to the compiler at the time of application and therefore compiler is able to select the appropriate function for a particular call at the compile time itself.

Advantages:

It helps to understand developed the program

It is easy to maintain

It eliminates the use of different functions name for the same operation

It helps to design a family of function that do essentially the same thing but using the different arguments.

For eg in c++ following two functions are defined:

float divide(int s, int y)

float divide(float x, float y)

Example:

```
#include<iostream.h>
#include<conio.h>
class shape{
float a,l,b,r;
public:
void area(float length, float breath){
l=length;
b=breath;
a=l*b;
cout<<"Area of rectangle"<<a;}
void area(float radius){
r=radius;
a=3.14*r*r;
cout<<"Area of circle:"<<a; } };
void main(){
shape s;
s.area(5.5,20);
s.area(5.75); getch(); }
```

Method overriding

The method overriding contributes to share the code. If the method defined in super class is into appropriate for base class and should slightly changes, at that time we can override the method. Without overriding, it would be necessary for all sub classes to provide their own method to respond to message.

Method overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a Method that has same name, same parameters or signature, and same return type as the method in the parent class. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, or If an object of the subclass is used to invoke the method, then the version in the child class will be executed

```
#include<iostream.h>
#include<conio.h>
class rectangleType{
public:
   void print();
private:
   double length;
   double width;};
void rectangleType::print(){
  // print() version of base class
cin>>length>>width;
   cout << "Length = " << length << "Width = " << width;}
class boxType : public rectangleType{
public:
   void print();
private:
   double height;};
void boxType::print()  // print() version of derived class
{    cin>>height;
  rectangleType::print();  // Invoke parent version of method print().
  cout << "Height= " << height;}
void main(){
boxType r;
r.print();
//r.rectangleType::print();
getch();}
```

Pure Polymorphism:

In case of C++ some function name can have different parameters and perform different functions. This situation is called overloading. This function is also called polymorphism function. In case of pure polymorphic, the type of arguments used in overloaded function is not known until run time. The message passing can be performed to distinguish the type of variable used in overloaded function.

Virtual function:

The object of different classes can respond to same message in different forms we can access the function by using object which is declared as single pointer variable. To access pointer object of base class this can independently access the member. The pointer of base class refers to all the objects of derived class. The function which is access by pointer object having same name in both base and derived class is called virtual function. The function is declared using key word virtual. When function is made virtual, C++ determines that function is used to at run time based on type of object pointed on base pointer.

```
#include<iostream.h>
#include<conio.h>
class base{
public:
void display(){ cout<<"\display Base class";}
virtual void show(){cout<<"\n show base class";} };
class derived: public base{
public:
void display(){ cout<<"\derived class";}
void show(){ cout<<"\n show derived class";}};
void main(){ clrscr();
base b;
derived d;
base*bpt;
cout<<"\pointer to base";
bpt=&b;
bpt->display(); // call base class
bpt-> show(); //call base class
cout<<"\n pointer to derived claps";
bpt=&d;
bpt-> display();
bpt-> show();
getch();}
```

In above bpt-> display () call only one base version but bpt->show() calls derived class version of show(. This is because show () method is virtual function.

We can access virtual member functions by using dot operator followed by object name but why we use pointer object. The main purpose of pointer object is to make runt time polymorphism. In run time polymorphism, the compiler checks the appropriate member form more members declared in various classes with same name.

Rules for virtual functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using objects pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. A virtual function in a base class version of a virtual function and the entire derived class version must be identical. If two functions with the same name have different prototype, c++ consider them as overloaded functions, and function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to type of the derived object, the reverse in not true. That is we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points a derived class, incrementing or decrementing it will not make it to point the next objects of derived class.
10. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
11. If a virtual function is defined in the base class. It need not be necessary redefine in the derived class.
In such case , call will invoke the base function

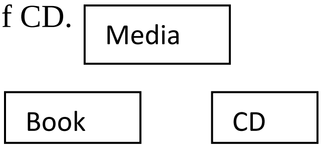
Pure virtual function (Deferred Method)

A pure virtual function is a member function of super class (base class) which is null and redefined in the derived class where it is used. This is also derived class whenever it is used. This is also called deferred method or pure polymorphism or does nothing function.

Function declaration

virtual return type function name()=0;
virtual void show()=0;

So the deferred methods are declared in abstract class (a class which is not used to create the object). Consider a book shop which sells both books and CD. Create a class name as media that store the title and price of publication. Then create two derived classes named as book which stores number of pages in book and another derived class named as tape which stores playing time of CD.



```
 #include<iostream.h>
#include<conio.h>
#include<string.h>
class media{
public:
char title[30];
float price;
public:
media(char *s, float p){
strcpy(title,s);
price=p;}
virtual void display()=0;  };
class book: public media{
int page;
public:
book(char *s,float k, int p):media(s,k){
```

```
page=p;}
void display(){
cout<<"Title:"<<title<<endl<<"Price:"<<price<<endl<<"Page:"<<page;}  };
class cd: public media{
int ptime;
public:
cd(char *s, float k, int pt): media(s,k){
ptime=pt;}
void display(){
cout<<"\nTitle:"<<title;
cout<<"\nPrice Rs:"<<price;
cout<<"\nTimetoplay:"<<ptime;}  };
void main(){ clrscr();
media *m;
book b("Computer:",400,890);
cd c("Objects Oriented",179,80);
m=&b;
m->display();
m=&c;
m->display();
getch();   }
```

Friend function

We know that private members cannot access outside the function and non member can access the private data from class. C++ provides a special function call; friend function or friendly function which is shared by more classes and also can access the private members of class.

Syntax:

```
class test
public:
friend void function ();};
```

The friend function should be preceded by keyword friend. The function can define anywhere in the program. The function definition does not contain either keyword friend and scope resolution (::)

Some features are:

- It cannot access by the object of class
- It can invoke like normal function
- It cannot access member data directly. It should use data member of class by object of class.
- It can be declared by where in the program

```
#include<iostream.h>
#include<conio.h>
class average{
 private: int a,b;
public:
void setvalue(){  a=21,b=48; }
friend float mean(average av);};
float mean(average av) {
return float(av.a+av.b)/2.0;}
void main(){
average avg;
avg.setvalue();
cout<<"Final average is";
cout<<mean(avg);
getch();}
```

Here, friend function access the members a and b outside the class even they are private members.

```
#include<iostream.h>
#include<conio.h>
class test{
private:  int x;
public:
```

```
void getdata(){ cout<<"Enter the number"; cin>>x; }
friend void display(test t);};
void display(test  t) { cout<<"I am getting pvt member:"<<t.x;}//calling friend function
void main(){
test tt;
tt.getdata();
display(tt);
getch();}
```

Friend class

The class which is able to access the private data member of another class is known as friend class.

```
#include<iostream.h>
#include<conio.h>
class alpha{
friend class beta;
private: char name [30];
int roll;
public:
void takedata(){cout<<"Enter nam e and roll";
cin>>name>>roll;} };
class beta{
public:
void display(alpha a){
cout<<"\nname is:"<<a.name;
cout<<"\nroll is:"<<a.roll;}};
void main(){
alpha a;
beta b;
a.takedata();
b.display(a);
getch(); }
```

Generic and template

A *template* describes a set of rules to classes or functions in which a list of parameters (types) are declared it is designed how these members classes are set out according to the declaration vary. The compiler generates new classes or functions when you supply these parameters; this process is called *template instantiation.* Therefore class or function definition generated from a template and a set of template parameters is called a *specialization* in the program.

The *declaration* is one of the following:

- a declaration or definition of a function or a class
- a definition of a member function or a member class of a class template
- a definition of a static data member of a class template
- a definition of a static data member of a class nested within a class template
- a definition of a member template of a class or class template

The *identifier* of a *type* is defined to be a *type_name* in the scope of the template declaration. A template declaration can appear as a namespace scope or class scope declaration.

Generic is the form of polymorphism. Generic provides a way of parameter to a class or a function by use of type like normal parameter to the functions and classes. Parameter provides a way of defining an abstract algorithm without specific value. In generic a variable can be define as type parameter although its property may not known. In description of class that unknown parameter is matched & used. In C++ generic programming is implemented by template so temples are used to create a family of classes or functions.

Templates are used to class or function in which different types of variables will be created in future. When an object of a specific type is defined or actual use, the template definition for that class or function is substituted with the required data type. The templates are parameters that can be replace by specified data type of the time of actual use of class or function. The temples are also called parameterized class.

Templates are a mechanism that makes it possible to use one function or class to handle many different data types.

Syntax:

```
 template<class T>
return type( targs1, targs2------,targs){
body of templet;;   }
```

Therefore template are the mechanism that make it possible to use one function or class to handle assign single c class or function so function that operate on data of many types instead of having to create separate classes or functions for each type. When used with function, they are called as function template where as when used with class they are called class template.

Function template:

Suppose we want to write a function that returns the sum of two variables accordingly so we just write function as

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

Here the function is defined to take arguments of integer type and written the value of the same type. What if we want to find the sum of float, long int and double type, then we need to again redefine another function for those types to handled those data structure . We should rewrite the same code for different data types are time consuming process. If we have an errors in a code then we need to checked the errors in each code functions, it would be more better if we can just write the one function and reuse it for all other types.

```
#include<iostream.h>
#include<conio.h>
template<class T>
T sum(T a,T b){
T add;
add=a+b;
return (add);}
void main(){
clrscr();
int i=10;
int j=20;
cout<<"\nsum is in int is:"<<sum(i,j);
float a=15.4,b=16.7;
cout<<"\nsum in float is:"<<sum(a,b);
getch();}
```

In the above code T is used for whenever data type is required. We can use any type of data after function called the compiler decides what type of arguments is used in the function and then return the value. When again compiler find the sum (a, b) then it will checked for in a and b data type then value returned from the function.

Function template with multiple parameters we can have more than generic data type in any template status like

Function temple with multiple parameters

template< class T1, class T2>

return type function name(T1 arg,T2 arg){ function body}

```
#include<iostream.h>
#include<conio.h>
template<class T, class S, class Z>
void display(T a, S b, Z z){
cout<<"\na:"<<a<<endl;
cout<<"\nb"<<b<<endl;
cout<<"\np"<<z;}
void main(){
clrscr();
int i=10;
float j=20.89;
float p=90.6;
display(i,j,p);
getch(); }

#include<iostream.h>
#include<conio.h>
template<class T>
T min(T a, T b){
return (a<b)? a:b;}
void main(){
int i=10;
 int j=20;
cout<<"\nmin in int is:"<<min(i,j);
float a=15.4,b=16.6;
cout<<"\nminimum in float:"<<min(a,b);
getch();}

#include<iostream.h>
#include<conio.h>
template <class T>
class stdinfo{
T value1,value2;//temples variables having temples type
public:
void takedata(){
cout<<"Enter the two number";
cin>>value1>>value2;}
T sum() {  //templet function
T value;//new templet variable
value=value1+value2; //adding values
return value;}};
void main(){ clrscr();
stdinfo <int>s1;//templet must behave as temp
stdinfo <float>s2;//temples must behave as float
s1.takedata();//display information
s2.takedata();
cout<<s1.sum()<<endl;
cout<<"the float sum is:"<<s2.sum();
getch(); }
```

Temple function to swap the content of two variables

```
#include<iostream.h>
#include<conio.h>
template<class T>
T swap(T &a,T &b){
T m; m=a;
a=b; b=m;
return a;}
void main(){
int i=10;  int j=20;
 swap(i,j);
 cout<<"After swap";
cout<<i<<"&"<<j;
getch(); }
```

Template class

When a temple used with classes, they are called template class. In template classes the actual type of data being manipulated will be specified as parameters when object of class are created.

Syntax:

```
template <class T>
class classname{    };
void main(){
classname <type> object name----;
----;   }
```

Create a template class stack to show push and pop operation on stack.

```
#include<iostream.h>
#include<conio.h>
const int max=10;
template <class T>
class stack{
 private:
 T stk[max];
 int top;
 public:
 stack(){  top=-1;  }
 void push(T data){
 if(top==max-1)
 cout<<"Stack is full";
 else{ top++;
 stk[top]=data; }}
 T pop(){
 if(top==-1){
 cout<<"Stack is empty";
 return NULL; }
 else {
 T data=stk[top];
 top--;
 return data; }}};
 void main(){ clrscr();
 stack<int>s1;
 s1.push(10);
 s1.push(20);
 s1.push(30);
 cout<<s1.pop();
 cout<<s1.pop();
 cout<<s1.pop();
 getch();  }
```

This stack has a disadvantage and that is int can store only integer values. If you need a stack to store float or character data, you have to define a second object for float data and third class for char data. If you could generalize the stack template concept for their data type.

If the class stack is generalized, you can use it in many programs. So it would be helpful if you create a header file with this class. The header files have an extension of .h. as your include iostream.h in any of your program your can also include the user defined header files of that programs. This is another example of code reasonability.

The following example illustrates how to create generalized stack without using templates.

```
template <class t>
```

Here the data type of stack required is donated by a variable name. In the program, this requires the stack header file. Stack file is including after defining what the variables stand for. This can be done by using typedeft statement.

```
#include<iostream.h>
#define max 10
class stackk{
private: D st[max];
int top;
public: stackk(){top=-1;}
void push(D var){ st[++top]=var;}
D pop(){ return(st[top--]);}};
```

In the above example a header file is created for the stackk class. When this file is to be included in a program, the name of the file has to be enclosed in double quotes. This signifies that the included header files is a user defined header file.

```
typedef int D;
#include "stackk.h"
#include<conio.h>
```

```
void main(){
int s;
stackk s1;
s1.push(50); s2.push(60); s3.push(70);
s=s1.pop();ss=s2.pop();sss=s3.pop();
cout<<s<<”,”<<ss<<”,”<<sss;
getch();}
```

In C++, you have a structure that defines such general classes. This structure is called template, or parameterized data types. Whew template is a key word, class indicates that a data type is the parameter and t is the name given instead of the data type for declaring the variables of the class. If you have any other arguments it can be specified by using comma as parameter. After this statement you give the class specifies using the data type as t. This template is stored in separate header files to promote code reusability.

```
#include<iostream.h>
template <class t, int max>
class stack{
public:
t st[max];
int top;
public:
stack(){top=-1;}
void push(t var){st[++top]=var;}
t pop(){ return (st[top--]);} };

#include"stack.h"
#include<conio.h>
void main(){
stack <int,3>s1;
stack <char,5>s2;
s1.push(50);
s2.push('a');
int x1=s1.pop();
char x2=s2.pop();
cout<<x1<<endl;
cout<<x2;
getch(); }
```

Write a class Template to represent a generic vector includes member functions to perform the following task;

a. To create a vector
b. To modify value of given element
c. To multiply scalar value
d. To display the vector

```
#include<vector.h>
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#define vector vector
template<class T>class vector{
T* v;
int size;
public:
 vector(int m);
 vector(T* a);
T operator*(vector& y);};
template<class T>
vector<T>::vector(int m){
v=new int[size=m];
for(unsigned i=0;i<=3;i++)
v[i]=0;}
```

```
   template<class T>
vector<T>::vector(T* a){
for(unsigned i=0;i<size;i++)
v[i]=a[i];}
template<class T>
vector<T>::operator*(vector& y){
T sum;
for(unsigned i=0;i<size;i++){
    sum=0;
     sum+=this->v[i]*y.v[i]; }
     return sum;}
void main(){
int x[3]={1,2,3};
int y[3]={4,5,6};
vector<int> v1(3);
vector<int> v2(3);
v1=x;
v2=y;
int r=v1*v2;
cout<<r;
getch();}
```

WAP using temple to add two integers, two float and one integer & one float respectively display the final result in float.

```
       #include<iostream.h>
       #include<conio.h>
       template <class t1, class t2>
       class add{
       t1 a; t2 b;
       public:
       add(t1 x,t2 y){ a=x;b=y;}
       void sum(){cout<<"sum of numbers"<<(a+b)<<endl; }};
       void main(){
       clrscr();
       add<int,int> obj1(22,23);
       add<float,float>obj2(12.8,12.8);
       add<int,float>obj3(22,23.4);
       obj1.sum();
       obj2.sum();
       obj3.sum();
       getch();}
```

Exception handling

Exception handling is a mechanism that that detects and separate the code then handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions
- Exception specifications (C++ only)

It is the latest features added to ANSI C++. It refers to unusual conditions in a program. They could be errors that casus the program to fail or certain conditions that lead to errors.

There are two kinds of exception

i. Synchronous exception eg. Out of range index overflow

ii. Asynchronous exception errors that are caused by events beyond the control of program are called asynchronous exception.

The purpose of exception handling mechanism is to provide means to detect and report an exceptional circumstance so that appropriate action can be taken

This mechanism includes following task.

1. Find the problem(hit the exception)
2. Inform that an error has occurred(throw the exception)
3. Receive the error information(catch the exception)
4. Take corrective action(handled the exceptions)

Syntax

```
try{
throws exception }
catch(args  ){
}
```

```
#include<iostream.h>
#include<conio.h>
void main (){
int f,s;
clrscr();
cout<<"Enter two numbers";
cin>>f>>s;
try{
if(s!=0){
cout<<"Division/quotient";
cout<<f/s;}
else throw(s);}
catch(int s){
cout<<"Errors has  found";}
getch();}
```

```
#include<iostream.h>
#include<conio.h>
void main(){
int a;int b;
double d;
try{
cout<<"Enter first numemerator";
cin>>a;
cout<<"Enter second denumerator";
cin>>b;
if(a<=0) throw a;
d=a/b;
cout<<"if numerator  is larger than 0"<<d;}
catch( int e){
cout<<"Exception occur because numerator is less than 0";}
getch();}
```

```
#include<iostream.h>
#include<conio.h>
class dividebyzero
{ };
double safedivide(int top,int bottom) throw(dividebyzero);
void main(){
int numera;
int denom;
double quotient;
cout<<"Enter numerator";
cin>>numera;
```

```
        cout<<"Enter denomainator";
        cin>>denom;
        try{
        quotient=safedivide(numera, denom);
        cout<<quotient; }
        catch(dividebyzero){cout<<"Errors";  }
        cout<<numera<<"/"<<denom ;
        getch(); }
        double safedivide(int top,int bottom) throw(dividebyzero){
        if (bottom==0)  throw dividebyzero();
        return top/bottom;}
```

Standard Template Library (STL)

We already saw templates are used to crate generic classes and functions that could support for generic programming in order to help the users in programming. It set out general purpose temptalized classes (Data structure) and functions that could be used as a standard approach for storing and processing of data. The collection of those classes and function is called standard template library.

STL is very large and complex concept its important features are. It helps in saving time, efforts, load top high quality programming because STL provides well written and tested components which can be reuse in our programming to make our program more robustness. STL contains several components but its core contains three components they are 1 Container 2 Algorithm 3 Iterators

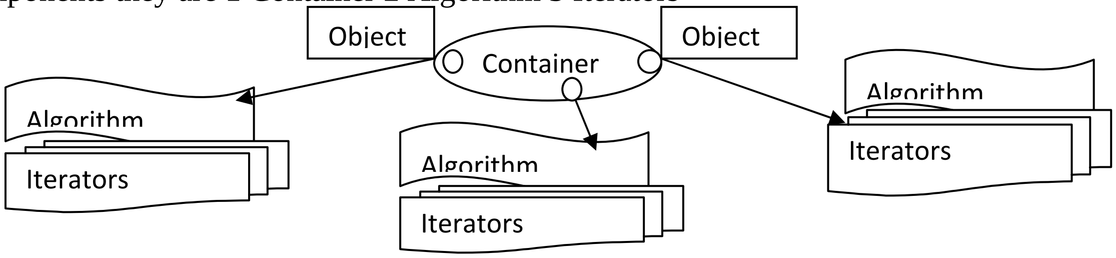


Fig: Relationship between STL and Components

Container: The container may content single or multiple objects that store the data. The STL containers are implemented by template, classes and procedure that is used to process a data type.

There are some inbuilt functions inside container class

container()	empty container
container(n)	n elements/default value
container(n,x)	n copies of x
Container (first ,last)	Initial elements form
~container()	destroy the container

Algorithm: It is a procedure that is used to process a data contain in the container STL includes different kind of algorithm to support different task like initializing, searching copying and sorting. Algorithm is implemented by template.

Iterates: An iterates is an object that can be used with a container to gain access to particular elements in the container. It is generalization of the notion of a pointer that points to elements in a container. Iterates are used to move through the containers. Iterates connects algorithm with containers.

There are some inbuilt functions inside Iterates

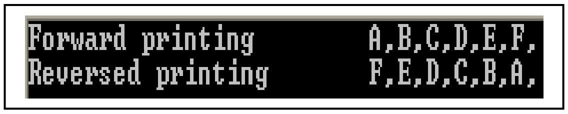
begins()	Points to first elements
end()	Points to last elements
rbegin()	Points to first elements of reverse sequence
rend()	Points to last elements of reverse sequence

Eg

iterator p;

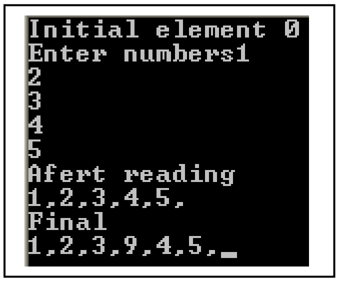
for(p=container.beginb(); p!=container.end(); p++)

process_element p;



```
Forward printing        A,B,C,D,E,F,
Reversed printing       F,E,D,C,B,A,
#include<iostream.h>
#include<vector.h>
#include<conio.h>
using std:: vector;
using std::vector<char>::iterator;
using std::vector<char>::reverse_iterator;
void main(){
vector<char>container;
container.push_back('A');
container.push_back('B');
container.push_back('C');
container.push_back('D');
container.push_back('E');
container.push_back('F');
cout<<"Forward printing\t";
iterator p;
for(p=container.begin();p!=container.end();p++)
cout<<*p<<",";
cout<<endl;
cout<<"Reversed printing\t";
reverse_iterator rp;
for(rp=container.rbegin(); rp!=container.rend();rp++)
cout<<*rp<<",";
cout<<endl;
getch();}
```

```
#include<iostream.h>
#include<vector.h>
#include<conio.h>
using std:: vector;
using std::vector<int>::iterator;
using std::vector<int>::reverse_iterator;
void display(vector<int>&v){
for(int i=0;i<v.size();i++){
cout<<v[i]<<",";}}
void main(){
vector<int>v;
cout<<"Initial element\t"<<v.size();
int x;
cout<<"\nEnter numbers";
for(int i=0;i<5;i++){
cin>>x;
v.push_back(x);}
cout<<"Afert reading "<<endl;
display(v);
vector<int>::iterator itr=v.begin();
itr=itr+3;
v.insert(itr,1,9);
cout<<"\nFinal"<<endl;
display(v);
getch();}
```



```
Initial element 0
Enter numbers1
2
3
4
5
Afert reading
1,2,3,4,5,
Final
1,2,3,9,4,5,_
```

Is it possible to used different editor for single program in C++? Give an example to display the some character of ‘H’E’L’L’O’   to display in reversed OLLEH by using stack?

Yes write this code in separate editor and save it as pu.h file extension then compiled this code

```
#include<iostream.h>
class pu{
char faculty[30];
int top;
public: pu(){top=-1;}
void push( char vari){ faculty[++top]=vari;}
char pop(){
return faculty[top--];}};
```

Again write this code in different editor and compiled and run

```
#include"pu.h"
#include<iostream.h>
#include<conio.h>
void main(){
pu p;
p.push('H');
p.push('E');
p.push('L');
p.push('L');
p.push('O');
cout<<p.pop();
cout<<p.pop();
cout<<p.pop();
cout<<p.pop();
cout<<p.pop();
getch();}
```

Similar type code reusability

```
#include<vector.h>
#include<iostream.h>
#include<conio.h>
using namespace std;
void main(){
vector<int>v;
cout<<"Enter numbers";
int next;
cin>>next;
while(next>0){
v.push_back(next);
cout<<next<<"added";
cout<<"v.size()"<<v.size();
cin>>next;}
for(unsigned i=0;i<v.size();i++)
cout<<v[i]<<",";
getch();}
```