Evaluation:

	Theory	Practical	Total
Sessional	30	20	50
Final	50	-	50
Total	80	20	100

Course Objectives:

Object-oriented programming is an approach to thinking about computation and problem solving in Object Oriented Paradigms. This course lays out the principles of object-oriented programming in C++ Programming Language. The Course helps the target audience to discover the basic concepts of object-oriented programming in C++. The basic objective behind this course is:

- To introduce the fundamentals of C++ programming
- To present the defensive programming style required by the C/C++ programming language
- To explore the facilities offered by C++ for object-oriented programming.

Course Contents:

1. Introduction to Object Oriented Paradigm (8 Hrs.)

Object-Oriented Programming as a New Paradigm, A Way of Viewing the World-Agents, Computation as Simulation, Coping with Complexity: The Nonlinear Behavior of Complexity: Abstraction Mechanisms, reusable Software. Object-Oriented Design: Responsibility Implies Noninterference, Programming in the Small and Programming in the Large, Role of Behavior in OOP, Case Study: Responsibility-Driven Design, CRC Cards, Components and Behavior, Software Components, Formalize the Interface, Design the Representation for Components, Implementing Components, Integration of Components, Maintenance and Evolution

2. Classes, Methods and Messages (8 Hrs.)

Encapsulation, Varieties of Classes, Interface and Implementation, Classes and Methods in C++, Message-Passing Formalism, Message Passing Syntax in C++, Issues in Creation and Initialization; Stack Versus Heap Storage Allocation, Memory recovery, Pointers; Mechanisms for Creation and Initialization in C++, Case Study: The Eight Queen Puzzle in C++.

3. Inheritance and Software Reusability (7 Hrs.)

Introduction to Inheritance, Subclass, Subtype, and Substitutability; Forms of Inheritance, Inheritance in C++, Inheritance- Its merit and demerits, Inheritance and Substitutability, The *is-a* rule and the *has-a* rule, Composition and Inheritance, Software reusability

4. Polymorphism (7 Hrs.)

Polymorphism in Programming languages, Varieties of Polymorphism, Polymorphic Variables, Overloading, Overriding, Deferred Methods, Pure Polymorphism, Generic and Templates, Polymorphism in C++ and, Case Study: Container Classes and The Standard Template Library

Laboratory Work

There shall be 20 exercises in minimum, as decided by the faculty. The exercises shall encompass a broad spectrum of real-life and scientific problems, development of small program to the development of fairly complex subroutines, programs for engineering applications and problem solving situations. Laboratory assignments will be offered in groups of two to four for evaluation purpose. In general, the Laboratory Work must cover assignments and exercises from the following areas:

1. Data types – control structures, functions and scoping rules.
2. Composite data types, C++ strings, use of " Constant " keyword, pointers and references
3. Classes and data abstraction
4. Inheritance, abstract classes and multiple inheritance
5. Friend functions, friend classes and operator overloading.
6. Static class members
7. Polymorphism, early binding and late binding
8. C++ type conversion
9. Exception handling
10. Function templates, class templates and container classes.

Textbooks:

1. R. Lafore, *Object Oriented Programming in Turbo C++*, Galgotia Publications Ltd. India, 1999
2. Budd, T., *an Introduction to Object Oriented Programming*, Second Edition, Addison-Wesley, Pearson Education Asia, ISBN: 81-7808-228-4.
3. Savitch, W, *Problem Solving with C++ - The Object of Programming*, 2nd Edition, Addison – Wesley, Pearson Education Asia (LPE), 2001, ISBN: 81-7808-173-3

Reference Books:

1. Lippman, S.B., Lahore. J., *C++ Primer*, 3rd Ed., Addison Wesley, 1998
2. Eckel, B., *Thinking in C++*, 2nd Edition, Prentice Hall, 2000

A Survey of Programming Techniques

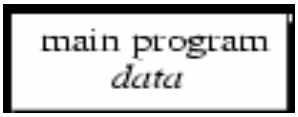
Pokhara University (PU) revised

This chapter is a short survey of programming techniques, generally used to design a program. We use a simple example to illustrate the particular properties and to point out their main ideas. Roughly speaking, we can distinguish the following learning curve of someone who learns to program are:

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

1.1 Unstructured Programming

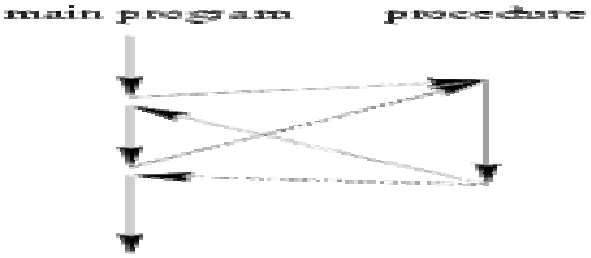
Usually, people start learning programming by writing small and simple programs consisting only one main program. Here "main program" stands for a list of commands or *statements* which modify data throughout the whole program. We can illustrate this as shown in **Figure**. The main program directly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

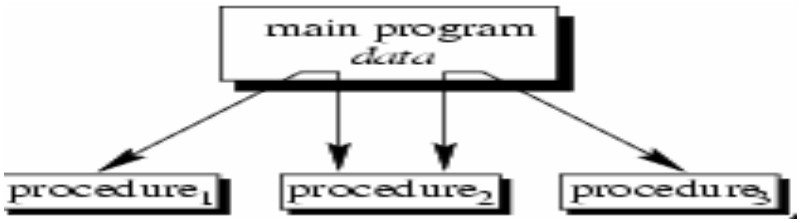
1.2 Procedural Programming

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

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



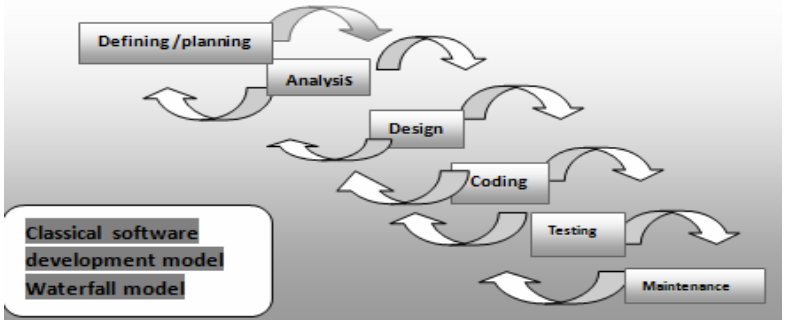
Now we have a single program which is divided into small pieces called procedures. To enable usage of general procedures or groups of procedures also in other programs, they must be separately available. For that reason, modular programming allows grouping of procedures into modules. Therefore, a program 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 are

- Emphasis is one doing things (algorithm).
- Large programs are divided into smaller program known as function/module.
- 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 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.

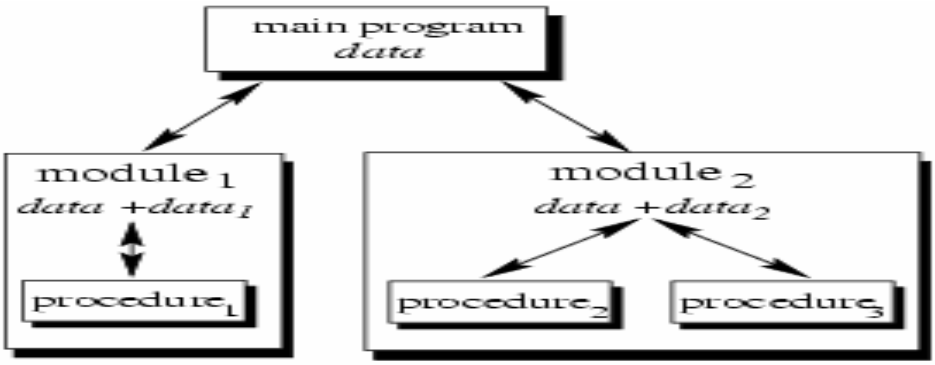
Testing: Once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole system. It requires a detailed plans as to what, when and how to test.

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 decomposition 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

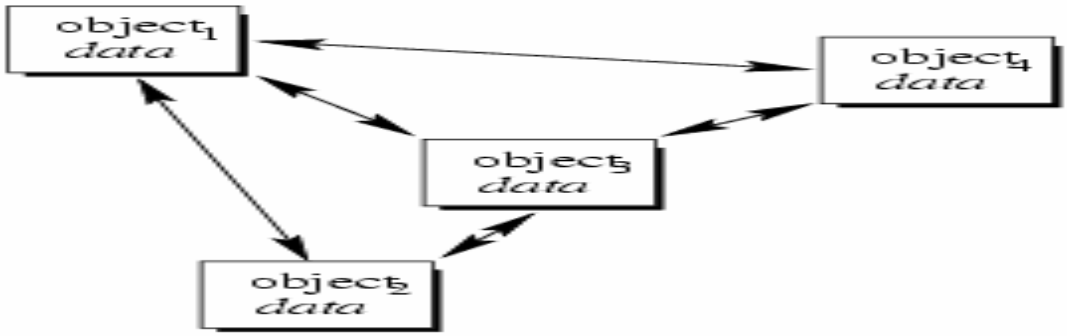
With modular programming procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program in as Figure bellow.
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* as **Figure.** 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, 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. 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:

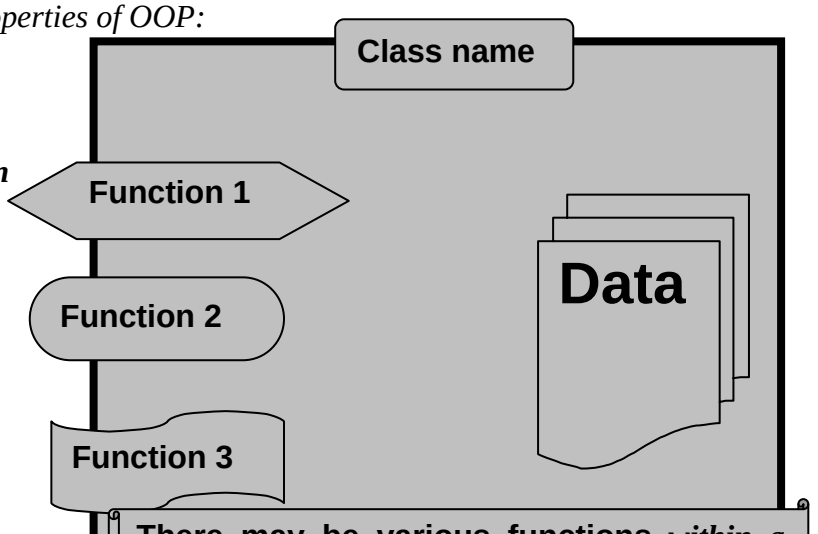
- Emphasis is on data rather than procedure.
- Programs are divided into what are known as object.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in data structure.
- Data is hidden and cannot be access by external function (increase security level).
- 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. The object oriented model work together closely with various objects. In one phase, the problem domain objects are identified. 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***

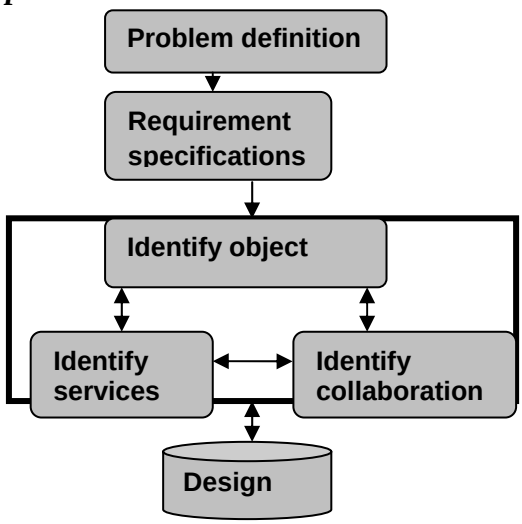


Steps in object oriented analysis

- *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 and behaviors.*
- *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 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. The cat object 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 class is divided into sub class as our previous example. Animal class can be divided into sub class like water and land animal. Land animal can inherit some or all properties of higher animal and the cat class 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 by 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 behaviors depend upon data types used in the 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 cannot 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 code redundant (repetition).
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 (window and outlooks).
- Real business applications because it supports inheritance and polymorphism
- Real time system application.
- 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.

Difference between Procedural Programming and Object Oriented Programming

Object Oriented Programming	**Procedural Oriented Programming**
It gives emphasis on data rather than procedure	It gives more emphasis to procedure only
We can declared local as well as global variable	We can declare global variable only.
Designing approaches is button to top	It is just as collection of procedure list to complete task.
Data can be hidden / secret	There is no any type of protection
Oops program can be divided into various objects	Procedural programming divide into various procedures.
Encapsulation, inheritance and reusability are	Encapsulation, inheritance, polymorphism are not

common features.	available.
We can add programming functions and objects so that objects can communicate with one another	To design program long we should create function first then called inside main procedure
New data can be added easily as requirement increase	New data cannot easily be added
We can defined real world problem into various small units/ subunits	The traditional type way of programming.
Mostly software designer used	Less useful in design large software
Java, C++, C≠, Oracle, PHP,	FORTRAN, C, Basic

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, and Tata Mc Geaw Hill
Timothy Budd, 2009: An Introduction to Object Oriented Programming, Delhi, Anand Sons
Berne 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

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

Goto,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 go to 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. The compiler creates filename.obj which links filename.exe on that platform then run by pressing CTRL+F9 or directly goes to run menu.

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

```
#include<iostream.h>
#include<conio.h>
class info{ // class declaration
int a; char s1[34];                    // member data(properties)
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
```

```
Enter your First nameBimal
Enter your rollno34
Your name is:::Bimal
Your Rollno is:::34
```

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 friend 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 (i.e. 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 responsible person to deliver my message to my friend in Germany. He might use some methods and algorithm to deliver that information without any change. Therefore the first florist established the communication to the florist in the Germany. That florist must have to manage 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.

In conclusion

- Object oriented problem can be solved with the help of many other individuals/objects.
- Without 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. If so we can communicate whenever we need some task as an object. 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 problem. 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 to computer)

The way of programming in C++ is to create **main method.** This is the many ways similar to the computer simulation called event driven style. If so we know computer takes data and instruction through keyboard then those data and instructions are processed in sequential manner. Through this style we can design the methods inside the class to fulfill our requirement. The designer has fully rights to design the various elements called objects. Finally those objects established the relationship to one another objects. Thus object oriented programming style is similar to computation as simulation.

There are some procedures

- Analysis the real world problem with various dimension/elements.
- Analysis the problem with its related objects.

- Analysis the objects according to their characteristics.
- Develop and establish the relationship.
- Developed main method.
- Develop the algorithm.
- Design coding.
- Develop project first then software.

Coping with Complexity

The needs and requirements of 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 objects behavior. 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 problem 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, and 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 various smaller modules single processor computer architecture then the designer often consider the use of multiprocessor system. Those decisions are often feed with multiprocessors have to get well-managed to overcome. Similarly fir the large code, the cost of parallel software development process can easily solved, but in needs additional hardware on which the code is intended to run. Therefore programmers often have given sense of the complexity of a language as well as problem. However, programmers typically have experience with only a subset of the parallel languages, a comprehensive comparison cannot be based on only opinion designer but also the device developed and advance software innovations. The relationship between complexity and performance are pros and cons, because if we increase the complexity of the problem, it increases time, money, resources and additional hardware too.

For Reference Reading

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 and many more. In this manner, we find in certain information has been included and certain information has been omitted in every level of abstraction. 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

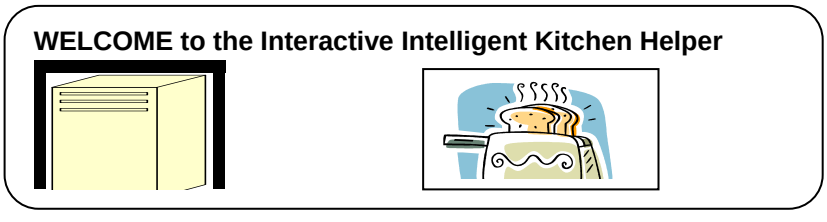
The typical program written in object oriented style give significant features level of abstraction. The higher levels of abstraction provide the detail parts of the program. This can be analyzed by searching more related object its nature of interaction to one another. The highest level of abstraction 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. It is concluded that the more the level you search the more the detail information you will get. There are some points to be considered while software analysis and design as bellow.
Division into parts: The most common techniques use to solve the complex problem is 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 problem can be easily solved. The most benefit of encapsulation is that it permits us to consider the possible interchangeability from one class or object to other objects.
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. It has limited used.
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. It can solve large problem and complete task.

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 various concern objects should have to analyze before designing the software system. It is impossible to design good software without analyzing each and every component and their behaviors of interaction apply in the real situation.
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 that you are software manager in a big company. One day your boss walks into your office with an idea that he want to develop kitchen helper i.e. (**Interacting Intelligent Kitchen Helper**). So being the manager you should have to design all the requirement of software components 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 clarifying 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 and manage desired data from huge data item.

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 add 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 and works inside kitchen.

CRC Cards (Components, Responsibility and Collaborator)

Components: *A component is 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 card/document that describes many related information of an individual's (name, address, id, organization name issue date etc.). Similarly CRC card is another type of card that describes name (components), responsibilities and collaborators of particular items which is used during the process of software 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 place in identified and assigned to some component as a responsibility.

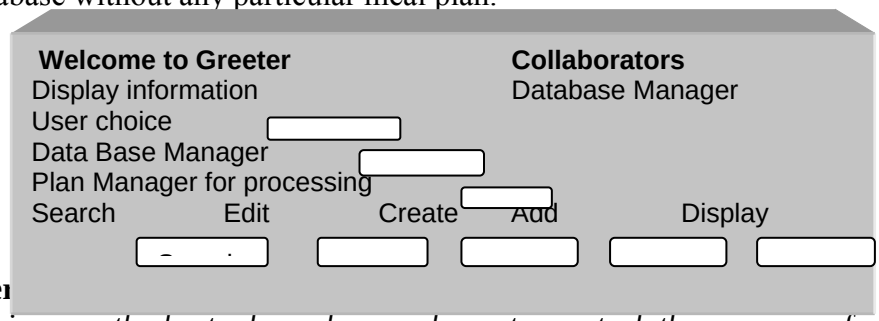
Components Name (Department store) (Description of responsibilities)	**Collaborators**
Name: **Id:** **Quantity:** **Order date:** **Delivery date:**	**Order** (Other Related Components)

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

When the system begins, we should further decide to add some interactive window before released the software. The responsibility for displaying those components inside the main method is called composition. The selective tools may design to communicate inner other components. So the developer again realized the following action to be put down on kitchen helper interactive software. There are some important interactive tools to be added. 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 (KHI)
software may be like



Software Components

A set of instructions written for computer

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 is very general pre-requirements. However there are various paradigms or set of methods and tools but, the selection of particular paradigm depends on the nature of application programming language used and requirement and so on. Good successful software must include:

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

But there are some software components tools used in software development process

Behavior and state

We know that object oriented programming has given more importance to the creation of small components; these components certainly hold some information that should be fulfilled later. 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. Those set of requirements will be found only when we search more information on the problem. The complete description of all behaviors for a component is sometimes called the protocol/rules. 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 instruction and perform in a similar manner.

Coupling and cohesion

Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component on the basic of similar type. In software design, the component has to achieve high cohesion. So that we get more strong and large component.

Coupling describes the relationship between software components. While software component development process begin the coupling should have to reduce because if the relationship between components are more. There will be very difficult to edit or modify and integrated into single unit. 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. Therefore, the components should not have same value, nature and behaviors.

Interface and Implementation

In software design, large requirement is divided into small components and those components are assigned to different programmer to design component may be related with each other therefore the programmer have

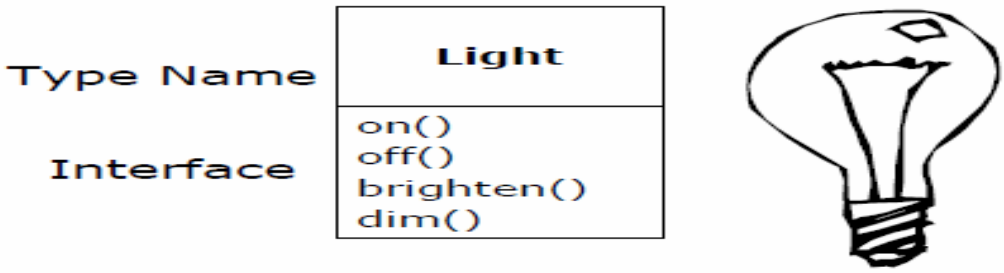
possibility to know how to combine another component developed by other programmer. But it is not necessary to know how the component is implemented/execute/does the task.
Let's have an e.g. of account management software of a college which is divided into various components. Suppose there is fee data base components which keep the record of all fee structure of all faculties the next component may be faculty of civil engineering, which manage all fees of civil engineering department and so on. Here the programmer of civil engineering component interface with fee database but no need to know detail description of fee database components fulfills the task. The separation of interface & implementation are most important concept in software engineering. It also describes the concept of information hiding in software design. Interface and implementation leads two different views of the system software. The interface view is outlook view when combine all modules. The interface view describes what a software component can perform. An implementation view is the face seen by the programmer working within particular component. It describes how the component completes its task.

Formalize the Interface
Formalize the interface is the process of design the components in which the patterns and channels of communication are formalized. The design components interaction decision should make properly so that it will be used to implement each other components. 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 into uppercase is a function but components with more task and more behaviors are implemented by the class notion. The components must have named & responsibilities that must be done by procedure/ function. The required database types of arguments are passed into the function. Each component requires data, those data are passed through argument or global variables are maintained by function. The data type must be identified clearly, which is used in component/function.
Integration of Components
Once software components 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 component 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 software.
Maintenance and Evolution
After completing the design of software system, we delivered the product in to the market, there may be some requirements will be added like changing the operating 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 users' requirements. The varieties of activities fall into software maintenance and evolution that are listed below:

- Errors or bugs can be occurred in while delivering the product. These must be corrected and upgraded.
- Requirement may change according the change in rules and regulation of government/norms etc.
- Hardware may change. If we change any change in hardware the simultaneous change in hardware is required or vice versa.
- User demand may change (The user may demand greater functionality, with lower cost and user friendly tools etc).
- Better documentation may be required by user (help).

An Object has an Interface
Aristotle was probably the first to begin a careful study of the concept of type; he spoke of "the class of fishes and the class of birds." The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword class that introduces a new type into a program. Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have a bunch of tellers, customers, accounts, transactions, and units of money—a

lot of "objects." Objects which are identical except for their state during a program's execution are grouped together into "classes of objects" and that's where the keyword class came from. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You can create variables of a type (called objects or instances in object-oriented parlance) and manipulate those variables (called sending messages or requests; you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state: each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors. So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versaSince a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system welcomes the new classes and gives them all the care and type-checking that it gives to built-in types. The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution. Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of thechallenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space.But how do you get an object to do useful work for you? There must be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch. And each object can satisfy only certain requests. The requests youcan make of an object are defined by its interface, and the type is what determines the interface. A simple example might be a representation of a light bulb:
Light




```
Light lt = new Light();
lt.on();
```

The interface establishes what requests you can make for a particular object. However, there must be code somewhere to satisfy that request.This, along with the hidden data, comprises the implementation. From a procedural programming standpoint, it's not that complicated. A type has a method associated with each possible request, and when you make a particular request to an object, that method is called. This process is usually summarized by saying that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code). Here, the name of the type/class is Light, the name of this particular Light object is lt, and the requests that you can make of a Light object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a Light object by defining a "reference" (lt) for that object and calling new to request a new object of that type. To send a message to the object, you

state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

```
#include<iostream.h>
#include<conio.h>
class light{
public: void on(){cout<<"\nlight on";}
void dim(){cout<<"\nlight dim";}
void off(){cout<<"\nlight out";}
void briten(){cout<<"\nlight briten";} };
void main(){
light *lt=new light();
lt->on();  lt->off();  lt->dim();  lt->briten();
getch();}
```

Character Set

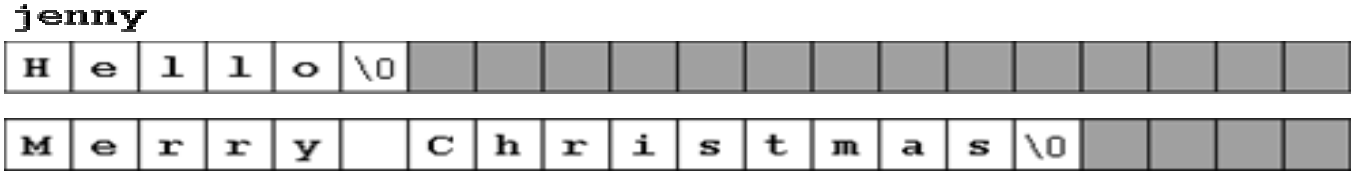
Obviously the set of legal characters of C++ can contribute a legal word in C++ . The set of such characters refers as character set of C++. The characters are grouped as follows

Letters Digits Special characters Whitespace Alphabets a..z A..Z Numerals 1..9.

Character Sequences

As you may already know, the C++ Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements. For example, the following array: char jenny [20]; is an array that can store up to 20 elements of type char. It can be represented as:

Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, jenny could store at some point in a program either the sequence "Hello" or the sequence "Merry Christmas", since both are shorter than 20 characters. Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the null character, whose literal constant can be written as '\0' (backslash, zero). Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and "Merry Christmas" as:



Notice how after the valid content a null character ('\0') has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array: char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' }; In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end. But arrays of char elements have an additional method to initialize their values: using string literals. In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example: "the result is: " is a constant string literal that we have probably used already. Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0')

automatically appended at the end. Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword [] = "Hello";

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically. Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation. Assuming mystext is a char[] variable, expressions within a source code like**: mystext = "Hello"; mystext[] = "Hello"; would not be valid, like neither would be: mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };** The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (char[]) and can also be used in most cases. For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

```
#include <iostream>
#include<conio.h>
int main (){
char question[] = "Please, enter your first name: ";
char greeting[] = "Hello, ";
char yourname [80];
cout << question; cin >> yourname;
cout << greeting << yourname << "!";
getch();return 0;}
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars. Finally, sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:

```
string mystring;
char myntcs[]="some text";
mystring = myntcs;
```

Special characters: , comma . period :colon ; semicolon ‘apostrophe ? question marks “” quotation marks ! exclamation marks / slash \back slash ~ tilde % percentage # number sign () parenthesis [] square bracket {} brackets ^ catet

White Space: Blank space, horizontal tab, carriage return, new line.

Keywords

The set of C++ words that can be formed form these characters can be classified as keywords. The keywords which have a predefined meaning and cannot be changed. Al key words must be written in lower case. Auto do for return type deft break double got start union case else if size of char enum, int static void continue long struct while default register switch.

Variables

Variables enables the programmer to assign and manipulates data using symbolic names. Strings and numeric values can be stored in the memory of computer for subsequent recall, whenever the memory is used for this purpose the programmer must assigned a unique name to each area in memory. In brief, a variable is a data name that may be used to store a data value for easy retrievals. Tot salary, counter stdname.

The naming conventions of variables has already been discussed. The maximum length of variable is machine dependent. Usually 32 is maximum number of characters permissible in variable name.

Data Types

Data type is defining an attribute to the variable. It defines the set of legal that the variable can store. C++ is rich data types. The basic data types are int char and float.

Integer: The data type int are whole numbers with range of value supported a particular machine for instance, in a16 bit word length matching integer values lies in between -32768 to 32 767. C++ facilitates some control over the integer data type by providing sub data types namely short int int , long int.

Short int represents fairly small integer values and requires half the amount of storage as a normal int uses.

Similarly a long int represent fairly higher integer values and requires generally twice the number of bits as a normal int uses. Nevertheless these are highly machine dependent.

Another option in integer data type is declaring it as unsigned. This unsigned integer uses all bits for the magnitude of the numbers and are always positive. For instance in a 16 bit machine the range of using integer is 0 to 65,535. thus long and unsigned are intended for increasing the range of values.

Floating Point Type: Floating numbers are numbers that have a decimal point. This data type in C++, is an attribute for real numbers. The corresponding declaration is float a;

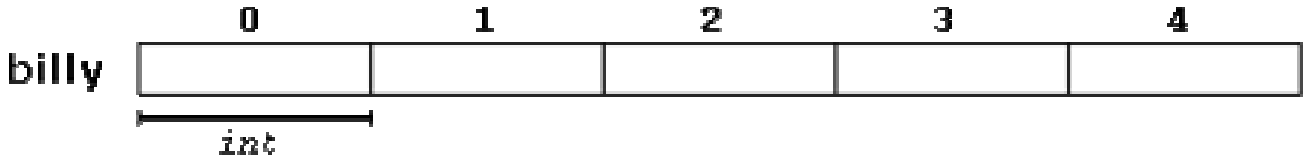
Which instruct the compiler that the value a belongs to the data type real. If you want to initialize the variable then float a; a=24.9; Or we can done directly float a=24.9;

When more accurately is required, another sub type double under the float data type can be used. Obviously this double data type uses twice the storage as that of the float data type. To display a double value the format specifies %f is sued to obtain the standard floating point notion and %e is for scientific exponential notion.

Integer	**Floating**	**Char type**
Int 2 bytes (32768 to 32767)	Float 4 bytes (3.4 E- -38 to 3.4E+ 38)	Char 1 byte (-128- 127)
Short int 2 bytes(31768 to 32767)	Double 8 (1.7 E- 308 to 1.7E 308)	Unsigned char 1(0 to 255)
Long int 4 bytes(214783648 to 2147483647)	Long double 10 bytes(3.4 E-4932 to 1.1E +4932)	Signed char 1() -128-127
Unsigned int 2(-31768 to 32767)		
Unsigned short int 2 (-31768 to 32767)		
Signed short int 2(-32768 to 32767)		
Singed long int 4(0 to 429467295)		

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier. For example, an array to contain 5 integer values of type int called billy could be Represented like this:



Where each blank panel represents an element of the array, that in this case are integer values of type int. These elements are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length. Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is: type name [elements]; where type is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain. Therefore, in order to declare an array called billy as the one shown in the above diagram it is as simple as: int billy [5]; **NOTE**: The elements field within brackets [] which represents the number of elements the array is going to hold, must be a **constant** value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

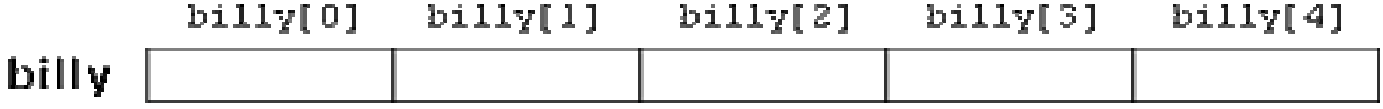
Initializing Arrays

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros. In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces { }. For example: int billy [5] = {16, 2, 77, 40, 12071}; this declaration would have created an array like this:

The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array billy we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element. When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }: int billy [] = {16, 2, 77, 40, 12071}; after this declaration, array billy would be 5 int long, since we have provided 5 initialization values.

Accessing the values of an array

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as: name [index]. Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:



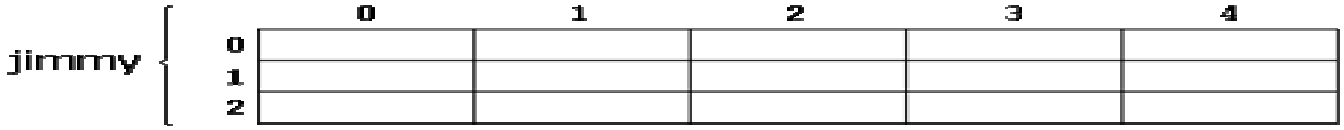
For example, to store the value 75 in the third element of billy, we could write the following statement: billy[2] = 75; and, for example, to pass the value of the third element of billy to a variable called a, we could write: a = billy[2]; Therefore, the expression billy[2] is for all purposes like a variable of type int. Notice that the third element of billy is specified billy[2], since the first one is billy[0], the second one is billy[1], and therefore, the third one is billy[2]. By this same reason, its last element is billy [4]. Therefore, if we write billy [5], we would be accessing the sixth element of billy and therefore exceeding the size of the array. In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers. At this point it is important to be able to clearly distinguish between the two uses those brackets [ ] have

related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays.
int billy[5]; // declaration of a new array
billy [2] = 75; // access to an element of the array.
If you read carefully, you will see that a type specifies always precedes a variable or array declaration, while it never precedes an access. Some other valid operations with arrays:

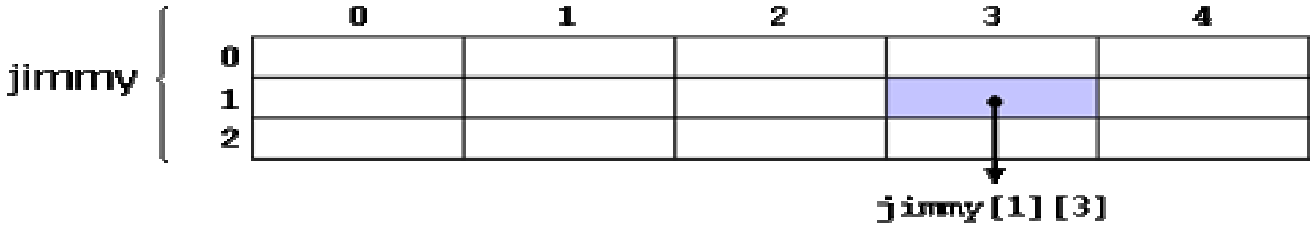
```
billy[0] = a;  billy[a] = 75;  b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
#include <iostream>
int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;
int main (){
for ( n=0 ; n<5 ; n++ )  {    result += billy[n];  }
cout << result;
return 0;} 12206
```

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bid-mensional array can be imagined as a bi-dimensional table made of elements, all of them of a same uniform data type.



Jimmy represents a bi-dimensional array of 3 per 5 elements of type int. The way to declare this array in C++ would be: int jimmy [3] [5]; and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be: jimmy [1] [3]



(Remember that array indices always begin by zero).
Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example: char century [100][365][24][60][60]; declares an array with a char element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 Gigabytes of memory!
Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:
int jimmy [3][5]; // is equivalent to
int jimmy [15]; // (3 * 5 = 15)
With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bi-dimensional array and the other one uses a simple array:

Multidimensional array	Pseudo-multidimensional array
#define WIDTH 5 **#define HEIGHT 3** **int jimmy [HEIGHT][WIDTH];** **int n,m;**	**#define WIDTH 5** **#define HEIGHT 3** **int jimmy [HEIGHT * WIDTH];** **int n,m;**

```
int main (){
for (n=0;n<HEIGHT;n++)
for (m=0;m<WIDTH;m++){
jimmy[n][m]=(n+1)*(m+1);}
return 0;}
```

```
int main (){
for (n=0;n<HEIGHT;n++)
for (m=0;m<WIDTH;m++){
jimmy[n*WIDTH+m]=(n+1)*(m+1);}
return 0;}
```

None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

We have used "defined constants" (#define) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line: with no need to make any other modifications to the program.

Array as Parameter

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function: void procedure (int arg[]) accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as: int myarray [40]; it would be enough to write a call like this: procedure (myarray); Here you have a complete example:

```
#include <iostream>
void printarray (int arg[], int length) {
for (int n=0; n<length; n++)    cout << arg[n] << " ";  cout << "\n";}
int main (){
int firstarray[] = {5, 10, 15};
int secondarray[] = {2, 4, 6, 8, 10};
printarray (firstarray,3);
printarray (secondarray,5);
return 0;} Outputs:
                510 15
                2 4 6 8 10
```

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range. In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is: base_type [][depth][depth] for example, a function with a multidimensional array as argument could be: void procedure (int myarray[][3][4]). Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension. Arrays, both simple and multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate.

Function in C++

We know that object oriented programming is based on the principle of dividing the whole 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.

Syntax: void main () {
…..;//statements ……; }

The functions that have any return type (int, float, and char) 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.

Syntax: 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 do add (int a, int b); // It says that do add is the function that return floating values after getting a and b in integer type in function call**.**

The function can be called in a program as: cube=do add (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

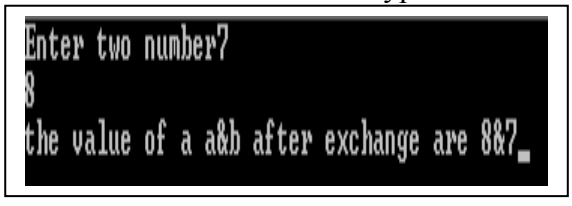
```
#include<iostream.h> //pre processor
#include<conio.h>
void square();// function declaration
      void main(){clrscr(); // main method
      square(); //function calling in main method
      getch(); }// close main method
void square(){// function design start here
int p; cout<<"enter the values";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

Here void square() is predefined function declared just below the preprocessors. It is called inside main method and designed its functionality outside the void main class.

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
```

Here function swap(int , int) is declared just below the preprocessor having two data type as an arguments is called inside main method. The functionality swap is deigned outside the main method using temp variables to hold intermediate results.

Inline Function

As we know function saves memory space because same code can be called 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 particular function. At the end of function, it jumps back to the calling program. This sequence of events save memory space but it needs extra time to complete its tasks. To save execution time function the code in function body directly inline with the code in calling program. When the function is called, the actual code execution starts. 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;                      // float variables declaration
avg=average(10.5,20.6);                 // function calling and passing values
cout<<"The average is"<<avg;
getch();}                       //main method close
```

```
The average is15.55
```

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time. The format for its declaration is:

inline type name (arguments ...) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Recursive Function

We know various types of control structure may execute automatically till the user defined condition become true. The function is the block of statement 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 function is said to be recursive function. Recursion is a process by which a function calls itself repeatedly until some specified condition become true. Recursively is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be: n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be: 5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
3 to power0is:1
3 to power1is:3
```

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() {                  clrscr();//main
for(int n=0;n<=5;n++)//for loop
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); }
```

In above code int power(int ,int) is function declared at header and called inside main method. The calculation is done outside if the something power 0 is impossible to calculate power. Else the calculation of power goes up to n-1 times while doing so the base is constant and n is increased y to loop times. When first loop jumps to power(3,0) that it directly jumps to calculation if possible it returns power else in such case if n is not up to n-1 it again repeat inside return power process. Then finally return value .

WAP program to demonstrate factorial of given number by using 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_
```

```
#include <iostream>
#include<conio.h>
long factorial (long a){
if (a > 1)    return (a * factorial (a-1));
else     return (1);}
int main (){    long number;
cout << "Please type a number: ";   cin >> number;
cout << number << "! = " << factorial (number); getch();
return 0;}
```

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

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

In this process void vertical (int) function receive the number as integer type then it again transfer through vertical(p) form main method to outside function. This function again checked if value from user is less than 10 or not if value is grater than then that value is divided by 10 produce only leading digits than -1, again this process repeats up to less than 10 then cout 1 then again this process repeats the remainder number

Encapsulation

Encapsulation is most important feature of object oriented programming language. Warping data and function 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 ideas that support abstraction in various levels. Data abstraction means hiding and searching information from outside world. In class member variable and function can be declared as private so that they cannot access outside the class. Therefore data become more secured and protected from other users.

The variables which are declared inside class as private are also called instance variable because these variables can only change by methods of same class.

Access Specifies for Visibility

We know OOPs can be applicable in large varieties of object. Objects are various types because they have various type and nature. To handle those properties, C++ supports 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 from outside the class without any restriction. Class functions are usually public by default. Public member can be accessed by member 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 same 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 from 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 and derived
        void show(){ cout<<"\data is"<<sum;}   };          // class closed
void main(){// main method
clrscr();          obj o;              // object of class
p.setdata(2);                  o.show();  //object .function
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 objects of that type using declarations, which are similar to the built-in type declaration.

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 structure it holds homogenous 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)

```
Enter your phone:124687
Enter the age::24
Enter the name:Sunita
Enter u address:Lekhnath
Enter occupation:Student

The name is:Sunita

The address:Lekhnath

The occupation is:Student

The phone is::124687

The age is:24_
```

```
#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
        cout<<"Enter the age::"; cin>>i.age;
        cout<<"Enter the name:"; cin>>i1.name;
        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 class is the group of objects that can share common properties and established the 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 within a class are also called methods. When we design a class we use new abstract data type that can be treated as 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.

```
4 And45.6_
```

```
#include<iostream.h>
#include<conio.h>
class item { int num; float cost;
public:   void getdata ( int a, float b)  {num=a; cost=b;}
void putdata(){cout<<num<<cost;}   };
void main(){   item x;
x.getdata(4,45.6);              x.putdata();
getch();}
```

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, bank, buffalo, bench etc. This name becomes a new type identifier that can be used to declare instance of class (object). The class item contains two function members are **int num and float cost**. **void getdata (int a, float b) a**nd **void putdata()** are private by default but here both the functions are public by declaration. The function getdata (int, float) 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 the member of that class.

Difference between structure and class

Structure	**Class**
Structure is data type that can handle homogenous data type.	Classes are mixture of function and member data and objects.
To declare structure we use struct in front of structure name.	To declare class we just use class keyword in front of class name.
Each structure ends with ;(semicolon)	Class also ends with semicolon;
Dot operator is used to access variable	The dot operator is used to access function of particular class
Structure provides template of data type.	Class provides structure to all objects, dat items and functions.
We cannot add more objects as requirement increases.	We can add more objects as requirement increases.
Give Example code	Give Example code

Creating Object

Once a class has been declared, we can create object of that class. The class representative is known as object or instance. Therefore x is called object of that class. We may also declare more than one object in one statement. Definition: 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-oriented programming gives more importance to the class, object and function. Object-oriented programming is therefore the implementation of abstract data type. At runtime the instances of class and the objects achieve the goal of the program by changing its states. Consequently, in the running program it is like as a collection of objects. The question arises how these objects *interact to satisfy your requirement.*

Message Passing

A class is made up with objects, function and data items where objects are created, destroyed and interact with objects. 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 send the messages with the help of "." operator. 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, the 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:*** *A **method** is associated with a class. An object invokes a method as a reaction to receipt of a message.*

Accessing Class Member Function

The member function of any classes can be defined in two ways:

- **Either outside the classes' definition**
- **Inside class definition.**

The private data of a class can be accessed only through the member functions of that class. The main () cannot contain statements that access num and cost directly in the above code. The syntax:

object name. function (arguments);

x.getdata(4,45.6); x.putdata();

```
Enter name :Puja
Enter the roll2
Enter the  marks34
The name isPuja
Roll is:2
The marks is:34_
```

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/instance
   t.putdata(); getch(); }
```

In the above code the getdata member function is designed what is the functionality it should do to fulfill the test task is designed inside the class. Similarly the function putdata function also designed to do its task inside the class encapsulation. In such case we just create the object of test class and used dot operator to access these member function in main method.

Object as Physical Objects

```
#include<iostream.h>
#include<conio.h>
class part{   private:     int model;     int partno;    float cost;
public:         void setpart(int mn, int  pn,float c){
model=mn;     partno=pn;cost=c;}
void showpart(){    cout<<"\nModelno is\t"<<model;
cout<<"\nPartno is:\t"<<partno;    cout<<"\nCost is:\t"<<cost;}    };
void main(){     part p;
p.setpart(654,3900,1090.56);
p.showpart();
```

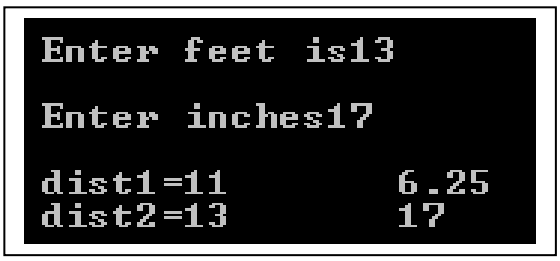
```
Modelno is      3900
Partno is:      654
Cost is:        1090.56
```

```
getch();  }
```

Here single set part function is used to supply all three data items at once. Similarly showpart() sets the three data times in all three items. In this example only one object is created i.e. p.

Object as Data Types

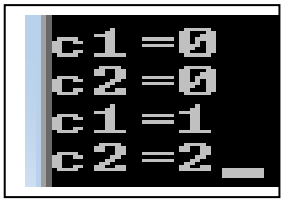
```
#include<iostream.h>
#include<conio.h>
class Distance{  private:  int feet;  float inches;
public:  void setdist(int ft,float in){  feet=ft;  inches=in;}
void getdist(){  cout<<"\n Enter feet is";cin>>feet;
 cout<<"\n Enter inches";cin>>inches;}
void showdist(){  cout<<feet<<"\t"<<inches;}  };
void main(){  Distance dist1,dist2;
dist1.setdist(11,6.25);  dist2.getdist();
cout<<"\n dist1=";dist1.showdist();
cout<<"\n dist2";dist2.showdist();getch();  }
```



Here C++ objects can be represent variables of a user defined data type, we used objects to represent distance in inches and feet.In this program class distance contains two data items feet and inches. And the class distance has three member functions setdist () which uses arguments to set feet and inches, getdist () which gets for feet and inches form user at the key board; and finally showdist () which display the distance in feet and inches formats. The value of object of class can be set either of two ways. In main () we defined two objects of class distance dist1 and dist2. One is used given a value using the setdist () member function with the arguments 11 and 6.25 and the other is given a value supplied by user.

Counter Example

```
#include<iostream.h>
#include<conio.h>
class counter{
private: int count;
public :  counter(){ count=0;}
void increased(){count++;}
int getcount(){ return count;}  };
void main(){  counter c1,c2;
cout<<"\nc1="<<c1.getcount();
cout<<"\nc2="<<c2.getcount();
c1.increased();  c2.increased();  c2.increased();
cout<<"\nc1="<<c1.getcount();
cout<<"\nc2="<<c2.getcount();
getch();  }
```



We ll create a class of objects that might be useful as general purpose. A count is a variable that counts things or number of times. Each time such events takes place the counter incremented. The counter can also be accessed to find the current count. The counter class has one data items count, of type int. is always positive number. It has three member function counter () which is as constructor. Increased () is the function which increased 1 to count variable and getcount () with returns the current value of counter class.

String related syntaxes

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main(){
char str1[]="Oh no Mr. Profssors of my college is not ";  char str2[80];
//int roll;
//cout<<"Enter character\t";
```

```
//cin>>str;// used to read one word only
//cin.get(str,20); //to read all data
//cin.getline(str,40);
//cin.get(str,80,'$'); // used for to get up to $ sign all other after $ is ignored
//cin>>roll;
//cin.ignore(); //to control only integer numbers say 5tyy' just reads 5 only
//cout<<roll;  int i;
for(i=0;i<strlen(str1);i++)
str2[i]=str1[i];    str2[i]='\0';   cout<<endl;   cout<<str2;
getch();   }//cin.get();//similar to getch but do not wait the output
```

```
#include<iostream.h>
#include<conio.h>
const int days=7;    const int max=10;
void main(){  char   str[days][max]={"sunday","monday",
"tuesday","wednesday","thursday","friday","saturday"};
for(int i=0;i<days;i++)
cout<<str[i]<<endl;
getch();  }
```

```
sunday
monday
tuesday
wednesday
thursday
friday
saturday
```

User defined string type

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string{
private: char str[80];
public:     string(){str[0]='\0';}
string(char s[]){strcpy(str,s);}
void display(){    cout<<str;}
void concat(string s2){     if( strlen(str)+strlen(s2.str)<80)     strcat(str,s2.str);
else cout<<"\n string is too long";   }   };
void main(){          string s1("Marry Chrismas!");
string s2="Best season";
string s3;
cout<<"\n 1 inf=";    s1.display();
cout<<"\n 2nd inf=";    s2.display();
s3=s1;   //the message can be transferred form one object to another object
s3.concat(s2);  // s2 object is added to s3 object
cout<<"\n 3rd inf=";    s3.display();
getch();   }
```

```
1 inf=Marry Chrismas!
2nd inf=Best season
3rd inf=Marry Chrismas!Best season
```

In this string class contains data items array of char type. It may see that out newly defined class is just the same as original string. s3=s1 line shows the object can be passed to another object of same class when the object are created. Similarly member function connate concatenates one string to another. The original string is added from s2 object.

```
#include<iostream.h>
#include<conio.h>
typedef struct {  int i, j, k;}
ThreeDpoint;
int main() {   ThreeDpoint p[10];
for(int i = 0; i < 10; i++) {
cout<<(p[i].i = i + 1)<<"\t"; cout<<(p[i].j = i + 2)<<"\t"; cout<<(p[i].k = i + 3)<<endl;  }
getch();   return 0;}
```

```
1       2       3
2       3       4
3       4       5
4       5       6
5       6       7
6       7       8
7       8       9
8       9       10
9       10      11
10      11      12
```

Enumerated Data Types

An enum specifies defines the set of all names that will be permissible values of the type. These values are called members of the data variable. Enumerated means that all values are listed. These data types are treated as integer. The first member gets value 0 and so on. Changes can be done by assigning it in the declaration.

```
#include<iostream.h>
#include<conio.h>
enum days{Sun=1,Mon,Tue,Wed,Thu,Fri,Sat};
void main(){      days day1,day2;
day1=Sun; day2=Sat;
  int diff=day2-day1;
  cout<<"The different is"<<diff;
  if(day1<day2)      cout<<"\n"<<day1<<"\t comes before\t"<<day2;
  getch();}
```

```
The different is6
1        comes before   7
```

```
#include<iostream.h>
#include<conio.h>
class part{    clrscr();
private:   char partname[30];
int partnumber;  float cost;
public: void setpart(char pname[],int pn,float c){
strcpy(partname,pname);   partnumber=pn;  cost=c;}
void showpart(){  cout<<"\n Name is:"<<partname;
cout<<"\n Number is:="<<partnumber;
cout<<"\n cost $="<<cost<<endl; } };
void main(){       part part1,part2;
part1.setpart("Handled IT",34,200.5);
part2.setpart("Start lever",44,20.8);
cout<<"First part ";  part1.showpart();
cout<<"Second part ";  part2.showpart();
getch();}
```

```
First part
 Name is:Handled IT
 Number is:=34
 cost $=200.5
Second part
 Name is:Start lever
 Number is:=44
 cost $=20.8
```

This program defines two objects of class part and gives them values with the setpart member function. Then it displays them with the showpart() member function. In setpart member function we use strcpy string library function to copy string from the argument pname to the class data member partname. This function saves the same purpose with string variables that an assignment statement does with simple variables.

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. Inside the main () method 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 employee::read(){        //scope resolution operator for first function
cout<<"Enter empno, name, address"; cin>>empno;  cin>>name;
cin>>add; }         //end first function
void employee::display(){       // 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.read();          //obj. First member functi
t.display();
getch();}        // main method function
```

```
Enter empno, name, address3
Sila
Pokhara
Empno:3
Name:Sila
address:Pokhara
```

In the above code the member function read and display designed inside class but its structure and its functionality is designed outside the class with the help of scope resolution operator. Therefore scope resolution operator provides the facility to access class member functions outside class definition.

Create a class distance, a constructor is used to take feet and inches as arguments. The another function getdist is used to take inches and feet front user then add function having two objects as parameter **which add these values respectively. (Object as function arguments)**

```
#include<iostream.h>
#include<conio.h>
class distance{   private:  int feet;    float inches;
public:  distance(){}
distance(int ft,float in){    feet=ft;inches=in;}
void getdist(){    cout<<"Enter feet";cin>>feet;    cout<<"Enter inches";cin>>inches;}
void show(){    cout<<"The feet is:"<<feet;    cout<<"\tThe inches is:"<<inches;}
void add(distance, distance);    };
void distance::add( distance d2, distance d3){ inches=d2.inches+d3.inches;     feet=0;
if(inches>=12.0){   inches-=12.0; feet++;}
feet+=d2.feet+d3.feet;}
void main(){
distance dist1,dist3;
distance dist2(11,6.25);
dist1.getdist();
dist3.add(dist1,dist2);
cout<<"\ndist1=\t";dist1.show();
cout<<"\ndist2=\t";dist2.show();
cout<<"\ndist3=\t";dist3.show();
getch();  }
```

```
Enter feet23
Enter inches26

dist1=  The feet is:23  The inches is:26
dist2=  The feet is:11  The inches is:6.25
dist3=  The feet is:35  The inches is:20.25
```

This program demonstrates constructor overloading defining member function form outside class and object as function arguments. The class distance having feet and inches are two data members. Distance is one constructor having no arguments as default constructor. Similarly distance having two arguments int ft and float in is another constructor shows constructor overloading having same name but different arguments. Both are used for to initialize the values. Similarly getdist() is another function used to asked supply feet and inches to the system. And another function showdist is function just display the information. In the main method we create two object of distance class to are used to ask the user supply two items of feet and inches separately and another object that add up them with the help of void add() function outside the class is shown. Distance object 2 is initialized 11,6.25 as parameter is one items and dist1.getdist() is another function which asked user to supply the feet ad inches. The third object dist3 called add function having two

data type of object type. Which add both items separately and checked if the value of two items is checked conditionally inside add function. This line demonstrate object as function arguments.

Varieties of Classes

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

The varieties of classes are categorized according to 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

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 that 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.

Data source or data sink: A class which supports two way data interchange is known as data source class. Data source class accepts some data and processor those data for further used is known as data sink class. This type of class does not hold data for long period of time but it generates data & supply to other classes. Data sink means **accept** data and data source means **supply** data therefore this type of class can supply data and accept those supplied data requires 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 class and another class which accepts the data generated by another class fall into data sinks categories.

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

Facilitator and helper class: This class is helper class of data manager class which facilities the proper maintaining of data, proper display of information, accelerate programming software and hardware too.

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 used before constructor.
- **constructor method must be public**
- Constructor is executed when the object of the class is created; no special call is to design to execute the constructor.

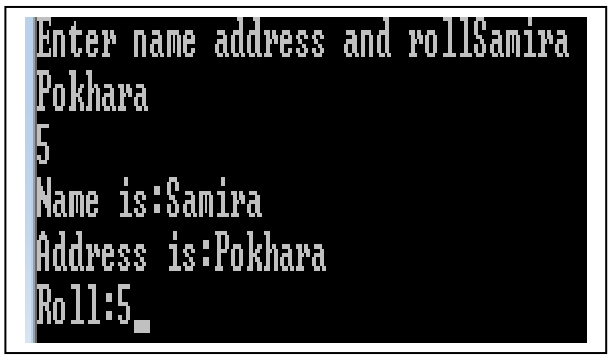
Rules of Destructor

Destructor destroyed the dynamic values from memory.

- 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 be used 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
~test();//destructor
};// class closed



Example:

```
#include<iostream.h>//preprocessor
#include<conio.h>
class test{          //Class declaration
char name[20] ;              char add[25];  int roll;  // class properties
public : test();          // constructor method must be public and without return type
void display(); //for display method
~ test();   };          //destructor and encapsulation
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(){          clrscr();test t;//          main method
//class object void test constructor is called
t.display();//but constructor is not called because it execute automatically
getch();  }
```

Types of Constructor

Default Constructor

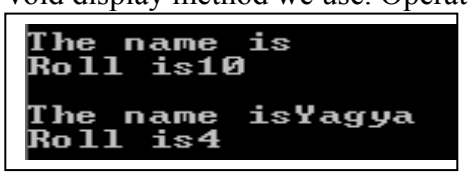
We have seen varieties of classes with interaction of its member functions interacting one another in classes putdata () get data () are member function provides interaction with class object like object. This is very convenient if we have larger numbers of functions used in our program. A constructor that accepts no parameter is called the default constructor. The default constructor integer can be called just creating object of that class.

```
#include<iostream.h>
#include<conio.h>
class integerr{ public: int m,n;
public:  integerr(){  m=1;n=2; }//default constructor//initialized member data items
void display(){  cout<<endl<<m<<"\t"<<n;}  };
void main(){  integerr ii;                  // object of class
cout<<"\nPassed value are";  ii.display();getch();}
```

As above class integer class have two data members int m , int n and one member function integer() without any parameter . There is no need to write any statements to invoke the constructor function in main method. In integer constructor function passed 1 to m and 2 to n. This is process of initialization. We can invoke this function for each of the objects separately. If we design our program to be designed outside function as integer::integer () {}. If no match constructor is found then the computer compiler supplies a default constructor in main function. To get result in void display method we use. Operator then function name.

Similarly;

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class apple{  char name[34];  int roll;
```




```
void main(){
apple
a("Marphali",4);
a.display();
apple
b("Manange",4);
```

```
public:  apple(){name[0]='\0'; roll=10;} //default constructor
apple(char c[], int r){ strcpy(name,c);  roll=r;   }
void display(){  cout<<"\nThe name is"<<name;
cout<<"\nRoll is"<<roll<<endl;  }   };
void main(){ apple a;
apple aa("Yagya",4);
a.display();
aa.display();
getch();  }
```

```
void main(){
char p[23];
int t;
for(int i=0;i<5;i++){
cout<<"Enter  name  and
number"<<i<<"Items";
cin>>p>>t;
apple a(p,t);
a.display(); }
```

Parameterized Constructors

Object oriented programming permits us to have put some data items inside function as arguments. Similarly constructor also supports to kept data items in as arguments. The constructors that can take arguments are called parameterized constructors.

```
#include <iostream.h>
#include<conio.h>
class Z {   int i, j;
public:    Z(int ii, int jj);
void print();  };
Z::Z(int ii, int jj) {   i = ii;   j = jj;  }
void Z::print() {   cout << "i = " << i << ", j = " << j << endl; }
int main() {    Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
for(int i = 0; i <4; i++)
zz[i].print(); getch();}
```

```
i  =  1,  j  =  2
i  =  3,  j  =  4
i  =  5,  j  =  6
i  =  7,  j  =  8
```

In above code class Z having some data types int i,j. the member function Z(int ii, int jj) is constructor having two arguments that takes two integer values as parameter. And similarly void print function print these values. The Z constructor having two parameters is accessing outside the class with the help of scope resolution operator. Similarly the print function is also printing int i and jj from the constructor methods. In main method Z zz[] is object of constructor having array type is passed to object zz. To print them we just used for loop up to array items the called print function before getch.

Copy Constructor

A copy constructor takes pointer reference to an object of the same class name as itself as an argument. Let us consider a simple example of reconstructing and using a copy constructor below. Here, code (code &x) is the constructor having argument as class type. x. id is the pointer reference code is used to access particular location.

```
1 0 0
1 0 0
1 0 0
1 0 0
```

```
#include<iostream.h>
#include<conio.h>
class code{  int id;
        public: code(){}                // default constructor
        code(int a){    id=a;}      // constructor with one arguments//passing values to id
        code(code & x){//copy constructor that takes class name as data type and pointer object
        id=x.id; }                  //dot operator is used to passed value form address to variable
        void display(){         cout<<id;} };         // to display id   //class used
                void main(){
                code A(100);// constructor object A is initialization
                code B(A);      // constructor object B again initialization same value of A
                code C=A;// constructor object c again initialization to same value of A
                code D;         // another object d
```

```
D=A;// D object passing value of A
cout<<"\n";A.display();//calling display functions
cout<<"\n";B.display();   cout<<"\n";C.display();   cout<<"\n";D.display();
getch(); }
```

Dynamic Constructor

The constructor can be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of same size, this resulting the save memory. The allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of new operator.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string{  private: char *str;
public: string(){str=new char[0]; }
string(char *s){     str=new char[strlen(s)+1];
 strcpy(str,s);    }
void getdata(){     cout<<"Enter New string\n";
cin.getline(str,'\n');}
void display(){     cout<<"\nEntered string is\t"<<str<<endl;}
~string(){delete str;
cout<<"\nDestroyed";}    };
void main(){    clrscr();
string s1("Welcome");
string s2;// creating objects of string one with Welcome string.
s1.display();//display s1 object string
s2.getdata();//to take  string from user
s2.display();//calling display function with user supply string
string();       getch();    }//calling destructor
```

```
Entered string is       Welcome
Enter New string
Hi student

Entered string is       Hi studen

Destroyed
```

Here class string having * pointer data item is STR which is not designed to hold how many characters is to be hold. The constructor string having null parameter is designed to initialize only with the help of new operator that is initialized with the help of strcpy function with null value inside default constructor. Similarly string constructor having one pointer parameter is designed new char type whose dimension is declared string length+1, which is again copy from s as parameter form object of main method. The getdata function is used to read all line except next line input form user. The display function is used to display char string. The destructor is used to delete the memory whose memory is again and again reallocated with the help of new operator in constructors.

In main method first constructor is populated with welcome with double cote goes to first constructor with argument it copy string Welcome text then transfer str to display function. The second string constructor that object s2 again getdata function to read second information form user then display new text information is also hold by str as pointer type char . At last the delete destructor is called just to release pointer memory area.

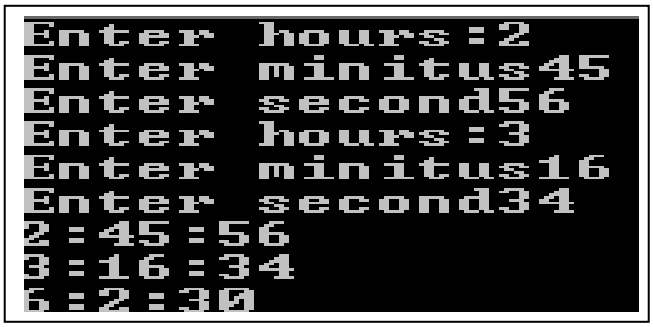
```
#include<conio.h>
#include<iostream.h>
class string{  char *name;  int length;
public:  string(){ length=0;
name=new char[length+1];}
string (char *s){ length= strlen(s);
name= new char[ length+1];
strcpy(name,s);}
```

```
Yagya
Nath
Rimal
YagyaNath
YagyaNathRimal
```

```
void display(){  cout<<name<<"\n";}
void join(string & a, string & b);      };
void string::join( string & a, string & b){
length=a.length+b.length;
delete name;
name=new char[ length+1];
strcpy(name,a.name);   strcat(name,b.name);};
void main(){  char *first="Yagya";
string name1(first),name2("Nath"),name3("Rimal"),s1,s2;
s1.join(name1,name2);   s2.join(s1,name3);
name1.display();  name2.display();  name3.display();
s1.display();  s2.display();
getch();}
```

Create a class time a constructor having hours minutes and second as arguments is used to take two data form user. The add function that takes two class objects an arguments adds them respectively. Then display aggregate results.

```
#include<iostream.h>
#include<conio.h>
class time{  private : int hrs,mins,sec;
public:  time(){hrs=0;mins=0;sec=0;}
time(int h,int m,int s){hrs=h;mins=m;sec=s;}
void getdata()  {cout<<"Enter hours:";cin>>hrs;  cout<<"Enter minitus";cin>>mins;
cout<<"Enter second";cin>>sec;
while (sec>=60){  sec=sec-60; mins++;}
while(mins>59){ mins=mins-60; hrs++;}     }
void display(){  cout<<hrs<<":"<<mins<<":"<<sec<<endl;}
void add(time t1,time t2){  sec=t1.sec+t2.sec; mins=t1.mins+t2.mins;  hrs=t1.hrs+t2.hrs;
if(sec>=60){ sec=sec-60; mins++;}
if(mins>=60){mins=mins-60; hrs++;}}     };
void main(){    time time1, time2, time3;
time1.getdata();   time2.getdata();
time1.display();   time2.display();
time3.add(time1,time2);
time3.display();
getch();   }
```



In above code time is class name and constructors are overloading having one default constructor without any parameter but this is used to initialize its member function to be zero, similarly another constructor having three parameter us processed the value form main function to its global variable. Similarly void getdata is another member function is used to give data to the system further it checked and calculate minutes and second and hours respectively inside it. Similarly the void display function is used to display hours minutes and seconds form the console. The void add(class ,class) is another function that takes constructor as two objects then time second, minutes and hours are added separately then again calculate minutes and hours inside it. These all members' functions are inside the classes definite are called with time objects time1, time2 tieme3 inside main method. What activities are to be done is designed according to our requirements. thee time3 objects having two arguments as constructor type objects is passed to add two data read form users the n this is also displayed by same display function.

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 becomes released. 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 there is using in somewhere. The memory allocation of automatic variable is like a stack. Stack means Last in First 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 variable and value can be changed if memory space specifies no longer pointing to the memory area. Hence memory can create when the programmer send request to that new objects. 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 calls ends. There is one another category of variables namely global variables. Global variables are variables that are declared outside the function definition.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class add{   char*str;
public:  add(char*ptr){  int i= strlen( ptr);
       str=new char(i+1);  strcpy(str,ptr);}
~add(){  delete str; };
     void display(){  cout<<"the string will
     be:"<<str; }};
void main(){  clrscr();
add a("\nGoodmorning");
a.display();  getch();}
```

```
#include<iostream.h>
#include<conio.h>
class studentinfo {
char name[30];   char add[30];    int roll;
public:void get(){
     cout<<"\nEnter the name,add,roll"<<endl;
     cin>>name>>add>>roll ; }
  void
put(){cout<<"name"<<name<<endl<<"address:"<<add<<en
dl<<"roll:"<<roll; }};
void main(){ clrscr();
    studentinfo s;s.get(); s.put();
studentinfo*s1;
s1=new studentinfo[20];int i;
for(i=0;i<5;i++){
s1[i].get(); }
for(i=0;i<5;i++){
s1[i].put();}
delete s1;  getch();}
```

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 we declared large memory space in the design time but all the variables may not use always used completely in the program. Therefore, in the above program unused memory space can recover and save memory area. This may be reused to another variable in the program.

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

C++ provides a third visibility modifiers protected, which serve a limited purpose in inheritance. A member declared as protected as accessible by the member functions within its class and any class immediately derived from it. It cannot be accessible by the function outside these two classes. When protected member is inherited in public mode, it becomes protected in the derived class therefore it is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member inherited in the private mode derivation becomes private in the derived class. Although it is available to the member functions of the derived class, it is not visible for further inheritance.

Base class visibility	Public derivation	Private derivation
Private	Not inherited	Not inherited
Protected	Protected	Private
Public	Public	Private

The keyword private, public, and protected may appear in any order in any number of times in the program.

Single Inheritance

Inheritance is a property of OOP in which the characteristics of one class comes into derived 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
cout<<"\nPhone:"<<ph<<"\nFees:"<<fees;}} };//both class clos
void main(){  clrscr();
test1 t1;        //object of derived class
t1.get(); t  1.put();
getch();         }
```

In 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 handle 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.

Constructor in Derived Class

```
#include<iostream.h>
#include<conio.h>
class counter{  protected: int count;
public:  counter(){count=0;}
counter(int c){ count=c;}
int getcount(){return count;}  };
class countdown: public counter{
public: countdown(): counter(){};
countdown(int c): counter(c){};  };
void main(){    countdown c1; countdown c2(100);
cout<<"c1="<<c1.getcount()<<"\t";
cout<<"c1="<<c2.getcount()<<"\t";
getch(); }
```

In above code class counter has protected data items and counter having no arguments is default constructor is used for to initialized counter variable 0 and counter having one argument int c is another constructor and one getcount is return type functions used in counter class. Similarly countdown is another class derived from counter class having countdown null parameter and countdown is another constructor having one argument is used . the derived class countdown is publicly derived from counter so that we can access super class constructor with using colon function and terminate with semicolon at end of constructor member. the countdown member constructor can be initialized 0 in first case and similarly countdown having one argument is again passed value form main methods. Inside main method we just create two objects of countdown then call getcount function which reside inside counter class.

```
#include<iostream.h>
#include<conio.h>
class alpha{ int x;
public:  alpha(int i){   x=i;cout<<"\nAlpha initialized"; }
void showa(){   cout<<"\nThe value is:"<<x;}};
class beta{   float y; public:  beta(float j){
y=j;cout<<"\nBeta initialized";}
void showb(){  cout<<"\nY is:"<<y;} };
class gamma : public beta, public alpha{   int m,n;
public:gamma(int a, float b,int c, int d): alpha(a),beta(b){
m=c;n=d; cout<<"\nGamma initialized";}
void showg(){  cout<<"\nM is:"<<m;  cout<<"\nN is:"<<n;}  };
void main(){
gamma g( 1,20.3,20,40);
cout<<"\n";  g.showa();  g.showb();  g.showg();
getch();}
```

```
Beta initialized
Alpha initialized
Gamma initialized

The value is:1
Y is:20.3
M is:20
N is:40
```

The beta is initialized first although it appears second in the derived constructor. This because it has been declared firist in the derived class header line. Alpha(a) and beta(b) are function calls. Therefore the

parameters should not include type. The C++ supports another type of initialization class objects. This method used what is known as initialization list in the constructor function. Constructor gamma just after : is known as initialization section. We can use this section to provide the initial values to the base constructors and also to initialize its own class members. **Similarly**

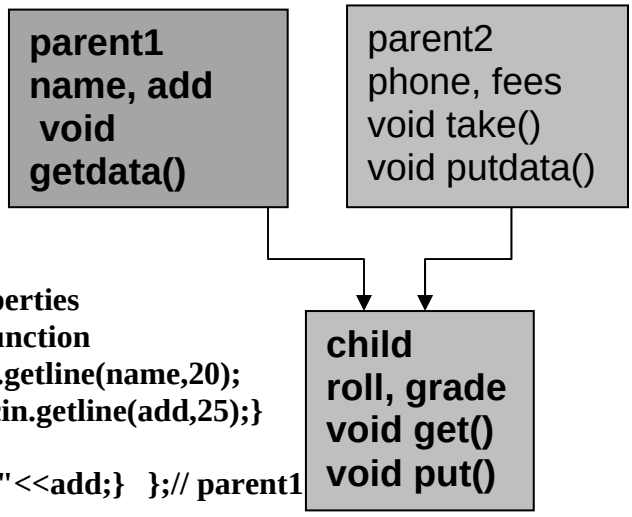
```
#include<iostream.h>
#include<conio.h>
class alpha{  int x;
public:  alpha(int i){   x=i;cout<<"\nAlpha constructor"; }
void showa(){   cout<<"\nThe value is:"<<x;}};
class beta{        float p,q; public:
beta(float a, float b): p(a), q(b+p){   cout<<"\nBeta constructor";}
void showb(){  cout<<"\nP:"<<p;        cout<<"q is:"<<q;} };
class gamma : public beta, public alpha{          int u,v;
public:     gamma(int a, float b, int c): alpha(a*2),beta(c,c),u(a){
v=b; cout<<"\nGamma constructor";}
void showg(){  cout<<"\nu is:"<<u;
cout<<"\nv is:"<<v;}  };
void main(){
gamma g( 2,4,2.5);   cout<<"\n";
g.showa();   g.showb();  g.showg();
getch();}
```

```
Beta constructor
Alpha constructor
Gamma constructor

The value is:4
P:2q is:4
u is:2
v is:4
```

All objects of gamma class contains the objects a and b. This kind of relation is called containership of nesting. The arguments list of arguments this is to be supplied when a gamma object is defined. These parameters are used for initialization the members of gamma. The arguments a and argument b are constructor. The constructor a and constructor b may or may not use the arguments . Remember, a (argument) and b(argument) are function calls the arguments do not contain the data types. They are variables or constants.

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 as child inherit the genetic characteristic of their parents.



Class A{}
class B: visibility A{}
class C: visibility B {}
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
class parent2{//another parent2  super class
int ph;  int fees;//data items
public :     void take();//function to givedata
```

```
Enter Your Name:
Sila

Enter Your Address:
```

```
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();          getch();}//obj.derived class function name
```

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 taken () 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 {.*** The derived child class gets all the properties of its super classes and roll and grade are two properties set out of its own. In the child 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.

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write: class CRectangle: public CPolygon, public COutput;

```
#include <iostream.h>
#include<conio.h>
class CPolygon {
protected:    int width, height;
public:    void set_values (int a, int b)   { width=a; height=b;}  };
class COutput {
public:    void output (int i);  };
void COutput::output (int i) {  cout << i << endl;  }
class CRectangle: public CPolygon, public COutput {
public:    int area ()      { return (width * height); }  };
class CTriangle: public CPolygon, public COutput {
public:    int area ()      { return (width * height / 2); }  };
int main () {    CRectangle rect;     CTriangle trgl;
rect.set_values (4,5);     trgl.set_values (4,5);
rect.output (rect.area());    trgl.output (trgl.area());
getch();  return 0;}
```

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:

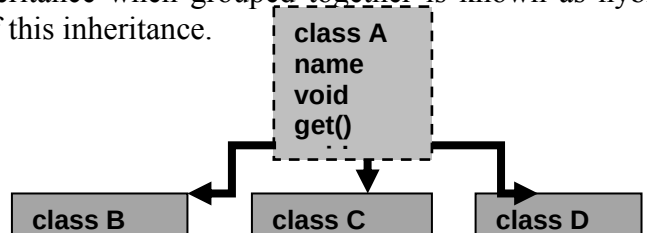
```
#include<iostream.h>
#include<conio.h>
class parent1 {//parent class
char name[30];  char add[25];//data items
public: void getdata(){//methods to give data
 cout<<"\n Enter Name and Address:";cin.getline(name,30);
 cin.getline(add,25);  }//method closed
 void putdata(){// method display data
 cout<<"\n Name is:"<<name<<"\n Address is:"<<add;}  };//parent class closed
 class child1:public parent1{//child1 derived from parent1
 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 close
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 giveda
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();          c2.display();//obj.child2//obj.child2
getch();  }
```

```
 Enter Name and Address:Puja
Pokhara

Enter roll and phone12
98460

Enter Age:34

Enter Price5000

 Name is:Puja
 Address is:Pokhara
Roll is:98460
Phone is:12
Age is:34
Price is:5000
```

In 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 own properties which are operated with the help of get and put methods. Likewise in the third level 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 classes to the lower level.

Hybrid Inheritance

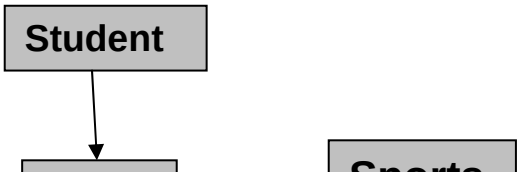
In the hybrid inheritance it possible to access two or more types of inheritance is used to design a program. It is the combination of single as well as multilevel inheritance when grouped together is known as hybrid inheritance. The following figure shows clear picture of this inheritance.




```
#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 class
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 functi
cout<<"\nEnter your age";cin>>age;}
void put4(){//display data function of E
B:: put1();C::put2();//calling display function of B and C
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
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
```

```
Enter Your name:
Yagya

Enter Your address:
Lekhnath

Enter Your roll8

Enter your age23

Enter Your phone8907

Enter the price 9000

Enter your occupation:

Your Name is:Yagya
The address is:Lekhnath
Your Roll is:8
Your ageis:23
Your phoneno is:8907
The price is:9000
Your occu is:
```



```
G g1;//object of G class
g1.get6();//obj.G class givedata function
g1.put6();//obj.G class duisplay data function
getch();  }
```

WAP to demonstrate this figure (Hybrid inheritance)

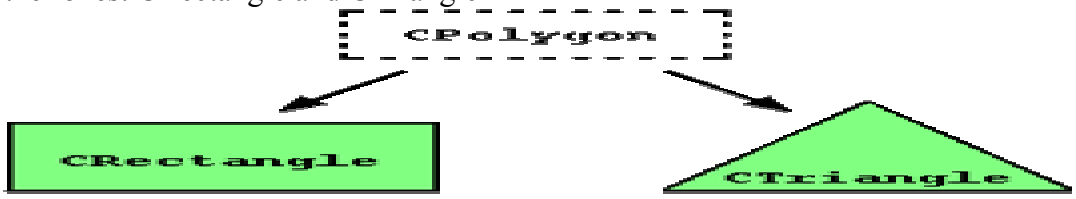
```
#include<iostream.h>
#include<conio.h>
class student{  protected: int roll;
public: void nums(int b){roll=b;}
void puts(){ cout<<"Roll is"<<roll;}};
class test: public student{
protected: float p,q;
public:  void gett(float y,float z){ p=y;q=z;}
void putt(){cout<<"Marks is"<<p;cout<<"Marks is"<<q;}  };
class sport{    protected:  float sum;
public:  void getss(float z){sum=z;}
void putss(){ cout<<"sport"<<sum;}  };
class resutl: public test, public sport {float tot;
public:  void putrt(){   tot=p+q+sum;
puts();  putss();   putt();
cout<<"total is"<<tot;}};
void main(){   resutl r;r.nums(1234);
r.gett(80.5,62.5);   r.getss(6.2);
r.putrt();  getch();     }
```

```
Roll is1234
sport6.2
Marks is80.5
Marks is62.5
total is149.2
```

Inheritance between Classes

A key feature of C++ classes is inheritance. Inheritance allows creating classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle



.

The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B. In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

class derived_class_name: public base_class_name { /*...*/ };

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifies protected and private. This access specifier limits the most accessible level for the members inherited from

the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

```
#include <iostream.h>
#include<conio.h>
class CPolygon {  protected:     int width, height;
public:    void set_values (int a, int b)      { width=a; height=b;}  };
class CRectangle: public CPolygon {
public:    int area ()      { return (width * height); }  };
class CTriangle: public CPolygon {
public:    int area ()      { return (width * height / 2); }  };
int main () {  CRectangle rect;  CTriangle trgl;
rect.set_values (4,5);    trgl.set_values (4,5);
cout << rect.area() << endl;
cout << trgl.area() << endl;
getch();  return 0;} output 20
                                    10
```

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are: width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members. Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, have we used protected access instead of private. We can summarize the different access types according to who can access them in the following way:

Access	Public	Protected	Private
Members of the same class	yes	yes	yes
Members of derived classes	yes	yes	no
Not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main (), from another class or from a function. In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width          // protected access
CRectangle::width        // protected access
CPolygon::set_values()   // public access
CRectangle::set_values() // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes: class CRectangle: public CPolygon { ... }

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case CPolygon) will have. Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class. If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private. For example, if daughter was a class derived from mother that we defined as: class daughter: protected mother; This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother. If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

What is inherited from the base class

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class: **derived_constructor_name (parameters) : base_constructor_name (parameters) {...}**

For example:

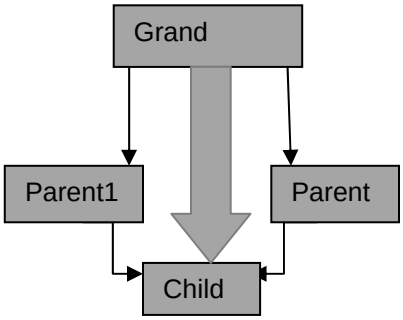
```
#include <iostream.h>
#include<conio.h>
class mother {
public:   mother (){ cout << "No Kids\n"; }
mother (int a)   { cout << "\nHave Kids\n"; }};
class daughter : public mother {
public:    daughter (int a) { cout << "Daughter: \t"<<a; }};
class son : public mother {
public:    son (int a) : mother (a)      { cout << "son: \t"<<a; }};
int main () {
daughter cynthia (1);  son daniel(2);
getch();  return 0;}
```

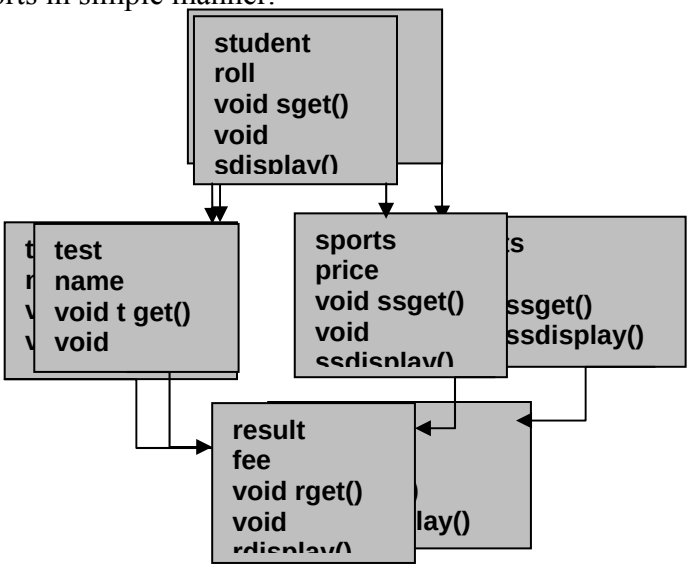
```
No Kids
daughter:       1
Have Kids
son:    2_
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

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 parent members is due to the multiple inheritances can be avoided by making the common base class is as virtual base class. When a class is made virtual base class, C++ takes necessary care to see that only one copy of that base class is inherited, regardless of many inheritance path exists between the virtual base class and the derived class the keyword virtual is placed in front of the inheritance. In the below figure class student is the simple base class having one property roll and void sget() and sdisplay () are the two member functions student. Similarly the test and sports are the two derived classes are inherited form the same base class virtual student. The virtual student is required to get single copy of parent student. In the third level result class will inherited form the both class test and sports in simple manner.






```
#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(); }
```

```
Enter nameSubash
Enter roll3
Enter your price4000
Enter fees6000

Your roll is:3
Your name is:Subash
Your price is:4000
Your fee is:6000_
```

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 becomes large mesh relationship among parents to child. As the requirements increase the dependency on parents and child automatically increased. There are some forms of inheritance:

a) **Sub classing/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 the class Window in operating system provides the general windows properties like move, resize, maximum, minimum icons and so on. The same window properties are reused to design Notepad, WordPad, Word, and Excel etc. This is only possible through the use of inheritance. Therefore Microsoft office package is subclass of the parent window operating system. It will take all the properties of window have.

b) **Sub classing 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 how the behavior will be implemented in the child classes. E.g. When we declared protected member function in the parent class this member can access only one level to the derived class.

c) **Sub classing for construction:** The child class can be constructed form parent classes by implementing the behaviors of parent classes. The desired behavior from the base classes can be collected and constructed in the child class. Some arguments and behaviors provided by the parent class may redesign inside the sub class construction. Inheritance provides the facility to construct new class with the help of parent class.

d) **Sub classing for generalization:** We know, the child class modifies some of the methods of the parent class. It is applicable when we design the child class from the existing base. The child class modifies or override (replace) some method of the parent class. A child class can be used the general behavior of the parent class. Sub classing for generalization is the opposite to sub classing for specifications. When we do inheritance, the general properties of the parent class can be easily accessible to the child. The base class holds the common properties that will inherit to the derived class.

e) **Sub classing 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 function other than parent class have. The child class adds new functionality to the parent class but does not change any inherited behaviors. So child class is look like as extension parts of the base class.

f) **Sub classing 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 properties. In sometime the child class restricts the use of some of the behaviors and functions inherited form the parent class.

g) **Sub classing for variance:** The child class and parent class are varied when the level of inheritance increased, and the class and subclass relationship is imaginary. When we increased the level of inheritance , the subclass is varied in nature.

h) **Sub classing for combination:** The child class inherits features form more than one classes. There is a multiple inheritance in which the child class inherits features form more than one classes. The child class has coexistence only from the base class. It may form when we access the properties form single or various base 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. The behaviors of inheritance increase the productivity of the software designer, because the programmers do not 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:** when two or more sub classes inherit the behavior and function from base classes are some in all times inheriting form the same super class. We are assured that the behavior they inherit will be the same in all cases and times. That provides consistence in the program design. The super class provides the same outlooks for all sub classes.

d) **Software component:** If we design big project the super class behaves like as universal components that provide programmer with the ability to construct reusable software components. We can construct the reusable software component by collecting/ reusing the behaviors of those components.

e) **Rapid prototyping:** When we inherit the properties form super class we can reuse the code and construct new components. This process reduce coding time is known as rapid prototyping. With the help of inheritance software systems can be generated more quickly and easily is known as software prototype because the components that behave exactly what we needs had already done before.

f) **Polymorphism:** Polymorphism in programming language permits the programmer to generate high-level reusable component that can be suitable in different applications.

g) **Information hiding: The** programmer who reuses the software component needs only to understand the nature and behaviors of the particular component 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 because the inheritance provides interface of different objects the more the objects the more the execution time required. The programmer should spend more time while developing same output.
2) **Program size:** Due to inheritance the programming codes become very 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 more complexity. It is very difficult to understand. Inheritance adds more components to the child class.
4) **Message passing overloading:** Due to message passing among components the exception speed becomes less.

The is a Rule and the Has a Rules of Abstraction

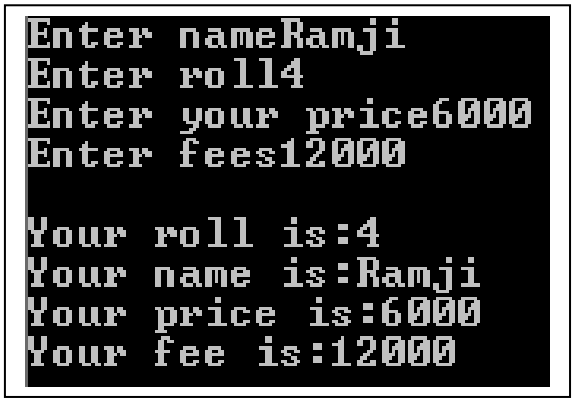
The idea of division into parts and division into specializations represent two most important forms of abstraction used in OOP. They are commonly known as is-a and has an abstraction. Division into parts is has an abstraction, means the term "**A car has an engine", "A bus has an engine. In such abstraction engine is common type they have specialized in terms of engine's variation. We cannot use engine the super class properties to all child classes because the engine has different for all transportation means.** There

are two relationships between the components of the system by knowing the relationship, we can easily understand and apply object oriented techniques. The has relationship occurred when second component of the first but both components are not same. In programming language similar type of inheritance concept is used that manage data and function associated within child classes are always 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 the animal classes can be inherited to the many sub classes. e.g. s A **dog is an animal, a cow is an animal**. In such case we can inherit some or all properties form the same class animal class. There relations derive from a simple rule that test the relationship between two concepts. In such, a case a way that animal behaves as super class for a dog and a cow are similar type objects. Therefore while design the program we should know the relationship among objects of components through analyzing the behaviors in multidimensional manner.

Is a relationship:

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




```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class employee{
char name[23];   int num;
public: void getdata(){ cout<<"\n Enter lastname";
cin>>name;          cout<<"Enter number is";cin>>num;}
void display(){ cout<<"\nName is"<<name;cout<<"\nNumber is=:"<<num;}};
class manager: public employee{   char title[23];
public: void getdata(){    employee::getdata();
cout<<"\nEnter title";cin>>title;}
void dispdata(){   employee::display();
cout<<"\nTitle is:"<<title;}   };
class scientist: public employee{   int papers;
public: void getdata(){  employee::getdata();
cout<<"\nEnter papers NO ";cin>>papers;}
void dispdata(){ employee::display();
cout<<"\nNumber of publication="<<papers;} };
void main(){    manager m;    scientist s;
cout<<"\nManager "; m.getdata();cout<<"\Data is:=";m.dispdata();
cout<<"\nScientist";s.getdata();cout<<"\Data is:";s.dispdata();
getch();}
```

//"A dog is a mammal" Here, a cow is an animal and a dog is an animal. The animal is super class. Its property can directly take to the any objects of same type classes. In the above code class employee is super class then, class manager and scientist are derived from the same parent class. While deriving the derived class may call all characteriscts of super class in addition to its own. Hence you can say that manager and scientist classes have "is kind of" relationship so inheritance is sometimes called "Kind of" relationship.

Has a Relationship

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

Similarly there is another relationship called "has a" relationship or containership. This relationship is occurs when we create object of super class inside derived class, rather than calling member function of old class. **"A car has an engine", a bus has an engine.** The engine is independent properties that cannot be directly called to the car, bus classes. While deriving from the super classes.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student{  int id; char name[20];
public: student(){ id=0;strcpy(name,"\0");}
student(int i,char *n){ id=i; strcpy(name,n);}
void getdata(){cout<<"Enter id and name";cin>>id>>name;
void display(){
cout<<"\nid is:"<<id<<"and name is"<<name;}  };
class M{   int makes[3];
public: student s;
marks(int a,int b,int c){  mks[0]=a; mks[1]=b; mks[2]=c;}
void  getdata(){  s.getdata();
for( int i=0;i<3;i++){
cout<<"\n Enter marks"<<i+1; cin>>mks[i]; }   }
void display(){     s.display();
for(int i=0;i<3;i++){   cout<<"\n Marks in subject";cout<<(i+1);cout<<"is";
cout<<mks[i];}   }  };
void main(){  M mm;
mm.getdata();  mm.display();
getch();    }
```

```
Enter id and name3
Rimal

 Enter marks178

 Enter marks290

 Enter marks387

id is:3and name isRimal
 Marks in subject1is78
 Marks in subject2is90
 Marks in subject3is87_
```

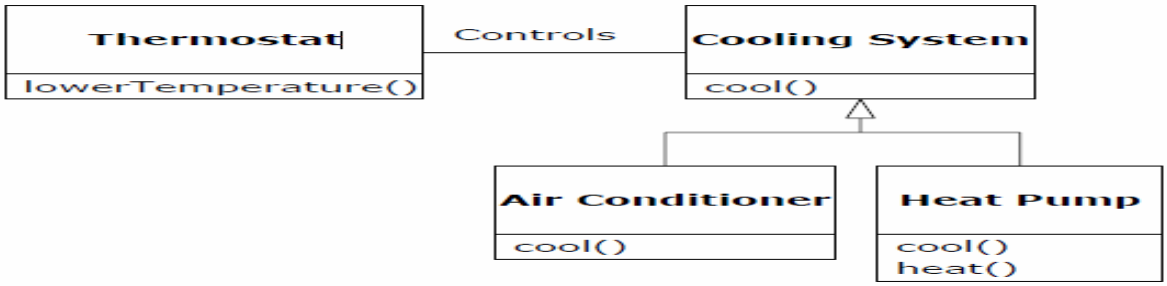
```
   #include<iostream.h>
   #include<conio.h>
   class Engine {
   public:void start() {
   cout<<"\nI am form engine and started";}
   void stop() {cout<<"\nI am form engine and stop";}  };
   class Wheel { public:   Engine e;
   void flat(){  e.start();
   cout<<"\nI am form Wheel class";} };
   class Door {   public:void open()  {cout<<"\nI am form Door class and OPen";}
   void close()  {cout<<"\nI am form Door and closed";}  };
   class Car {    public:Engine engine;     Wheel wheel;       Door d;   };
   void main() {clrscr();
   Car car;
   car.wheel.flat();
   car.d.open(); car.d.close();
   car.engine.stop();   getch();}
```

```
I am form engine and started
I am form Wheel class
I am form Door class and OPen
I am form Door and closed
I am form engine and stop
```

Is-a vs. is-like-a relationships (For reference reading)

There's a certain debate that can occur about inheritance: Should inheritance override only base-class methods (and not add new methods that aren't in the base class)? This would mean that the derived type is exactly the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base class. This can be thought of as pure substitution, and it's often referred to as the substitution principle. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an is-a relationship, because you can say "a circle is a shape." A test for inheritance is to determine whether you can state the is-a relationship about the classes and have it make sense. There are times when you must add new interface elements to a derived type, thus extending the interface and creating a new type. The new type can still be substituted for the base type, but the substitution isn't perfect because your new methods are not accessible from the base type. This can be described as an is-like-a relationship (my term). The new type

has the interface of the old type but it also contains other methods, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that allows you to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump is-like-an air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object has been extended, and the existing system doesn't know about anything except the original interface.



Of course, once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so that it can also include heating—at which point the substitution principle will work. However, the diagram above is an example of what can happen with design in the real world. When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it is nice if your design works out that way. But you'll find that there are times when it's equally clear that you must add new methods to the interface of a derived class. With inspection both cases should be reasonably obvious.

Class Hierarchies

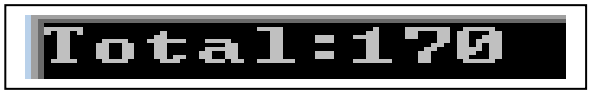
```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class employee{  char name[23];  int num;
public: void getdata(){ cout<<"\n Enter lastname";
cin>>name;        cout<<"Enter number is\t";cin>>num;}
void dispdata(){ cout<<"\nName is:\t"<<name;
cout<<"\nNumber is=:"<<num;}};
class manager: public employee{   char title[23];
public: void getdata(){   employee::getdata();
cout<<"\nEnter title";cin>>title;}
void dispdata(){   employee::dispdata();
cout<<"\nTitle is:"<<title;}  };
class scientist: public employee{      int papers;
public:   void getdata(){  employee::getdata();
cout<<"\nEnter papers NO ";cin>>papers;}
void dispdata(){   employee::dispdata();
cout<<"\nNumber of publication="<<papers;}    };
void main(){ manager m;  scientist s;
cout<<"\nManager "; m.getdata();
cout<<"\Data is:=";m.dispdata();
cout<<"\nScientist";s.getdata();
cout<<"\Data is:=";s.dispdata();
getch();}
```

```
Manager
 Enter lastnameSapkota
Enter number is 2

Enter titleMr.
Data is:=
Name is:          Sapkota
Number is=:2
Title is:Mr.
Scientist
 Enter lastnameDhurba
Enter number is 3

Enter papers NO 3456
Data is:=
Name is:          Dhurba
Number is=:3
Number of publication=3456
```

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 in nature the 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);//we cannot crate any instance of class A
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: **class 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: **class A {**

```
virtual void f() = 0;};
class B : A { virtual void f() { }};      // Error: Class A is an abstract class,same function
A g();    // Error: Class A is an abstract class
void h(A);  // Error: Class A is an abstract class
A& i(A&);   // Error: Class A is an abstract class
int main() {  A; // Error: Class A is an abstract class
A* pa;  B 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 no abstract 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) pure virtual functions 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 its 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 **a**bstract 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 {   float dim1, dim2;
public:  void setarea(float d1, float d2){ dim1 = d1; dim2 = d2;}
void getdim(float &d1, float &d2){ d1=dim1;d2=dim2; }
virtual float getarea() = 0; };  // pure virtual function
class rectangle : public area {
public:  float getarea(){  float d1, d2;
getdim(d1, d2);    cout<<"\nd1:"<<d1<<"\td2:"<<d2;
return d1 * d2;} };
class triangle : public area {
public: float  getarea(){ float d1, d2;
getdim(d1, d2);       cout<<"\nd1:"<<d1<<"\td2:"<<d2;
return 0.5 * d1 * d2; } };
void main() { area *p,*pp;    //object of abstract class
rectangle r;   triangle t;  r.setarea(3.3, 4.5);  t.setarea(4.0, 5.0);
pp = &r;  cout <<"\nRectangle has area: " << pp->getarea() << '\n';
pp = &t;   cout <<"\nTriangle has area: " << pp->getarea() << '\n';
p = &r;   cout <<"\rectangle has area: " << p->getarea() << '\n';
p = &t;  cout <<"\nTriangle has area: " << p->getarea() << '\n';
getch();}
```

```
d1:3.3  d2:4.5
Rectangle has area: 14.85

d1:4    d2:5
Triangle has area: 10

ectangle has area: 14.85

d1:4    d2:5
Triangle has area: 10
```

Subclass Subtype and Substitutability

A class which inherits the features of upper class is known as subclass. Subclass provides a way of constructing new components by 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 can never hold a value of string in derived class. 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

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

Substitutability

The principle of substitutability referred to the relationship between same variable name declared in parent class and sub class. The concept of substitutability is provided by inheritance. Substitutability is a feature of programming in which certain behavior 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. There's a certain debate that can occur about inheritance, inheritance override *only* base-class functions This would mean that the derived type is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute


```
Enter speed of car70
```

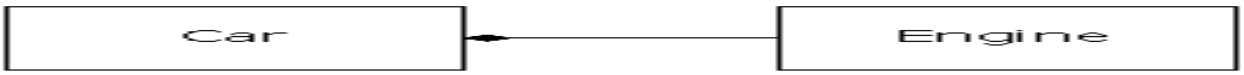
an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often referred to as the *substitution principle*.

```
#include<iostream.h>
#include<conio.h>
class car { int speed;
public:  void give(){cout<<"\nEnter speed of car";
cin>>speed;}
void display(){ cout<<"\n Speed of car is"<<speed;}};
class truck{   int speed;   public:  void give(){
cout<<"\nEnter speed of truck";cin>>speed;}
void display(){ cout<<"\nSpeed of truck is:"<<speed;} };
class speed : public car, public truck{
public:  void give(){car:: give();truck::give();}
void display(){car::display();truck::display();} };
void main(){  speed s;s.give();   s.display(); getch();}
```

Composition and Inheritance

We know object of a class is simply a concept of substitution of data and behaviors of that class, The concept of composition is applied to reduce existing data value of new data type. The parts of new data structure are simply the instance of existing data structure. The composition clearly indicates what operations can be called on a particular data structure.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class. Because you are composing a new class from existing classes, this concept is called *composition* (or more generally, *aggregation*). Composition is often referred to as a "has-a" relationship, as in "a car has an engine."



(The above UML diagram indicates composition with the filled diamond, which states there is one car. I will typically use a simpler form: just a line, without the diamond, to indicate an association.

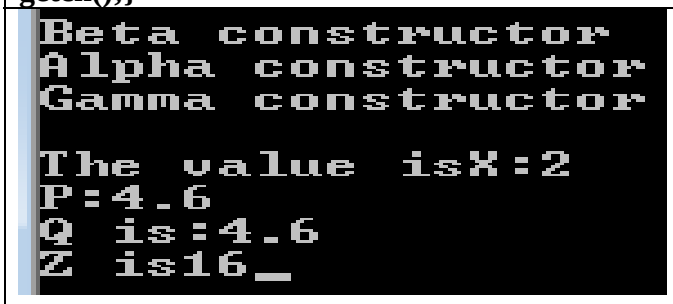
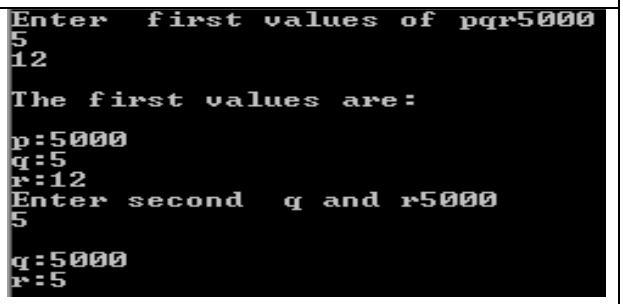
Composition comes with a great deal of flexibility. The member objects of your new class are usually private, making them inaccessible to the client programmers who are using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at runtime, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance. Because inheritance is so important in object-oriented programming it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overly-complicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will stay cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

The data members are **private** in this class, so it's completely safe to embed an object of type **X** as a **public** object in a new class, which makes the interface straightforward:

Comparison between Composition and Inheritance

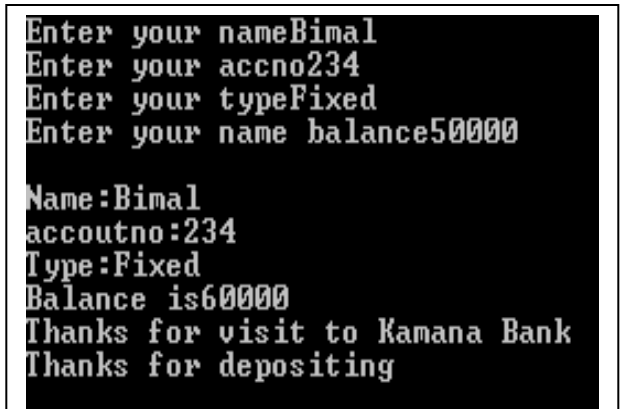
Composition	**Inheritance**
1.Composition indicates the operation of an existing structure.	1.Inheritance is a super set of existing structure when object is located.
2.In composition cannot reuse code directly but provide greater functionality.	2.Inheritance can be directly reused the code and function provided by parent class.
3.It is code become shorter then inheritance	3.Its code become longer than composition

4.It is very easy to re-implement the behaviors and functions.
Syntax: derived class Function name(arguments): base class(argument), base class(argument);
**#include<iostream.h>
#include<conio.h>
class alpha{ int x; public:
alpha(int i){ x=i; cout<<"\nAlpha constructor"; }
void showa(){ cout<<"\nThe value isX:"<<x;}};
class beta{ float p,q; public:
beta(float a, float b): p(a*2), q(b*2){
cout<<"\nBeta constructor";}
void showb(){ cout<<"\nP:"<<p; cout<<"\nQ is:"<<q;} };
class gamma : public beta, public alpha{
int x,y,z;
public: gamma(int a, float b, float c): alpha(a),beta(c,c),z(b*4){
cout<<"\nGamma constructor";}
void showg(){cout<<"\nZ is"<<z;} };
void main(){ gamma g(2,4,2.3); cout<<"\n";
g.showa(); g.showb(); g.showg();
getch();}**
	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;
```



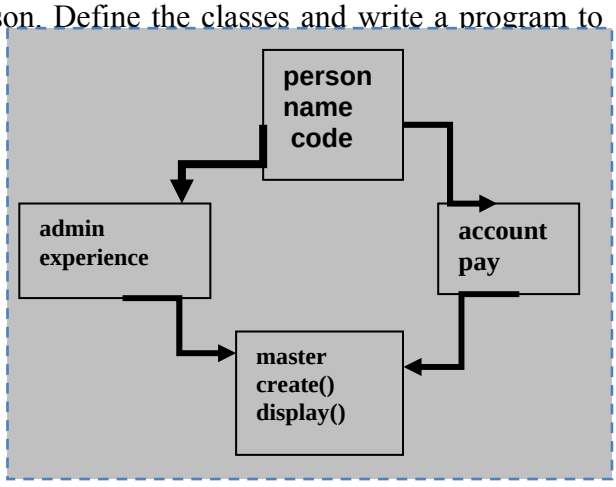
```
cout<<"Balance is"<<amount;
cout<<"Thanks for visit to Kamana Bank";}};
void main(){   clrscr();
bbalance a1; a1.create();
a1.deposi(10000);
a1.display();   cout<<"\nThanks for depositing\n";
a1.withdraw(2000); cout<<"Thanks for withdraw";
 a1.display();  cout<<"Final thanks";
getch();}
```

Create a class city that have two member variables city name char [30], distance form in float, add member functions to setdata() 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 Pokhara";  cin>>dfktm;  }
void retrive(){  cout<<"Cityname is:"<<name;
cout<<"Distance form Pokhara"<<dfktm; }
float adddistance(city c1, city c2){
float d;// temporary variable
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();}
```

```
Enter CitynameKathmandu
Enter distance form Pokhara200
Enter CitynameDhanghadi
Enter distance form Pokhara800
Cityname is:KathmanduDistance form Pokhara200
Cityname is:DhanghadiDistance form Pokhara800
The total distance form Kathmandu: 1000
```

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

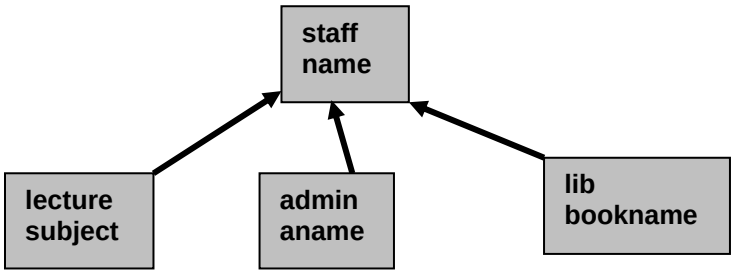



```
#include<iostream.h>
#include<conio.h>
class person{
protected:     char name[30]; int code; };
class admin: virtual public person{
protected: int exp;};
class account:  virtual public person{
protected: float pay;};
class master: public admin, public account{
public:  void create(){   cout<<"Enter 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;
```

```
Enter your name:Suresh
Enter code:124
Enter experience:5
Enter pay:60000

Name is:Suresh
```

```
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 derived in numbers of classes whose relationship is shown in below figure. That specifies all the classes and display constructor and function top create database and retrieve the individual information as 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();}
```

```
Your id is:1
Your nameis:Yagya
Your subject is:Computer
Your admname is:ChandParkah
You book name is:Object Oriented in C++
```

Create a class time represent with a member to take represent second, minute, hour. The time function tread 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();}
```

```
hh:1mm:1ss:41
```

Polymorphism

In programming languages, *polymorphism* means that some code/ operations or objects that behave differently in different contexts. For example + (plus) operator in C++ may be used as

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

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 in above code 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 design a member functions for each that makes each shape. Another convenient approach is that 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 of desired shape, by inserting pointer into the defined array. This approach leads to different function executed by the same function call. Polymorphism is used to give different meanings to the same concept. Polymorphism is the basic of *Virtual function* implementation. In polymorphism, a single function or an operator functioning in many ways depends upon the use to function. 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 concept the shape of (circle, triangle, and rectangle) are from the single base class called Shape.
- The member function must be declared virtual in the base class.

Features / 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 will not to recompile original programs by adding new types again and again. Only re-linking is necessary to give the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, it is necessary of creating for adding and customizing to the old program. By utilizing the concept of polymorphism, time, money and work effort is greatly reduced in addition it makes future maintenance easier too.
- 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:

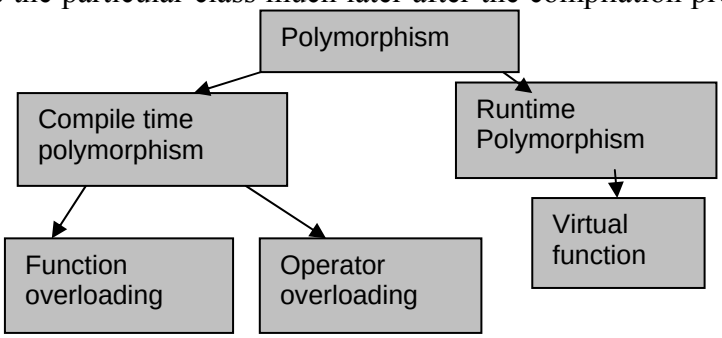
- run-time
- compile-time
- ad-hoc polymorphism
- parametric polymorphism

Other types of polymorphism defined:

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 compile time. And therefore, compiler is able to select the appropriate function for a particular call at the compile 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.



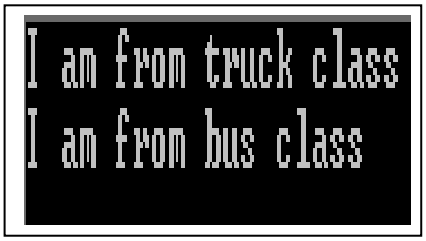
The dynamic binding is one of the powerful techniques it requires the use of pointers objects,

Run-Time (RLVDP)

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. The operator overloading and simple example of class are the example.




```
#include<iostream.h>
#include<conio.h>
class truck{
public:virtual void display(){cout<<"I am from truck class\n";}};
class bus: public truck{
public:void display(){cout<<"I am from bus class\n";}};
void main(){
```

RUNTIME Polymorp

```
truck *tc;
tc=new truck;
tc->display();
tc=new bus;
tc->display();
getch(); }
```

```
truck t;
bus b;
t.display();
b.display();
getch();}
```

Compile time

Late Binding	Early Binding
First compile the whole code the run the program while running it requires some information then process them respectively.	Compilers compiles all code once then run the program, while running it does not requires any further information.
It is like as dynamic binding because we may give input in run mode.	It is like as static binding because all the required information's are already set out.
Errors cannot be easily solved if program become long.	Errors can be easily solved
Virtual function and pointer are used	No virtual function and pointers are used
It halts the program if required data is mismatched	It does not halt the program while running
#include<iostream.h> **#include<conio.h>** **int sum(int a,int b){** **return (a+b);}** **void main(){** **int a,b,su; cout<<"Enter two number";cin>>a>>b;** **su=sum(a,b);** **cout<<"the sum is="<<su;** **getch();}** Enter two number5 7 the sum is=12	**#include<iostream.h>** **#include<conio.h>** **int sum(int a,int b){** **return (a+b);}** **void main(){** **int su;** **su=sum(10,20);** **cout<<"the sum is="<<su;** **getch();}** the sum is=30

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 overloading is 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 using notation "closer to the target call and allows user-defined 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. We can overloaded all C++ operator except.*, ::, sizefo and ?.

The mechanism of giving special meaning to an operator is known as operator overloading. To define additional task to an operator we must specify what is means in relation to the class to which the operator is applied. This is done with the help of a special function called operator function can defined as.

return type operator op(){}
class name operator op(argument, argument){} and called as
return type class:: operator op(arguments list){
….. ;;;;;; }

Here return type is value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. Operator op is function name. The operator functions must be either function (non-static) or fiend functions. The friend function will have one argument for unary operators and two for binary operators while a member functions have no arguments for unary operators and only one for binary operators this is not in case with friend functions. The arguments are passed either by value or by references.

vector operator +(vector);//vector addition
vector operator -();//unary –
friend vector operator +(vector, vector);
vector operator-(vector&a);
int operator ==(vector);
fiend int operator ==(vector,vector);

The process of overloading involves following steps:

First create a class that defines the data type that is to be in the overloading operations. Declare the operator function operator op() in the public part of class. It may be either a member function of friend function. Define the operator function can be invoked by expressions op x or x op; for unary x op y for binary and for friend functions : operator op(y);

Binary over loadable operators

Binary operators (operators with two arguments) are overloaded by declaring a function with an "identifier" *operator (something)* which calls one single argument. The variable on the left of the operator is the sender while that on the right is the argument.

integer i = 1; /* we can initialize a structure variable this way as
if calling a constructor with only the first argument specified. */
integer j = 3;
/* variable names are independent of the names of the member variables of the structure. */
integer k = i * j; cout << k.i << endl;
'4' would be printed.

The following is a list of binary overloadable operators:

Operator	General usage
+ - * / %	Arithmetic calculation
<< >>	Bitwise calculation
< > == != <= >=	Logical comparison
&&	Logical conjunction
= <<= >>=	Compound assignment
,	(no general usage)

The '=' (assignment) operator between two variables of the same structure type is overloaded by default to copy the entire content of the variables from one to another. It can be overwritten with something else, if necessary.

Operators must be overloaded one by one, in other words, no overloading is associated with one another. For example, < is not necessarily the opposite of >.

Unary overloadable operators

While some operators, as specified above, takes two terms, sender on the left and the argument on the right, some operators have only one argument - the sender, and they are said to be "unary". Examples are the negative sign (when nothing is put on the left of it) and the "logical NOT" (exclamation mark, !).

Sender of unary operators may be on the left or on the right of the operator. The following is a list of unary overloadable operators:

Operator	General usage	Position of sender
+ -	Positive / negative sign	right
* &	Dereference	right
! ~	Logical / bitwise NOT	right
++ --	Pre-increment / decrement	right
++ --	Post-increment / decrement	left

The syntax of an overloading of a unary operator, where the sender is on the right, is as follows:

return_type operator@ ()

When the sender is on the left, the declaration is: *return_type* operator@ (int)

@ above stands for the operator to be overloaded. Replace return_type with the data type of the return value (int, bool, structures etc.) The int parameter essentially means nothing but a convention to show that the sender is on the left of the operator. const arguments can be added to the end of the declaration if applicable.

Overloading brackets

The square bracket [] and the round bracket () can be overloaded in C++ structures. The square bracket must contain exactly one argument, while the round bracket can contain any specific number of arguments, or no arguments.

The following declaration overloads the square bracket. *Return_type* operator[] (*argument*)

The content inside the bracket is specified in the argument part.

Round bracket is overloaded a similar way. *return_type* operator() (*arg1, arg2, ...*)

Contents of the bracket in the operator call are specified in the second bracket.

In addition to the operators specified above, the arrow operator (->), the starred arrow (->*), the new keyword and the delete keyword can also be overloaded. These memory-or-pointer-related operators must process memory-allocating functions after overloading. Like the assignment (=) operator, they are also overloaded by default if no specific declaration is made.

Constructors

Sometimes software engineers may want their variables to take a default or specific value upon declaration. This can be done by declaring constructors. **person (string N, int A) { name = N; age = A;}**

Member variables can be initialized in an initializer list, with utilization of a colon, as in the example below. This differs from the above in that it initializes (using the constructor), rather than using the assignment operator. This is more efficient for class types, since it just needs to be constructed directly; whereas with assignment, they must be first initialized using the default constructor, and then assigned a different value. Also some types (like references and const types) cannot be assigned to and therefore must be initialized in the initializer list. **person (std::string N, int A) : name (N), age (A) {}**

Note that the curly braces cannot be omitted, even if empty.

Default values can be given to the last arguments to help initializing default values.

person (std::string N = "", int A = 0) : name (N), age (A) {}

When no arguments are given to the constructor in the example above, it is equivalent to calling the following constructor with no arguments (a default constructor): person () : name (""), age (0) {}

The declaration of a constructor looks like a function with the name as the same as the data type. In fact, we can really call the constructor in form of a function call. In that case a person type variable would be the return value: **int main () { person r = person ("Wales", 40); r.print ();}**

The above code creates a temporary person object, and then assigns it to r using the copy constructor. A better way of creating the object (without unnecessary copying) is:

int main () { person r ("Wales", 40); r.print ();}

Specific program actions, which may or may not relate to the variable, can be added as part of the constructor. **person () { std::cout << "Hello!" << endl;}**

With the above constructor, a "Hello!" will be printed in case a person variable with no specific value is initialized.

Destructors

A destructor is the reverse of a constructor. It is called when an instance of a class is destroyed, e.g. when an object of a class created in a block (set of curly braces "{}") is deleted after the closing brace, then the destructor is called automatically. It will be called upon emptying of the memory location storing the variable. Destructors can be used to release resources, such as heap-allocated memory and opened files when an instance of that class is destroyed. The syntax for declaring a destructor is similar to that of a constructor. There is no return value and the name of the method is the same as the name of the class with a tilde (~) in front. **~person () { cout << "I'm deleting " << name << " with age " << age << endl;}**

Unary Operator

An unary operator, that takes just one operand, this operator changes when applied to a basic data items. It can be applied in int or float variable. The unary minus minus is applied to an object should changes the sing of each of its data items when it is called inside main method.

(-)Unary Operator Overloading

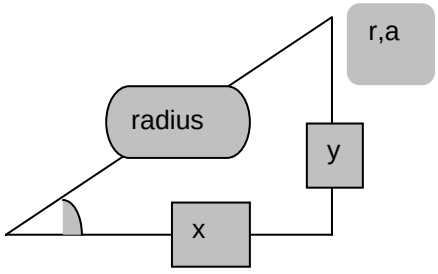
```
before oberloaded
p is:30
q is:-50
ris:40
After oberloadedp is:-30
q is:50
ris:-40
```

```
#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<<"\after overloaded"; -u;//overloading -operator
u.display(); getch(); }
```

In above program –ve operator is overloaded - 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.

Design a class polar which describes a point in the plane using polar coordinate radius and angle. A point in polar coordinates. (Note: 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)




```
cosx:10
sinex:5.25322
Tot x is15.2532
cosy:10
siney:5.25322
Y is:8.50904
r is17.4661
the a is1.06194
Radius10
Angle0
Radius10
Angle45
Radius17.4661
Angle1.06194
```

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class polar{
public: double radius,angle;
double getx(){return radius*cos(angle); }//polar to rectangular
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<<"\nRadius"<<radius; cout<<"\nAngle"<<angle; }
```

```
polar operator+(polar p2){cout<<"\ncosx:"<<getx();cout<<"\nsinex:"<<p2.getx(); double
x=getx()+p2.getx();
cout<<"\nTot x is"<<x;
double y=gety()+p2.gety();
cout<<"\ncosy:"<<getx();cout<<"\nsiney:"<<p2.getx();cout<<"\nY is:"<<y;
double r=sqrt(x*x+y*y); cout<<"\nr is"<<r;//to convert  rectangle to polar
double a=atan(x/y); cout<<"\nthe a is"<<a; return polar(r,a);}};
void main(){  polar p1(10.0,0.0);  polar p2(10.0,45);  polar p3;
p3=p1+p2;
p1.display();          p2.display();          p3.display();
getch();     }
```

Créate a class distance a constructor having two arguments, the distance are in feet and inches are taken form user. Then add distance operatopr function to add two user supplied distance respectively + Operator

```
Enter Feet inches 23
45
Enter Feet inches 4
14

First
feet is24       33
Second
feet is5        2
 sum is
feet is30       23_
```

```
#include<iostream.h>
#include<conio.h>
class distance{
private:    int feet;    float inches;
public:    distance(){    feet=0;    inches=0.0;}
distance(int ft,float i){   feet=f; inches=i;}
void getdist(){       cout<<"Enter Feet inches ";
cin>>feet>>inches;
while(inches>=12){      feet++;     inches-=12;}       }
void showdist(){
cout<<"\feet is"<<feet<<"\t"<<inches;}
// ~distance(){cout<<"Object destroyed";}
distance operator+(distance); };
distance distance::operator+(distance d){
distance d1;  d1.feet=feet+d.feet;
d1.inches=inches+d.inches;
if( d1.inches>=12.0){ d1.inches-=12.0;d1.feet++;} return d1; }
void main(){
distance dist1,dist2,dist3;//three times called distance function
dist1.getdist();      dist2.getdist();
dist3=dist1+dist2;
cout<<"\nFirst";dist1.showdist();
cout<<"\nSecond";dist2.showdist();
cout<<"\n sum is";dist3.showdist();
getch();  }
```

```
Enter hours:3
Enter minitus46
Enter second55
Enter hours:2
Enter minitus34
Enter second56
3:46:55
2:34:56
6:21:51
```

Time constructor overloading

```
#include<iostream.h>
#include<conio.h>
class time{
private : int hrs,mins,sec;
public:  time(){hrs=0;mins=0;sec=0;}
time(int h,int m,int s){hrs=h;mins=m;sec=s;}
void getdata()  {  cout<<"Enter hours:";cin>>hrs;  cout<<"Enter minitus";cin>>mins;
cout<<"Enter second";cin>>sec;
while (sec>=60){sec=sec-=60; mins++;}
while(mins>59){ mins=mins-=60; hrs++;}     }
```

```
void display(){  cout<<hrs<<":"<<mins<<":"<<sec<<endl;}
time operator+(time t1){ time t2;
t2.sec=sec+t1.sec;     t2.mins=mins+t1.mins;   t2.hrs=hrs+t1.hrs;
if(t2.sec>=60){t2.sec-=60;t2.mins++;}
if(t2.mins>=60){t2.mins-=60;t2.hrs++;}    return t2;}  };
void main(){
time time1, time2, time3;
time1.getdata();  time2.getdata();
time1.display();  time2.display();
time3=time1+time2;
time3.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 <<"\n"<<y<<endl;}
void main(){ clrscr();
complex c1,c2,c3;
c1=complex(2.5,3.5);   c2=complex(1.6,2.7);
c3=c1+c2;
cout<<"c1";c1.display();   cout<<"c2";c2.display();
cout<<"c3";c3.display();
getch(); }
```

```
c12.5    3.5
c21.6    2.7
c34.1    6.2
```

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 functions have only one parameter. The data member of c1 is access directly and c2 is access by using dot operator.

Complex complex:: operator+(complex c){

It receives only one complex type argument explicitly

It returns a complex value

It is a member function of complex class.

c3	**= c1+c2**
4.1(temp.x)	**= 2.5(c.x)+1.6(x)**
6.3(temp.y)	**=3.5(c.y)+2.7(y)**

```
Enter Feet inches 12
34

dist2=11        6.25
after addition
dist1=24        28.25
```

+= Operator Overloading

```
#include<iostream.h>
#include<conio.h>
class distance{
private:    int feet;    float inches;
public:    distance(){    feet=0;     inches=0.0;}
distance(int ft,float in){       feet=ft; inches=in;}
void getdist(){       cout<<"Enter Feet inches ";       cin>>feet>>inches;}
void showdist(){     cout<<feet<<"\t"<<inches;}
```

```
void operator +=(distance);   };
void distance::operator+=(distance d2){      feet+=d2.feet; inches+=d2.inches;
if( inches>=12){ inches-=12.0;feet++;} }
void main(){  distance dist1,dist2(11,6.25);
dist1.getdist();
cout<<"\ndist2="; dist2.showdist();
dist1+=dist2;
cout<<"\nafter addition";
cout<<"\ndist1=";dist1.showdist();
getch();  }
```

++ Operator Overloading

```
Original value of a: 1, 2, 3
Value after ++a: 2, 3, 4
```

```
#include <iostream>
#include<conio.h>
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(); }
```

++ Operator overloading

```
enter how many do u want
7
1,2,3,5,8,13,21,
```

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class test{  long int f1,f2,fib;
public:test(){    f1=0;      f2=1;       fib=f1+f2;}
void display(){ cout<<fib;   }
void operator++()    {f1=f2;        f2=fib;     fib=f1+f2;}
~test()     {}};
void main(){  clrscr();  test t;  int n;
cout<<"enter how many do u want"<<endl;  cin>>n;
for(int i=1;i<=n;i++)       {  t.display();  ++t; }getch();          }
```

Overloading Binary Operator using Friends

A friend class may use in the place of member functions for overloading a binary operator. The only difference being that a friend function requires two arguments to be explicitly passed to it while a member function requires only one. in the complex example above

friend complex operator+(complex a, complex b);

complex operator +(complex a, complex b){ return complex(a.x+b.x), a.y+b.y));}

incase of c3= c1+c2   c3=operator+ (c1,c3);

```
Enter elemetnes of m vector1
2
3


 m=(1,2,3)
m=(2,4,6)
```

```
#include<iostream.h>
#include<conio.h>
const size=3;
```

```
class vector{     int v[size];
public:    vector();  vector(int * x);
friend vector operator *(int a, vector b);
friend vector operator*(vector b, int a);
friend istream & operator>>(istream &, vector &);
friend ostream & operator<<(ostream &, vector &);   };
vector::vector(){       for(int i=0;i<size;i++)       v[i]=0;}
vector:: vector(int * x){     for(int i=0;i<size;i++)     v[i]=x[i];}
vector operator *( int a, vector b){vector c;
for(int i=0;i<size;i++)     c.v[i]=a*b.v[i]; return c;}
vector operator *(vector b,int a){         vector c;
for(int i=0;i<size;i++)   c.v[i]= (b.v[i])*a; return c;}
istream & operator>>(istream& in, vector&b){
for(int i=0;i<size;i++)    in>>b.v[i];    return in;}
ostream& operator<<(ostream& dout, vector &b) {   dout<<"("<<b.v[0];
for(int i=1;i<size;i++)   dout<<","<<b.v[i];   dout<<")";       return (dout);}
int x[size]={2,4,6};
void main(){  vector m;   vector n=x;
cout<<"\nEnter elements of m vector";
cin>>m;    cout<<"\n";cout<<" m="<<m;
vector p,q;
p=2*m;    q=n*2;
cout<<"\np="<<p<<"\n";    cout<<"\nq="<<q<<"\n";
getch();   }
```

The program overloads the operator * two times thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like other overloaded functions based on the types of its arguments. Iostream and ostream are the classes available inside iostream.h header files.

Overloading comparison operator <

```
#include<iostream.h>
#include<conio.h>
class distance{
private:    int feet;    float inches;
public:     distance(){    feet=0;     inches=0.0;}
distance(int ft,float in){        feet=ft; inches=in;}
void getdist(){        cout<<"Enter Feet inches ";
cin>>feet>>inches;}
void showdist(){    cout<<feet<<"\t"<<inches;}
bool operator < (distance d2){     float bf1=feet+(inches)/12;
float bf2=d2.feet+(d2.inches)/12;
return (bf1<bf2)? true:false;}              };
void main(){
distance dist1,dist2;
dist1.getdist();
dist2.getdist();
cout<<"\ndist1="; dist1.showdist();
cout<<"\ndist2="; dist2.showdist();
if(dist1<dist2) cout<<"\n dist1 is less than dist2";
```

```
Enter Feet inches 12
34
Enter Feet inches 23
42

dist1=12        34
dist2=23        42
 dist1 is less than dist2
```

```
The value will be: 10
The valeue2 will be: 20
The value will be: 50
The valeue2 will be: 60
The value will be: 50
The valeue2 will be: 50
The value will be: 50
```

```
else cout<<"dist1 is greater than dist2";
getch(); }
Operator loading=
#include<iostream.h>
#include<conio.h>
class test{      int value,value1;
public:          test(int a, int b){      value=a;       value1=b;   }
void display(){         cout<<"\nThe value will be: "<<value<<endl ;
cout<<"\nThe valeue2 will be: "<<value1<<endl; }
void operator=(test t)          {        value=t.value;          value1=t.value;}};
void main(){             clrscr();
test t(10,20);         test t1(50,60);
t.display();          t1.display();
t=t1;
t.display();          t1.display();
getch();    }
```

```
Enter employee no3
Enter employee nameSunita
Enter employee salary4000
Enter employee no123
Enter employee nameBinita
Enter employee salary4000

 Average salary is4000_
```

```
Average of two Object Salary
#include<iostream.h>
#include<conio.h>
class employee{
private: int enu; char name[23];   int basic;
public: void getdata(){   cout<<"Enter employee no";cin>>enu;
cout<<"Enter employee name";cin>>name;
cout<<"Enter employee salary"; cin>>basic; }
void display(){   cout<<"Employee no"<<enu;
cout<<"Name is"<<name;   cout<<"Salary is"<<basic;}
float cal(employee x){   float temp;    temp=(float(basic)+x.basic)/2;    return temp; } };
void main(){
employee a1,b1;
a1.getdata();   b1.getdata();
float average=a1.cal(b1);
cout<<"\n Average salary is"<<average;getch();   }
//overloading==
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string{  private: char *str;
public:   string(){str=new char[1]; strcpy(str,"\0");}
string(char*s){str=new char [strlen(s)+1],strcpy(str,s);}
~string(){delete str; cout<<"\nDestroyed previously";}
int operator==(string &st){  int a;
if(strcmp(str,st.str)==0)     return 1;
else   return 0;}
void display(){cout<<str<<endl;}
void getdata(){  cout<<"\nEnter string"<<endl;
cin.getline(str,'\n');}  };
void main(){   string s1,s2;
s1.getdata();   s2.getdata();
s1.display();  s2.display();
if(s1==s2)  cout<<"\nBoth string are equal"<<endl;
```

```
Destroyed previously
Enter string
Sunita

Enter string
Puja
Sunita
Puja

not equal_
```

```
Enter the feet: 23

enter the inch:43

The feet&incheses are:
The feet:=23    Inches:=43

Pre defined at constructor
```

```
else    cout<<"\not equal";
string s3;
getch();  }
```

Operator Overloading +=

```
#include<iostream.h>
#include<conio.h>
class test{      int feet;      float inch;
public:      test();
test(int f1,float in);
void takedata();
void display();
void operator+=(test t);};
test::test(){      feet=0;     inch=0.0;}
test::test(int ft,float in)     { feet=ft;     inch=in;     }
void test::takedata()     {     cout<<"\nEnter the feet: ";
cin>>feet ;     cout<<"\nenter the inch:";  cin>>inch;    }
void test::display(){ cout<<"\nThe feet&incheses are:";
cout<<"\nThe feet:="<<feet<<"\tInches:="<<inch<<endl;        }
void test::operator +=(test t){      feet += t.feet;        inch +=t.inch;
while(inch >12.0){       inch -=12.0;    feet++;     } }
void main(){     clrscr();       test t;       t.takedata();
test t1(10,20.25);
t.display();       cout<<"\nPre defined at constructor\n" ;
t1.display();
t += t1;       cout<<"After used+=\n";
t.display();     getch();    }
```

```
Enter string
Pokharacity
Enter string
kathmanducity
Pokharacity
kathmanducity

String are not equal
```

//==Operator Overloading

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string{  private: char *str;
public:       string(){str=new char[0]; strcpy(str,"\0"); }
string(char *s){  str=new char[strlen(s)+1]; strcpy(str,s);    }
~string(){delete str; cout<<"\nDestroyed";}
int operator==(string &st){       int a;
if(strcmp(str,st.str)==0)        return 1;        else return 0;    }
void display(){        cout<<str<<endl;}
void getdata(){       cout<<"Enter string"<<endl;        cin>>str;}};
void main(){       clrscr();
string s1,s2;
s1.getdata();      s2.getdata();      s1.display();       s2.display();
if(s1==s2)      cout<<"\nBoth strings are equal";
else cout<<"\nString are not equal";
getch(); }
```

```
Enter Feet inches 23
54
Enter Feet inches 3
65

feet is24        42
feet is4         53
```

-= Operator Overloading

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

```
private:    int feet;     float inches;
public:  distance(){   feet=0;    inches=0.0;}
distance(int f,float i){   feet=f; inches=i;}
void getdist(){       cout<<"Enter Feet inches ";
cin>>feet>>inches;
while(inches>=12){feet++; inches-=12;}       }
void showdist(){    cout<<"\nfeet is"<<feet<<"\t"<<inches;}
//~distance(){cout<<"Object destroyed";}
void operator-=(distance); };
void distance::operator-=(distance d){   //distance d1;
feet-=d.feet;   inches-=d.inches;
if( inches>=12.0){ inches-=12.0;feet++;} }
void main(){   distance dist1(10,13.5),dist2;
dist1.showdist();    dist2.getdist();    dist2.showdist();
dist1 -= dist2;
dist1.showdist();getch();  }
```

String manipulation using operator overloading + and <= operator overloading

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string {
char *p;    int len;
public:  string(){len=0;p=0;}
string(char *s);
string (string&s);
~ string(){delete p;}
friend string operator+(const string &z,const string &t);
friend int operator<=(string & s,string &t );
friend void show(string s);   };
string:: string(char*s){ len=strlen(s);
p=new char[len+1];     strcpy(p,s);}
string:: string(string &s){ len=s.len;   p=new char[len+1];
strcpy(p,s.p);}
string operator+( const string &s, const string &t){
string temp;
temp.len=s.len+t.len;
temp.p=new char[temp.len+1];
strcpy(temp.p,s.p);   strcat(temp.p,t.p);    return(temp);}
int operator<=(string&s,string &t){
int m=strlen(s.p);     int n=strlen(t.p);
if(m<=n) return (1);
else return(0);}
void show( string s){   cout<<s.p;}
void main(){ clrscr();
string s1="New";    string s2="York";    string s3="Delhi";
string t1,t2,t3,t4;
t1=s1; t2=s2;       t3=s1+s2;       t4=s1+s3;
cout<<"\n T1=";show(t1);  cout<<"\n T2=";show(t2); cout<<"\n";
cout<<"\n T3=";show(t3);cout<<"\n T4=";show(t4);    cout<<"\n";
if(t3<=t4){  show(t3); cout<<"Smaller than"; show(t4);     }
```

```
 T1=New
 T2=York

NewDelhi
 T4=NewDelhi
NewDelhismaller than
```

```
else{ show(t4); cout<<"smaller than ";show(t3);cout<<"\n";}
getch();}
```

Ruels for Overloading

- Only existing operators can be overloaded. New operators cannot created.
- We cannot change the basic meaning of an operator. That is we cannot redefined the plus operator to subtract one value form the other.
- Overloaded operators fellow the syntax rules of the original operators. That cannot be overridden.
- Binary operators oberloaded through a friend function take two explicit arguments.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

Type conversion	Conversion takes
Basic to class	Constructor
Class to basic	Operator
Class to class	Constructor

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 right side 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 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 one type to another type by applying type conversion rule provide by compiler. **The automatic type conversion means that the right side of data is automatically converted to left side type. This type of conversion rule cannot apply in case of user defined data type**. In user defined data type, we must define conversion routines ourselves. The possible type conversions are as follows:

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 use of constructor. The compiler is built to time object. Note: that the constructor used for the type conversion takes a single argument whose type is to be converted.

```
#include<iostream.h>
#include<conio.h>
class time{  int h,m;
public: time(){h=0;m=0;}
        time(int t){ h=t/60;   m=t%60;}              //constructor
     void display(){cout<<"Hour:"<<h;  cout<<"Minutes"<<m;}};
void main(){time t1;   int a=4566;
t1=a;    //type conversion
t1.display();
getch();}
```

```
Hour:76Minutes6
```

In above the constructor time (int t) converts a basic type into 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 from 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 calls the casting operator function to do the job, casting operator function should satisfied:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
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(){              feet=0; inches =0;}//constructor
distance(float meter){//constructor with one argument
float ft= m*meter;              feet=int(ft);   //convert meter to feet
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;//meter
    cout<<"Distanceis :\n";
    d.show(); d=1.00;   cout<<"\ndistancwe";
    d.show();
    distance d2(10,10.25);//feet,inches
    float met=float(d2);  cout<<"\ndistance";
    cout<<met;  met=d2; cout<<met;
    getch();}
```

```
Distanceis :
7       8.51949
distancwe3      3.37
distance3.308363.30836
```

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 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.

A data conversion example

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class invent1{
int code;    int items;      float price;
public:  invent1(int a,int b,float c){
```

```
Product detail=
Code=100
Items=5
Price=140
stock ValueValue is=700
Product details is=
```

```
code=a; items=b;price=c;}
void putdata(){   cout<<"\nCode="<<code;
cout<<"\nItems="<<items;
cout<<"\nPrice="<<price;}
int getcode(){return code;}
int getitems(){return items;}
int getprice(){return price;}
operator float(){return (items*price);}  };
class invent2{  int code;     float value;
public:invent2(){   code=0; value=0;}
invent2(int x,float y){  code=x; value=y;}
void putdata(){ cout<<"\nCode="<<code;  cout<<"\nValue="<<value;}
invent2(invent1 p){ code=p.getcode();
 value=p.getitems()*p.getprice();}  };
void main(){   clrscr();
invent1 s1(100,5,140.0);
invent2 d1;  float total_value;
total_value=s1; d1=s1;
cout<<"\nProduct detail="; s1.putdata();
cout<<"\nstock Value";cout<<"Value is="<<total_value;
cout<<"\nProduct details is=";
d1.putdata();getch(); }
```

Pointer to Objects

We have already seen how to use pointer to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider item x;

Where item is a class and x is object defined to be of type item. Similarly we can defined a pointer int *ptr;

The object pointer is useful in creating objects at runtime, we also use an object pointer to accesses the public members of an object.

```
#include<iostream.h>
#include<conio.h>
class item{ int code;float price;
public: void getdata(int a,float b){code=a; price=b;}
void show(){cout<<""code"<<code;cout<<"price"<<price;}};
```

let us declared item variable x and a pointer ptr to x

as item x; item*ptr=&x;

the ptr is initialized with the address of x. we can refer to member function of item in two ways, one by using the dot operator and the object and another by using the arrow operator to object.

x.getdata(100,55,5);

x.show(); are equivalent ptr->getdata(100,55.5);

ptr->show(); since *ptr is an alias of x we can also use the following method.

(*ptr).show(); the paranethissi is necessary because the dot operator has higher precedence than the indirection operator*.

We can also create objects using pointers and new operators

Item*ptr=new item;

This statements allocates enough memory for the data members in objects structure and assignment address of memory space ptr. Then ptr can be used to refer to other members at shown below. Ptr->show();

If a class has constructor with arguments and doesn't include any empty constructor then we must supply the arguments when the object is created. we can also create an array of objects using pointers.

int *item[10]; creates memory space for an any of 10 objects of items. If a class contains constructor it must also contain an empty constructor**.**

```
#include<iostream.h>
#include<conio.h>
class item{  int code;  float price;
public: void getdata(int a,float b){
code=a; price=b;}
void show(){cout<<"\ncode:"<<code;
cout<<"\nprice:"<<price;}};
const int size=3;
void main(){
item *p=new item[size];
item *d=p;
int x,i; float y;
for(i=0;i<size;i++){
cout<<"Input code and price" <<(i+1);
cin>>x>>y;
p->getdata(x,y);  p++;}
for(i=0;i<size;i++){  cout<<"\nItems:\t"<<(i+1);
d->show();  d++;}   getch();}
#include<iostream.h>
#include<conio.h>
#include<cstring.h>
class city{
protected:char *name;int len;
public:city(){ len=0;name=new name[len+1];}
void getname(){char *s;s=new char[30];
cout<<"Enter city name";cin>>s;
len=strlen(s); name=new char[len+1];
strcpy(name,s);}
void printname(){
cout<<name<<"\n";}};
void main(){
city *ct[10];   int n=1;  int option;
do{ct[n]=new city;
ct[n]->getname();n++;
cout<<"Do you want to enter one more name\n";
cout<<"Enter 1 for yes and 0 for exit";
cin>>option;}
while(option);   cout<<"\n";
for(int i=1;i<=n;i++){
ct[i]->printname();}
getch();}
```

```
Enter city namePokhara
Do you want to enter one more name
Enter 1 for yes and 0 for exit1
Enter city nameKathmandu
Do you want to enter one more name
Enter 1 for yes and 0 for exit1
Enter city nameDhangadi
Do you want to enter one more name
Enter 1 for yes and 0 for exit1
Enter city nameJanakpur
Do you want to enter one more name
Enter 1 for yes and 0 for exit0

Pokhara
Kathmandu
Dhangadi
Janakpur
```

this Pointer

C++ use unique keyword called this to represents an object this invokes a member function. This is a pointer that points for which this function was called. For example function a.mas() will set the pointer this to address of the object a. the starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member when it is called. This pointer this acts an implicitly arguments to all the members function. int a; a=123; or we can also use this->a=123;

Since C++ permits the use of shorthand form a=123; we have not been using the pointer this explicitly so far, however we have been implicitly using the pointer the pointer this when overloading the operator using

member function. When binary operator overloaded using the member function we pass only one argument to the function. The other argument is implicitly passed using the pointer this. Once important application of the pointer this is to return the object is points to. return * this;
Inside the member function will return this object that invoked the function. This function and return the invoking object as result.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person{
char name[20];  float age;
public: person(char*s, float a){
strcpy(name, s); age=a; }
person & person::grater(person & x){
if(x.age>=age) return x; else return *this;}
void display(){  cout<<"\nName is"<<name;   cout<<"age is"<<age;} };
int main(){  person p1("Yagya",32),p2("Bimala",22),p3("Bindu",24);
person p=p1.grater(p3); cout<<"\nElder person";
p.display(); p=p1.grater(p2);
cout<<"\nElder person is\n";
p.display();   getch();}
```

```
Elder person
Name isYagyaage is32
Elder person is

Name isYagyaage is32
```

Accessing Data with this

When you call a member function it comes into existence with the value of this set to the address of the objects for which it was called. The this pointer can be treated like any other pointer to an object and can thus be used to access the data in the object it points to as shown in

```
#include<iostream.h>
#include<conio.h>
class what{  int alpha;
public: void tester(){  this->alpha=11;//same as alpha=11
cout<<this->alpha;  }};              //same as cout<<alpha
void main(){what w;  w.tester();
getch();}  output: 11
```

Using Returning Value

A more practical use for this is in returning values form members functions and oberloaded operations.
The object of which a function is a member is more permanent then its individual member functions an objects member functions are created and destroyed every time they are called, but the object itself endures until it is destroyed by outside .thus returning by reference the object the object of which a member of function.

```
#include<iostream.h>
#include<conio.h>
class alpha{  int data;
public:  alpha(){}
alpha(int d){data=d;}
void display(){cout<<data;}
alpha& operator=(alpha & a){data=a.data;
cout<<"\assignment operator is called";
return*this;}};
void main(){ alpha a1(37), a2,a3;
a3=a2=a1; cout<<"\na2=";a2.display(); cout<<"\na3=";a3.display();
getch();}
```

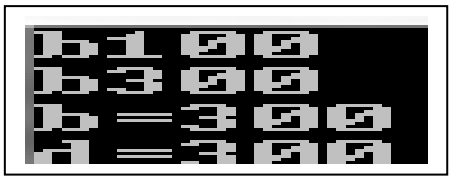
```
Assignment operator is called
Assignment operator is called
a2=37
a3=37
```

Pointer to Derived Class

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

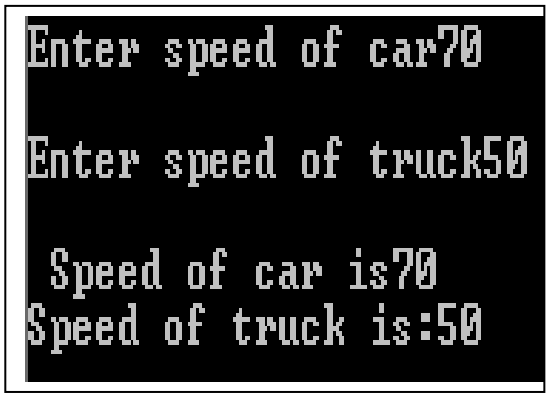


```
#include<iostream.h>
#include<conio.h>
class B{  public: int b;
void show(){ cout<<"b"<<b<<endl;}};
class D: public B{  public: int d;
void show(){cout<<"b="<<b<<endl<<"d="<<d;}};
void main(){ B *ptr; //pointer object
B b;  //normal object
ptr=&b;//passing normal to pointer object
ptr->b=100;
ptr->show();
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 variables are possible to use only by using pointer or references objects. For example:

```
int a=10;  float b=20.4; int *p=&b;
```




```
#include<iostream.h>
#include<conio.h>
class car { int speed;
public:  void give(){cout<<"\nEnter speed of car";
cin>>speed;}
void display(){ cout<<"\n Speed of car is"<<speed;}};
class truck{   int speed;
public:  void give(){
cout<<"\nEnter speed of truck";cin>>speed;}
void display(){
cout<<"\nSpeed of truck is:"<<speed;} };
class speed : public car, public truck{
public:  void give(){car:: give();truck::give();}
```

```
void display(){car::display();truck::display();} };
void main(){  speed s;
s.give();   s.display();getch();}
```

```
Enter code and price5
6000
Code5price6000
```

```
#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. Their 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 example:

```
float divide(int s, int y)
float divide( float x, float y)
```

```
Area of rectangle110
Area of circle:103.816
```

```
#include<iostream.h>
#include<conio.h>
class shape{  float a,l,b,r;
public:  void area(float length, float brea
            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 child 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 the message.

Method overriding, in object oriented programming, is a language feature that allows a superior child class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes. The implementation in the subclass overrides (replaces) the implementation in the super class 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{  double length;     double width;
public:     void print();};
```

```
void rectangleType::print(){cout<<"Enter L and H"; // print() version of base class
cin>>length>>width;
cout << "\nLength = " << length << "\nWidth = " << width;}
class boxType : public rectangleType{
double height;
public:     void print();          };
void boxType::print()  // print() version of derived class
{   cout<<"Enter Height"; cin>>height;
rectangleType::print();  // Invoke parent version of method prin
cout << "\nHeight= " << height;}
void main(){boxType r;
r.print();  //r.rectangleType::print();
getch();}
```

```
Enter Height45
Enter L and H76
90

Length = 76
Width = 90
Height= 45
```

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 polymorphic 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 keyword 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();}
```

```
pointer to base
display Base class
 show base class
 pointer to derived claps
display Base class
 show derived class_
```

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 Function

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 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 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 to derived object it will not make it to point the next objects of derived class.in such case we create another object of base 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

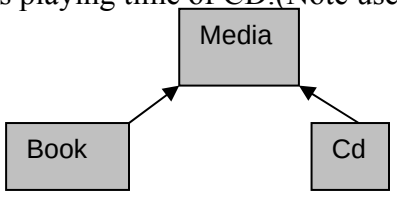
A Pure Virtual Function (Deferred Method)

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 called deferred method or pure polymorphism .This method does not have directly implementation. We use pointer object to passed values to that object.

Syntax: **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.(Note use deferred method)




```
#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);
```

```
Title:Computer:
Price:400
Page:890
Title:Objects Oriented
Price Rs:179
Timetoplay:80
```

```
m=&b;   m->display();
m=&c;   m->display();
getch();   }
```

Create a class pet having two member functions speak and eat. A derived class dog is used both members functions speak and eat virtually. The deferred method used both instance of real world objects Richard's dog poppy and Rimal's dog tommy.

```
#include <iostream.h>
#include<conio.h>
class Pet {
public:  virtual void speak()  = 0;
virtual void eat()  = 0;  };
void Pet::eat()  {  cout << "sarap sarap" << endl;  }
void Pet::speak()  {  cout << "Hau Hau" << endl;   }
class Dog : public Pet {
public:    void speak()  { Pet::speak(); }
void eat()  { Pet::eat(); } };
int main() {
Dog poppy;  // Richard's dog
poppy.speak();    poppy.eat();
getch();}
```

```
Pet *p,*pp;
Dog poppy;  // Richard's
dog
Dog g;
pp=& g;  //Shyam's dog
p=& poppy;
p->speak();
p->eat();
pp->speak();
pp->eat();
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: friend return type function name(class obj){}

```
class test{…;;
int fun();          };
class test2{  ;  ;
friend int test:: fun (){ }};
```

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

Some features are:

- It cannot access by the object of class
- It cannot be called using object of that class.
- It can invoke like normal function without help of object
- It cannot access member data directly. Object name dot membership operator
- It should use data member of class by object of class.
- It can be declared anywhere in the program
- It can declared either public or private part of class without affecting its meanings.
- Usually it has objects as arguments

```
#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 of a and b outside the class even they are private members. The function call mean avg passes the object by value to the friend function. The member functions of one class can be fiend function of another class. In each case they are defined using the scope resolution operators as

```
class x{
int fun();   };
class y{    fiend int x:: fun();  };
```

The function fun is member of class x and friend of class y. we can declared all the member function of one class as the friend of another class.

```
Enter the number23
I  am getting pvt member:23
```

```
#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();}
```

Create classes called ABC and class XYZ with each of having private members. Add member function to set a value say setvalue on each class. Add one more function max that is friendly to both classes. Max function should compare two private members of each class and show maximum among them. Create one object of each class then set a value on them. Display the maximum numbers among them.

```
#include<iostream.h>
#include<conio.h>
class ABC;//forward declaration
class XYZ{  int x;
public: void setvalue(int a){x=a;}
friend void max(XYZ,ABC); };
class ABC{  int a;
public: void setvalue(int i){a=i;}
friend void max(XYZ,ABC);  };
void max(XYZ m,ABC n){
if(m.x>n.a) cout<< m.x;
else  cout<<n.a;}
void main(){
ABC a;XYZ p;
a.setvalue(50);  p.setvalue(20);
cout<<"The Max:";max(p,a);
getch();}
Output:The Max : 50
```

```
Value exchange before
Value 1 100
Value 2 200
After exchange
Value 1 200
Value 2 100
```

Swap Private Data of Classes

In case of pointer to the address of the object is passed and the called function directly works on the actual object used in the call. This method can be used to alter the values of private members of a class. Remember altering the values of private members is assigned the basic principles of data hiding. A friend function to exchange private values of two classes.

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

```
#include<conio.h>
class class2;
class class1{  int val;
public: void indata(int a){val=a;}
void display(){ cout<<"\nValue 1\t"<<val;}
friend void exchange(class1,class2);   };
class class2{  int val2;
public: void indata(int a){val2=a;}
void display(){ cout<<"\nValue 2\t"<<val2;}
friend void exchange(class1,class2);   };
void exchange(class1 x,class2 y){
int temp=x.val;    x.val=y.val2;   y.val2=temp;}
void main(){  class1 c1;   class2 c2;
c1.indata(100);   c2.indata(200);
cout<<"\nValue exchange before ";
c1.display(); c2.display();
exchange(c1,c2);
cout<<"\nAfter exchange";
c1.display();    c2.display();
getch();}
```

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*. Friends are functions or classes declared with the friend keyword. If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
#include<iostream.h>
#include<conio.h>
class CRectangle {    int width, height;
public:      void set_values (int, int);
int area () {return (width * height);}
friend CRectangle duplicate (CRectangle);};
void CRectangle::set_values (int a, int b) {    width = a;  height = b;}
CRectangle duplicate (CRectangle rectparam){
CRectangle rectres;
rectres.width = rectparam.width*2;    rectres.height = rectparam.height*2;
return (rectres);}
int main () {
CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout << rectb.area();
getch();   return 0;}  output:24
```

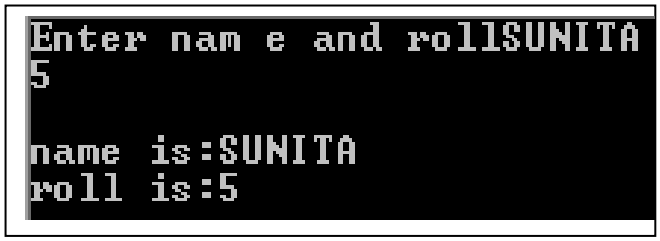
The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member. The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever

possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

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(); }
```



Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
#include <iostream.h>
#include<conio.h>
class CSquare;
class CRectangle {    int width, height;
public:    int area ()      {return (width * height);}
void convert (CSquare a);};
class CSquare {  private:    int side;
public:    void set_side (int a)      {side=a;}
friend class CRectangle;};
void CRectangle::convert (CSquare a) {  width = a.side;  height = a.side;}
int main () {
CSquare sqr;
CRectangle rect;       sqr.set_side(4);
rect.convert(sqr);
cout << rect.area();
getch();   return 0;} output: 16
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square. You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle. Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to. Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

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 or 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 know. In description of class that unknown parameter is matched & used. In C++ generic programming is implemented by template so templates are used to create a family of classes or functions. Generic programming use approach where generic types are used as parameters in algorithm so that they work for variety of suitable data types and data structure. 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 used, the template definition for that class or function is substituted with the required data type. The templates are parameters that can be replaced by specified data type of the time of actual used 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 templates are the mechanisms that make it possible to use one function or class to handle assign single class or function. The 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 whereas when used with class they are called class template.
Template can be used as a kind of macro. When an object of a specific type is defined fir actual use. The template definition for that class is substituted with the required data type. Sometime the template replaced by specific data type at the **time** of actual use of class and function. So template are called parameterized class and function.

```
#include<iostream.h>
#include<conio.h>
const size=3;
class vector{ int *v;int i;
public: vector(){v=new int[size];
for(i=0;i<size;i++)v[i]=0;}
vector(int *a){for(int i=0;i<size;i++)v[i]=a[i];}
int operator *(vector &y){int sum=0;
for(i=0;i<size;i++)sum+=this->v[i]*y.v[i];
return sum;}};
void main(){  int x[3]={1,2,3};  int y[3]={4,5,6};
vector v1;  vector  v2;
v1=x;   v2=y;
int r=v1*v2;cout<<r;
getch();} output: 32
```

Template Class

When a temple is used in class, 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>//we defined template with T use data type as t thn T may be replacd any data type
                  class classname{    };
                  void main(){
                  classname <type> object name----;      ----;   }
```

WAP to illustrate the use of vector class template for performing scalar product of int type vector s as well as float type vectors.

```
#include<iostream.h>
#include<conio.h>
const size=3;
template<class t>
class vector{ t *v; int i;
public: vector(){v=new t[size]; for(i=0;i<size;i++)v[i]=0;}
vector(t *a){for(int i=0;i<size;i++)v[i]=a[i];}
t operator *(vector &y){t sum=0; for(i=0;i<size;i++)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;  vector <int> v2;
v1=x;   v2=y;
int r=v1*v2;cout<<r;
getch();}
output 32(1*4+2*5+3*6)
#include<iostream.h>
#include<conio.h>
template<class T>
class vector{ T* v;     int i;
public: vector(){  v=new T[3];
for(i=0;i<3;i++) v[i]=0;  }
vector(T* a){for(i=0;i<3;i++)v[i]=a[i];}
T operator* (vector &y){T sum=0;for(i=0;i<3;i++)
sum+=this->v[i]*y.v[i];return sum;}  };
void main(){float x[3]={1.1,2.2,3.3};float y[3]={4.4,5.5,6.6};
vector<float>v1;vector<float>v2;
v1=x;  v2=y;
float R=v1*v2;
cout<<"\nR="<<R;
getch();  }
output 38.72(1.1*4.4+2.2*5.5+3.3*6.6)
```

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 because the template 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 object for char data. If you could generalize the stack template concept for their data type. After the template in the class stack is generalized, we 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 we include iostream.h in our program you 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 the 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. Template is a keyword in class that indicates data type is the parameter. 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 declaring we can give the class specifies using the data type as t. The template is stored in separate header files to execute code reusability.

```
#include<iostream.h>
Const int max=3;
template <class t>
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(); }
```

Class template with multiple parameters two generic data types in a class definition

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>
class Test{ T1 a;   T2 b;
public:  Test(T1 x,T2 y){a=x;  b=y;  }
void show(){cout<<a<<"\tand\t"<<b<<"\n";}};
int main(){Test<float,int>test1(1.23,123);
Test<int,char>test2(100,'W');
test1.show();test2.show();
getch();return 0;  };
```

output:1.23 and 123
100 and W

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<iostream.h>
#include<conio.h>
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 T[size=m];for(int i=0;i<size;i++) v[i]=0;}
template <class T>
vector<T>::vector(T* a){for(int i=0;i<size;i++)v[i]=a[i];}
template<class T>
T vector<T>::operator*(vector &y){T sum=0;   for(int i=0;i<size;i++)sum+=this->v[i]*y.v[i];
return sum;}
int 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();return 0;}
```

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 two arguments of integer type and return the value of the same type. What if we want to find the sum of two float, long int and double type, then we need to again redefine another function for those types to handle those data. We should rewrite the same code for different data types are time consuming process. If we have errors in a code then we need to check the errors in each code functions. It would be better if we can just write the one function and reuse it for all other types.

```
#include<iostream>
#include<conio>
template <class T>
void swap(T&x,T&y){T temp=x; x=y;y=temp;}
void fun(int m,int n,float a,float b){
cout<<"\nM and n : before"<<m<<"\t"<<n;
swap(m,n);
cout<<"\nM and n : after"<<m<<"\t"<<n;
cout<<"\nM and n : before"<<a<<"\t"<<b;
swap(a,b);
cout<<"\nM and n : after"<<a<<"\t"<<b;   }
int main(){
fun(100,200,11.22,33.44);
getch(); return 0;}
```

```
M and n : before100    200
M and n : after200     100
M and n : before11.22  33.44
M and n : after33.44   11.22
```

```
#include<iostream>
#include<conio>
template<class T>
void bubble(T a[],int n){
for (int i=0;i<n-1;i++) for(int j=n-1;i<j;j--)
if(a[j]<a[j-1]){swap(a[j],a[j-1]);}}
template <class T>
void swap(T &x,T &y){
T temp=x; x=y;y=temp;}
int main(){ int x[5]={10,50,30,40,20}; float y[5]={1.1,2.3,4.2,2.5,6.5};
bubble(x,5);bubble(y,5);
cout<<"\nSorted x=";for(int i=0;i<5;i++)
cout<<x[i]<<" ";cout<<endl;
cout<<"\nSorted Y=";for(int j=0;j<5;j++)
cout<<y[j]<<" "; cout<<endl;
getch();return 0;}
```

```
Sorted x=10 20 30 40 50

Sorted Y=1.1 2.3 2.5 4.2 6.5
```

```
#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;
```

```
sum is in int is:30
sum in float is:32.1
```

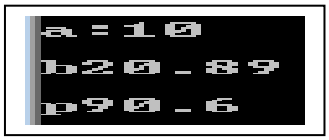
```
        cout<<"\nsum is in int is:"<<sum(i,j);
        float a=15.4,b=16.7;
        cout<<"\nsum in float is:"<<sum(a,b);
        getch();}
```

In 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 returns 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:

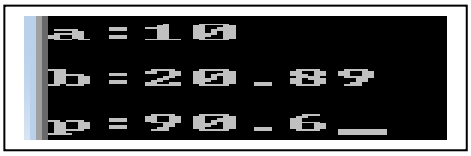
Function temple with multiple parameters

```
template< class T1, class T2>
return type function name(T1 arg,T2 arg){ function body}
```

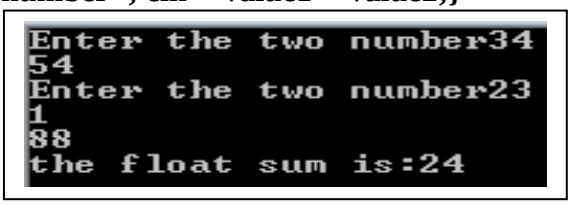
```
        #include<iostream.h>
        #include<conio.h>
        template<class T, class S, class Z>
void display(T a, S b, Z z){  cout<<"\na:"<<a<<endl;
cout<<"\nb"<<b<<endl;  cout<<"\np"<<z;}
        void main(){  clrscr();
        int i=10;  float j=20.89;    float p=90.6;
        display(i,j,p);getch();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




```
        #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(); }
```

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

```
sum of numbers45
sum of numbers25.6
sum of numbers45.4
```

```
#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();}
```

```
Integer cofficients are
 R1=3and R2=2

 Float cofficient

Real part is R1=-1.2and
Immaginary part R2=1.37598
```

```
#include<iostream>
#include<iomanip>
#include<math>
#include<conio>
template<class T>
void root(T a,T b,T c){  T d=b*b-(4*a*c);
if(d==0){   cout<<"R1=R2="<<-b/(2*a)<<endl;}
else if(d>0){cout<"\nRoots are real";
float R=sqrt(d);
float R1=(-b+R)/(2*a);
float R2=(-b-R)/(2*a);
cout<<"R1="<<R1<<"and " ;
cout<<"R2="<<R2<<endl;}
else{ //complex roots
cout<"\nRoots are complex";
float R1=-b/(2*a);
float R2=sqrt(-d)/(2*a);
cout<<"\nReal part is R1="<<R1<<"and " ;
cout<<"\nImmaginary part R2="<<R2<<endl;}}
int main(){
cout<<"\nInteger coefficients are\n ";
root(1,-5,6);
cout<<"\n Float coefficient\n";
root(1.5,3.6,5.0);  getch();return 0;}
```

Function with multiple parameters

```
#include<iostream>
```

```
#include<conio>
#include<string.h>
template <class T1,class T2>
void display(T1 x,T2 y){
cout<<x<<" "<<y<<"\n";}
int main(){
display(1999,"YAGYA");
display(12.4,1234);
getch();return 0;}
```

Exception Handling

Exception 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. ***Exception handling* is a mechanism that first detects and separate the code then handles exceptional circumstances from the rest of our program. Note that an exceptional circumstance is not necessarily an error.** When a function detects an exceptional situation, we represent this with an object. This object is called an *exception object*. In order to catch the exceptional situation you *throw the exception*. This process control, as well as generate the exception, to the particular block of code in a direct or indirectly then 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 compiler will pass to that object to the appropriate handler with is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *re-throw* 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)

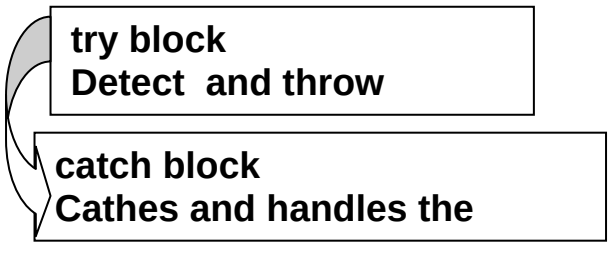
There are two kinds of exception

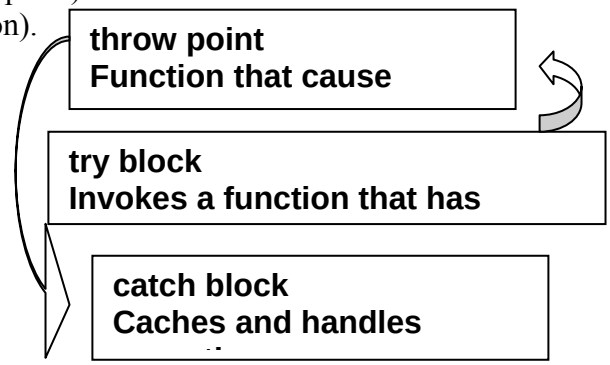
i. Synchronous exception (e.g. Out of range index overflow)
ii. Asynchronous exception (errors that are caused by events beyond the control of program are called asynchronous exception).

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).

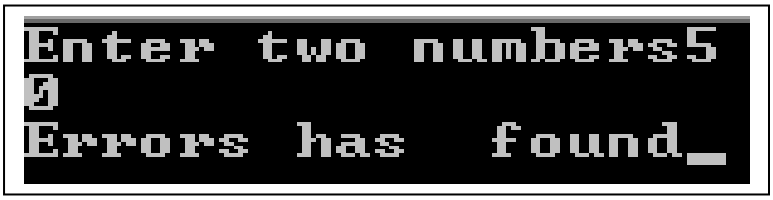




Syntax:

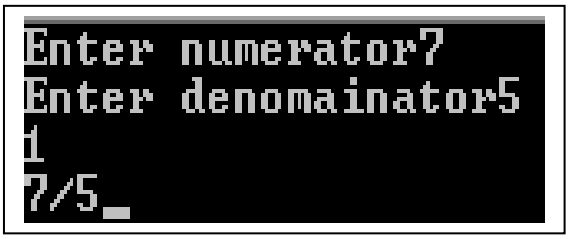
```
try{                          throw(obj)   }
throws exception }            try{    }
catch(args ){   }             catch(obj){  }
```

C++ exception handling mechanism is basically built upon three keywords namely try throw, and catch. The keyword try is used to preference a block of statements (surrounded by brace) which may generate execptions. This block of statements is known as try block. When an exception is detected, it is thrown using throw statements in the try block. When an exception is detected it is thrown using a throw statement in the try block. A catch block defined by the catch, the exceptions thrown by the throw statements in the try block and handles it appropriate. The catch block that catches an exception must immediately follow the try block that throws the exception.




```
#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 numerator";  cin>>a;
cout<<"Enter second enumerator";  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 denominator"; 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;}
```



Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
#include <iostream>
int main () {
try  {    throw 20;  }
catch (int e)  {
cout << "An exception occurred. Exception Nr. " << e << endl;  }
return 0;} An exception occurred. Exception Nr. 20
```

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception: throw 20; A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler. The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught. We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed. If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last: try { // code here}

```
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char. After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {     }     // code here
catch (int n) {      throw;  }}
catch (...) {  cout << "Exception occurred";}
```

Exception Specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration: float my function (char param) throw (int);. This declares a function called my function which takes one augments of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler. If this throw specified is left empty with no type, this means the function is not allowed to throw exceptions. A function with no throw specifies (regular functions) are allowed to throw exceptions with any type:

```
int my function (int param) throw (); // no exceptions allowed
int myfunction (int param);         // all exceptions allowed
```

Standard Exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
#include <iostream>
#include <exception>
```

```
using namespace std;
class myexception: public exception{
virtual const char* what() const throw()  {    return "My exception happened";  }
} myex;
int main () {  try
{    throw myex;  }
catch (exception& e)  {
cout << e.what() << endl;  }
return 0;}  Output: My exception happened.
```

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception. All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class. These are:

Exception	Description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typed
ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try{  int * myarray= new int[1000];}
catch (bad_alloc&){  cout << "Error allocating memory." << endl;}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a bad_alloc exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a bad_alloc exception. Because bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
int main () {  try  {
int* myarray= new int[1000];  }  catch (exception& e)  {
  cout << "Standard exception: " << e.what() << endl;  }
return 0;}
```

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 functions are called standard template library.

STL contents several components but at its core there are three components these three components works in conjunction with one another to provide support to a variety of programming solutions. The relationship between the components are as shown below.STL is very large and complex concept. There are some 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 program to make our program more robustness. STL contains several components but its core contains three components they are 1) Container 2) Algorithm 3) Iterates

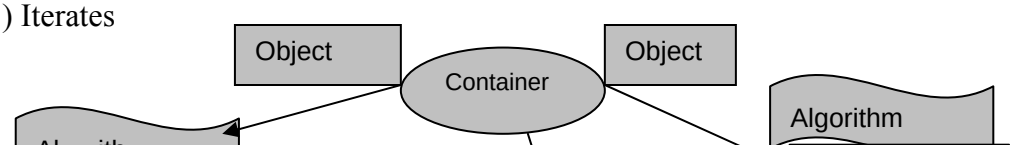


Fig: Relationship between STL and Components

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 data type. A container is an object that actually stores data. It is a way data is organized in memory. The stl containers are implemented by template classes and therefore can be easily customized to hold different data types.

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 kinds of algorithm to support different task like initializing, searching copying and sorting. The algorithm is implemented by template. An algorithum is procedure that is used to process that data contained in the containers. STL includes many different kinds of algorithum to provide support to take such as initializing, searching, copying, sorting and merging. Algorithms are implemented by templates functions.

Iterates: Iterates is an object that can be used with a container to gain access to particular elements in the container. It is generalize the concept of a pointer that points to elements in a container. Iterates are used to move through the containers. Iterates connects algorithm with containers. Iterators are an object (like pointer) that points to elements in containers. We can use iterators to move through the contents of containers. Iterators are handled just like pointers. We can increment or decrement them. Iterators connects algorithms s with containers and plays a key role in the manipulation of data stored in the 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

```
iterator p;//int p
for( p=container.beginb(); p!=container.end(); p++)//for loop
process_element p;
```

```
#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++)
```

```
Forward printing        A,B,C,D,E,F,
Reversed printing       F,E,D,C,B,A,
```

```
cout<<*p<<","; cout<<endl;
cout<<"Reversed printing\t";
reverse_iterator rp;
for(rp=container.rbegin(); rp!=container.rend();rp++)
cout<<*rp<<","; cout<<endl;
getch();}
```

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

```
#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();}
```

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();}
```

```
#include<vector.h>
#include<iostream.h>
#include<conio.h>
using namespace std;
void main(){
```

```
Enter numbers2
2
Added
v.size()13
3
Added
v.size()24
4
Added
```

```
vector<int>v;  cout<<"Enter numbers";
int next;  cin>>next;
while(next>0){
v.push_back(next);
cout<<next<<"\nAdded";
cout<<"v.size()"<<v.size();
cin>>next;}
for(unsigned i=0;i<v.size();i++)
cout<<v[i]<<",";
getch();}
```

Application of Container classes

There are three important are widely used in vector, list and map

Vectors: The vector is the most widely used container . it stores elements in contigenous memory location and enable s directly access to any element using the subscript operator[]. A vector can change its size dynamically and therefore allocates memory as need at runtime. The vector contains supports random access iterator and wide range of iterator operator may be applied to a vector iterator. Class vector supports member of constructor for creating vector objects

Vector<int> v1;
Vector<int> (v4)
Vector<int>v(5,2).

The vector class supports several member functions

Function	Task
at()	Give a reference to an element
back()	Give reference to last elements
begin()	Give reference to first element
erse()	Delete specific elements
pop_bsck()	Deletes last elements
size()	Gives the number of elements
swap()	Exchange elements in the specifed two vectors

```
#include<iostream>
#include<vector>
#include<conio>
using namespace std;
void display(vector<int>  &v){
 for( int i=0;i<v.size();i++){
cout<<v[i]<<" ";}
cout<<"\n";}
void main(){
        vector<int> v;
cout<<"Initialial size="<<v.size()<<"\n";
int x[5]; cout<<"Enter five integer";
for(int i=0;i<5;i++){ cin>>x[i];
v.push_back(x);}
cout<<"Size of aftr adding 5";
cout<<v.size()<<" \n"; cout<<"Current contents\n";
display(v);
v.push_back(6.6);
cout<<"\n Size ="<<v.size()<<"\n";
```

```
cout<<"contents are\n";
display(v);
vector<int>::iterator itr=v.begin();
itr=itr+3;
v.insert(itr,1,9);
cout<<"\n Display";
display(v);
v.erase(v.begin()+3,v.begin()+5);
cout<<"\n content after deletion";
display(v);
cout<<"End";
getch();}
```

List: List is another components that is popular used. It supports a bidirectional , linear list and provides an efficient implementation for deletion and insertion operations. Unlike a vector which supports random access, a list can be accessed sequentially only. Any algorithum that requires input output forward, bidirectional iterators can operate list. Class list provides many members functions for manipulating the elements of a list .

```
#include<iostream>
#include<list>
#include<cstdlib>
using namespace std;
void display(list<int>&lst){
list<int>::iterator p;
 for(p=lst.begin();p!=lst.end();++p)
cout<<*p<<",";
cout<<"\n";}
int main(){
        list<int>list1;
        list<int>list2(5);
        for(int i=0;i<3;i++)
                list1.push_back(rand()/100);
        list<int>::iterator p;
for(p=list2.begin();p!=list2.end();++p)
*p=rand()/100;
cout<<"List1\n";
display(list1);
cout<<"List2\n";
display(list2);
list1.push_front(100);
list1.push_back(200);
list2.pop_front();
cout<<"Now list\n";
display(list1);
cout<<"List2list\n";
display(list2);
list<int>lista,listb;
lista=list1;
listb=list2;
list1.merge(list2);
cout<<"Merged\n";
display(list1);
```

```
List1
0,184,63,
List2
265,191,157,114,293,
Now list
100,0,184,63,200,
List2list
191,157,114,293,
```

```
lista.sort();
listb.sort();
lista.merge(listb);
cout<<"Merged sorted\n";
display(lista);
lista.reverse();
cout<<"Reversed\n";
display(lista);
return (0);}
```

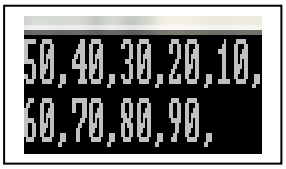
Map: A map is a sequences of keys and values pairs whre a single value is associated with each queue key. retrive of value is based on the key and is very fast. We should specify the key to obtain the associated value. A map is commonly called an associative array. The key is specific using the subscript operator{} .
Phone["Puja"]=1111;

```
#include<iostream.h>
#include<map>
#include<conio.h>
typedef map<string,int> phoneMap;
int main(){ string name;
int number;
phoneMap phone;
cout<<"Enter ";
for(int i=0;i<3;i++){
cin>>name>>number;
phone[name]=number;}
phone["Jabca"]=4444;//insert
phone.insert(pair<string,int>("Base",5555);
int n= phone.size();
cout<<"size"<<n;
phoneMap::iterator p;
for(p=phone.begin();p!=phone.end();p++){
cout<<(*p).first<<" "<<(*p).second<<"\n";    }
cout<<"\n";
cout<<"Enter name:";
cin >>name;
number=phone[name];
cout<<"Number"<<number;
return(0);}
```

Function Objects

A function object is a function that has been wrapped in a class so that it looks like an object. The class has only one member function. The overloaded operator and no data.

```
#include<iostream>
#include<algorithm>
#include<functional>
using namespace std;
int main(){ int z[9]; int i;int j;
int x[]={10,50,30,40,20};
int y[]={60,70,80,90};
sort(x,x+5,greater<int>());
sort(y,y+4);
```



```
for(i=0;i<5;i++)
        cout<<x[i]<<",";
cout<<"\n";
for(j=0;j<4;j++)
cout<<y[j]<<",";
z[9];
std::merge(x,x+5,y,y+4,z);
for( i=0;i<9;i++)
        cout<<z[i]<<",";
cout<<"\n";
return(0);}
```

Some Important Codes

Create class employee having data items number,name and basic salary, the two member function getdata and display are used to read information and display information of two employee. The cal is another member function that takes two objects as arguments that calculates highest and average of three employees among them.

```
        #include<iostream.h>
        #include<conio.h>
        class employee{
        private:  int num; char name[23];  float basic;
        public:   void getdata(){   cout<<"Enter no";cin>>num;   cout<<"Enter your name";
    cin>>name;
        cout<<"Enter Basic"; cin>>basic; }
        void display(){  cout<<"\nEmp name is:"<<name;  cout<<"\nEmp id is:"<<num;
        cout<<"\nSalary is:"<<basic; }
        float cal(employee x,employee y){  float temp;
        if((basic>=x.basic)&&(basic>=y.basic)){
        temp=basic; display();}
        else if((x.basic>= y.basic)&&(x.basic>=basic)){
        temp=x.basic; x.display();}
        else temp=x.basic;y.display();
        temp=(float)(basic+x.basic+ y.basic)/3;
        return temp;  }};
        void main(){   clrscr();
        employee a1,b1,c1;
        a1.getdata();  b1.getdata();   c1.getdata();
        float average=(a1.cal(b1,c1));
        cout<<"\nAverage"<<average;getch();  }
```

```
Enter no2
Enter your nameShyam
Enter Basic9000
Enter no3
Enter your nameDhurba
Enter Basic6000
Enter no1
Enter your nameUrmila
Enter Basic8000

Emp name is:Shyam
Emp id is:2
Salary is:9000
Emp name is:Urmila
Emp id is:1
Salary is:8000
Average7666.67
```

```
        #include<iostream.h>
        #include<conio.h>
        void main(){ int num[10],i,j,temp;
        cout<<"Enter 10 numbers"<<endl;
        for(i=0;i<10;i++)  cin>>num[i];
        for(i=0;i<9;i++){  for(j=i+1;j<10;j++){
        if(num[i]>num[j]){  temp=num[i];  num[i]=num[j];  num[j]=temp;}}}
        cout<<"Numbers in ascending order"<<endl;
        for(i=0;i<10;i++)  cout<<num[i]<<"\t";
        getch(); }
```

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

```
Enter 10 numbers
2
3
4
5
3
2
1
3
5
```

```
#include<conio.h>
#include<string.h>
void main(){ clrscr();
char name[50];
cout<<"input a string"<<endl;  cin>>name;    //cin.getline(name,'\n');
int n=strlen(name); cout<<n;
for(int i=n-1;i>=0;i--) cout<<name[i]<<endl;
getch(); }
```

```
Enter a stringYagyarimal
a       ###
e
i
o       #
u
```

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main(){ clrscr();
char s[80], row[5];
int a[]={0,0,0,0,0};
row[0]='a'; row[1]='e'; row[2]='i';
row[3]='o'; row[4]='u';
cout<<"Enter a string"; cin>>s;
for( int i=0;s[i]!='\0';i++){
switch(s[i]){
case 'A':
case 'a': a[0]++; break;
case 'E':
case 'e': a[1]++; break;
case 'I':
case 'i': a[3]++; break;
case 'O':
case 'o': a[4]++; break;
case 'U':
case 'u': a[5]++; break;
default: break;    }   }
for(int i=0;i<5;i++){ cout<<row[i]<<"\t";
for(int j=1;j<=a[i];j++) cout<<"#";
cout<<endl; }
getch(); }
```

```
#include<iostream.h>
#include<conio.h>
int main(){  clrscr(); int a;
cout<<"Enter number";  cin>>a;
cin.ignore();
cout<<"you typed"<<a<<"\n";
cin.get();
//getch();
return 1;  }
```

```
Enter number8
you typed8
```

```
#include<iostream.h>
#include<conio.h>
int main(){  clrscr();
int age;  cout<<"Enter age";cin>>age;
cin.ignore();
if(age<100)cout<<"you are preety young"<<age<<"\n";
else cout<<"you are really old";
```

```
Enter age23
you are preety young23
```

```
1
2
```

```
cin.get();
return 1; }
#include<iostream.h>
#include<conio.h>
int main(){  clrscr();
for(int i=1;i<=5;i++){  cout<<i<<endl;}
cin.get();
return 1;}
```

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is: **namespace identifier{entities }**

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:**namespace myNamespace { int a, b; }**

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write: myNamespace::a myNamespace::b

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
#include <iostream>
namespace first  {  int var = 5;}
namespace second{  double var = 3.1416;}
int main () {
cout << first::var << endl;
cout << second::var << endl;
return 0;}
```

5
3.1416

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

Using: The keyword using is used to introduce a name from a namespace into the current declarative region. For example:

```
#include <iostream>
using namespace std;
namespace first{  int x = 5;  int y = 10;}
namespace second{     double x = 3.1416;  double y = 2.7183;}
int main () {
using first::x;
using second::y;
cout << x << endl;
cout << y << endl;
cout << first::y << endl;
cout << second::x << endl;
return 0;}
```

5
2.7183
10
3.1416

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.
The keyword using can also be used as a directive to introduce an entire namespace:

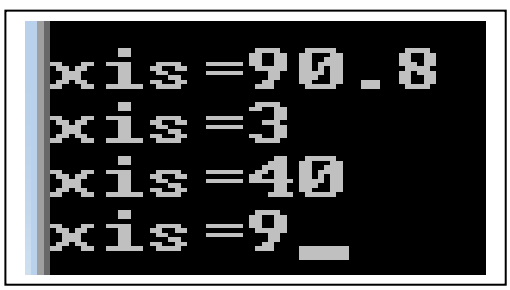
```
#include <iostream>
using namespace std;
namespace first{  int x = 5;  int y = 10;}
namespace second{
double x = 3.1416;
double y = 2.7183;}
int main () {
using namespace first;
cout << x << endl;
cout << y << endl;
cout << second::x << endl;
cout << second::y << endl;
return 0;}
```

5
10
3.1416
2.7183
In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers were referring to their declarations in namespace first.
using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
#include <iostream>
using namespace std;
namespace first{  int x = 5;}
namespace second{  double x = 3.1416;}
int main () {  {
using namespace first;
cout << x << endl;  }  {
using namespace second;
cout << x << endl;  }
return 0;}
```

5
3.1416

```
#include<iostream.h>
#include<conio.h>
namespace one{ float x=90.8;
int p=3;}
namespace two{ float x=40.0; int q=9;}
int main(){
cout<<"\nxis="<<one::x;
cout<<"\nxis="<<one::p;
cout<<"\nxis="<<two::x;
cout<<"\nxis="<<two::q;
getch();return 0;}
```



Namespace alias

We can declare alternate names for existing namespaces according to the following format: **namespace new_name = current_name;**

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

```
// An example of the using directive
#include <iostream.h>
namespace F {
float x = 9;       }
namespace G {
using namespace F;
float y = 2.0;
namespace INNER_G {
float z = 10.01;           }       }
int main() {
using namespace G;  // THIS DIRECTIVE GIVES YOU EVERYTHING DECLARED IN "G"
using namespace G::INNER_G;  // THIS DIRECTIVE GIVES YOU ONLY "INNER_G"
float x = 19.1;     // LOCAL DECLARATION TAKES PRECEDENCE
cout << "x = " << x << endl;
cout << "y = " << y << endl;
cout << "z = " << z << endl;
return 0;      }
Output:
x = 19.1
y = 2
z = 10.01
#include <iostream.h>
#include<conio.h>
inline int hello(){
int k;   k=10;
cout<<k;   return 0;  }
void main(){   clrscr();
hello();  cout<<"\nhello student";
getch(); }


#include<iostream.h>
#include<conio.h>
void swap(int *a,int *b);
void main(){   int a,b;
cout<<"Enter two numbers";   cin>>a>>b;
cout<<"after swap are"<<a<<"\t"<<b;   swap(&a,&b);
cout<<"After swap are "<<a<<"\t"<<b;
getch();   }
void swap(int *m,int *n){
int temp;  temp=*m;  *m=*n;  *n=temp;  }

#include<iostream.h>
```

```
#include<conio.h>
#include<string.h>
class string{
private: char *str;
public: string(){str=new char[1]; strcpy(str,"\0");}
string(char *s){str=new char[strlen(s)+1]; strcpy(str,s);}
void getdata(){    cout<<"Enter New string";    cin.getline(str,'\n');}
void display(){    cout<<"Entered string is"<<str<<endl;}
~string(){delete str;    cout<<"Destroyed";}   };
void main(){   clrscr();
string s1("Welcome"),s2;
s1.display();
s2.getdata();
s2.display();
cout<<"puja";
string(); getch();   }
```

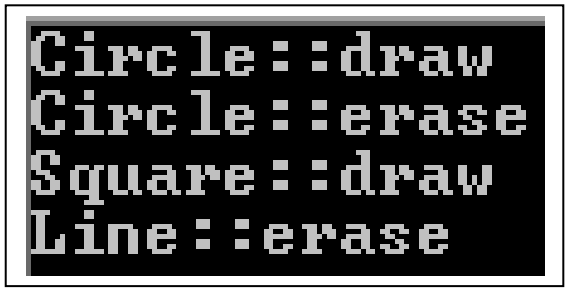
```
Enter titleObject

Enter price900

Enter number of pages450

Enter play time45

TitleObject
Price is900
Pages:450
Playingtime is45_
```

```
Publication derived class
#include<iostream.h>
#include<conio.h>
const int len=80;
class publication{
char title[len];   float price;
public: void getdata(){  cout<<"\nEnter title";cin>>title; cout<<"\nEnter price";cin>>price;}
void putdata(){  cout<<"\nTitle"<<title;   cout<<"\nPrice is"<<price;}  };
class book: private publication{int page;
public:  void getdata(){ publication::getdata(); cout<<"\nEnter number of pages";cin>>page;}
void putdata(){ publication:: putdata(); cout<<"\nPages:"<<page;} };
class tape: private publication{float time;
public:void getdata(){  cout<<"\nEnter play time"; cin>>time;}
void putdata(){  cout<<"\nPlayingtime is"<<time;}    };
void main(){book b; tape t;
b.getdata(); t.getdata(); b.putdata(); t.putdata();
getch(); }

//Mainpulate class
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
class student{
private: int id;   char name[20];   float fees;
public:  student(){id=0; strcpy(name,"\0"); fees=0.0;}
student(int i, char*n,float f){id=i; strcpy(name,n); fees=f;}
void getdata(){  cout<<"Enter id";cin>>id;  cout<<"Enter name ";  cin>>name;
cout<<"Enter fees";cin>>fees;}
void display(){ cout<<"ID is:"<<id<<"\tName is"<<name<<"\tFees
is"<<setiosflags(ios::fixed)<<setiosflags(ios::showpoint)<<setprecision(2)<<setw(10)<<fees;   }};
class marks: private student{
int marks[3];
public: marks(): student(){marks[0]=marks[1]=marks[2]=0;}
marks(int i, char*n,float f,int a, int b,int c): student(i,n,f){
```

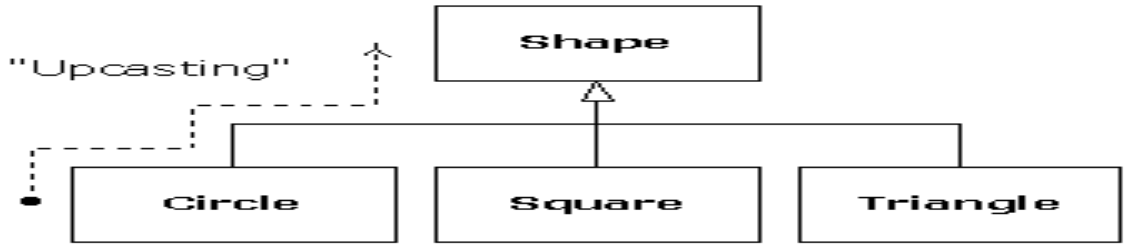
```
marks[0]=a;   marks[1]=b;marks[2]=c;}
void getdata(){ student::getdata();
for(int i=0;i<3;i++){   cout<<"\n Enter marks";cin>>marks[i];}}
void display(){  student:: display();
for(int i=0;i<3;i++)  cout<<"Marks is"<<marks[i];} };
void main(){
marks m1(9901,"Asha",4000,60,70,80);
m1.getdata();  m1.display();
getch();  }
```

```
#include<iostream.h>
#include<conio.h>
class person{char name[20];float age;
public:person(char *s,float a){
strcpy(name,s);age=a;}
person person:: grater(person &x){
if(x.age>age) return x;
else return *this;}
void display(){
cout<<"\nname is"<<name<<"\t"<< age;}  };
void main(){
person p1("Yagya", 33.5),p2("apple", 50.9),
p3("Help",30.9);
p=p1.grater(p3);
cout<<"Elder is\n";p.display();
p=p1.grater(p2);cout<<"Elder is\n";
p.display();}
```

```
#include<iostream.h>
#include<conio.h>
class Shape {
public:virtual void draw() = 0;
virtual void erase() = 0;
virtual ~Shape() {}};
class Circle : public Shape {
public:Circle() {}
~Circle() { cout << "Circle::~Circle\n"; }
void draw() { cout << "Circle::draw\n";}
void erase() { cout << "Circle::erase\n";}};
class Square : public Shape {
public:Square() {}
~Square() { cout << "Square::~Square\n"; }
void draw() { cout << "Square::draw\n";}
void erase() {cout << "Square::erase\n";}};
class Line : public Shape {
public:Line() {}
~Line() { cout << "Line::~Line\n"; }
void draw() {cout << "Line::draw\n";}
void erase() { cout << "Line::erase\n";}};
void main(){ Circle c;   Line l;   Square s;
c.draw();  c.erase();   s.draw();   l.erase();
getch(); }
```



Type *upcasting* :We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "up casting."



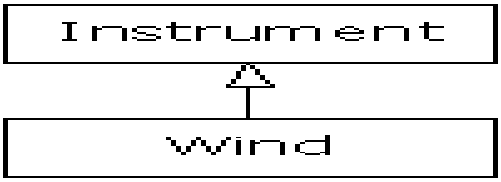
An object-oriented program contains some up casting somewhere, because that's how you decouple yourself from knowing about the exact type you're working with.

Notice that it doesn't say "If you're a **Circle**, does this, if you're a **Square**, do that, etc." If you write that kind of code, which checks for all the possible types that a **Shape** can actually be, it's messy and you need to change it every time you add a new kind of **Shape**. Here, you just say "You're a shape, I know you can **erase( )** and **draw( )** yourself, do it, and take care of the details correctly."

And yet the right thing happens because of polymorphism. The compiler and runtime system handle the details; all you need to know is that it happens and more importantly how to design with it. If a member function is **virtual**, then when you send a message to an object, the object will do the right thing, even when up casting is involved.

Why "up casting?"

The reason for the term is historical and is based on the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for **Instrument.cpp** is then:



Casting from derived to base moves *up* on the inheritance diagram, so it's commonly referred to as upcasting. Upcasting is always safe because you're going from a more specific type to a more general type – the only thing that can occur to the class interface is that it can lose member functions, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

Upcasting and the copy-constructor

If you allow the compiler to synthesize a copy-constructor for a derived class, it will automatically call the base-class copy-constructor, and then the copy-constructors for all the member objects (or perform a bitcopy on built-in types) so you'll get the right behavior:

```
#include<iostream.h>
#include<conio.h>
class Parent {int i;
public:Parent(int ii) : i(ii) {
cout << "\nParent(int ii)"<<i;   }
Parent( Parent& b) : i(b.i) {
cout << "\nParent( Parent&)"<<b.i;   }
Parent() : i(0) { cout << "\nParent()"<<i; }};
class Member {int i;
public:Member(int ii) : i(ii) {
cout << "\nMember(int ii)"<<ii;   }
```

```
Parent(int ii)2
Member(int ii)2
Child(int ii)2
calling from main:
```

```
    Member( Member& m) : i(m.i) {
    cout << "\nMember(Member&)"<<m.i;    }};
    class Child : public Parent {
    int i;Member m;
    public:Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "\nChild(int ii)"<<i;   }};
    void main() {Child c(2);
    cout << "\ncalling from main: " << endl;
    getch();}

#include<iostream.h>
#include<conio.h>
class person{char name[20];  float age;
public:  person(char *s,float a){strcpy(name,s);  age=a;}
person person:: grater(person &x){
if(x.age>age) return x;
else return *this;}
void display(){  cout<<"\nname is"<<name<<"\t"<< age;}   };
void main(){
person p1("Yagya", 33.5),p2("apple", 50.9), p3("Help",30.9);
person p('\0',0);
p=p1.grater(p3);  cout<<"Elder is\n";
p.display();
p=p1.grater(p2);
cout<<"Elder is\n";  p.display(); }
```