Contents

1 Thinking Object Oriented

1.1 Procedure-Oriented Programming

- Conventional Programming, using high level languages such as COBOL (Common Business Oriented Language), FORTAN (Formula Translation) and C, is commonly called as procedural oriented programming (POP).
- POP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as function.
- Due to this POP is called Function Oriented Programming or Structured Programming.
- The list of instructions or actions is represented using flowchart or algorithms which helps to depict the flow of control from one action to another.
- The technique of hierarchical decomposition has been used to specify the task to be completed for solving a problem.
- A typical program structure for procedural programming is shown in the figure below.

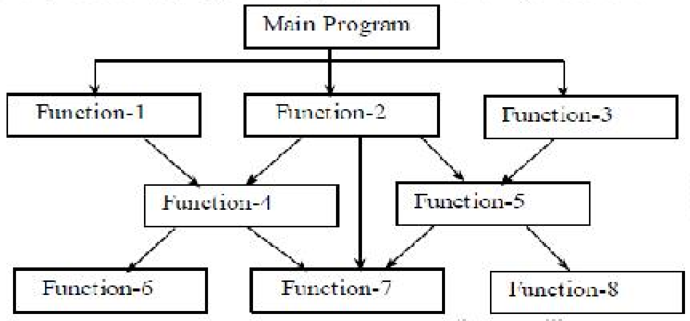


Figure 1-1: Typical Structure of Procedure Oriented Programs

- In a multi-function program, many important data items are placed as globally. So that they may be accessed by all the functions. Each function may have its own local data. The figure shown below shows the relationship of data and function in a procedure-oriented program.

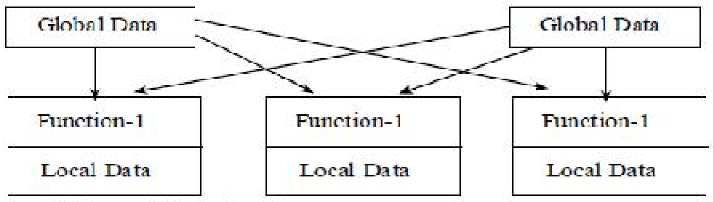


Figure 1-2: Relationship of data and functions in procedure programming

<u>Some features of procedure-oriented programming are:</u>

- Emphasis is on doing things (Algorithms).
- Large programs are divided into smaller programs called as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.

<u>Difficulties or Problems of POP</u>

- Complexity in handling large programs
- Data is undervalued
  In POP the focus is on development of function and flow of control, very little importance given to the data. The important data items are placed as global so they can be accessed to all functions. The global data can be modified by any of the function and this may result in accidental change by any functions.
  Also such global data can be destroyed by any function. Further in large programs it is very difficult to identify what data is used by which function. In case we need to revise any such data we need to revise all the functions that access and this may result in problems.
- POP cannot model real world entities (person, place, thing, event etc.). POP is action oriented and don't really correspond to element of problem. It cannot depict the characteristics and function or entities in combined form.
- Creation of new data type is difficult. Different data type like complex number s and two dimensional coordinate cannot be easily represented by POP.

1.2 Object Oriented Programming Paradigm

- The object oriented approach is to remove some of the flows encountered in the procedure approach.
- OOP treats data as critical element i.e. it doesn't allow the data to move freely around the systems.
- It ties the data closely with the related function there by protecting it from any accidental modification and deletion.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.
- The organization of data and functions in the object-oriented programs shown in the figure:

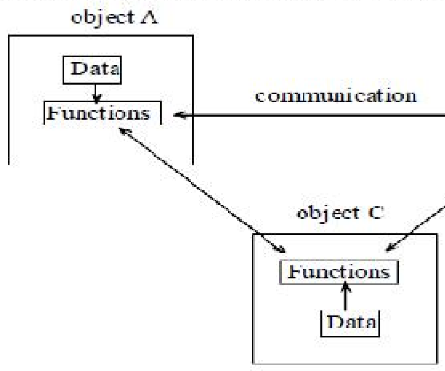


Figure 1-3: Organization of data & functions in OOP

- The data of an object can be accessed only by the function associated with that object. However, functions of one object can access the function of other objects.

Some features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are called objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed be external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

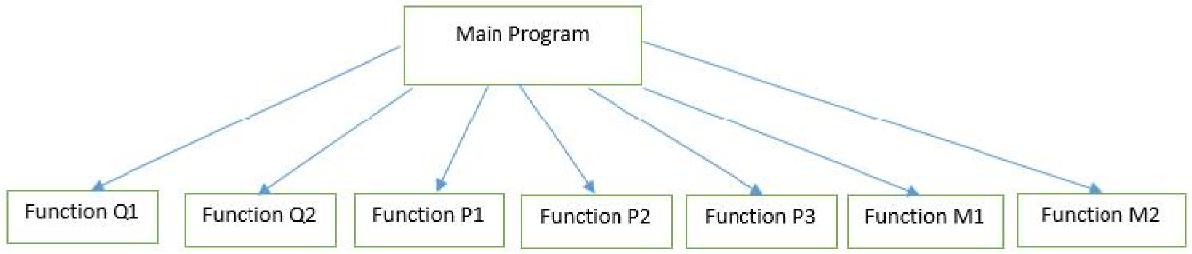
- So, **object oriented programming** is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and function that can be used as templates for creating copies of such modules on demand.
- Object oriented programming has other features also like inheritance, polymorphism, overloading, data obstruction and encapsulation.

Example 1-1: Comparing Procedural Programming & Object Oriented Programming Paradigm

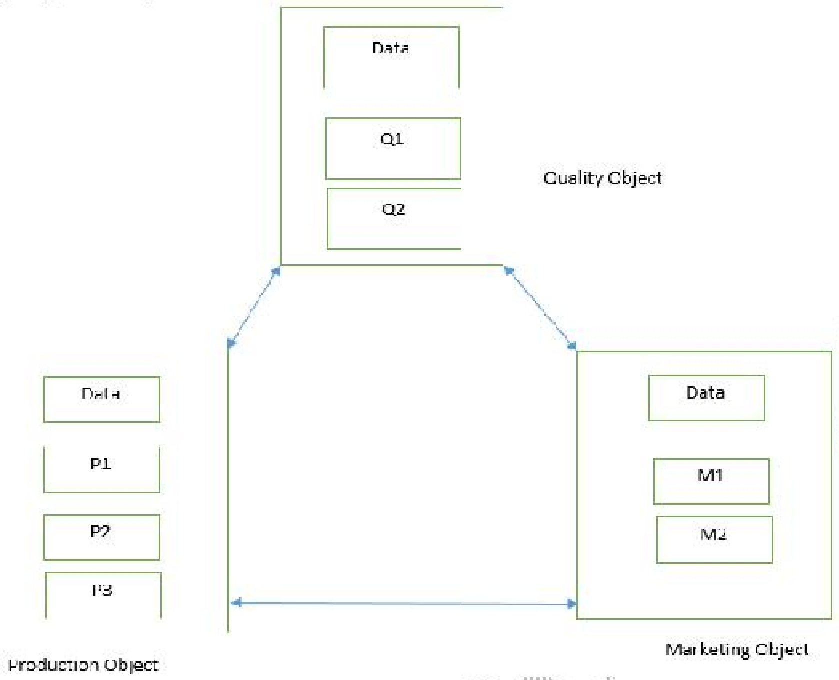
Problem: Write a program which implement a company, in which there are following departments:

(1) Quality (2) Production (3) Marketing

- If we solve the above program with the help of structured programming, we first divide the total department into different functions. Suppose we need two functions Q1 and Q2 for Quality, three functions P1, P2 and P3 for production, two functions for marketing M1 and M2 then the program structure look like:



- If we solve the same problem which we solved with the help of structured programing, using object oriented programming then the programming structure look like:



❖ Difference between C and C++

C	C++
C is a structural or procedural programming language.	C++ is an object oriented programming language
Emphasis is on procedure or steps to solve any problem.	Emphasis is on objects rather than procedure.
Functions are the fundamental building blocks.	Objects are the fundamental building blocks.
In C, the data is not secured.	Data is hidden and can't be accessed by external functions
C follows top down approach	C++ follows bottom up approach
C uses scanf() and printf() function for standard input and output	C++ uses cin>> and cout<< for standard input and output.
Variables must be defined at the beginning in the function.	Variables can be defined anywhere in the function.
In C, namespace feature is absent.	In C++, namespace feature is present.
C is a middle level language.	C++ is a high level language.
Programs are divided into modules and functions	Programs are divided into classes and functions.
C doesn't support exception handling directly. Can be done by using some other functions	C++ supports exception handling. Done by using try and catch block
C doesn't support exception handling directly. Can be done by using some other functions.	C++ supports exception handling. Done by using try and catch block.
Features like function overloading and operator overloading is not present	C++ supports function overloading and operator overloading.
C program file is saved with .C extension.	C++ program file is saved with .CPP extension

❖ **Difference between POP (Procedural Oriented Programming) and OOP (Object Oriented Programming)**

ASPECTS	POP	OOP
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

1.2.1 Why OOP is Popular

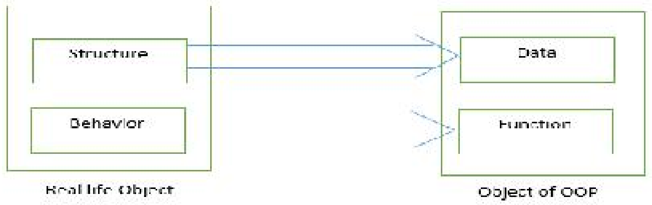
➢ The concept of object oriented programming can directly be mapped with real life problem. Even we can say that the idea of OOP are taken from the real life. Now there are two questions.

➢ **Question1:** How we can say that OOP concepts are taken from real life?
In a real life the objects are: Pen, Pencil, Car, book etc. These objects have two main things:
- Structure (or attribute)
- Behaviour (or Nature)

Example A: For example the structure of pen consists: color of pen, Length of pen, the material by using which body of pen is made etc. The behavior of pen means the job performed by pen that is writing.

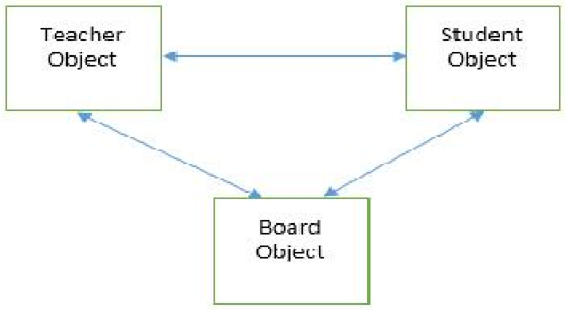
Example B: Suppose the object is teacher then the structure of teacher consists: the height of teacher, Age of teacher etc. The behavior of teacher consists: Teaching, Writing, Research etc.

Now we can compare the real objects with object of OOP. We know that the object of OOP consists data and functions. The structure of real life object can map with data member of object of OOP and behavior can map with function.



By above analysis we can say that the object concept is taken from real life object. And in real life the objects communicate with each other like in object oriented programming

Suppose a teacher is teaching to student, then following objects interact with, each other



Teacher teach students by using Board (i.e. Black Board). Means sometimes speak and sometimes write onto Board for explaining topic.

By above analysis we can say that the interaction of object in OOP is also taken from real life object interaction.

So we can say that the object oriented programming concept are taken from real life object.

- **Question2:** If concepts are taken from real life then what is the benefit of that i.e. how this increase the popularity of OOP?
  We know that programming concepts are used for designing the program, and we know that program are designed for solving problem. Problems are real life problems. That means program are designed for real life problems so if we have a programming concept which is based on real life concept then to design program by using that concept is easy.
  So we can say that designing program by OOP concept is easy, that is why OOP is more popular than other programming concept.

1.3 A way of Viewing World Agent (Object)

To illustrate the major ideas in object-oriented programming, let us consider how we might go about handling a real-world situation and then ask how we could make the computer more closely model the techniques employed. Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who happens to be named Flora), tell her the variety and quantity of flowers I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.

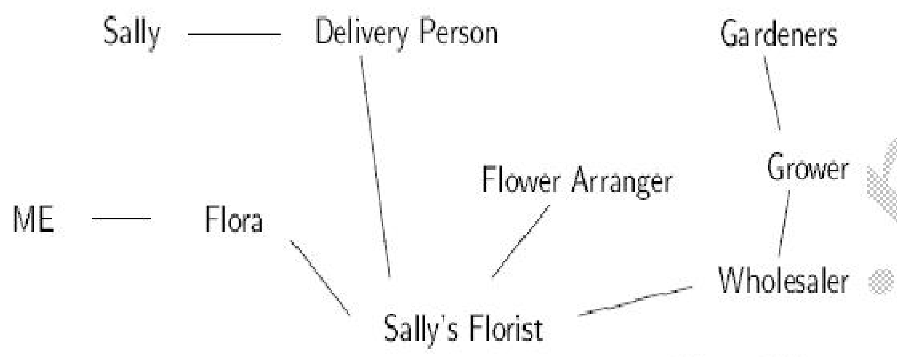


Figure 1-4: The community of agents helping me

<u>Agents and Communities</u>

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

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

<u>Message and Methods</u>

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

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

- **Message versus Procedure Calls**

  In message passing, there is a designated receiver, and the interpretation the selection of a method to execute in response to the message may vary with different receivers. Usually, the specific receiver for any given message will not be known until run time, so the determination of which method to invoke cannot be made until then.

  Thus, we say there is late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message. This situation is contrast to the very early (compile-time or link-time) binding of name to code fragment in conventional procedure calls.

<u>Responsibilities</u>

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

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

<u>Classes and Instances</u>

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

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

Class Hierarchies Inheritance

In above example, I have more information about Flora-not necessarily because she is a florist but because she is a shopkeeper. Since the category Florist is a more specialized form of the category Shopkeeper, any knowledge I have of Shopkeepers is also true of Florists and hence of Flora. Flora is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so I know, for example, that Flora is probably bipedal. A Human is a Mammal, and a Mammal is an Animal, and an Animal is a Material Object (therefore it has mass and weight).

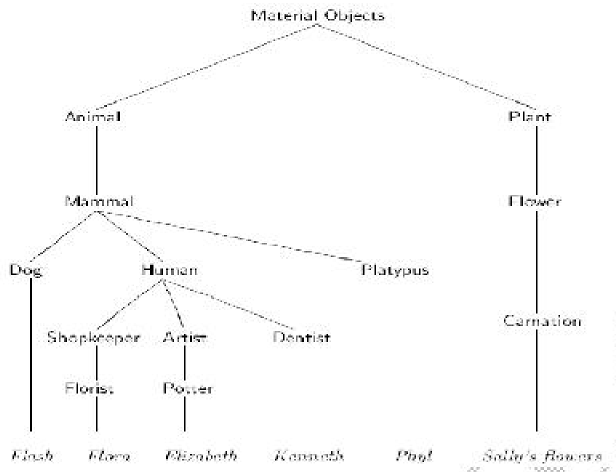


Figure 1-5: A class hierarchy for various material objects.

Classes can be organized into a hierarchical inheritance structure. A child class (or subclass) will inherit attributes from a parent class higher in the tree. An abstract parent class is a class (such as Mammal) for which there are no direct instances; it is used only to create subclasses.

❖ Summary of A way of Viewing World Agent

1. Everything is an object.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

1.4 Computation as Simulation

The Traditional Model

- ➢ In traditional view, computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various memory transforming them in some manner and pushing the results back into other memory.
- ➢ The behavior of computer executing a program is a process-state or pigeon-hole model.
- ➢ By examining the values in the slots, one can determine the state of the machine or the results produced by a computation.
- ➢ This model may be a more or less accurate picture of what takes place inside a computer.
- ➢ Real world problem solving is difficult in the traditional model.

The Object Oriented Model

- ➢ Never mention memory addresses, variables, assignments, or any of the conventional programming terms.
- ➢ Instead, we speak of objects, messages and responsibility for some action

- This model is process of creating a host of helpers that forms a community and assists the programmer in the solution of a problem (Like in flower example).
- The view of programming as creating a universe is in many ways similar to s style of computer simulation called "discrete even-driven simulation"
- In a discrete event-driven simulation the user create computer models of the various elements of the simulation, describes how they will interact with one another, and sets them moving.
- Object oriented programming is also similar to event driven simulation.

1.5 Copying with Complexity

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

<u>The Nonlinear Behavior of Complexity</u>

- As programming projects become larger, an interesting phenomenon was observed.
- A task that would be take one programmer 2 months to perform could not be accomplished by two programmer working for one month.
- In Fred Brook's memorable phrase, "*the bearing of child takes nine months, no matter how many women are assigned to the task*".
- The reason for this nonlinear behavior was complexity in particular, the interconnection between software components were complicated, and large amount of information had to be communicated among various members of programming team.
- Interconnectedness means the dependence of one portion of code on another section of the code.
- Time is not directly related with number of man hours.

1.6 Abstraction Mechanism

- The abstraction is the process of getting detail information according to the level of deep sight to the problem.
- If you want to get detail information about the topic, you should go deeper to the problem.
- Classes use theory of abstraction and defined list of abstract properties
- Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.
- In the same way that abstraction sometimes works in art, the objects that remains is a representation of the original, with unwanted detail omitted.
- The resulting object itself can be referred to as an abstraction, meaning a named entity made up of selected attributes and behavior (method) specify to particular usage of originating entity.
- Abstraction is related to both encapsulation and data hiding.
- Example: of adding complex number, considering three objects c1,c2, c3. C1 and c2 object is used to call input function, c3 is use for adding data in c1 and c2 and c3 is also used for displaying output.
- Consider Laptop computer, we view laptop as single unit but really laptop consists of input units (keyboard, touchpad) output unit (screen), processor unit (processor, motherboard) storage unit (RAM, HDD) etc.

2 Classes and Methods

2.1 Historical Background of C++

C++ is object oriented programming language. It was developed by Bjarne Stroustrup at Bell-Laboratories. Stroustrup take the best features of simula67 and C, and designed a language which support object oriented programming features. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However in 1983 the name was changed to C++. The idea of C++ comes from C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading and operator overloading. These features enable creation of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large program with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

2.2 A simple C++ Program

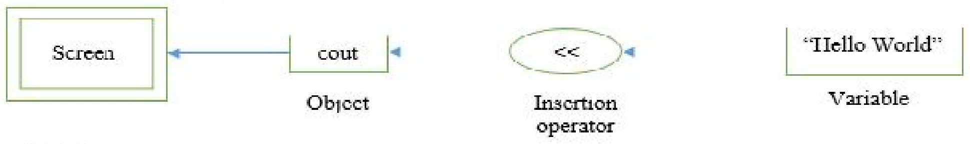
Example 2-1:

```
#include<iostream>                  //include header file
using namespace std;
int main()
{
        cout<<"Hello World";
        return 0;
}
```

❖ Output Operator

cout<< "Hello World";

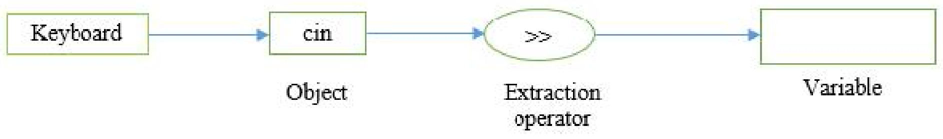
- ➢ The above statement print the string Hello world on to screen.
- ➢ cout (pronounced "c out") is a predefined object that represent the standard output stream in C++.
- ➢ In above statement the operator << is called insertion or put to operator. It inserts the contents of the variable on its right to the cout object.



❖ Input Operator

cin>>n;

- ➢ The above statement takes a value assign that value to n.
- ➢ Same like cout, cin (pronounced "c in") is also predefined object in C++, cin represent standard input stream.
- ➢ The operator >> is known as extraction or get from operator. It extracts the value from the keyboard and assign it to variable.



- **Cascading of input and output operator is also possible.**
- If suppose we want to read the value of variable n1, n2 and n3 then we can write statement

  cin >> n1 >> n2 >> n3;

  This is equivalent to

  cin >> n1;
  cin >> n2;
  cin >> n3;
- Same we can do with output operator also for example suppose we want to print the value of a and b then we can write the statement like

  cout << "a= " << a << "b= " << b;

  Above statement is equivalent to

  cout <<"a= ";
  cout << a;
  cout <<"b= ";
  cout <<b;

❖ Directive

The two lines that begin the example 2.1 program are directive. The first is a preprocessor directive, and the second is a using directive.

Preprocessor Directive

#include <iostream>

- The first line of the program might look like a program statement, but it is not. It isn't part of function body and doesn't end with a semicolon, as program statements must. Instead, it starts with a number sign (#).
- The program statement are instruction to the computer to do something, such as adding two numbers or printing a sentence. A **preprocessor** directive, on the other hand, is an instruction to the compiler. A part of the compiler called the **preprocessor** deals with these directive before it begins the real compilation process.
- The preprocessor directive **#include** tells the compiler to insert another file into your source file. In effect, the **#include** directive is replaced by the contents of the file indicated.
- The **#include** is only one of many preprocessor directives, all of which can be identified by the initial # sign.
- iostream is an example of header file (sometimes called an include file). It's concerned with basic input/output operations, and contains declarations that are needed by the cout identifier and the << operator. Without these declarations, the compiler won't recognize cout and will think << is being used incorrectly.
- There are many such include files. The newer standard C++ header files don't have a file extension, but some older header files, left over from the days of the C language, have the extension .h.
- We can add more than one header file in the program, if necessary.

The using Directive

- A C++ program can be divided into different ***namespaces***. A **namespaces** is a part of the program in which certain names are recognized, outside of the namespaces they are unknown.
- The directive

  using namespace std;

Says that the program statements that follow are within the std namespace. Various program component such as cout are declared within this namespace. If we didn't use the **using directive**, we would need to add the **std** name to many program elements. For example, the program of example 2.1 we'd need to say

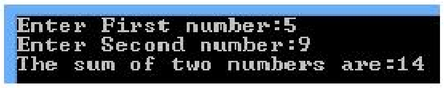
std::cout << "Hello World";

- To avoid adding **std::** dozens of time in program we use the using directive instead.

Example 2-2: Following Program read two number and then print the sum of these two.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a,b,s;
        cout<<"Enter First number:";
        cin>>a;
        cout<<"Enter Second number:";
        cin>>b;
        s=a+b;
        cout<<"The sum of two numbers are:"<<s;
        getch();
        return 0;
}
```

Output:



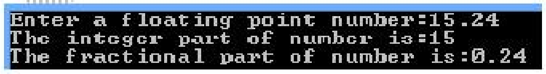
```
Enter First number:5
Enter Second number:9
The sum of two numbers are:14
```

Note: If in cout we write the message within " " then that message is printed as it is onto screen, otherwise the value of that variable is printed onto screen.

Example 2-3: Write s program which read a floating point number then print the integer and fractional part of that number

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
        float n,fpart;
        int ipart;
        cout<<"Enter a floating point number:";
        cin>>n;
        ipart=n;
        fpart=n-ipart;
        cout<<"The integer part of number is:"<<ipart<<endl;
        cout<<"The fractional part of number is:"<<fpart;
        getch();
        return 0;
}
```

Output:



```
Enter a floating point number:15.24
The integer part of number is:15
The fractional part of number is:0.24
```

Note: In above program **endl** is used in cout, **endl** transfer the cursor to new line i.e. the next statement is printed from the next line. This is equivalent to **"\n"**; in place of **endl** we can write **"\n"** also.

Q. Write a program which read three numbers *u, a, t* and then calculate function *s* where $s = u * t + \frac{1}{2} * a * t^2$

2.3 Review of Structure

- A structure is a collection of simple variable. The variable in s structure can be of different types: Some can be int, some can be float, and so on.
- The data item of structure is called member of the structure.
- The difference between array and structure is the element of an array has the same type while the element of structure can be of different type.
- For C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. The only difference being that in a class, all members are private by default. But in a C++ structure, all members are public by default. In C, there is no concept of public or private.
- So by learning about structures we'll be paving the way for an understanding of classes and objects.

2.3.1 Defining the Structure

- The structure definition tells how the structure is organized: It specifies what member the structure will have

Syntax:

```
struct  tag_name
{
        data_type variable1;
        data_type variable1;
        ……………………
        ……………………
};
```

- The keyword **struct** introduces the structure definition. Next comes the structure name or tag. The declaration of the structure members are enclosed in braces. A semicolon follows the closing brace, terminating the entire structure.
- Example:

```
struct student
{
        char name[50];
        int roll;
        char branch[20];
};
```

2.3.2 Creating structure variable

We can also create structure variable at the time of declaration:

```
struct student
{
        char name[50];
        int roll;
        char branch[20];
}s1,s2,s3;
```

We can create structure variable like following also

- Structure declaration:

```
struct student
{
        char name[50];
        int roll;
        char branch[20];
};
```

- Structure variable creation:

```
student s1,s2,s3;
```

Note: One of the aims of C++ is to make the syntax and operation of user-defined data types as similar as possible to that of built-in data types. In C we need to include the keyword **struct** in structure type variable creation as in **struct student s1,s2,s3;**. In C++ the **keyword** is not necessary.

2.3.3 Accessing member of structure

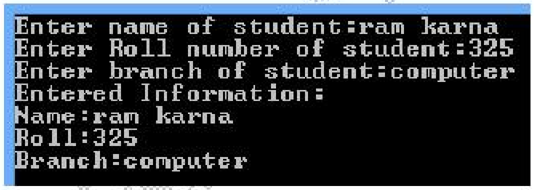
- Any member of a structure can be accessed as:

  Structure_variable_name . member_name
- The structure member is written in three parts: the name of the structure variable, the dot operator which consisit of a period (.), and the member name.
- For example: **s1.roll = 325;**

Example 2-4:

```
#include<iostream>
#include<conio.h>
using namespace std;
struct student
{
        char name[50];
        int roll;
        char branch[20];
};
int main()
{
        student s;
        cout<<"Enter name of student:";
        cin.get(s.name,50);
        cout<<"Enter Roll number of student:";
        cin>>s.roll;
        cout<<"Enter branch of student:";
        cin>>s.branch;
        cout<<"Entered Information:"<<endl;
        cout<<"Name:"<<s.name<<endl;
        cout<<"Roll:"<<s.roll<<endl;
        cout<<"Branch:"<<s.branch;
        getch();
        return 0;
}
```

Output:



```
Enter name of student:ram karna
Enter Roll number of student:325
Enter branch of student:computer
Entered Information:
Name:ram karna
Roll:325
Branch:computer
```

Note:

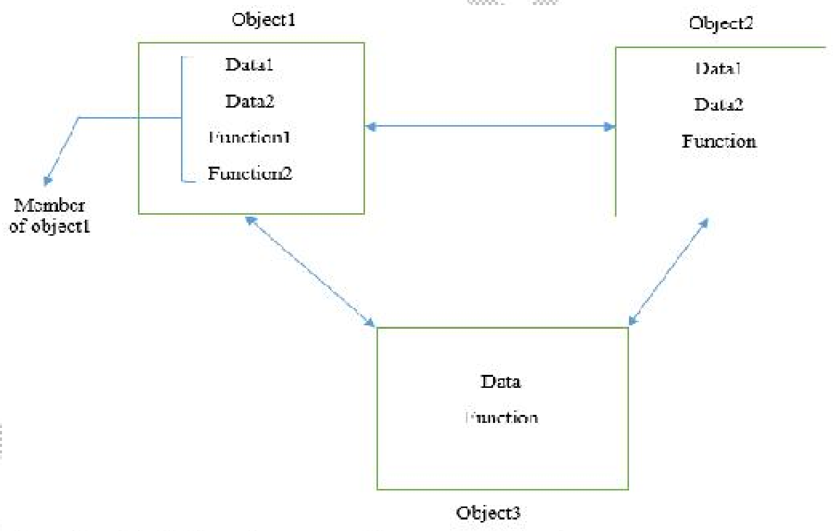
- C structure can't contain functions means only data members are allowed, but structure in C++ can have both functions and data members.
- **struct** keyword is necessary in C to create structure type variable, but it is redundant & not necessary in C++.
- Structure in C can't have static members, but C++ structure can have static members.
- Structure members can't be directly initialized inside the **struct** in C, but it is allowed in C++

2.4 Basic Concept of Object-Oriented Programming

- In object oriented programming data and functions are organized in one entity.
- It is necessary to understand some of the concepts used extensively in object-oriented programming. These include
  - Objects and Classes
  - Method
  - Encapsulation
  - Data abstraction
  - Inheritance
  - Polymorphism
  - Dynamic binding
  - Message passing

❖ Objects and Classes

- Classes is an entity in which data and functions (which operate these data) are organized.
- Object is a class variable. When a program is executed, the object interact by sending messages to one another.
- Object are the basic run-time entities in an object-oriented system. They may represent a person, a place, a table of data or any item that the program has to handle.
- Programming problem is analyzed in terms of objects and the nature of communication between them.
- Program objects should be chosen such that they match closely with the real-world objects.
- The entire set of data and code of an object can be made a user-defined data with the help of a classes.
- Once a class has been defined, we can create any number of objects belonging to that class.
- A class is thus a collection of objects of similar type. For example: mango, apple and orange are members of the class fruit.



- If object3 wants to access function1 then that access throws the object1.
- The data and function inside a classes are known as member of that class.

❖ Method

- An operation required for an object or entity when coded in a class is called method.
- The operation required for an object is defined in a class.
- Functions of class is referred as methods or member functions.
- Usually data members are declared private and methods as public in a class.

❖ Encapsulation

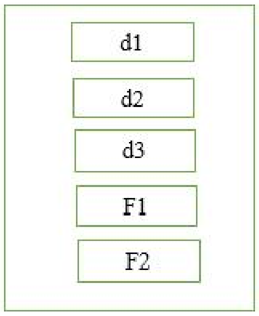
- The wrapping up of data and functions into a single unit (called class) is known as encapsulation.
- Data encapsulation is the most striking features of a class. The data is not accessible to the outside of class. Only the function which are inside the class can access the data.
- The insulation of the data from direct access by the program is called data hiding or information hiding.

❖ Data abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Class use the concept of abstraction and are defined as a list of abstract data and functions to operate on these data.
- Classes encapsulate all the essential properties of the object that are to be created.
- Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

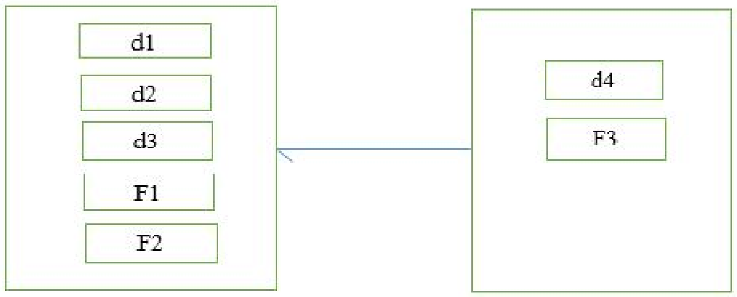
❖ Inheritance

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- Inheritance is a method by using which we can create a new class by extending and enhancing existing class. The existing class is called the base class and the new class is called derived class.
- Inheritance is very powerful features of oop.
- **For example** we have a class X which has following member: two functions F1 and F2 and three data d1, d2 and d3.



Class X

Suppose after sometimes we want to add one function, F3 and data d4 in that class X. We can do this by designing the class again but the main drawback of this is we have again to check the class X (test & debug). But in oop with the help of inheritance we can do this. Drive a new class Y from the base class X. In class Y add only function F3 and data d4, because the class Y is inherited from the class X. Therefore the member of class X are automatically copied into class Y.



Class X
(Base Class)

Class Y
(Derived Class)

If we create an object of class Y then in that object the member are d1, d2, d3, d4, F1, F2 and F3. But if we create an object of class X, in that object the member are d1, d3, d3, F1 and F2. That means the base member are transformed into derived class but not in reverse order.

❖ Polymorphism

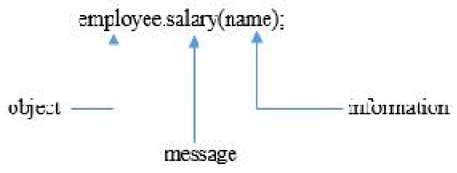
- Polymorphism, a Greek term, means the ability to take more than one form.
- Using function in different ways, depending on what they are operating on is called the polymorphism. i.e. one thing with several distinct forms.
- For example suppose we are writing a program, which calculates the area of circle, area of triangle, area of rectangle. With the help of polymorphism we can give same name area to all the area functions. In other words function overloading is called polymorphism.
- We can overload the operator also that is known as operator overloading for example + operator is used to add, numeric data (int or float). If we use same + operator for adding two objects, then this is known as operator overloading.

❖ **Dynamic binding**

- Dynamic binding refers to linking a procedure call to the code that will be executed only at run time.
- The code associated with the procedure is not known until the program is executed, which is also known as late binding.

❖ **Message Passing**

- An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:
  - Creating classes that define objects and their behavior.
  - Creating objects from class definitions, and
  - Establishing communication among objects.
- A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- **Example:**



- Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

2.5 Specifying a class

- A class is a way to bind the data and its associated functions together. It allows the data (and function) to be hidden, if necessary, from external use.
- When defining a class we are creating a new abstract data type that can be treated like any other built-in data type.
- The data inside the class are called member data and the functions are called member function.
- The binding of data and functions together into a single class type variable is called encapsulation, which is one of the benefit of object-oriented programming.
- The general form of declaring class is:

```
class class_name
{
        access-specifier1: member_data1;
                           member_data2;
                           - - - - - - - -
                           - - - - - - - -
                           member_function1;
                           - - - - - - - -
                           - - - - - - - -
        access-specifier2: member_data1;
                           - - - - - - - -
                           member_function1;
        access-specifier: member_data;
                          - - - - - - - -
                         member_function;
};
```

In above declaration, **class** is keyword. class_name is any identifier name. The number of member data and member function depends on the requirements. An object is an instance of a class i.e. variable of a class. The general form of declaring an object is

class_name object_name;

OR

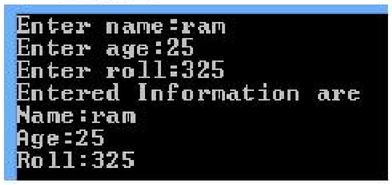
```
class class_name
{
        private:
                variable_declaration;
                function_declaration;
        public:
                variable_declaration;
                function_declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword class specifies that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called class members. They are usually grouped under section, namely private and public to denote which of the members are private and which of them are public. The keyword private and public are known as visibility labels. Note that these keywords are followed by a colon.

Example 2-5:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
        private:
                char name[50];
                int age;
                int roll;
        public:
                void getdata()
                {
                        cout<<"Enter name:";
                        cin>>name;
                        cout<<"Enter age:";
                        cin>>age;
                        cout<<"Enter roll:";
                        cin>>roll;
                }
                void showdata()
                {
                        cout<<"Name:"<<name<<endl;
                        cout<<"Age:"<<age<<endl;
                        cout<<"Roll:"<<roll<<endl;
                }

};
int main()
{
        student s;
        s.getdata();
        cout<<"Entered Information are"<<endl;
        s.showdata();
        getch();
        return 0;
}
```

Output:



```
Enter name:ram
Enter age:25
Enter roll:325
Entered Information are
Name:ram
Age:25
Roll:325
```

2.5.1 Access Specifiers (Visibility Labels)

- The access specifier tells about the visibility of member.
- The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords **public, private,** and **protected** are called access specifiers.
- A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.
- Access specifier are followed by colon(**:**).

❖ Public

- The members defined in **public** section can be accessed anywhere from the program. This allows the class to expose its data members and member function to other functions and objects.
- When this **public** access specifier is used all the details of the data member and member function are visible to other class.
- Usually data members(i.e. variables) are not declared in this section
- If data members are declared in this section. Data hiding rule is violated in oop.

❖ Private

- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access **private** members.
- This allows a class to hide its member variables and member functions from other class objects and functions.
- Usually data member are declared in this section.
- If member function is declared in private other public function should call that member function.

❖ Protected

- A **protected** member variable or function is very similar to a **private** member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

2.5.2 Creating Objects

- A typical class declaration would look like:

```
class item
{
                int number;                          //variable declaration
                float cost;                          //private by default
        public:
                void getdata(int a, float b);        //function declaration using prototype
                void showdata ();
};
```

- Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies what they will contain. Once a class has been declared, we can create variables of that type by using the class name.
- For example

  item p; //memory for p is created

  create a variable **p** of type item. In C++, the class variables are known as objects. Therefore, **p** is called an object of type **item**. We may also declare more than one object in one statement. Example

  item p,q,r;
- Object can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structure.
- For Example

```
        class item
        {
                ……….
                ……….
        }p,q,r;
```

Would create the objects **p**, **q** and **r** of type **item**.

2.5.3 Accessing class members

- Private member cannot access outside the class. Therefore the private data of a class can be accessed only by the member functions of that class.
- The public member can be access outside the class like from **main( )**. The format for calling a member function is:

  Object-name.member-name;

  If member is function then calling format is:

  Object-name.function-name(argument);
- That means the public member of a class can access through the object of that class. The calling procedure of a public member function is same as calling procedure of function. Only difference is the object-name is also necessary in case of member function.
- For example, the function call statement

  p.getdata(50,250.65);

  is valid and assigns the value 50 to **number** and 250.65 to **cost** of the object p by implementing the **getdata( )** function. Similarly, the statement

  p. showdata( );

  would display the values of data members.
- Remember, a member function can be invoked only by using an object (of the same class). The statement like

  getdata(50,250.65);

  has no meaning. Similarly, the statement

  p.number = 50;

  is also illegal. Although **p** is an object of the type **item** to which number belongs, the **number** (declared private) can be accessed only through a member function and not by the object directly.

- A variable declared as public can be accessed by the objects directly. **For Example:**

```
class xyz
{
            int x;
            int y;
      Public:
            int z;
};
………..
………..
xyz p;
p.x = 20;           //Error, x is private
p.z = 50;           //Ok, z is public
```

2.5.4 Member Function

- The variable declared inside the class are known as data members and the functions are known as methods or member functions.
- The operation required for an object is defined in a class.
- Member function can be defined private or public sections, usually methods are declare in public section of class.
- Member function can be defined in two ways
  - Outside the class definition
  - Inside the class definition

❖ Outside the Class Definition

- Member function that are declared inside a class have to be defined separately outside the class.
- Their definition are very much like the normal functions. They should have a function header and a function body.
- An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to.
- The general form of a member function definition is:

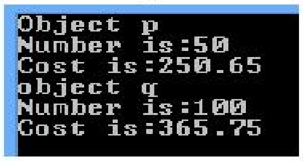
return-type ***class-name*** *:: function-name (argument declaration)*
{
Function body
}

- The membership label **class-name::** tells the compiler that the function **function-name** belongs to the class **class-name**. That is, the scope of the function is restricted to the class-name specified in header line. The symbol **::** is called the scope resolution operator.

Example 2-6:

```
#include<iostream>
#include<conio.h>
using namespace std;
class item
{
            int number;                                   //private by default
            float cost;                                   //private by default
      public:
            void getdata(int a, float b);
            void showdata();
};
void item::getdata(int a, float b)                        // use membership label
{
      number=a;                                           //private variables
      cost=b;                                             //directly used
}
void item::showdata()
{
      cout<<"Number is:"<<number<<endl;
      cout<<"Cost is:"<<cost;
}
int main()
{
      item p;                                             //create object p
      cout<<"Object p"<<endl;
      p.getdata(50,250.65);                               //call member function
      p.showdata();
      item q;                                             //create another object q
      cout<<"\nobject q\n";
      q.getdata(100,365.75);
      q.showdata();
      getch();
      return 0;
}
```

Output:



```
Object p
Number is:50
Cost is:250.65
object q
Number is:100
Cost is:365.75
```

❖ Inside the class definition

- Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.
- When a function is defined inside a class, it is treated as an inline function.

Example 2-7:

```
#include<iostream>
#include<conio.h>
using namespace std;
```

```
class item
{
            int number;                                         //private by default
            float cost;                                         //private by default
        public:
            void getdata(int a, float b)
            {
                number=a;
                cost=b;
            }
            void showdata()
            {
                cout<<"Number is:"<<number<<endl;
                cout<<"Cost is:"<<cost;
            }
};
int main()
{
        item p;                                                 //create object p
        cout<<"Object p"<<endl;
        p.getdata(50,250.65);                                   //call member function
        p.showdata();
        item q;                                                 //create another object q
        cout<<"\nobject q\n";
        q.getdata(100,365.75);
        q.showdata();
        getch();
        return 0;
}
```

<u>Making an Outside Function Inline</u>

- We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition.
- **Example:**

```
class item
{
        ......
        ......
        public:
            void getdata(int a, float b);                   //declaration
};
inline void item::getdata(int a, float b)                   //definition
{
        number=a;
        cost=b;
}
```

2.5.5 Private Member Functions

- Although it is normal practice to place all the data items in a private section and all the functions in public. Some situations may require certain functions to be hidden (like private data) from outside calls. We can place these functions in the private section.
- A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.
- Consider a class as defined below:

```
class sample
{
            int m;
            void read();                //private member function
      public:
            void update();
            void write();

}s1;
```

If s1 is an object of sample, then

```
s1.read( );           // is illegal, object cannot access private member
```

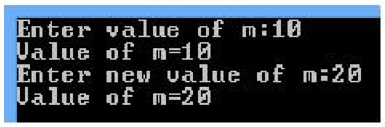
the function **read( )** can be called by the function **update( )** and function **update( )** is called by object.

```
void sample::update()
{
      read();                           //simple call, no object used
}
```

Example 2-8:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
            int m;
            void read()
            {
                  cout<<"Value of m="<<m<<endl;
            }
      public:
            void write()
            {
                  cout<<"Enter value of m:";
                  cin>>m;
            }
            void update()
            {
                  read();
                  cout<<"Enter new value of m:";
                  cin>>m;
                  read();
            }

};
int main()
{
      sample s1;
      s1.write();
      s1.update();
      getch();
      return 0;
}
```

Output:



```
Enter value of m:10
Value of m=10
Enter new value of m:20
Value of m=20
```

2.5.6 Arrays within a class

- The array can be used within a class as a member.
- For example

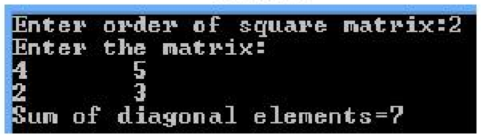
```
class xyz
{
        int a[10],matrix[2][3];
        public:
                ........
                ........
};
```

Example 2-9: Write a program to find the sum of diagonal elements of *N x N* matrix.

```
#include<iostream>
#include<conio.h>
using namespace std;
class matrix
{
                int n,i,j,a[10][10],sum;
        public:
                void readmatrix();
                void sumdiagonal();
};
void matrix::readmatrix()
{
        cout<<"Enter order of square matrix:";
        cin>>n;
        cout<<"Enter the matrix:"<<endl;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        cin>>a[i][j];
                }
        }
}

void matrix::sumdiagonal()
{
        sum=0;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(i==j)
                        {
                                sum+=a[i][j];
                        }
                }
        }
        cout<<"Sum of diagonal element="<<sum;
}
int main()
{
```

```
                matrix m;
                m.readmatrix();
                m.sumdiagonal();
                getch();
                return 0;
        }
```

Output:



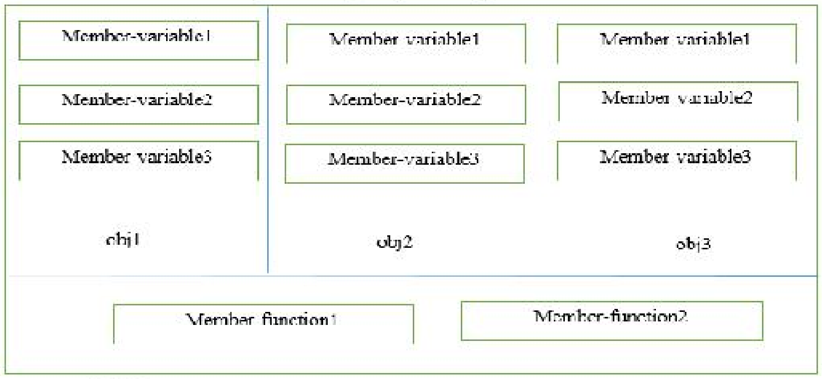
```
Enter order of square matrix:2
Enter the matrix:
4       5
2       3
Sum of diagonal elements=7
```

2.5.7 Memory Allocation for Object

- When we create the objects space for member data variable is allocated separately for each object but no separate space is allocated for member function, the space for member function are created only once when they are declared as a part of class. Because all the objects belong to that class, therefor all use the same member function.
- For example:

```
class abc
{
                Member-variable1;
                Member-variable2;
                Member-variable3;
        public:
                Member-function1;
                Member-function2;

};
............
............
abc obj1, obj2, obj3;
```

The space for obj1, obj2 and obj3 is as follows



2.6 Array of Object

- Array is a collection of homogenous data items. Therefore we can create an array in which each data item is class type such array is called array of objects.
- For example

```
        class student
        {
                char name[50];
                char branch[30];
                int roll;
            public:
                void getdata( );
                void showdata( );
        };

        student biodata[30];
```

biodata is an array of 30 elements each is student type i.e. biodata[0], biodata[1], ……….biodata[29] all are object o student class.

The accessing of public member is as

biodata[i]. getdata( );   //whrere i is 0 to 29

if we want to access getdata( ) for object biodata[2] the we will write

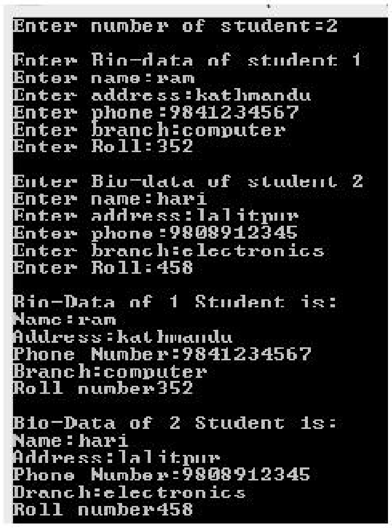
biodata[2].getdata( );

Example 2-10: Write a program which reads the bio-data of n students and then prints.

```
#include<iostream>
#include<conio.h>
using namespace std;
```

```
class student
{
                char name[50];
                char address[40];
                char phone[14];
                char branch[30];
                int roll;
        public:
                void getdata();
                void showdata();
};
void student::getdata()
{
        cout<<"Enter name:";
        cin>>name;
        cout<<"Enter address:";
        cin>>address;
        cout<<"Enter phone:";
        cin>>phone;
        cout<<"Enter branch:";
        cin>>branch;
        cout<<"Enter Roll:";
        cin>>roll;
}

void student::showdata()
{
        cout<<"Name:"<<name<<endl;
        cout<<"Address:"<<address<<endl;
        cout<<"Phone Number:"<<phone<<endl;
        cout<<"Branch:"<<branch<<endl;
        cout<<"Roll number"<<roll<<endl;
}
int main()
{
        student biodata[100];
        int n,i;
        cout<<"Enter number of student:";
        cin>>n;
        for(i=0;i<n;i++)
        {
                cout<<"\nEnter Bio-data of student "<<i+1<<endl;
                biodata[i].getdata();
        }
        for(i=0;i<n;i++)
        {
                cout<<"\nBio-Data of "<<i+1<<" Student is:"<<endl;
                biodata[i].showdata();
        }
        getch();
        return 0;
}
```

Output:



```
Enter number of student:2

Enter Bio-data of student 1
Enter name:ram
Enter address:kathmandu
Enter phone:9841234567
Enter branch:computer
Enter Roll:352

Enter Bio-data of student 2
Enter name:hari
Enter address:lalitpur
Enter phone:9808912345
Enter branch:electronics
Enter Roll:458

Bio-Data of 1 Student is:
Name:ram
Address:kathmandu
Phone Number:9841234567
Branch:computer
Roll number352

Bio-Data of 2 Student is:
Name:hari
Address:lalitpur
Phone Number:9808912345
Branch:electronics
Roll number458
```

2.7 Objects as Function Arguments

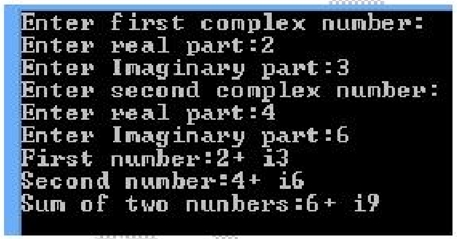
- An entire object can be passed to a function. There are two concepts for passing an object to function
  - Call by value
  - Call by reference or call by address
- If we pass object value to function that is called call by value, in this case any change made to the object inside the function do not affect the actual object.
- If we pass object as a call by reference or call by address, we pass the reference or address of the object. Therefore any change made to the object inside the function will reflect in the actual object.
- The object which used as an argument when we call the function is known as actual object, and the object which is used as an argument within the function header is called formal object.

Example 2-11: Write a program which reads two complex numbers and then calculate sum of these two.

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
            float realp;
            float imagp;
      public:
            void getdata()
            {
                  cout<<"Enter real part:";
                  cin>>realp;
                  cout<<"Enter Imaginary part:";
                  cin>>imagp;
            }
            void output()
            {
                  cout<<realp<<"+ i"<<imagp<<endl;
            }
            void sum(complex c1, complex c2)
            {
                  realp=c1.realp+c2.realp;
                  imagp=c1.imagp+c2.imagp;
            }
};
```

```
int main()
{
        complex x,y,z;
        cout<<"Enter first complex number:"<<endl;
        x.getdata();
        cout<<"Enter second complex number:"<<endl;
        y.getdata();
        z.sum(x,y);
        cout<<"First number:";
        x.output();
        cout<<"Second number:";
        y.output();
        cout<<"Sum of two nunbers:";
        z.output();
        getch();
        return 0;
}
```

Output:



```
Enter first complex number:
Enter real part:2
Enter Imaginary part:3
Enter second complex nunber:
Enter real part:4
Enter Imaginary part:6
First nunber:2+ i3
Second nunber:4+ i6
Sum of two nunbers:6+ i9
```

- In example 2-11, the member function **sum( )** is called by object **z** and we have passed two argument **x** and **y** both are object. When the statement **z.sum(x, y);** is executed the value of actual object **x** is copied to formal object **c1** and actual object **y** is copied to formal object **c2** and after that body of function **sum()** is executed. Inside the body of **sum()** there are following statements:

  realp=c1.realp+c2.realp;
  imagp=c1.imagp+c2.imagp;

  By the first statement the **realp** of **c1** (i.e. the realp of object x) is added with **realp** of **c2** (i.e. the realp of object y) and assigned to **realp** variable. This **realp** is member of object **z** because the function **sum( )** is called by object **z**. Similarly **imagp** is calculate by second statement.

2.8 Returning object from function

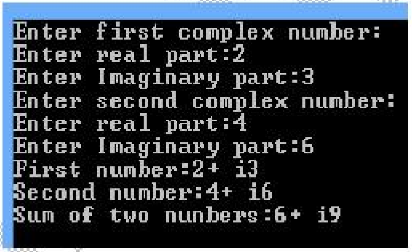
- A function cannot only receive objects as arguments but also can return them.

Example 2-12:

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
                float realp;
                float imagp;
        public:
                void getdata()
                {
                        cout<<"Enter real part:";
                        cin>>realp;
                        cout<<"Enter Imaginary part:";
                        cin>>imagp;
                }
                void output()
                {
                        cout<<realp<<"+ i"<<imagp<<endl;
                }
                complex sum(complex c1)
                {
                        complex temp;
                        temp.realp=c1.realp+realp;
```

```
                    temp.imagp=c1.imagp+imagp;
                    return (temp);                          //returning an object
            }
};
int main()
{
        complex x,y,z;
        cout<<"Enter first complex number:"<<endl;
        x.getdata();
        cout<<"Enter second complex number:"<<endl;
        y.getdata();
        z=y.sum(x);
        cout<<"First number:";
        x.output();
        cout<<"Second number:";
        y.output();
        cout<<"Sum of two nunbers:";
        z.output();
        getch();
        return 0;
}
```

Output:



```
Enter first complex number:
Enter real part:2
Enter Imaginary part:3
Enter second complex number:
Enter real part:4
Enter Imaginary part:6
First number:2+ i3
Second number:4+ i6
Sum of two nunbers:6+ i9
```

- In above program the function sum is called by object y therefore inside the function **sum( )** in statement

  temp.realp=c1.realp+realp;

  realp is member of y.
- **c1.realp** is actually **x.realp** because **x** is copied to **c1** at the time of calling by the statement:

  z = y. sum(x);

  So,

  temp.realp=c1.realp+realp;

  means realp of x is added with realp of y and assigned to realp of temp.
- Similarly

  temp.imagp=c1.imagp+imagp;

  means the **imagp** of **x** is added with **imagp** of y and assigned to **imagp** of **temp**.
- By

  return(temp);

  statement object temp is returned to main and copied to z because we are assigning **y.sum(x)** to **z** in main program by the calling statement

  z=y.sum(x);

2.9 Static Member of a Class

- The member function of a class can be made static (data member and function member both).

2.9.1 Static data member

- A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable.
- Static variables are normally used to maintain values common to the entire class.
- A static member variable has certain special characteristics. These are:
  - It is initialized to zero when the first object of its class is created. No other initialization is permitted.
  - Only one copy of that member is created for the entire class and is shared by all the objects of that class.
  - It is visible only within the class, but its lifetime is the entire program.
- The static variable are declared as follows:

```
class item
{
        static int count;
```

```
                        ................
            public:
                        ................
                        ................
        };
```

- The type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the **static** data member are stored separately rather than as a part of an object. Since they are associated with the class itself rather than any class object, they are also known as class variables.
- For example for above class **item**, **count** is defined as

  int item::count;

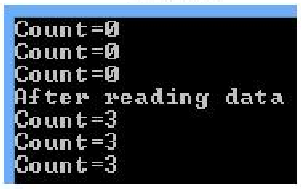
  or

  int item::count = 10;
- If we write, **int item::count;** then count is automatically assigned to zero.
- But by statement **int item::count = 10;** count is initialized 10.

Example 2-13:

```
#include<iostream>
#include<conio.h>
using namespace std;
class item
{
            static int count;
            int n;
    public:
            void getdata(int a)
            {
                    n=a;
                    count++;
            }
            void showcount()
            {
                    cout<<"Count="<<count<<endl;
            }
};
int item::count;                                    //definition of static data memebr
int main()
{
    item x,y,z;                                     //count is initialized to zero
    x.showcount();
    y.showcount();
    z.showcount();
    x.getdata(100);
    y.getdata(200);
    z.getdata(300);
    cout<<"After reading data"<<endl;
    x.showcount();
    y.showcount();
    z.showcount();
    getch();
    return 0;

}
```

Output:



```
Count=0
Count=0
Count=0
After reading data
Count=3
Count=3
Count=3
```

- In above example **count** is static variable, this is initialized **zero** when the objects are created. The variable **count** is increment whenever the **getdata( )** function is called. Since the **getdata( )** function called three times by object **x, y** and **z,** the variable **count** is incremented three times. Because there is only one copy of count shared by all the three objects.

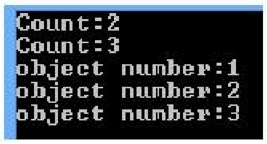
2.9.2 Static member function

- Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:
  - A static function can have access to only other static members (functions or variables) declared in the same class.
  - A static member function can be called using the class-name (instead of its object) as follows:
    class-name :: function-name;

Example 2-14:

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
                int code;
                static int count;                                   //static member variable
        public:
                void setcode()
                {
                        code=++count;
                }
                void showcode()
                {
                        cout<<"object number:"<<code<<endl;
                }
                static void showcount()                             //static member function
                {
                        cout<<"Count:"<<count<<endl;
                }
};
int test::count;
int main()
{
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();                                          //accessing static function
        test t3;
        t3.setcode();
        test::showcount();
        t1.showcode();
        t2.showcode();
        t3.showcode();
        getch();
        return 0;
}
```

Output:



```
Count:2
Count:3
object number:1
object number:2
object number:3
```

- Remember, the following function definition will not work

static void showcount( )
{
        cout<<code;   **//code is not static**
}

2.10 const Member function

- If a member function does not alter any data in the class, then we may declare it as a **const** member function.
- The compiler will generate an error message if const member function try to alter the data value.
- If we write the body of function outside then qualifier const is appended in both declaration and definition.

Example 2-15:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
            int data;
      public:
            void assign()
            {
                  data=20;
            }
            void changedata() const
            {
                  data=40;          //Error because changedata( ) is const member function
            }
            void showdata() const
            {
                  cout<<"Data="<<data<<endl;
            }
};
int main()
{
      sample s;
      s.assign();
      s.changedata();
      s.showdata();
      getch();
      return 0;
}
```

- If we pass object as constant argument to a function then function cannot modify these arguments, if modify compiler give error message.

Example 2-16:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
            int data;
      public:
            void assign()
            {
                  data=20;
            }
            void sum(sample const &a, sample const &b)
            {
                  data=a.data+b.data;
            }
```

```
      void showdata()
      {
               cout<<"Data="<<data<<endl;
      }
};
      int main()
      {
         sample x,y,z;
         x.assign();
         y.assign();
         z.sum(x,y);
         z.showdata();
         getch();
         return 0;
      }
```

2.11 Friend function

- We know that private member of a class cannot be accessed from outside the class i.e. a non-member function cannot access to the private data of a class. But may be in some situation one class wants to access private data of second class and second wants to access private data of first class, or may be an outside function wants to access the private data of a class. Then for solving these situations in C++ there is a concept that is called friend function.
- If an outside function can access the private data of a class that is called friendly to class.
- To make an outside function friendly to a class, we have to declare this function as a friend function.
- Declaration of friend function need keyword friend but in definition keyword friend is not necessary.
- The friend function declares as follows:

```
class class-name
{
            ........
    public:
            ........
            friend return-type function-name(arg);
}
```

- The body of function is written outside the class (i.e. the definition of function is written outside the class).

```
return-type function-name(arg)
{
        Body of function
}
```

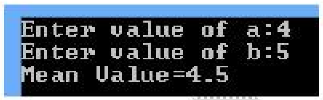
- A friend function possesses certain special characteristics:
  - It is not in the scope of the class to which it has been declared as friend.
  - Since it is not in the scope of the class, it cannot be called using the object of that class.
  - It can be invoked like a normal function without the help of any object.
  - Unlike member function, it cannot access the member names directly and has to use object name and dot membership operator with each member name.
  - It can be declared either in the public or the private part of a class without affecting its meaning.
  - Usually, it has the objects as arguments.

Example 2-17:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
            int a,b;
    public:
            void getdata()
            {
                    cout<<"Enter value of a:";
                    cin>>a;
                    cout<<"Enter value of b:";
                    cin>>b;
            }
            friend float mean(sample s);
};
float mean(sample s)
{
        float temp;
        temp=(s.a+s.b)/2.0;
        return temp;
}
```

```
int main()
{
        sample x;
        float r;
        x.getdata();
        r=mean(x);
        cout<<"Mean Value="<<r;
        getch();
        return 0;
}
```

Output:



```
Enter value of a:4
Enter value of b:5
Mean Value=4.5
```

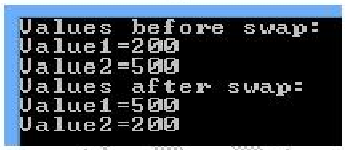
- **Note that we can make a friend function which is friend of two different classes by declaring that into both classes, and body of function is written outside the both classes.**

Example 2-18: Swapping Private Data of Classes

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample2;                  //forward declaration
class sample1
{
                int value1;
        public:
                void getdata(int a)
                {
                        value1=a;
                }
                void showdata()
                {
                        cout<<"Value1="<<value1<<endl;
                }
                friend void swap(sample1 &x, sample2 &y);
};
class sample2
{
                int value2;
        public:
                void getdata(int a)
                {
                        value2=a;
                }
                void showdata()
                {
                        cout<<"Value2="<<value2<<endl;
                }
                        friend void swap(sample1 &x, sample2 &y);
};
void swap(sample1 &x, sample2 &y)
{
        int temp;
        temp=x.value1;
        x.value1=y.value2;
        y.value2=temp;
}
```

```
int main()
{
        sample1 s1;
        sample2 s2;
        s1.getdata(200);
        s2.getdata(500);
        cout<<"Values before swap:"<<endl;
        s1.showdata();
        s2.showdata();
        swap(s1,s2);                    //swapping
        cout<<"Values after swap:"<<endl;
        s1.showdata();
        s2.showdata();
        getch();
        return 0;
}
```

Output:



```
Values before swap:
Value1=200
Value2=500
Values after swap:
Value1=500
Value2=200
```

- In above program class **sample2** is declared before class **sample1** but the body of class **sample2** is written after class **sample1**, this is necessary because within friend function we are using argument which is object of class **sample2**, so compiler must know that class **sample2** exist. This type of declaration is called **forward declaration**.

Note:

- Member function of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
        .........
        int fun1();                     //member function of X
        .........
};
class Y
{
        ..........
        friend int X::fun1();           //fun1 of X is friend of Y
        ..........
};
```

2.12 Friend class

- We can declared a **class** as a friend of another class. A fiend class can use all the data member of a class for which it is friend. For example

```
        class B;
        class A
        {
                .......
                friend B;
        };
```

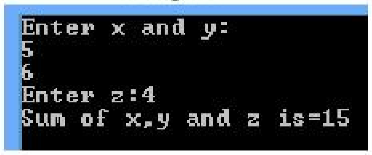
In this declaration class **B** is declared as a friend as of class **A**. That's why class **B** can use all the members of class **A** including private member

Example 2-19:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B;                        //forward declaration
class A
{
        int x,y;
    public:
```

```
                void getdata()
                {
                        cout<<"Enter x and y:"<<endl;
                        cin>>x>>y;
                }
                friend B;
};
class B
{
        int z;
        public:
                void getdata()
                {
                        cout<<"Enter z:";
                        cin>>z;
                }
                int sum(A t);
};
int B::sum(A t)
{
        int s;
        s=t.x+t.y+z;
        return (s);
}
int main()
{
        A p;
        B q;
        p.getdata();
        q.getdata();
        cout<<"Sum of x,y and z is="<<q.sum(p);
        getch();
        return 0;
}
```

Output:



```
Enter x and y:
5
6
Enter z:4
Sum of x,y and z is=15
```

2.13 Inline function

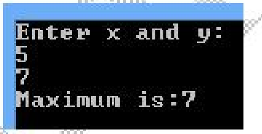
- When function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving register, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.
- One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.
- To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline* function.
- An inline function is a function that is expanded in line when it is invoked. That is, the compiler replace the function call with the corresponding function code.
- Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicate code. For this reason, it is best to inline only very small functions.
- Declaration of inline function:

```
inline return-type function-name (argument list)
{
        ........................
        Body of the function
        ........................
}
```

Example 2-20:

```
#include<iostream>
#include<conio.h>
using namespace std;
inline int max(int a, int b)
{
        return (a>b)?a:b;
}
int main()
{
        int x,y;
        cout<<"Enter x and y:"<<endl;
        cin>>x>>y;
        cout<<"Maximum is:"<<max(x,y);
        getch();
        return 0;
}
```

Output:



```
Enter x and y:
5
7
Maximum is:7
```

2.14 Reference Variable

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

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

- For Example

```
int x=5;
int &y=x;
```

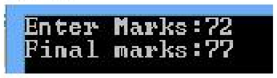
Value of x is 5 and after initialization y also becomes 5

- **Principles for declaring reference variable**
  - Reference variable should be initialized
  - Reference variable should be referenced only to one variable
  - A variable can have more than one reference
  - Array reference not allowed

Example 2-21:

```
#include<iostream>
#include<conio.h>
using namespace std;
void addgracemarks(float &);          //Function declaration with reference argument
int main()
{
        float marks;
        cout<<"Enter Marks:";
        cin>>marks;
        addgracemarks(marks);
        cout<<"Final marks:"<<marks;
        getch();
        return 0;
}
void addgracemarks(float &m)
{
        m+=5;
}
```

Output:



```
Enter Marks:72
Final marks:77
```

2.15 Default argument

- When declaring a function, we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.
- It is used mainly in function call whose parameters are always same.
- One important point to note is that the trailing argument can have default values. That is the default argument must add from right to left. We cannot provide a default value to a particular argument in the middle or first of an argument list.
- For example:

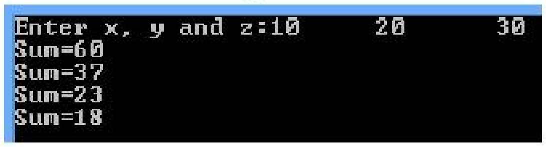
```
float add (int a, int b = 2, int c = 5);        //This is legal because trailing argument is default.
int add (int a = 6, int b);                     // This is illegal because trailing argument is not default
int add (int a, int b = 5, int c);              // This is illegal because trailing argument is not default
int add (int a = 5, int b, int c = 6);          // This is illegal because middle argument is not default
int add (int a = 5, int b = 6, int c = 7);      //This is legal because the argument are default from right to left
```

Example 2-22:

```
#include<iostream>
#include<conio.h>
using namespace std;
int add(int a=5, int b=6, int c=7);
int main()
{
        int x,y,z;
        cout<<"Enter x, y and z:";
        cin>>x>>y>>z;
        cout<<"Sum="<<add(x,y,z)<<endl;
        cout<<"Sum="<<add(x,y)<<endl;
        cout<<"Sum="<<add(x)<<endl;
        cout<<"Sum="<<add()<<endl;
        getch();
        return 0;
}
```

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

Output:



```
Enter x, y and z:10     20      30
Sum=60
Sum=37
Sum=23
Sum=18
```

2.16 State and Behavior of Object

❖ **State**

- An objects state is defined by the attributes (i.e. data members or variables) of the object
- In OOP state is defined as data members
- It is determined by the values of its attributes
- What the objects have, Example: Student have a first name, last name, age, etc...

❖ **Behavior**

- An objects behavior is defined by the methods or action (i.e. Member functions) of the object
- In OOP behaviors are defined as member function
- It determines the actions of an object
- What the objects do, Example Student attend a course "OOP", "C programming" etc…
- **Example:** Consider Lamp as an Object
  Its states are on/off and behavior are turn on/ turn of

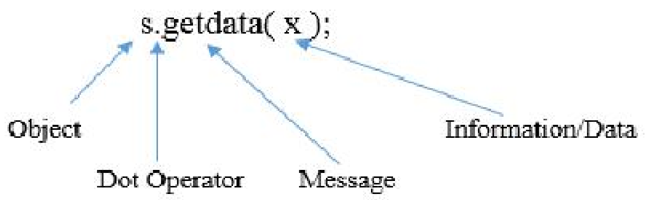
2.17 Responsibility of Object

An object must contain the data (attributes) and code (methods) necessary to perform any and all services that are required by the object. This means that the object must have the capability to perform required services itself or at least know how to find and invoke these services.

3 Message, Instance and Initialization

3.1 Message Passing

- Message is piece of communication (Exchange of data/information between sender and receiver)
- In OOP object communicate with help of message passing.
- Data is transferred from one object to another using message
- Execution of member function is response guaranteed due to receipt of message
- An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:
  - Creating classes that define objects and their behavior.
  - Creating objects from class definitions, and
  - Establishing communication among objects.
- Objects communicate with one another by sending and receiving information much the same way as people pass message to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.
- A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Message passing syntax using object **s**



Example 3-1: Example of message passing

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
                int roll;
        public:
                void getdata(int x)
                {
                        roll=x;
                }
                void display()
                {
                        cout<<"Roll number="<<roll;
                }
};
int main()
{
        student s;
        s.getdata(325);          //object s passing message
        s.display();             //object s passing message
        getch();
        return 0;
}
```

3.2 Constructor

A constructor is a special member function which initializes the objects of its class. It is called special because its name is same as that of the class name. The constructor is automatically executed whenever an object is created. Thus, a constructor helps to initialize the objects without making a separates call to a member function. It is called constructor because it constructs values of data members of a class.

Characteristics of constructor

- Constructor has same name as that of its class name.
- It should be declared in public section of the class.
- It is invoked automatically when objects are created.
- It cannot return any value because it does not have a return type even void.
- Like other C++ functions, they can have default arguments.
- It cannot be a virtual function and we cannot refer to its address.
- It cannot be inherited, though a derived class can call the base class constructor
- An object with a constructor (or destructor) cannot be used as a member of a union.
- It makes implicit call to operators **new** and **delete** when memory allocation is required.

❖ Default Constructors (Constructor without argument)

- A constructor that accepts no arguments (parameters) is called the default constructor.

Example 3-2:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
        int a,b;
        public:
                sample()                //default constructor
                {
                        a=10;
                        b=20;
                }
                int sum()
                {
                        return (a+b);
                }
};
int main()
{
        sample s;               //constructor called
        cout<<"Output is:"<<s.sum();
        getch();
        return 0;
}
```

- In above example **sample** is class, the constructor is

```
        sample()
        {
                a=10;
                b=20;
        }
```

  When we create the object of above class this is automatically called and therefore 10 is assigned to a, 20 is assigned to b. In the body of constructor we can write any statement comments also.
- The body of constructor can also be written outside the class like other member function.
- For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
        int a,b;
        public:
                sample();
                int sum();

};
sample::sample()
                {
                        a=10;
                        b=20;
                }
int sample::sum()
                {
                        return (a+b);
                }
int main()
{
        sample s;               //constructor called
        cout<<"Output is:"<<s.sum();
        getch();
        return 0;
}
```

❖ Constructor with argument (Parameterized Constructor)

- It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when we declare an object, specify the arguments.
- When argument are used in constructor they are known as parameterized constructor.
- We can use argument in constructor as follows:

```
class sample
{
        int a;
        public:
                sample(int x)           //constructor with argument
                {
                        a=x;
                }
                void display()
                {
                        cout<<"Value of a="<<a;
                }
};
```

- The object of such type of class is created as follows:

```
sample s1(20)           //By calling the constructor implicitly
        or
sample s2 = sample(20)  //By calling the constructor explicitly
```

Example 3-3:

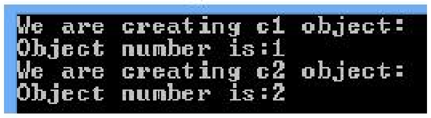
```
#include<iostream>
```

```
#include<conio.h>
using namespace std;
class sample
{
        int a,b;
        public:
                sample(int x, int y)
                {
                        a=x;
                        b=y;
                }
                int sum()
                {
                        cout<<a+b<<endl;
                }
};
int main()
{
        sample s1(10,20);                       //implicit call
        cout<<"sum of object s1 is:";
        s1.sum();
        sample s2=sample(30,20);                //explicit call
        cout<<"Sum of object s2 is:";
        s2.sum();
        getch();
        return 0;
}
```

Example 3-4: Write a program which print the object number whenever we create an object

```
#include<iostream>
#include<conio.h>
using namespace std;
class counter
{
        static int count;
        public:
                counter()
                {
                        count++;
                        cout<<"Object number is:"<<count<<endl;
                }
};
int counter::count;
int main()
{
        cout<<"We are creating c1 object:"<<endl;
        counter c1;
        cout<<"We are creating c2 object:"<<endl;
        counter c2;
        getch();
        return 0;
}
```

Output:



```
We are creating c1 object:
Object number is:1
We are creating c2 object:
Object number is:2
```

3.2.1 Multiple Constructor (Constructor Overloading)

- Like function Overloading Constructors also can be overloaded.
- In one class it is possible to declare more than one constructors

- Class contains more than one constructors, this is called constructor overloading.
- All constructor have same name of class, But contains different number of arguments
- Depending upon the arguments number, compiler executes appropriate constructor.
- For example:

```
class sample
{
        int a,b,c;
        public:
                sample()
                {
                        a=0; b=0; c=0;
                }
                sample(int x, int y)
                {
                        a=x; b=y; c=0;
                }
                sample(int x, int y, int z)
                {
                        a=x; b=y; c=z;
                }
};
```

In above example, we have declared three constructors in class **sample**. The first constructor receives no argument, second receives two arguments and third receives three arguments.
We can create three different types of objects for above class **sample** like

1. sample s1;
2. sample s2(10, 20);
3. sample s3(5, 6, 7);

Here, object **s1** automatically invokes the constructor which has no argument so, **a, b** and **c** of object s1 are initialized by value zero (0).
In object **s2**, this automatically invokes the constructor which has two arguments, so the value of **a, b** and **c** are 10, 20, 0.
In object **s3**, this automatically invokes the constructor which has three arguments, so the value of **a, b** and **c** are 5, 6, 7.

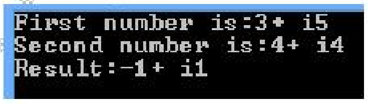
- Thus, more than one constructor is possible in a class. We know that sharing the same name by two or more functions is called function overloading. Similarly, when more than one constructor is defined in a class, this is known as constructor overloading.

Example 3-5: Write a Program which calculates the subtraction of two complex numbers: a and b, when a = 3+i5 and b = 4+i4

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
        float realp,imagp;
        public:
                complex(){ }                    //constructor no argument
                complex(float x)                //constructor one argument
                {
                        realp=x;
                        imagp=x;
                }
                complex(float x, float y)//constructor two argument
                {
                        realp=x;
                        imagp=y;
```

```
            }
            void sub(complex c1, complex c2)
            {
                realp=c1.realp-c2.realp;
                imagp=c1.imagp-c2.imagp;
            }
            void display()
            {
                cout<<realp<<"+ i"<<imagp<<endl;
            }
};
int main()
{
    complex a(3,5);
    complex b(4);
    complex c;
    c.sub(a,b);
    cout<<"First number is:";
    a.display();
    cout<<"Second number is:";
    b.display();
    cout<<"Result:";
    c.dispchlay();
    get ();
    return 0;
}
```

Output:



```
First number is:3+ i5
Second number is:4+ i4
Result:-1+ i1
```

3.3 Copy Constructor

- Copy of constructor are used to copy one object to other one.
- The constructor which pass reference of object as argument to constructor.
- All copy constructor required one object argument.
- The general declaration for copy constructor is:

```
        class sample
        {
            int a,b;
            public:
                sample(){ }                     //constructor
                sample(sample & s1)             //copy constructor
                {
                    a=s1.a;
                    b=s1.b;
                }
        };
```

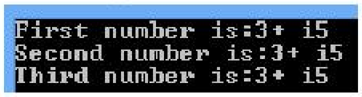
- We can create object for above as follows:
  1. sample s1;
  2. sample s2(s1);
     or
     sample s2 = s1;
- The statement **sample s1;** calls the no argument constructor. While the statement **sample s2(s1);** calls the copy constructor. The statement **sample s2 = s1** also called copy constructor.
- Note the header of copy constructor is written as

class-name (class-name & object-ref)

Example 3-6: Write a program which reads a complex number, copy that into another. Use copy constructor for writing program.

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
        float realp,imagp;
        public:
                complex(float x, float y)//constructor two argument
                {
                        realp=x;
                        imagp=y;
                }
                complex(complex & c1)//copy constructor
                {
                        realp=c1.realp;
                        imagp=c1.imagp;
                }
                void display()
                {
                        cout<<realp<<"+ i"<<imagp<<endl;
                }
};
int main()
{
        complex a(3,5);                 //object a is created and initialized
        complex b(a);                   //copy constructor called
        complex c=a;                    //copy constructor called again
        cout<<"First number is:";
        a.display();
        cout<<"Second number is:";
        b.display();
        cout<<"Third number is:";
        c.display();
        getch();
        return 0;
}
```

Output:



```
First number is:3+ i5
Second number is:3+ i5
Third number is:3+ i5
```

Note: The above constructor is called user defined copy constructor. Actually we can classify the copy constructor into three categories:

- **User defined copy constructor**
- **Default copy constructor**
- **Hybrid copy constructor**

❖ Default copy constructor

A copy constructor is a special type of constructor by using which we can initialize the data members of newly created object by existing object. The compiler provides a default copy constructor. The syntax of calling a copy constructor is:

complex c2(c1);

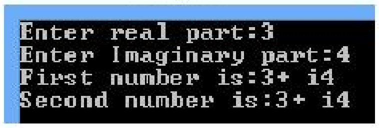
where complex is name of class, c2 is new object and c1 is existing object.

Example 3-7: Write a program which creates a class for complex number, and by using default copy constructor initialize an object of class.

```
#include<iostream>
#include<conio.h>
using namespace std;
```

```
class complex
{
        float realp,imagp;
        public:
                complex(float x, float y)//constructor two argument
                {
                        realp=x;
                        imagp=y;
                }
                void display()
                {
                        cout<<realp<<"+ i"<<imagp<<endl;
                }
};
int main()
{
        float x,y;
        cout<<"Enter real part:";
        cin>>x;
        cout<<"Enter Imaginary part:";
        cin>>y;
        complex c1(x,y);                //object c1 is created and initialized
        complex c2(c1);                 //default copy constructor called
        cout<<"First number is:";
        c1.display();
        cout<<"Second number is:";
        c2.display();
        getch();
        return 0;
}
```

Output:



```
Enter real part:3
Enter Imaginary part:4
First number is:3+ i4
Second number is:3+ i4
```

- In above example we have not written the user defined copy constructor, compiler provide the default copy constructor when the statement **complex c2(c1);** is executed. i.e. the data member of c2 is initialized by data member of c1.

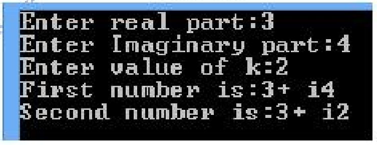
❖ Hybrid Copy Constructor

- The hybrid constructor initialize the data members of an object by using an existing object and variable. For example suppose we want to initialize object **c2**(object of complex number) partially by object **c1** and partially by a variable **k**, i.e. we want to initialize real part of **c2** by real part of **c1** and image part of **c2** by a variable **k**.

Example 3-8:

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
        float realp,imagp;
        public:
                complex(float a, float b)//constructor two argument
                {
                        realp=a;
                        imagp=b;
                }
                complex(complex &c1, float k)  //hybrid constructor
                {
                        realp=c1.realp;
                        imagp=k;
                }
                void display()
                {
                        cout<<realp<<"+ i"<<imagp<<endl;
                }
        };
        int main()
        {
              float x,y,k;
              cout<<"Enter real part:";
              cin>>x;
              cout<<"Enter Imaginary part:";
              cin>>y;
              complex c1(x,y);                    //object c1 is created and initialized
              cout<<"Enter value of k:";
              cin>>k;
              complex c2(c1,k);                   //hybrid constructor called
              cout<<"First number is:";
              c1.display();
              cout<<"Second number is:";
              c2.display();
              getch();
              return 0;
        }
```

Output:



```
Enter real part:3
Enter Imaginary part:4
Enter value of k:2
First number is:3+ i4
Second number is:3+ i2
```

- In above example real part of **c2** is initialize by real part of **c1** and image part of **c2** is initialized by value of **k**.

3.4 Constructors with default Argument

- It is possible to define a constructor with default argument like in normal function.
- For example:

```
class complex
{
        int realp,imagp;
        public:
                complex(int a, int b=0)
                {
                        realp=a;
                        imagp=b;
                }
};
```

- We can create the following type of objects for above class
  (1) complex c1(5);
  (2) complex c2(5,6);

If we create object like **(1)** then **5** is assigned to **realp** and **0** is assigned to **imagp** of **c1**, because default value of b is zero.

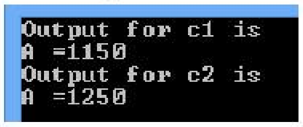
If we create object like **(2)** then **5** is assigned to **realp** and **6** is assigned to **imagp**.

Example 3-9: Write a program which calculates **A** where **A = (1 + r/100)^2** and **r** is **15** in some case. Design program by oop approach.

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
class cal
```

```
{
        int n;
        float p,r,A;
        public:
                cal(int x, float y, float z=15)                    //constructor with default argument
                {
                        p=x; n=y; r=z;
                }
                void output()
                {
                        A=p*pow((1+r/100),n);
                        cout<<"A ="<<A<<endl;
                }
};
int main()
{
        cal c1(1000,1);
        cal c2(1000,1,25);
        cout<<"Output for c1 is"<<endl;
        c1.output();
        cout<<"Output for c2 is"<<endl;
        c2.output();
        getch();
        return 0;
}
```

Output:



```
Output for c1 is
A =1150
Output for c2 is
A =1250
```

3.5 Destructors

- A destructor is used to destroy the objects that have been created by a constructor. A destructor is a member function like constructor.
- The name of destructor is same as constructor (i.e. class name) but is preceded by a tild(~).
- Destructor will automatically be called by compiler upon exit from the program to clean up storage which was taken by object.
- Note that objects are destroyed in the reverse order of creation.
- Like constructor, destructor do not have a return value. They also take no arguments (the assumption being that there's only one way to destroy an object).
- For example

```
class complex
{
        int realp,imagp;
        public:
                complex()
                {
                        realp=10; imagp=20;
                }
                ~complex(){     }                   //destructor
};
```

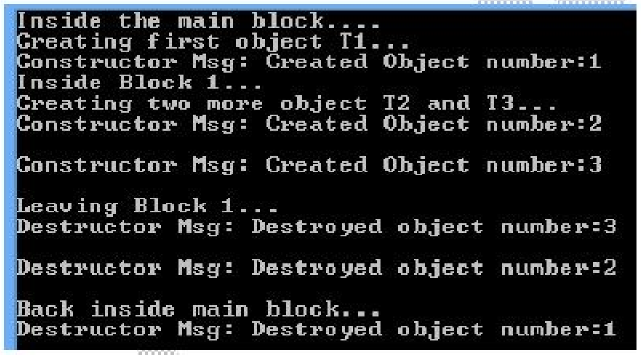
We can write statement within the body of destructor also.

Example 3-10:

```
#include<iostream>
using namespace std;
int count=0;
class test
{
        public:
```

```
                    test()
                    {
                            count++;
                            cout<<"\nConstructor Msg: Created Object number:"<<count<<endl;
                    }
                    ~test()
                    {
                            cout<<"\nDestructor Msg: Destroyed object number:"<<count<<endl;
                            count--;
                    }
    };
    int main()
    {
            cout<<"Inside the main block....";
            cout<<"\nCreating first object T1...";
            test T1;
            {
                    //block1
                    cout<<"Inside Block 1...";
                    cout<<"\nCreating two more object T2 and T3...";
                    test T2,T3;
                    cout<<"\nLeaving Block 1...";
            }
            cout<<"\nBack inside main block...";
            return 0;
    }
```

Output:



```
Inside the main block....
Creating first object T1...
Constructor Msg: Created Object number:1
Inside Block 1...
Creating two more object T2 and T3...
Constructor Msg: Created Object number:2

Constructor Msg: Created Object number:3

Leaving Block 1...
Destructor Msg: Destroyed object number:3

Destructor Msg: Destroyed object number:2

Back inside main block...
Destructor Msg: Destroyed object number:1
```

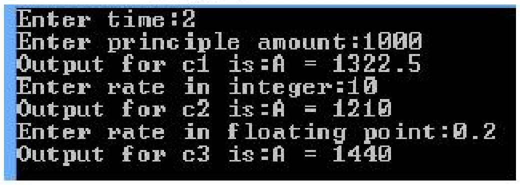
3.6 Dynamic Initialization of Objects

- The initial value of an object may be provided during run time
- One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors.
- This provides the flexibility of using different format of data at run time depending upon the situations

Example 3-11:

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
class calc
{
        int n;
        float p,r,A;
        public:
                calc(){  }
                calc(float x, int y,float z=0.15)
                {
                        p=x;
                        n=y;
                        r=z;
                }
                calc(float x,int y,int z)
                {
                        p=x;
                        n=y;
                        r=float(z)/100;
                }
                void output()
                {
                        A=p*pow((1+r),n);
                        cout<<"A = "<<A<<endl;
                }
};
int main()
{
        int t,irate;
        float m,frate;
        calc c1,c2,c3;
        cout<<"Enter time:";
        cin>>t;
        cout<<"Enter principle amount:";
        cin>>m;
        c1=calc(m,t);
        cout<<"Output for c1 is:";
        c1.output();
        cout<<"Enter rate in integer:";
        cin>>irate;
        c2=calc(m,t,irate);
        cout<<"Output for c2 is:";
        c2.output();
        cout<<"Enter rate in floating point:";
        cin>>frate;
        c3=calc(m,t,frate);
        cout<<"Output for c3 is:";
        c3.output();
        getch();
        return 0;
}
```

Output:



```
Enter time:2
Enter principle amount:1000
Output for c1 is:A = 1322.5
Enter rate in integer:10
Output for c2 is:A = 1210
Enter rate in floating point:0.2
Output for c3 is:A = 1440
```

3.7 Memory Mapping, Allocation and Recovery

3.7.1 Stack & Heap Memory

❖ **Stack**

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

Advantages

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

Limitations

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

❖ **Heap**

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

Stack vs. Heap Pros and Cons

Stack

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

Heap

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

3.7.2 New and Delete Operator

- C uses **malloc( )** and **calloc( )** functions to allocate memory dynamically at run time. Similarly, it uses the function **free( )** to free dynamically allocated memory. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as free store operators.
- An object can be created by using new and destroyed by using delete as and when required.

❖ new Operator

- The **new** operator can be used to create objects of any type. Its syntax is

  pointer_variable=new data-type;
- Here, pointer-variable is a pointer of type data-type. The **new** operator allocates sufficient memory to hold a data object of type data-type and return the address of the object.
- The data-type may be any valid data type. The pointer variables holds the address of the memory space allocated.
- For example:

```
int *a;
a = new int;
```

- Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p=new int ;
float *q=new float ;
```

- subsequently, the statements

```
*p=25 ;
*q=7.5 ;
```

  assign 25 to the newly created int object and 7.5 to the float object.
- We can also initialize the memory using new operator. This is done as

  pointer_variable=new data-type(value)

  E.g.

```
int *p=new int(25);
float *q=new float(7.5);
```

- **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes.
- The general form for a one-dimension array is

  pointer_variable=new data_type[size];

  Here, size specifies the number of elements in the array. For example,

  int *p=new int[10]; creates a memory space for an array of 10 integers.
- For example we are creating a matrix dynamically

  Note that pointer to pointer means pointer to array i.e. pointer to matrix (two dimensional array)

```
int **p;
```

  Now p is pointer to matrix and we can create the space as follows:

```
p = new int *[r];
```

  where r is any integer value. This statement creates an array of pointer and assigns that to p. Now we can create space for each row as follows:

```
for ( int i = 0; i<r; i++)
{
        p[i] = new int[c];
}
```

  The statement **p[i] = new int[c];** is executed **r** times therefore this creates space for **r** rows and each space length is equal to space for **c** int type elements.

❖ delete Operator

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program.
- When a data object is no longer needed, it is destroyed to release the memory space for reuse. For this purpose, we use delete operator. The general syntax is

  delete pointer_variable;

  The pointer_variable is the pointer that points to a data object created with new. For e.g.

delete p ;
delete q ;

- If we want to free a dynamically allocated array, we must use the following form of delete.

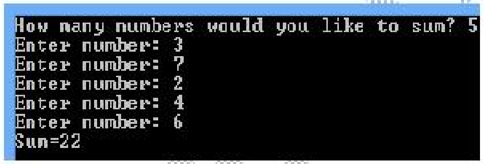
delete [size] pointer_variable ;

- The size specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent version of C++ do not require the size to be specified.

Example 3-12:

```
#include <iostream>
#include <conio.h>
using namespace std;
int main ()
{
        int i,n,sum=0;
        int *p;                         //Declaration of pointer
        cout << "How many numbers would you like to sum? ";
        cin >>n;
        p=new int[n];                   //Dynamic memory allocation for p
        for (i=0; i<n; i++)
        {
                cout << "Enter number: ";
                cin >> p[i];
        }
        for (i=0; i<n; i++)
        {
                sum+=p[i];
        }
        cout<<"Sum="<<sum;
        delete[] p;                     //Deallocation of variable p
        getch();
        return 0;
}
```

Output:



```
How many numbers would you like to sum? 5
Enter number: 3
Enter number: 7
Enter number: 2
Enter number: 4
Enter number: 6
Sum=22
```

3.8 Dynamic Constructor & Destructor (Dynamic Memory allocation using constructor & Destructor)

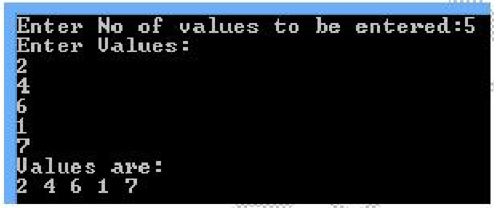
- The constructor can also be used to allocate memory while creating objects. Allocation of memory to object at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of new operator.
- Memory is deallocate using delete operator inside destructor.

Example 3-13:

```
#include <iostream>
#include <conio.h>
using namespace std;
class sample{
                int *a;
                int n,i;
        public:
                sample (int x){                 //Dynamic constructor
```

```
                        n=x;
                        a= new int [n];            // a is n number of locations for intger type
                    }
            void input(){
                        cout<<"Enter Values:"<<endl;
                        for(int i=0;i<n;i++)
                        cin>>a[i];
                    }
            void output(){
                    cout<<"Values are: "<<endl;
                    for(int i=0;i<n;i++)
                    cout<<a[i]<<" ";
                    }
            ~sample() {
                        delete a;
                      }
        };
        int main ()
        {
            int num;
            cout<<"Enter No of values to be entered:";
            cin>>num;
            sample e(num);                  //calling dynamic constructor
            e.input();
            e.output() ;
            getch();
            return 0;                       //Automatically call destructor
        }
```

Output:



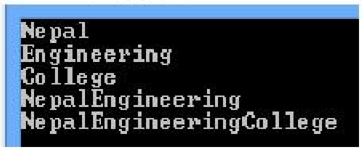
```
Enter No of values to be entered:5
Enter Values:
2
4
6
1
7
Values are:
2 4 6 1 7
```

In above example from main object passes the no of integer memory to be allocated to the constructor and inside constructor there is new operator which will allocates the memory according to the value of argument passed from main. At end of program it automatically call destructor

Example 3-14:

```
            #include <iostream>
            #include<string.h>
            #include <conio.h>
            using namespace std;
            class sample
            {
                char *name;
                int length;
                public:
                    sample()
                    {
                        length=0;
                        name=new char[length+1];
                }
                sample(char *p)
                {
                        length=strlen(p);
                        name=new char[length+1];
                        strcpy(name,p);
                }
                void display()
                {
                        cout<<name<<endl;
                }
                void join(sample &a, sample &b);
};
void sample::join(sample &a, sample &b)
{
        length=a.length+b.length;
        delete name;
        name=new char[length+1];
        strcpy(name,a.name);
        strcat(name,b.name);
}
int main()
{
        char *first="Nepal";
        sample name1(first),name2("Engineering"),name3("College"),s1,s2;
        s1.join(name1,name2);
        s2.join(s1,name3);
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();
        getch();
        return 0;
}
```

Output:



```
Nepal
Engineering
College
NepalEngineering
NepalEngineeringCollege
```

- This program uses two constructors. The first is an empty constructor that allows us to declare an array of string. The second constructor initialize the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end of string character **'\0'**.
- The member function **join( )** concatenates the two string. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string function **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. the **main( )** function program concatenates three string into one string.

4 Object Inheritance and Reusability

4.1 Introduction to inheritance

- Inheritance is the most powerful feature of object-oriented programming after classes and objects.
- Inheritance is the process of creating a new class, called derived class from existing class, called base class. The derived class inherits some or all the traits from base class. The base class is unchanged by this.
- Most important advantage of inheritance is reusability. Once a base class is written and debugged, it need not be touched again and we can use this class for deriving another class if we need.
- Reusing existing code saves time and money. By reusability a programmer can use a class created by another person or company and without modifying it derive other class from it.

REUSABILITY

- Reusability means reusing the properties of base class in the derived class
- Reusability permits usage of members of base class
- Reusability is the outcome of inheritance

Visibility Modifier (Access Specifier):

- Private member can be accessed only within the class in which they are defined while public can be accessed from outside the class also.
- The protected member can be accessed within the class in which they are defined and they are also accessed in derived class which is derived by its own class. Note that protected member cannot be accessed from outside these classes.

Visibility Modifier (Access Specifier)	Accessible from own class	Accessible from derived class	Accessible from objects outside class
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

Base Class Visibility	Derived Class Visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

4.2 FORMS OF INHERITANCE (SUB CLASS/SUB TYPES)

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

- **Sub classing for specialization (subtype):** The derived child class is a specialized form of the parent class, in other words, the child class is subtype/subclass of the parent
- **Sub classing for specification:** The parent class defines 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
- **Sub classing for construction:** The child class can be constructed form parent classes by implementing the behaviors of parent classes
- **Sub classing for generalization:** Sub classing for generalization is the opposite to sub classing for specifications. The base class holds the common properties that will inherit to the derived class.
- **Sub classing for extension:** The subclass for extension adds new functionality in child class form base class while designing new child class. It is done simply add new function other than parent class have.
- **Sub classing for limitation:** The subclass for limitation occurs when the behavior of subclass is similar or more dependent to the behavior of parent class.
- **Sub classing for variance:** The child class and parent class are varied when the level of inheritance increased, and the class and subclass relationship is imaginary.
- **Sub classing for combination:** The child class inherits features from more than one classes

4.3 Defining Derived Class (Specifying Derived Class)

- A derived class can be defined by specifying its relationship with the base class in addition to its own detail.
- The general syntax is

  class derived_class_name : visibility_mode **base_class_name**

  {

  //members of derived class

  };

  where, the colon (:) indicates that the derived_class_name is derived from the base_class_name. The visibility_mode is optional, if present, may be either private or public. The default visibility mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived or derived on protected.

- Example:

```
class ABC : private XYZ              // private derivation
{
        //members of ABC
};
class ABC : public XYZ               // public derivation
{
        //members of ABC
};
class ABC : protected XYZ            // protected derivation
{
        //members of ABC
};
class ABC : XYZ                      // private derivation by default
{
        //members of ABC
};
```

- While any **derived_class** is inherited from a **base_class**, following things should be understood:
  - When a base class is publicly inherited by a derived class the private members are not inherited, the public and protected are inherited. The public members of base class becomes public in derived class whereas protected members of base class becomes protected in derived class.
  - When a base class is protectly inherited by a derived class, then public members of base class becomes protected in derived class, protected members of base class becomes protected in the derived class, the private members of the base class are not inherited to derived class but note that we can access private member through inherited member function of the base class.
  - When a base class is privately inherited by a derived class, only the public and protected members of base class can be accessed by the member functions of derived class. This means no private member of the base class can be accessed by the objects of the derived class. Public and protected member of base class becomes private in derived class.

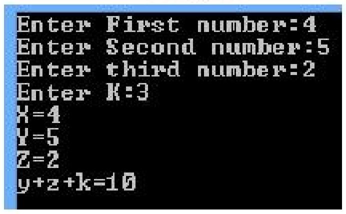
4.3.1 Public Inheritance

If the derivation type is public then the inheritance is known as public inheritance.

Example 4-1:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        private:
                int x;
        protected:
                int y;
        public:
                int z;
                void getdata()
                {
                        cout<<"Enter First number:";
                        cin>>x;
                        cout<<"Enter Second number:";
                        cin>>y;
                        cout<<"Enter third number:";
                        cin>>z;
                }
                void showdata()
                {
                        cout<<"X="<<x<<endl;
                        cout<<"Y="<<y<<endl;
                        cout<<"Z="<<z<<endl;
                }
};

class D:public B
{
        private:
                int k;
        public:
                void getk()
                {
                        cout<<"Enter K:";
                        cin>>k;
                }
                void output()
                {
                        int s;
                        s=y+z+k;
                        cout<<"y+z+k="<<s<<endl;
                }
};
int main()
{
        D d1;
        d1.getdata();
        d1.getk();
        d1.showdata();
        d1.output();
        getch();
        return 0;
}
```

Output:



```
Enter First number:4
Enter Second number:5
Enter third number:2
Enter K:3
X=4
Y=5
Z=2
y+z+k=10
```

Note: In above example x is private member of class B and cannot be inherited but objected of D are able to access it through an inherited member function of B (i.e. through getdata( ) and showdata( ) of class B.

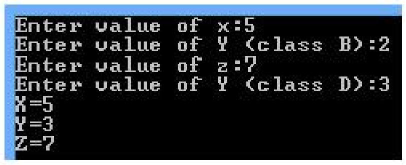
4.3.2 Protected Inheritance

- If the derivation type is protected then inheritance is known as protected inheritance.

Example 4-2:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        private:
                int x;
        protected:
                int y;
        public:
                void getdata();
                void showdata();
};
class D:protected B
{
        private:
                int z;
        public:
                void getd();
                void showd();
};
void B::getdata()
{
        cout<<"Enter value of x:";
        cin>>x;
        cout<<"Enter value of Y (class B):";
        cin>>y;
}
void B::showdata()
{
        cout<<"X="<<x<<endl;
        cout<<"Y="<<y<<endl;
}
void D::getd()
{
        getdata();
        cout<<"Enter value of z:";
        cin>>z;
        cout <<"Enter value of Y (class D):";
        cin>>y;
}
void D::showd()
{
        showdata();
        cout<<"Z="<<z<<endl;
}
int main()
{
        D d1;
        d1.getd();
        d1.showd();
        getch();
        return 0;
    }
```

Output:



```
Enter value of x:5
Enter value of Y (class B):2
Enter value of z:7
Enter value of Y (class D):3
X=5
Y=3
Z=7
```

4.3.3 Private Inheritance

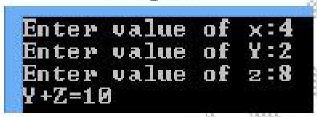
- If the derivation type is private then inheritance is known as private inheritance.

Example 4-3:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        private:
                int x;
        protected:
                int y;
        public:
                void getdata();
                void showdata();
};
class D:private B
{
        private:
                int z;
        public:
                void input();
                void output();
};
void B::getdata()
{
        cout<<"Enter value of x:";
        cin>>x;
        cout<<"Enter value of Y:";
        cin>>y;
}
void B::showdata()
{
        cout<<"X="<<x<<endl;
        cout<<"Y="<<y<<endl;
}
void D::input()
{
        getdata();
        cout<<"Enter value of z:";
        cin>>z;
}
```

```
void D::output()
{
        int f;
        f=y+z;
        cout<<"Y+Z="<<f<<endl;
}
int main()
{
        D d1;
        d1.input();
        d1.output();
        getch();
        return 0;
}
```

Output:



```
Enter value of x:4
Enter value of Y:2
Enter value of z:8
Y+Z=10
```

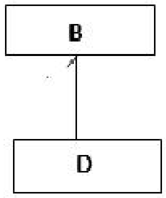
4.4 Type of Inheritance

A class can also inherit properties from more than one class or from more than one level. According to this we can classify inheritance into following type:

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

4.4.1 Single Inheritance

- If a class is derived from only one base class and derived class is not used as base class again then that is known as single inheritance.
- One base class and one derived class only involved.
  B is base class
  D is derived class
  D is not used as base class again
  No further extension of derived class



- All data members in protected and public of base class B can be accessed in derived class D.
- General form

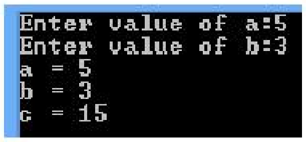
```
class B
{
        protected:
        private:
        public:
};
class D: public B
{
        private:
        public:
};
```

Example 4-4:

```
#include<iostream>
#include<conio.h>
using namespace std;
```

```
class B                                 //base class
{
                int a;                  //private not inheritable
        public:
                int b;
                void set_ab();
                int get_a();
};
class D:public B                        //derived class from B
{
                int c;
        public:
                void mul();
                void display();
};
void B::set_ab()
{
        cout<<"Enter value of a:";
        cin>>a;
        cout<<"Enter value of b:";
        cin>>b;
}
int B::get_a()
{
        return a;
}
void D::mul()
{
        c=b*get_a();
}
void D::display()
{
        cout<<"a = "<<get_a()<<endl;
        cout<<"b = "<<b<<endl;
        cout<<"c = "<<c<<endl;
}
int main()
{
        D d1;
        d1.set_ab();
        d1.mul();
        d1.display();
        getch();
        return 0;
}
```

Output:



```
Enter value of a:5
Enter value of b:3
a = 5
b = 3
c = 15
```

4.4.2 Multiple Inheritance

➢ If a class is derived from more than one base class then inheritance is called as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as starting point for defining new class. The syntax of multiple inheritance is:

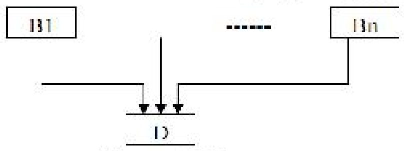
```
class D: derivation B1, derivation B2 ........
{
                //Member of class D
};
```

The derivation is private, public or protected. Note this is also possible that one derivation is public and another one is protected or private, etc.

- For example

```
(1) class D : public B1, public B2
        {
                private:
                        int a;

        };
(2) class D : public B1, protected B2
        {
                private:
                        int a;

        };
(3) class D : private B1, protected B2, public B3
        {
                private:
                        int a;

        };
```

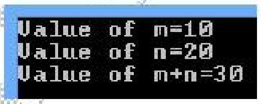
If B1, B2 and Bn are three classes from which class D is derived then we can draw the multiple inheritance as



Example 4-5:

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample1                   //base class
{
        protected:
                int m;
        public:
                void get_m(int);
};
class sample2                   //base class
{
        protected:
                int n;
        public:
                void get_n(int);
};
class sample3:public sample1, public sample2            //derived class from sampl1 and sample2
{
        public:
                void display();
};
void sample1::get_m(int x)
{
        m=x;
}
void sample2::get_n(int y)
{
        n=y;
}
void sample3::display()
{
        cout<<"Value of m="<<m<<endl;
        cout<<"Value of n="<<n<<endl;
        cout<<"Value of m+n="<<m+n<<endl;
}
int main()
{
        sample3 p;
        p.get_m(10);
        p.get_n(20);
        p.display();
        getch();
        return 0;
}
```

Output:

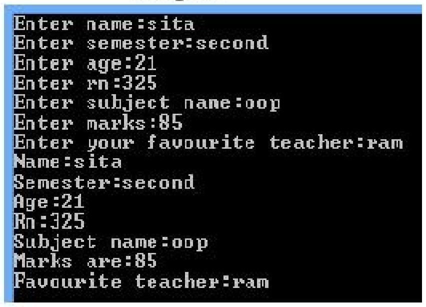


```
Value of m=10
Value of n=20
Value of m+n=30
```

Example 4-6:

```
#include <iostream>
#include <conio.h>
using namespace std;
class biodata                                   //base class
{
                char name[20];
                char semester[20];
                int age ;
                int rn ;
        public:
                void getbiodata();
                void showbiodata();
};
class marks                                     //base class
{
                char sub[10];
                float total;
        public:
                void getm();
                void showm();
};
class final: public biodata, public marks       //derived class from biodata and marks
{
                char fteacher[20];
        public:
                void getf();
                void showf();
};
void biodata:: getbiodata()
{
                cout<<"Enter name:";
                cin>>name;
                cout<<"Enter semester:";
                cin>>semester;
                cout<<"Enter age:";
                cin>>age;
                cout<<"Enter rn:";
                cin>>rn;
}
void biodata:: showbiodata()
{
                cout<<"Name:"<<name<<endl;
                cout<<"Semester:"<<semester<<endl;
                cout<<"Age:"<<age<<endl;
                cout<<"Rn:"<<rn<<endl;
}
void marks:: getm()
{
                cout<<"Enter subject name:";
                cin>>sub ;
                cout<<"Enter marks:";
                cin>>total;
}
void marks:: showm()
{
                cout<<"Subject name:"<<sub<<endl ;
                cout<<"Marks are:"<<total<<endl ; }
void final:: getf()
{
                cout<<"Enter your favourite teacher:";
                cin>>fteacher;
}
void final:: showf()
{
                cout<<"Favourite teacher:"<<fteacher<<endl;
}
int main()
{
        final f;
        f.getbiodata();
        f.getm();
        f.getf();
        f.showbiodata();
        f.showm();
        f.showf();
        getch();
        return 0;
}
```

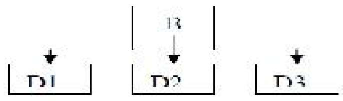
Output:



```
Enter name:sita
Enter semester:second
Enter age:21
Enter rn:325
Enter subject name:oop
Enter marks:85
Enter your favourite teacher:ram
Name:sita
Semester:second
Age:21
Rn:325
Subject name:oop
Marks are:85
Favourite teacher:ram
```

4.4.3 Hierarchical Inheritance

- When from one base class more than one classes are derived that is called hierarchical inheritance. The diagram for hierarchical inheritance is:



- Derived classes can not be used for deriving classes.
- It is also specialization (super class into sub class).
- The general format of hierarchical inheritance is:

```
class B{
            protected:
            private:
            public:
      };
class D1: public B{
            private:
            public:
      };
class D2: public B{
            private:
            public:
      };
class D3: public B{
            private:
            public:
      }
```

In above General form there are 4 classes B, D1 D2 and D3. Class B is base class and Class D1, D2, D3 is derived class. All data members in protected and public of class B can be accessed using object of classes D1, D2, D3

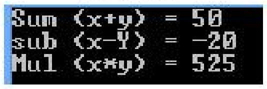
- With the help of hierarchical inheritance we can distribute the property of one class into many classes.

Example 4-7:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B                         //base class
{
      protected:
            int x,y;
      public:
            void assign()
            {
                  x=15;
                  y=35;
            }
};
class D1:public B               //derived class from B
{
      int s;
      public:
            void sum()
            {
                  s=x+y;
                  cout<<"Sum (x+y) = "<<s<<endl;
            }
};
class D2:public B               //derived class from B
{
      int t;
      public:
```

```
                void sub()
                {
                        t=x-y;
                        cout<<"sub (x-Y) = "<<t<<endl;
                }
};
class D3:public B                    //derived class from B
{
        int m;
        public:
                void mul()
                {
                        m=x*y;
                        cout<<"Mul (x*y) = "<<m<<endl;
                }
};
int main()
{
        D1 obj1;
        D2 obj2;
        D3 obj3;
        obj1.assign();
        obj1.sum();
        obj2.assign();
        obj2.sub();
        obj3.assign();
        obj3.mul();
        getch();
        return 0;
}
```

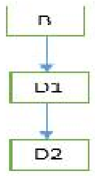
Output:



```
Sum (x+y) = 50
sub (x-Y) = -20
Mul (x*y) = 525
```

4.4.4 Multilevel Inheritance

- The mechanism of deriving a class from another derived class is called multilevel.
- When a class is derived from another derived class i.e. derived class acts as base class, such type of inheritance is called multilevel inheritance.



- In the figure, class D1 derived from class B and class D2 is derived from class D1. Thus class D1 provides a link for inheritance between B and D2 and hence it is called intermediate base class.
- Inheritance can be extended more by deriving another class from class D2.
- The general format of multilevel inheritance is:

```
class B{
                protected:
                private:
                public:
        };
class D1: public B{
                protected:
                private:
                public:
        };
```

```
class D2: public D1{
        private:
        public:
    };
```

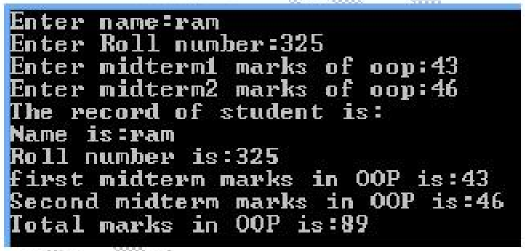
In above general form there are 3 classes B, D1 and D2. Class B is base class and class D1 is derived class from class B and also class D2 is derived from class D1. All data members in protected and public of class B can be accessed using object of class D1, D2 and data members in protected and public of class D1 can be accessed using object of class D2.

Example 4-8:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student                       //base class
{
        protected:
                char name[30];
                int rn;
        public:
                void getdata()
                {
                        cout<<"Enter name:";
                        cin>>name;
                        cout<<"Enter Roll number:";
                        cin>>rn;
                }
                void showdata()
                {
                        cout<<"Name is:"<<name<<endl;
                        cout<<"Roll number is:"<<rn<<endl;
                }
};
class marks:public student                      //derived class from student
{
        protected:
                int m1,m2;
        public:
                void getmarks()
                {
                        cout<<"Enter midterm1 marks of oop:";
                        cin>>m1;
                        cout<<"Enter midterm2 marks of oop:";
                        cin>>m2;
                }
                void showmarks()
                {
                        cout<<"first midterm marks in OOP is:"<<m1<<endl;
                        cout<<"Second midterm marks in OOP is:"<<m2<<endl;
                }
};
class result:public marks                       //derived class from marks
{
        int total;
```

```
                public:
                        void output()
                        {
                                total=m1+m2;
                                cout<<"Total marks in OOP is:"<<total<<endl;
                        }
        };
        int main()
        {
                result s;
                s.getdata();
                s.getmarks();
                cout<<"The record of student is:"<<endl;
                s.showdata();
                s.showmarks();
                s.output();
                getch();
                return 0;
        }
```

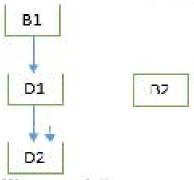
Output:



```
Enter name:ram
Enter Roll number:325
Enter midterm1 marks of oop:43
Enter midterm2 marks of oop:46
The record of student is:
Name is:ram
Roll number is:325
first midterm marks in OOP is:43
Second midterm marks in OOP is:46
Total marks in OOP is:89
```

4.4.5 Hybrid Inheritance

- If we apply more than one type of inheritance to design a problem then that is known as hybrid inheritance.
- The following diagram shows the hybrid inheritance:



Two types of inheritance is used i.e. single (B1->D1) and multiple (D1, B2 -> D2). Class D1 is derived from class B1 i.e. single inheritance and class D2 is derived from classes D1 and B2 i.e. multiple inheritance.

- There is two base classes (more than one base class involved).
- General form of Hybrid inheritance

```
class B1{
                protected:
                private:
                public:
        };
class B2{
                protected:
                private:
                public:
        };
class D1: public B1{
                protected:
                private: public:
        };
class D2: public D1, public B2{
                private:
                public:
        };
```

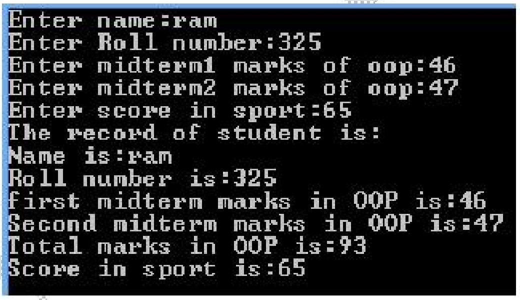
In above general form there are four classes B1, B2, D1, D2. Classes B1 & B2 are base classes and class D1 is derived class from class B1 and class D2 is derived class from D1 and B2. All the data members in protected and public of class B1 can be accessed using object of classes D1, D2 and classes D1 & B2 can be accessed by class D2.

Example 4-9:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student                                   //base class
{
        protected:
                char name[30];
                int rn;
        public:
                void getdata()
                {
                        cout<<"Enter name:";
                        cin>>name;
                        cout<<"Enter Roll number:";
                        cin>>rn;
                }
                void showdata()
                {
                        cout<<"Name is:"<<name<<endl;
                        cout<<"Roll number is:"<<rn<<endl;
                }
};
class sports                                    //base class
{
        protected:
                int score;
        public:
                void getscore()
                {
                        cout<<"Enter score in sport:";
                        cin>>score;
                }
};
class marks:public student                              //derived class from student
{
        protected:
                int m1,m2;
        public:
                void getmarks()
                {
                        cout<<"Enter midterm1 marks of oop:";
                        cin>>m1;
                        cout<<"Enter midterm2 marks of oop:";
                        cin>>m2;
                }
                void showmarks()
                {
                        cout<<"first midterm marks in OOP is:"<<m1<<endl;
                        cout<<"Second midterm marks in OOP is:"<<m2<<endl;
                }
};
```

```
class result:public marks,public sports                    //derived class from marks and sports
{
        int total;
        public:
                void output()
                {
                        total=m1+m2;
                        cout<<"Total marks in OOP is:"<<total<<endl;
                        cout<<"Score in sport is:"<<score<<endl;
                }
};
int main()
{
        result s;
        s.getdata();
        s.getmarks();
        s.getscore();
        cout<<"The record of student is:"<<endl;
        s.showdata();
        s.showmarks();
        s.output();
        getch();
        return 0;
}
```

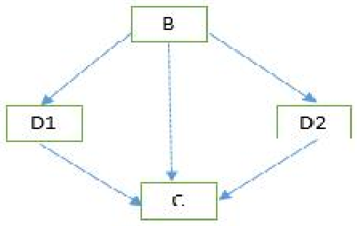
Output:



```
Enter name:ram
Enter Roll number:325
Enter midterm1 marks of oop:46
Enter midterm2 marks of oop:47
Enter score in sport:65
The record of student is:
Name is:ram
Roll number is:325
first midterm marks in OOP is:46
Second midterm marks in OOP is:47
Total marks in OOP is:93
Score in sport is:65
```

4.4.6 Multipath inheritance

- When a class is derived from 2 or more classes that are derived from same base class such type of inheritance
  - B is base class
  - Classes D1 and D2 are derived from class B
  - Both Class D1 and D2 inherit properties of Class B
  - Class C is derived from class D1 and D2
  - Ambiguity is generated
  - We make Base class as virtual for avoiding ambiguity



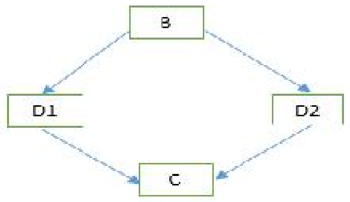
- **General form (Multipath inheritance)**

```
class B{
        protected:
        private:
        public:
};
class D1: virtual public B{
        protected:
        private:
        public:
};
class D2: virtual public B{
        protected:
        private:
        public:
};
class C:public D1, public D2{
        protected:
        private:
        public:
};
```

In General form there are 4 classes B, D1, D2 and C. Class B is base class and Class D1 & D2 is derived class from Class B and class C is derived class from classes D1 & D2, both contains properties of B. All data members in protected and public of class B can be accessed using object of classes D1, D2, C and of classes D1 & D2 can be accessed by class C. Here properties of class B is inherited to class C through two paths (B->D1->C & B->D2->C), so there will be ambiguity. To avoid ambiguity we make the base class i.e. class B as virtual

4.5 Virtual Base Class

- Consider a situation for which the diagram is shown below:

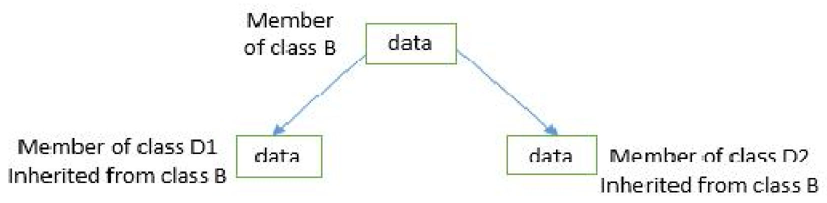


The class D1 and D2 are derived from class B and class C is derived from two classes D1 and D2. In above arrangement problem can arise if a member function in the C class want to access data or function in the class B. See the following example:

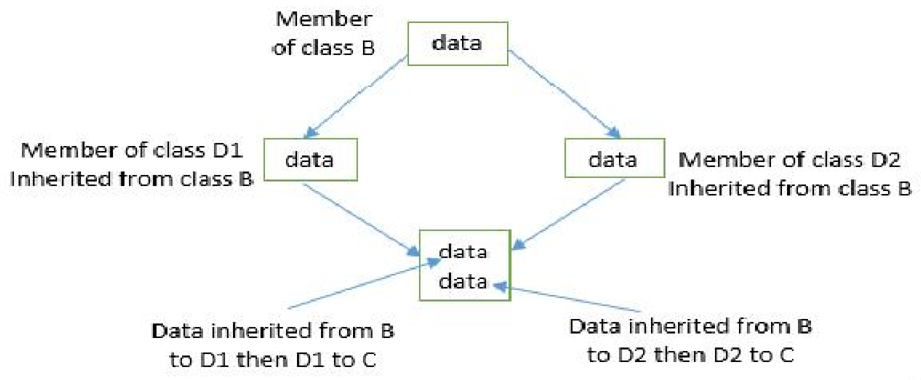
Example 4-10:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        protected:
                int data;
};
class D1:public B { };
class D2:public B { };
class C:public D1, public D2
{
        public:
                void showdata()
                {
                        data=20;                //Error
                        cout<<"Data="<<data;
                }
};
int main()
{
        C obj;
        obj.showdata();
        getch();
        return 0;
}
```

In Example 4-10 there is an error. Because showdata( ) function in class C attempts to access data which is in class B. When the class D1 and D2 are derived from class B, the member data of class B is inherit in class D1 and class D2.


Member of class B
data
Member of class D1 Inherited from class B
data
data
Member of class D2 Inherited from class B

When class C is derived from class D1 and class D2 then the member data is inherited from class D1 and from class D2 also. Therefore the class C has two copies of same data.


Member of class B
data
Member of class D1 Inherited from class B
data
data
Member of class D2 Inherited from class B
data
data
Data inherited from B to D1 then D1 to C
Data inherited from B to D2 then D2 to C

Therefore when the member function showdata( ) of class C access data the situation is ambiguous and therefore compiler gives error.

- To eliminate the ambiguity the common base class is declared as virtual base class by just writing virtual word.

for removing the error we change example 4-10 as follows:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        protected:
                int data;
};
class D1:virtual public B { };
class D2:virtual public B { };
class C:public D1, public D2
{
        public:
                void showdata()
                {
                        data=20;
                        cout<<"Data="<<data;
                }
};
int main()
{
        C obj;
        obj.showdata();
        getch();
        return 0;
}
```

- The use of keyword virtual in class D1 and D2 causes them to share a single common inherited copy of data. Now since there is only one copy of data, there is no ambiguity when it is referred to in class C. Or in other words when a class is made virtual base class, C++ takes necessary care to see the only one copy of that class is inherited, regardless of how many inheritance paths exist between virtual base class and derived class.
- Note that the keyword virtual and public may be used in either order.

4.6 Abstract class

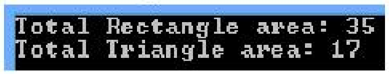
- The abstract class is one that is not used to create objects. An abstract class is used only for base class (to be inherited by other classes).
- It is design concept in program development and provides a base upon which other classes may be built.
- A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration.
- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with "= 0".

Example 4-11:

```
#include<iostream>
#include<conio.h>
using namespace std;
class Shape                     //base class
{
        protected:
          int width;
          int height;
        public:

          virtual int getArea() = 0;        // pure virtual function providing interface framework.
          void setWidth(int w)
          {
            width = w;
          }
          void setHeight(int h)
          {
            height = h;
          }
};
class Rectangle: public Shape    // Derived classes
{
        public:
          int getArea()
          {
            return (width * height);
          }
};
class Triangle: public Shape
{
        public:
          int getArea()
          {
            return (width * height)/2;
          }
};

int main( )
{
      Rectangle Rect;
      Triangle  Tri;
      Rect.setWidth(5);
      Rect.setHeight(7);
      // Print the area of the object.
      cout << "Total Rectangle area: " << Rect.getArea() << endl;
```

```
        Tri.setWidth(5);
        Tri.setHeight(7);
        // Print the area of the object.
        cout << "Total Triangle area: " << Tri.getArea() << endl;
        getch();
        return 0;
}
```

Output:



```
Total Rectangle area: 35
Total Triangle area: 17
```

- In above program class Shape is Abstract class.

4.7 Casting Base Class Pointer to Derived Class Pointers

- Pointers cannot be used only in the base class but also in derived class. Pointers to objects of a base class are type compatible with pointers to objects of derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes.
- For example, if B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be a pointer to D. Consider the following declaration:

```
B *cptr;      // pointer to class B type variable
B b;          // base object
D d;          // derived object
cptr=&b;      // cptr points to object b
```

We can make cptr to point to the objects d as follows:

```
cptr=&d;      // cptr points to object d
```

This is valid in C++ because d is an object derived from the class B. However, there is a problem in using cptr to access the public members of the derived class D. Using cptr, we can access only those members which are inherited from B and not the members that originally belong to D. In case a member of D has the same name as one of the members of B, then any reference to that member by cptr will always access the base class member.

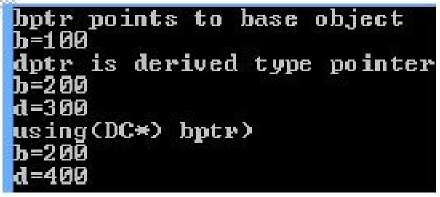
Although C++ permits a base pointer to point any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Example 4-12:

```
#include<iostream>
#include<conio.h>
using namespace std;
class BC
{
        public:
                int b;
        void show( )
        {
                cout<<"b="<<b<<endl;
        }
};
class DC: public BC
{
        public:
                int d;
        void show( )
        {
                cout<<"b="<<b<<endl<<"d="<<d<<endl;
        }
};
```

```
int main( )
{
        BC*bptr;                //base pointer
        BC base;
        bptr=&base;             //base address
        bptr->b=100;            //access BC via base pointer
        cout<<"bptr points to base object \n";
        bptr->show();
        DC derived;             //derived class
        bptr=&derived;          //address of derived class's object
        bptr->b=200;            // access DC via base pointer

        /*
        bptr->d=300;            //won't work
        cout<<"bptr now points to derived object \n";
        bptr->show();            //bptr now point to derived object
        */
        /*accessing d using a pointer to type derived class DC */
        DC*dptr;                //derived type pointer
        dptr=&derived;
        dptr->d=300;
        cout<<"dptr is derived type pointer \n";
        dptr->show();
        cout<<"using(DC*) bptr)\n";
        ((DC*)bptr)->d=400;      // cast bptr to DC type
        ((DC*)bptr)->show();
        getch();
        return 0;
}
```

Output:



```
bptr points to base object
b=100
dptr is derived type pointer
b=200
d=300
using(DC*) bptr)
b=200
d=400
```

4.8 Constructors and Destructors in inheritance

It is possible for the base class, the derived class or both to have constructor and / or destructor. When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order. That is the base class constructor is executed before the constructor in the derived class. The reverse is true for destructor functions: the destructor in the derived class is executed before the base class destructor.

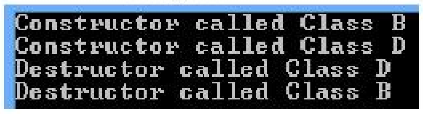
- Constructor is used to initialize variables and allocation of memory of object
- Destructor is used to destroy objects
- Compiler automatically calls constructer of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance constructor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

❖ Default Constructor (No argument) in Inheritance

Example 4-13:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B                 //Base Class
{
    public:
            B()                     //Base Class B Constructor
            {
                    cout<<"Constructor called Class B"<<endl;
            }
            ~B()
            {
                    cout<<"Destructor called Class B"<<endl;
            }
};
class D:public B            //Derived class
{
            public:
                    D()                     //Derived Class D Constructor
                    {
                            cout<<"Constructor called Class D"<<endl;
                    }
                    ~D()                    //Derived Class D Destructor
                    {
                            cout<<"Destructor called Class D"<<endl;
                    }
};
int main()
{
            D obj;                  //Derived class object obj
            return 0;
}
```

Output:



```
Constructor called Class B
Constructor called Class D
Destructor called Class D
Destructor called Class B
```

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

<u>**Note:**</u> If there is no constructor specify in derived class then derived class will use the appropriate constructor (i.e. no argument constructor) of base class.

❖ Parameterized constructor (with argument) in inheritance

- ➢ In parameterized constructor it is compulsory to have derived constructor if there base class constructor.
- ➢ Derived class constructor is used to pass arguments to the base class.
- ➢ If derived class constructor is not available, it is not possible to pass arguments from derived class object to base class constructor.
- ➢ If we need to pass an argument to the constructor of the base class, a little more effort is needed:
    - o All necessary arguments to both the class and derived class are passed to the derived class constructor.
    - o Using an expanded form of the derived class' constructor declaration, we then pass the appropriate arguments along to the base class.

The syntax for passing an argument from the derived class to the base class is as

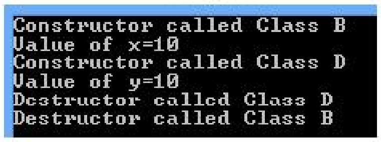
```
derived_constructor(arg_list) : base(arg_list)
{
        body of the derived class constructor
}
```

Here, base is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is possible for the derived class to ignore all arguments and just pass them along to the base.

Example 5-14: Parameterized constructor passing same argument for base class and derived class

```
#include<iostream>
#include<conio.h>
using namespace std;
class B                     //Base Class
{
        int x;
  public:
        B(int n)                    //Base Class B Constructor with one argument
        {
                x=n;
                cout<<"Constructor called Class B"<<endl;
                cout<<"Value of x="<<x<<endl;
   }
        ~B()
        {
                cout<<"Destructor called Class B"<<endl;
        }
};
class D:public B            //Derived class
{
        int y;
        public:
                D(int m):B(m)           //Derived Class D Constructor passing argument to base class B
                {
                        y=m;
                        cout<<"Constructor called Class D"<<endl;
                        cout<<"Value of y="<<y<<endl;
                }
                ~D()                //Derived Class D Destructor
                {
                        cout<<"Destructor called Class D"<<endl;
                }
};
int main()
{
        D obj(10);              //Derived class object obj passing argument to derived class
        return 0;
}
```

Output:



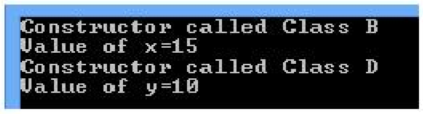
```
Constructor called Class B
Value of x=10
Constructor called Class D
Value of y=10
Destructor called Class D
Destructor called Class B
```

Example 4-15: Parameterized constructor passing different argument for base class and derived class

```
#include<iostream>
#include<conio.h>
using namespace std;
class B                     //Base Class
{
        int x;
  public:
        B(int n)                    //Base Class B Constructor with one argument
        {
                x=n;
                cout<<"Constructor called Class B"<<endl;
```

```
                    cout<<"Value of x="<<x<<endl;
    }
};
class D:public B                    //Derived class
{
          int y;
          public:
               D(int m,int n):B(n)      //Derived Class D Constructor m is used in derived class and n is passed to base class B
               {
                    y=m;
                    cout<<"Constructor called Class D"<<endl;
                    cout<<"Value of y="<<y<<endl;
               }
};
int main()
{
          D obj(10,15);             //Derived class object obj passing argument to derived class
          return 0;
}
```

Output:



```
Constructor called Class B
Value of x=15
Constructor called Class D
Value of y=10
```

__Note:__ If the inheritance is multiple and we want to call all the base constructor then we can call as follows:

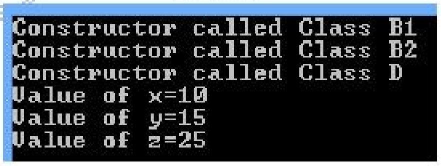
Derived_constructor(argument):base1_constructor(argument),base2_constructor(argument),…….

Example 4-16:

```
          #include<iostream>
          #include<conio.h>
          using namespace std;
          class B1                             //Base Class
          {
                    int x;
            public:
                    B1(int n)                  //Base Class B2 Constructor with one argument
                    {
                              x=n;
                              cout<<"Constructor called Class B1"<<endl;
              }
              void showx()
              {
                    cout<<"Value of x="<<x<<endl;
                    }
          };
          class B2                             //Base Class
          {
                    int y;
            public:
                    B2(int m)                  //Base Class B2 Constructor with one argument
                    {
                              y=m;
                              cout<<"Constructor called Class B2"<<endl;
              }
```

```
    void showy()
    {
        cout<<"Value of y="<<y<<endl;
     }
};
class D:public B1,public B2       //Derived class
{
        int z;
        public:
                D(int i,int j, int k):B1(i),B2(j)
                {
                        z=k;
                        cout<<"Constructor called Class D"<<endl;
                }
                void showz()
                {
                cout<<"Value of z="<<z<<endl;
                }
};
int main()
{
        D obj(10,15,25);
        obj.showx();
        obj.showy();
        obj.showz();
        return 0;
}
```

Output:



```
Constructor called Class B1
Constructor called Class B2
Constructor called Class D
Value of x=10
Value of y=15
Value of z=25
```

4.9 IS-A & HAS-A rule

- Division into parts or division into specialization will represent two most important rules of abstraction i.e. is-a rule and has-a rule
- While designing the program we should know the relationship among objects of components through analyzing the behaviors in different manner.
- Our idealization of inheritance is captured in a simple rule-of-thumb.
- Try forming the English sentences ``An A is-a B''. If it ``sounds right'' to your ear, then A can be made a subclass of B.
- A dog is-a mammal, and therefore a dog inherits from mammal
- A car is-a engine sounds wrong, and therefore inheritance is not natural. But a car has-a engine.

4.9.1 IS-A Rule

- Is a relationship says that the first component is specifies the instance of second component.
- The data and behavior related to the animal classes can be inherited to the many sub classes.
- IS-A rule basically imply some inheritance base class to derived class or super class sub class
- **Example:** Consider class fruit as base class or super class we can create sub classes like class apple, orange, banana from class fruit because all sub classes have some similar properties of fruit so those properties can be inherited.
- Write any example program of multiple inheritance.
- **Example:**

```
        class Vechicle
        {
                protected:
                        int price,
                        int number;
        };
```

```
class Car: Public Vehicle
{
        …….
};
class Bus: Public Vehicle
{
        ………
};
```

In above example common features (price, number) are in super class. Those with different will be sub class

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

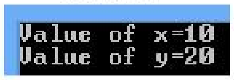
- HAS rule means division into parts.
- Consider the term "A car has an engine", "A bus has an engine". We cannot use engine the super class properties to all child classes because the engine has different for all transportation means. The engine is independent properties that cannot be directly called to the car, bus classes. We can include the class engine or class brake in class car not derive them.
- It is also called container class or containership because object of one class will be used inside other class which means member of one class is available in other class.
- **Example** for Container Class

Example 4-17:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        int x;
        public:
            B()
            {
                x=10;
            }
            void dispaly1()
            {
                cout<<"Value of x="<<x<<endl;
            }
};
class D
{
        int y;
        B objb;                 //Object of Class B
        public:
        D()
        {
            y=20;
        }
        void display2()
        {
            objb.dispaly1();        //Calling member of Class B from member of D
            cout<<"Value of y="<<y<<endl;
        }
};
int main()
{
```

```
            D objd;                  //object of class D
            objd.display2();
            return 0;
    }
```

Output:



```
Value of x=10
Value of y=20
```

In Above **Example program** there are 2 class B & D, Class B have a data member integer x and one default constructor and one member function display1(), Class D have 2 data member one is integer y and another is objb object of class B and one constructor and one member function i.e. display2() and inside diplay2 we call member function of class B i.e.display1(). Here class B's object is declared inside class D, so class D contains members of class B or class D HAS A object of class B (Class D is composite class).

4.10 Composition/Container Vs Inheritance

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

4.11 Pros and Cons of inheritance

❖ **Pros(Merits)**

➢ Generate more dominant object
➢ It supports the concept of hierarchical classification.
➢ The derived class inherits some or all the properties of base class.
➢ Inheritance provides the concept of reusability. This means additional feature can be added to an existing class without modifying the original class. This helps to save development time and reduce cost of maintenance.
➢ Code sharing can occur at several places.
➢ It will permit the construction of reusable software components. Already such libraries are commercially available.
➢ The new software system can be generated more quickly and conveniently by rapid prototyping.
➢ Programmers can divide their work themselves and later on combine their codes.

❖ **Cons(Demerits)**

➢ Compiler overhead- reduce execution speed
➢ Wastage of space in case some member function is unused
➢ Inappropriate usage will cause complexity in program
➢ The base and derived class get tightly coupled. This means one cannot be used independent of each other that is to say they are interconnected to each other.
➢ We know that inheritance uses the concept of reusability of program codes so the defects in the original code module might be transferred to the new module there by resulting in defective modules.

4.12 Subclass Subtype and Principle of 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 B and D then, B is super class and D is sub class.
- The behavior of B 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.
- 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.

Principle of 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 parts of the program. The principle of substitutability says that if we have two classes B and D. Then D is subclass of B that can shared possible substitute on the derived class D for instance of class B.

5 Polymorphism

5.1 Introduction

- Greek word poly means many/multiple and morphos means forms.
- Polymorphism simply means "the occurrence of something in different forms".
- Polymorphism is very important feature of object oriented programming. Polymorphism means one name, multiple form.
- Function overloading, Operator overloading and virtual functions are part of polymorphism.

❖ Polymorphic Variable

A polymorphic variable is a one which has many faces i.e. it can hold the value of different types. Polymorphic variable employs the principle of substitutability.

❖ Overriding

When a child class defines a method by the same name as that used for method in the parent class. The method in the child effectively hides or overrides the method in the parent class. In other words, a method in a class that has the same name as a method in a superclass is said to override the method in parent class.

❖ Overloading

Overloading means we can have several different functions with same name while overriding means that out of several functions the right one is selected at the run time depending on the dynamic type of objects.

5.2 Classification of Polymorphism

- Compile-time polymorphism
- Run-time polymorphism

5.2.1 Compile Time Polymorphism

- The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time, therefore, compiler can select appropriate function. So, this is known as compile time polymorphism.
- It is also called early binding or static binding. The example of compile time polymorphism are:
  - Function overloading
  - Operator overloading

5.2.2 Run Time Polymorphism

- If a member function is selected while program is running then this is called run time polymorphism. In run time polymorphism, the function link with a class very late (i.e. after compilation), therefore, this is called late binding or dynamic binding. This is called dynamic binding because function is selected dynamically at runtime. For example
  - Virtual function

5.3 Function Overloading In C++

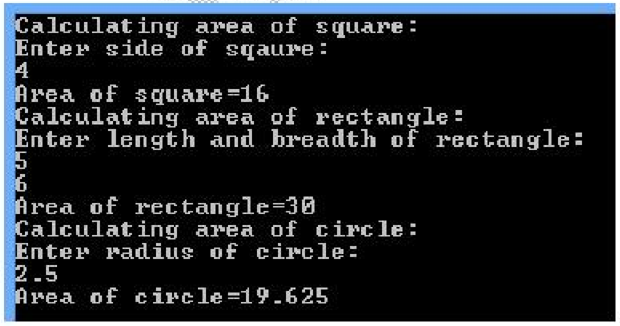
- Overloading refers to the use of the same thing for different purposes. C++ permits overloading of function. This means that we can use the same function name for a number of times for different purposes.
- We can have multiple definitions for the same function name in the same scope.
- The function would perform different operation depending on the argument list in the function call.
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
- The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.
- For example the function name area( ) can be used to find the area of square, rectangle and circle as follows:

Example 5-1:

```
#include<iostream>
#include<conio.h>
using namespace std;
int area(int);
int area(int, int);
```

```
float area(float);
int main()
{
        int l,b;
        float r;
        cout<<"Calculating area of square:"<<endl;
        cout<<"Enter side of sqaure:"<<endl;
        cin>>l;
        cout<<"Area of square="<<area(l)<<endl;
        cout<<"Calculating area of rectangle:"<<endl;
        cout<<"Enter length and breadth of rectangle:"<<endl;
        cin>>l>>b;
        cout<<"Area of rectangle="<<area(l,b)<<endl;
        cout<<"Calculating area of circle:"<<endl;
        cout<<"Enter radius of circle:"<<endl;
        cin>>r;
        cout<<"Area of circle="<<area(r)<<endl;
        getch();
        return 0;
}
int area(int x)
{
        return (x*x);
}
int area(int x,int y)
{
        return (x*y);
}
float area(float radius)
{
        return (3.14*radius*radius);
}
```

Output:



```
Calculating area of square:
Enter side of sqaure:
4
Area of square=16
Calculating area of rectangle:
Enter length and breadth of rectangle:
5
6
Area of rectangle=30
Calculating area of circle:
Enter radius of circle:
2.5
Area of circle=19.625
```

- A function call first matches the prototype having same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:
  - The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
  - If an exact match is not found, the compiler uses the integral promotions to the actual arguments to find a match, such as

    char to int
    float to double
  - When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

    long square(long n)
    double square(double x)

    A function call such as

    square(10)

    Will cause an error because **int** argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of **square( )** should be used.
  - If all of the steps fail, then the compiler will try the user-defined conversion with integral promotions and built-in conversion to find a unique match. User-defined conversions are often used in handling class objects.

5.4 Operator Overloading

- Operator overloading is one of the feature of C++ language. The concept by which we can give special meaning to an operator of C++ language is known as operator overloading.
- For example, + operator in C++ work only with basic type like int and float means c=a+b is calculated by compiler if a, b and c are basic types, suppose a, b and c are objects of user defined class, compiler give error. However, using operator overloading we can make this statement legal even if a, b and c are objects of class.
- Actually, when we write statement c=a+b (and suppose a, b and c are objects of class), the compiler call a member function of class. If a, b and c are basic type then compiler calculates a+b and assigns that to c.
- When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.
- We can overload all the C++ operators except the following:
  - Scope resolution operator (: :)
  - Membership operator (.)
  - Size of operator (sizeof)
  - Conditional operator (? :)
  - Pointer to member operator (.*)

Declaration of Operator Overloading:

- The declaration of operator overloading is done with the help of a special function, called operator function. The operator is keyword in C++. The general syntax of operator function is:

```
return_type operator op (arg list)
{
        function body
}
```

OR

```
return_type classname: : operator op (arg list)
{
        // function body
}
```

Where return_type is the type of value returned, operator is keyword. OP is the operator (+, -, *, etc) of C++ which is being overloaded as well as function name and arg list is argument passed to function.

- For Example:

```
void operator++( )
{
        body of function
}
```

In above example there is no argument. The above function is called operator function. When we write object with above written operator (++), the operator function is called i.e. when we write

obj ++;

The above function is called and executed (where obj is an object of class in which above function is written).

- Note that operator function must be either member function (non static) of friend function.

Operator Overloading Restrictions

- The precedence of the operator cannot be changed.
- The number of operands that an operator takes cannot be altered.

The process of overloading involves the following steps:

- Create a class that defines the data types that is to be used in the overloading operation.
- Declare the operator function operator op() in the public part of the class.
- Define the operator function to implement the required operation.

Overloaded operator function can be invoked using expression such as:-

op x or x op for unary operators (e.g. ++x or x++)

x op y for binary operator(e.g. x+y)

5.5 Overloading Unary and Binary Operators:

- Operator overloading is done through operator function. The operator function must be either a non-static or friend function of a class. Most important difference between a member function and friend function is that a friend function will have only one argument for unary operators and two for binary operators, while a member function will have no arguments for unary operators. The reason is object used to invoke the member function is passed implicitly and thus it is available for the member function. In other word, member function can always access the particular object for which they have been called.

  Types of Operator Overloading:

  There are two types of operator overloading:
  1. Unary operator overloading
  2. Binary operator overloading

5.5.1 Unary Operator Overloading

- As we know, an unary operator acts on only one operand. Examples of unary operators are the increment and decrement operators ++, -- and -
- When overloading of unary operator using member operator function no arguments is used
- Consider incrementing of one object of num class,operator function will look like this

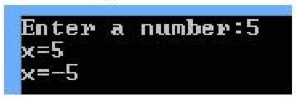
  void operator++();
- Member function can be called by object, n is object that calls the operator member function and called using no argument i.e only object n's value changes.

  ++n;   where n is only one operand

Example 5-2:

```
#include<iostream>
#include<conio.h>
using namespace std;
class num{
        int x;
        public:
                void getdata()
                {
                        cout<<"Enter a number:";
                        cin>>x;
                }
                void display();
                void operator-();
};
void num::display()
{
        cout<<"x="<<x<<endl;
}
void num::operator-()
{
        x=-x;
}
int main()
{
        num n;
        n.getdata();
        n.display();
        n.operator-();          //-n;
        n.display();
        getch();
        return 0;
}
```

Output:

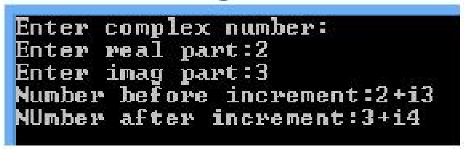


```
Enter a number:5
x=5
x=-5
```

Example 5-3: Write a program which reads a complex number increment real and imag part then print.

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
        int real,imag;
        public:
                void getdata()
                {
                        cout<<"Enter real part:";
                        cin>>real;
                        cout<<"Enter imag part:";
                        cin>>imag;
                }
                void display()
                {
                        cout<<real<<"+i"<<imag<<endl;
                }
                void operator++()
                {
                        ++real;
                        ++imag;
                }
};
int main()
{
        complex c1;
        cout<<"Enter complex number:"<<endl;
        c1.getdata();
        cout<<"Number before increment:";
        c1.display();
        ++c1;
        cout<<"NUmber after increment:";
        c1.display();
        getch();
        return 0;
}
```

Output:



```
Enter complex number:
Enter real part:2
Enter imag part:3
Number before increment:2+i3
NUmber after increment:3+i4
```

- Within **main( )** when we write **++c1** or **c1++** the member function **operator++ ( )** is called and executed. Actually compiler check the **c1** if this is basic type like **int**, the compiler increment that but if **c1** is object of a class compiler execute a member function **operator ++**.
- Note that within the body of operator function. We can write anything. But generally the statements written which match with the meaning of operator which we want to overload, in above case the operator is ++ and operator function is operator ++( ).
- In above **example 5.3** we cannot write the following statement:

  c2 = ++c1;

  Where c2 and c1 both are object of class complex because the return type of **operator ++( )** function is void.
- This type of problem will not come if **c1** and **c2** are basic type. But if **c1** and **c2** are object then **++c1** must return a value which is assigned to object **c2**. We can solve this problem by changing the return type of **operator ++( )** function.

Example 5-4:

```
#include<iostream>
#include<conio.h>
using namespace std;
```

```
class complex
{
        int real,imag;
        public:
                void getdata()
                {
                        cout<<"Enter real part:";
                        cin>>real;
                        cout<<"Enter imag part:";
                        cin>>imag;
                }
                void display()
                {
                        cout<<real<<"+i"<<imag<<endl;
                }
                complex operator++()
                {
                        complex temp;
                        temp.real=++real;
                        temp.imag=++imag;
                        return temp;
                }
};
int main()
{
        complex c1,c2;
        cout<<"Enter complex number:"<<endl;
        c1.getdata();
        cout<<"Number before increment:";
        c1.display();
        c2=++c1;
        cout<<"NUmber after increment:";
        c2.display();
        getch();
        return 0;
}
```

Limitation of above incremented overloaded operator

- In above **example 5-4**, there is no difference between following two statements:
  1. c2 = ++c1;
  2. c2 = c1++;

  If we write these in main( ). Both first increment the data member of c1 and then assign to data member of c2. i.e. the prefix and postfix cannot be differentiated by the above example 5-4.

- But in current version of turbo C++ and Borland C++, there is a facility by which we can differentiate prefix and postfix by changing the header of operator function like
  1. operator++( )   //does prefix
  2. operator++(int)   //does postfix

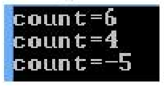
  the 2. Declaration uses a dummy int argument which is set to zero by the postfix ++operator. This extra argument allows the compiler to distinguish the two forms.

- Similarly this concept is applied for decrement operator (- -)
  1. Operator--( )   //does prefix
  2. Operator--(int)   //does postfix

Example 5-5: Like ++, unary operator – and unary – can be overloaded

```
#include<iostream>
#include<conio.h>
using namespace std;
class counter
{
        int count;
        public:
                counter()
                {
                        count=5;
                }
                void operator++(int)     //overloaded ++ operator
                {
                        count++;
                }
                void operator--()        //overloaded -- operator
                {
                        --count;
                }
                void operator-() //overloaded ++ operator
                {
                        count=-count;
                }
                void showdata()
                {
                        cout<<"count="<<count<<endl;
                }
};
int main()
{
        counter c1,c2,c3;
        c1++;
        --c2;
        -c3;
        c1.showdata();
        c2.showdata();
        c3.showdata();
        getch();
        return 0;
}
```

Output:



```
count=6
count=4
count=-5
```

5.5.2 Overloading Binary Operator

- Binary operator means operator which have two operands i.e +,-,*,/,%
- This takes two operands while overloading. For example c=a+b where a and b are two operands.
- When overloading of binary operator using member operator function one argument is necessary.
- Consider addition of two objects of complex class c3=c1+c2;operator function will look like this

  complex operator+(complex c2)

  Member function can be called by object c1 is object that calls the operator member function and called using one argument i.e c2 and the result is returned to c3.

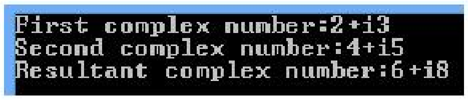
<u>Overloading of Arithmetic Operator:</u>

- Overloading of binary + operator is as follows:

Example 5-6: This add two complex number

```
#include<iostream>
#include<conio.h>
using namespace std;
class complex
{
        int real,imag;
        public:
                complex(){      }
                complex(int x,int y)
                {
                        real=x;
                        imag=y;
                }
                void display()
                {
                        cout<<real<<"+i"<<imag<<endl;
                }
                complex operator+(complex c)
                {
                        complex temp;
                        temp.real=real+c.real;
                        temp.imag=imag+c.imag;
                        return temp;
                }
};
int main()
{
        complex c1(2,3),c2(4,5),c3;
        cout<<"First complex number:";
        c1.display();
        cout<<"Second complex number:";
        c2.display();
        c3=c1+c2;
        cout<<"Resultant complex number:";
        c3.display();
        getch();
        return 0;
}
```

Output:



```
First complex number:2+i3
Second complex number:4+i5
Resultant complex number:6+i8
```

- In above example 5-6, when statement **c3=c1+c2** is executed the operator function is called. The operator function is called by object c1 and object c2 is passed as argument to operator function i.e. the c2 object is copied into object c which is written in the header of operator function.
- Inside the operator function, **temp.real = real + c.real;** calculate the sum of real member of object c1 and object c2 and assign this to real of temp object.
- Similarly the statement **temp.imag = imag+c.imag;** is calculated.
- The temp object is returned to object c3.

Note: The *, / and other binary operator are overloaded like + operator. For example the operator function for * is as follows:

```
complex operator*(complex c)
                {
                        complex temp;
                        temp.real=real*c.real-imag*c.imag;
                        temp.imag=real*c.imag+imag*c.real;
                        return temp;
                }
```

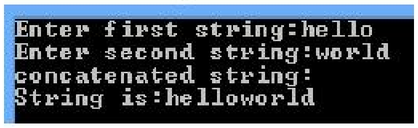
This perform multiplication of two complex numbers. Within **main( )** we call this like

```
c3 = c1*c2;
```

Example 5-7: Write a program which concatenates two strings by operator overloading

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class str
{
        char s[50];
        public:
                void getdata()
                {
                        cin>>s;
                }
                void showdata()
                {
                        cout<<"String is:"<<s<<endl;
                }
                str operator+(str x)
                {
                        str temp;
                        strcpy(temp.s,s);
                        strcat(temp.s,x.s);
                        return temp;
                }
};
int main()
{
        str s1,s2,s3;
        cout<<"Enter first string:";
        s1.getdata();
        cout<<"Enter second string:";
        s2.getdata();
        s3=s1+s2;
        cout<<"concatenated string:"<<endl;
        s3.showdata();
        getch();
        return 0;
}
```

Output:



```
Enter first string:hello
Enter second string:world
concatenated string:
String is:helloworld
```

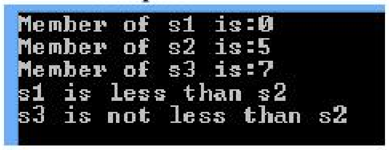
Overloading of Comparison Operator

- The comparison operator can also be overloaded like arithmetic but comparison operator return true or false, therefore we have to use enumerated data types.

Example 5-8: Overloading of < (less than) operator

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
        int n;
        public:
                sample()
                {
                        n=0;
                }
                sample(int x)
                {
                        n=x;
                }
                void display()
                {
                        cout<<n<<endl;
                }
                bool operator<(sample s)
                {
                        if(n<s.n)
                                return (true);
                        else
                                return (false);
                }
};
int main()
{
        sample s1;
        sample s2(5);
        sample s3(7);
        cout<<"Member of s1 is:";
        s1.display();
        cout<<"Member of s2 is:";
        s2.display();
        cout<<"Member of s3 is:";
        s3.display();
        if(s1<s2)
                cout<<"s1 is less than s2"<<endl;
        else
                cout<<"s1 is not less than s2"<<endl;
        if(s3<s2)
                cout<<"s3 is less than s2"<<endl;
        else
                cout<<"s3 is not less than s2"<<endl;
getch();
return 0;
}
```

Output:



```
Member of s1 is:0
Member of s2 is:5
Member of s3 is:7
s1 is less than s2
s3 is not less than s2
```

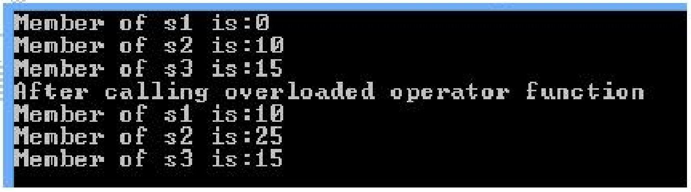
Overloading of Assignment Operator

- We can overload assignment operator =, +=, -=, *=, /= etc.

Example 5-9: Overloading of = and += operator

```
#include<iostream>
#include<conio.h>
using namespace std;
class sample
{
        int n;
        public:
                sample()
                {
                        n=0;
                }
                sample(int x)
                {
                        n=x;
                }
                void display()
                {
                        cout<<n<<endl;
                }
                void operator=(sample s)                    //overloaded = operator
                {
                        n=s.n;
                }
                void operator+=(sample p)                   //overloaded += operator
                {
                        n=n+p.n;
                }
};
int main()
{
        sample s1;
        sample s2(10);
        sample s3(15);
        cout<<"Member of s1 is:";
        s1.display();
        cout<<"Member of s2 is:";
        s2.display();
        cout<<"Member of s3 is:";
        s3.display();
        s1=s2;                                  //calling overloaded = operator
        s2+=s3;                                 //calling overloaded += operator
        cout<<"After calling overloaded operator function"<<endl;
        cout<<"Member of s1 is:";
        s1.display();
        cout<<"Member of s2 is:";
        s2.display();
        cout<<"Member of s3 is:";
        s3.display();
        getch();
        return 0;
}
```

Output:



```
Member of s1 is:0
Member of s2 is:10
Member of s3 is:15
After calling overloaded operator function
Member of s1 is:10
Member of s2 is:25
Member of s3 is:15
```

5.6 Operator Overloading using a Friend Function

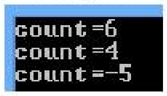
- When the overloaded operator function is a friend function, it takes two arguments for binary operator and takes one argument for the unary operator

Example 5-10: Overloading unary operator using friend function

```
#include<iostream>
#include<conio.h>
using namespace std;
class counter
{
        int count;
        public:
                counter()
                {
                        count=5;
                }
```

```
                    void showdata()
                    {
                            cout<<"count="<<count<<endl;
                    }
                    friend void operator++(counter &,int);          //postfix ++ operator overloading
                    friend void operator--(counter &);              //prefix – operator overloading
                    friend void operator-(counter &);
        };
                    void operator++(counter &p,int)
                    {
                            p.count=p.count+1;
                    }
                    void operator--(counter &q)
                    {
                            q.count=q.count-1;
                    }
                    void operator-(counter &r)
                    {
                            r.count=-r.count;
                    }

        int main()
        {
                counter c1,c2,c3;
                c1++;
                --c2;
                -c3;
                c1.showdata();
                c2.showdata();
                c3.showdata();
                getch();
                return 0;
        }
```

Output:



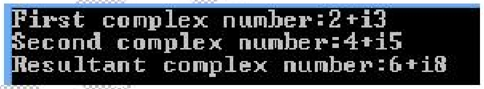
```
count=6
count=4
count=-5
```

Example 5-11: Overloading binary + operator using friend function

```
        #include<iostream>
        #include<conio.h>
        using namespace std;
        class complex
        {
                int real,imag;
                public:
                        complex(){      }
                        complex(int x,int y)
                        {
                                real=x;
                                imag=y;
                        }
                        void display()
                        {
                                cout<<real<<"+i"<<imag<<endl;
                        }
                friend complex operator+(complex,complex);
        };
```

```
            complex operator+(complex a,complex c)
                    {
                            complex temp;
                            temp.real=a.real+c.real;
                            temp.imag=a.imag+c.imag;
                            return temp;
                    }
    int main()
    {
            complex c1(2,3),c2(4,5),c3;
            cout<<"First complex number:";
            c1.display();
            cout<<"Second complex number:";
            c2.display();
            c3=c1+c2;
            cout<<"Resultant complex number:";
            c3.display();
            getch();
            return 0;
    }
```

Output:



```
First complex number:2+i3
Second complex number:4+i5
Resultant complex number:6+i8
```

5.7 Rules for Operator Overloading in C++

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded.

sizeof	Size of Operator
.	Membership Operator
.*	Pointer-to-member Operator
::	Scope resolution Operator
?:	Conditional Operator

5. We cannot use "friend" functions to overload certain operators. However, member function can be used to overload them.

=	Assignment Operator
( )	Function call Operator
[ ]	Subscripting Operator
->	Class member access Operator

6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

5.8 Type Conversions

- In C++ by assignment statement we can do automatic type conversion for basic data type. Compiler automatically convert from one type to another type by applying type conversion rule provided by compiler.
- For example

```
int x;
float y=3.14;
x=y;
```

In above example, we are assigning the value of variable y to variable x. To achieve this, the compiler first converts y into integer and then assign it to x.

- But the compiler does not support automatic (standard) conversion for the user-defined data types like classes. For this, we need to design our own data conversion routines. There are 3 types of data conversion available for user defined classes:
  1. Conversion from basic type to class type
  2. Conversion from class type to basic type
  3. Conversion from one class type to another class type

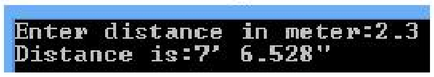
5.8.1 Conversion from Basic to Class type

- Basic data type i.e. int, float etc. To object of a class.
- Have to use one default and one parameterized constructor
- In this type left hand operand of =(equality sign) is class type (object) and right hand operand is basic type (int,float)
- Argument of constructor can be changed according to basic type, if int to object argument is int and if float to object argument is float and if int and float is converting in one program you can have float as an argument.
- The constructor used for type conversion takes a single argument whose type is to be converted.

Example 5-12:

```
#include<iostream>
#include<conio.h>
using namespace std;
class dist{
        int feet;
        float inches;
        public:
                dist(float mtr)
                {
                        float f;
                        f=mtr*3.28;             //1 meter=3.28feet(Approx)
                        feet=f;
                        inches=12*(f-feet);     // 1 feet=12inches
                }
                void showdata()
                {
                        cout<<feet<<"\' "<<inches<<"\""<<endl;
                }
};
int main()
{
        float m;
        cout<<"Enter distance in meter:";
        cin>>m;
        dist d=m;                       //this call constructor which convert float to class type data
        cout<<"Distance is:";
        d.showdata();
        getch();
        return 0;
}
```

Output:



```
Enter distance in meter:2.3
Distance is:7' 6.528"
```

5.8.2 Conversion from Class to Basic type

- Object of class i.e user defined data type to basic data type i.e. int,float etc..
- Use technique called type cast operator (casting operator function)
- In this type Left Hand Operand of = (equality sign) is basic type (int,float) and Right Hand Operand is a class type(object).
- By the constructor we cannot convert class to basic type. For converting class type to basic type we can define a overloaded casting operator in C++. The general format of an overloaded casting operator function is:

```
operator typename( )
{
        Function body
}
```

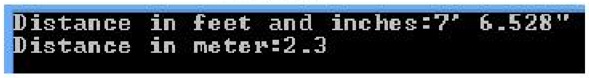
In above declaration operator is keyword and typename is basic type i.e. float, int etc.

- Conversion function shouldn't have any arguments or return value.
- It should be a member function

Example 5-13:

```
#include<iostream>
#include<conio.h>
using namespace std;
class dist
{
        int feet;
        float inches;
        public:
                dist(int f, float i)
                {
                        feet=f;
                        inches=i;
                }
                void showdata()
                {
                        cout<<feet<<"\' "<<inches<<"\""<<endl;
                }
                operator float()
                {
                        float ft;
                        ft=inches/12;
                        ft=ft+feet;
                        return (ft/3.28);
                }
};
int main()
{
        dist d(7,6.528);
        float m=d;
        cout<<"Distance in feet and inches:";
        d.showdata();
        cout<<"Distance in meter:"<<m;
        getch();
        return 0;
}
```

Output:



```
Distance in feet and inches:7' 6.528"
Distance in meter:2.3
```

- In example 5-13, the following function converts class type to basic type:

```
        operator float()
        {
                float ft;
                ft=inches/12;
                ft=ft+feet;
                return (ft/3.28);
        }
```

- When we write **float m = d;** in **main ( );** above function is called, which take feet and inches of dist d and return a float type data.

5.8.3 Conversion from One class type another class type

- We can convert one class(object) to another class type as follows:

  object of A= object of B

  A and B both are different type of classes. The conversion takes place from class B to Class A, therefore, B is source class and A is destination class.
- Class type one to another class can be converted by following way:
  - By constructor
  - By conversion function, i.e. the overloaded casting operator function.

<u>BY Constructor:</u>

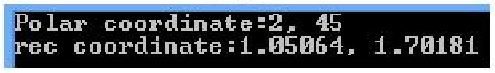
When the conversion routine is in destination class, it is commonly implemented as constructor

Example 5-14: Converts polar coordinate to rec coordinate

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
class polar
{
        float rd;
        float ang;
        public:
                polar()
                {
                        rd=0.0;ang=0.0;
                }
                polar(float r, float a)
                {
                        rd=r; ang=a;
                }
                float getrd()
                {
                        return (rd);
                }
                float getang()
                {
                        return (ang);
                }
                void showpolar()
                {
                        cout<<rd<<", "<<ang<<endl;
                }
};
```

```
class rec
{
        float x,y;
        public:
                rec()
                {
                        x=0.0;y=0.0;
                }
                rec(float xco, float yco)
                {
                        x=xco; y=yco;
                }
                void showrec()
                {
                        cout<<x<<", "<<y<<endl;
                }
                rec(polar p)
                {
                        float r=p.getrd();
                        float a=p.getang();
                        x=r*cos(a);
                        y=r*sin(a);
                }
};
int main()
{
        rec r1;
        polar p1(2.0,45.0);
        r1=p1;                          //convert polar to rec by calling one argument constructor
        cout<<"Polar coordinate:";
        p1.showpolar();
        cout<<"rec coordinate:";
        r1.showrec();
        getch();
        return 0;
}
```

Output:



```
Polar coordinate:2, 45
rec coordinate:1.05064, 1.70181
```

- In example 5-14, the following constructor converts polar class type to rec class type

```
rec(polar p)
        {
                float r=p.getrd();
                float a=p.getang();
                x=r*cos(a);
                y=r*sin(a);
        }
```

In this constructor object of polar class p is passed as an argument.
- Within **polar** class **rd** and **ang** are private data we cannot access these outside the class, therefore **getrd( )** and **getang( )** function are written within polar class by using these function, We can get **rd** and **ang**. By following ways:

```
p.getrd( );
p.getang( );
```

- The statement **r1= p1;** called one argument constructor of class rec.

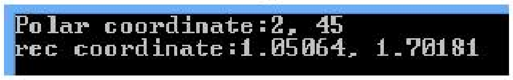
By conversion function

When conversion routine is in source class, it is implemented as a conversion function.

Example 5-15: Converts polar coordinate to rec coordinate

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
class rec
{
        float x,y;
        public:
                rec()
                {
                        x=0.0;y=0.0;
                }
                rec(float xco, float yco)
                {
                        x=xco; y=yco;
                }
                void showrec()
                {
                        cout<<x<<", "<<y<<endl;
                }
};
class polar
{
        float rd;
        float ang;
        public:
                polar()
                {
                        rd=0.0;ang=0.0;
                }
                polar(float r, float a)
                {
                        rd=r; ang=a;
                }
                void showpolar()
                {
                        cout<<rd<<", "<<ang<<endl;
                }
                operator rec()
                {
                        float x1=rd*cos(ang);
                        float y1=rd*sin(ang);
                        return rec(x1,y1);
                }
};
int main()
{
        rec r1;
        polar p1(2.0,45.0);
        r1=p1;                          //convert polar to rec by calling conversion function
        cout<<"Polar coordinate:";
        p1.showpolar();
        cout<<"rec coordinate:";
        r1.showrec();
        getch();
        return 0;
}
```

Output:



```
Polar coordinate:2, 45
rec coordinate:1.05064, 1.70181
```

- In example 5-15, the conversion function is written in class polar, which is source class, when we write the following statement **r1=p1;** This is called the conversion function

```
                operator rec()
                  {
                        float x1=rd*cos(ang);
                        float y1=rd*sin(ang);
                        return rec(x1,y1);
                  }
```

This function uses the **rd** and **ang** of **p1** and return an object of rec type class to **r1**.

- In above function return rec(x1, y1); create a temporary object and assign the data member of that by value x1 and y1.

5.9 Virtual Function

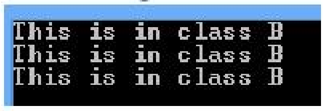
- Virtual means existing in effect but not in reality. A function is declared virtual by writing keyword 'virtual' in front of function header. A virtual function uses a single pointer to base class pointer to refer to all the derived objects. Virtual functions are useful when we have number of objects of different classes but want to put them all on a single list and perform operation on them using same function call.
- When we use the same function name in both the base and derived classes, the function in base class declared as virtual function. The virtual function is invoked at run time based on the type of pointer.
    - When function is made virtual, C++ determines that which function is used to at run time based on type of object pointed on base pointer.
    - Virtual function concept is used in inheritance when function having same name and with equal arguments in base class as well as in derived class.
    - The function is declared using **virtual** keyword.
    - The function which is access by pointer object having same name in both base and derived class is called virtual function.
    - The object of different class can respond to same message in different forms we can access the function by using object which is declared as single pointer variable. The pointer variable is called polymorphic variable

Example 5-16: When a base and derived classes have same functions with same name and these are accessed using pointers but without using virtual functions.

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        public:
                void show()
                {
                        cout<<"This is in class B"<<endl;
                }
};
class D1:public B
{
        public:
                void show()
```

```
                    {
                            cout<<"This is in class D1"<<endl;
                    }
    };
    class D2:public B
    {
            public:
                    void show()
                    {
                            cout<<"This is in class D2"<<endl;
                    }
    };
    int main()
    {
            B *p;
            B obj;
            D1 obj1;
            D2 obj2;
            p=&obj;
            p->show();
            p=&obj1;
            p->show();
            p=&obj2;
            p->show();
            getch();
            return 0;
    }
```

Output:



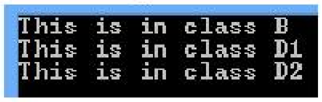
```
This is in class B
This is in class B
This is in class B
```

- In **example 5-16**, **p** is pointer of base class type. Base pointer is compatible to its drive classes. Therefore the statement
  p = &obj1;
  p = &obj2 are correct.
- The output of above program is same. That means the **p->show( );** statement execute every time the show function which is in class **B**.
- When we assign address of **obj** to **p** by statement **p = &obj;** then function of class **B** is called by the statement **p->show( )** because **obj** is object of class **B**.
- But when we assign address of **obj1** to **p** then **p->show( );** statement again called the function **show( )** of class **B** while **obj1** is object of class **D1** because in class **D1** there are two functions both have name **show( )**, one is inherited from class **B** and seond is its own. The **p** pointer is actually pointer of **B** type by the declaration **B *p;**
- The C++ compiler ignores the content of **p** (which is address of **obj1** after statement **p = &obj1**) and execute function which is inherited from class **B** because the type of **p** matches with class **B**.
- Similarly after statement **p = &obj2**; and **p ->show( )** compiler again executes the function which is inherited from class **B**.
- Whenever this type of situation occurs i.e. in derived class there are two functions both have same name, one is inherited from base, second is its own, and pointer is base type. Then if we want to execute the function through pointer compiler chose the function of base (i.e. which is inherited from base). But when this situation does not occur compiler executes the function of derive class because base type pointer is compatible with the derived class.
- If we want to execute the function of derived class in **example 5-16** that is possible in C++ by virtual function concept.
- If in **example 5-16**, we make the function **show( )** of base **B** as virtual function then the output of program is different.
- When a function is made virtual, C++ compiler determines which function to use at run time based on the content of base pointer rather than the type of pointer. (i.e. if base pointer has address of derived class objects then that executes the function of derived).
- Note that if base pointer contains the address of a derived class objects then that is called objects point to by base pointer.

Example 5-17: With use of virtual function

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        public:
                virtual void show()
                {
                        cout<<"This is in class B"<<endl;
                }
};
class D1:public B
{
        public:
                void show()
                {
                        cout<<"This is in class D1"<<endl;
                }
};
class D2:public B
{
        public:
                void show()
                {
                        cout<<"This is in class D2"<<endl;
                }
};
int main()
{
        B *p;
        B obj;
        D1 obj1;
        D2 obj2;
        p=&obj;
        p->show();
        p=&obj1;
        p->show();
        p=&obj2;
        p->show();
        getch();
        return 0;
}
```

Output:



```
This is in class B
This is in class D1
This is in class D2
```

- In above **example 5-17**, the member function of class **B** {i.e. **show( )** } is virtual function, therefore the output of the program is different from **example 5-16**.

5.9.1 Rules for Virtual Functions

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

1. A virtual function must be a member of certain class.
2. Such function cannot be a static member. But it can be a friend of another class.
3. A virtual function is accessed by using object pointer.
4. A virtual function must be defined, even though it may not be used.

5. The prototypes of the virtual in the base class and the corresponding member function in the derived class must be same. If not same, then C++ treats them as overloaded functions (having same name, different arguments) thereby the virtual function mechanism is ignored.
6. The base pointer can point to any type of the derived object, but vice-versa is not true i.e. the pointer to derived class object cannot be used to point the base class object.
7. Incrementing or decrementing the base class pointer (pointing derived object) will not make it point to the next object of derived class object.
8. If a virtual function is defined in the base class, it is not compulsory to redefine it in the derived class. In such case, the calls will invoke base class function.
9. There cannot be virtual constructors in a class but there can be virtual destructors.

5.10 Pure Virtual Function

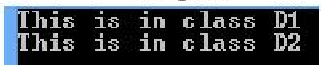
- A pure virtual function is a virtual function with no function body
- We know that in general we declare a function virtual inside the base class and redefine it in derived class. But the function defined inside the base class is seldom (rarely) used for performing any task.
- These functions only serve as a placeholder .Such functions are called **do-nothing functions** or **dummy function** or **deferred method**.
- It is always virtual function, uses keyword virtual
- Syntax:

  vitual return_type function_name()=0;
- Can't be used for any operation
- Can't create object of class when there is pure virtual function.
- The class containing pure virtual function is called abstract class or pure abstract class.

Example 5-18:

```
#include<iostream>
#include<conio.h>
using namespace std;
class B
{
        public:
                virtual void show()=0;                          //pure virtual function
};
class D1:public B
{
        public:
                void show()
                {
                        cout<<"This is in class D1"<<endl;
                }
};
class D2:public B
{
        public:
                void show()
                {
                        cout<<"This is in class D2"<<endl;
                }
};
int main()
{
        B *p[2];
        D1 obj1;
        D2 obj2;
        p[0]=&obj1;
        p[1]=&obj2;
        p[0]->show();
        p[1]->show();
        getch();
        return 0;
}
```

Output:



```
This is in class D1
This is in class D2
```

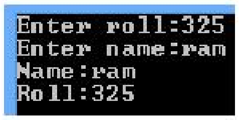
5.11 Object Pointer

- Object also have address in memory
- Object pointer stores address of specified object
- It can point to specified object
- Declaration syntax: **class_name *pointer_object**
- Binding syntax: **pointer_object=&object**
- Instead of dot operator for calling member function pointer uses (->) operator.

Example 5-19:

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
        int rn;
        char name[50];
        public:
                void getdata()
                {
                        cout<<"Enter roll:";
                        cin>>rn;
                        cout<<"Enter name:";
                        cin>>name;
                }
                void showdata()
                {
                        cout<<"Name:"<<name<<endl;
                        cout<<"Roll:"<<rn<<endl;
                }
};
int main()
{
        student s;
        student *p;
        p=&s;                   //now p points to s
        p->getdata();
        p->showdata();
        getch();
        return 0;
}
```

Output:



```
Enter roll:325
Enter name:ram
Name:ram
Roll:325
```

5.12 This Pointer

- In C++, **this** is a keyword. **this** represents an object that invokes a member function.
- Whenever any object called its member function **this** pointer is automatically set and contains the address of that object. The pointer acts as an implicit argument to all the member function.
- For example:

```
class sample
{
        float a;
        int b;
        …….
};
```

The private member **a** and **b** can be used directly inside any member function as follows:

```
        a = 5.5;
        b = 10;
```

We can also use the following statement inside any member function of sample.

```
        this -> a = 5.5;
        this -> b = 10;
```

- One important application of this is to return the object, it points to as follwos:

```
        return *this;
```

Characteristics:

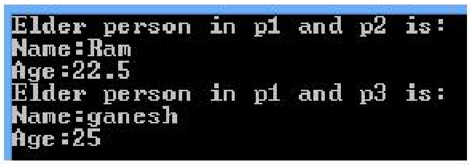
1. **this** pointer stores the address of the class instance, to enable pointer access of the member functions of the class.
2. **this** pointer is not counted for calculating the size of the object
3. **this** pointer is not accessible for static member functions.
4. **this** pointer are not modifiable.

Example 5-20:

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class person
{
        float age;
        char name[50];
        public:
                person(){ }
                person(char *s,float x)
                {
                        strcpy(name,s);
                        age=x;
                }
                person & greater(person &);
                void display();
};
        person & person::greater(person &p)
                {
                        if(p.age>age)
                                return p;
                        else
                                return *this;
                }
```

```
                    void person::display()
                    {
                            cout<<"Name:"<<name<<endl;
                            cout<<"Age:"<<age<<endl;
                    }
        int main()
        {
                person p1("Ram",22.5);
                person p2("Hari",21.25);
                person p3("ganesh",25.0);
                person p;
                p=p1.greater(p2);
                cout<<"Elder person in p1 and p2 is:"<<endl;
                p.display();
                p=p1.greater(p3);
                cout<<"Elder person in p1 and p3 is:"<<endl;
                p.display();
                getch();
                return 0;
        }
```

Output:



```
Elder person in p1 and p2 is:
Name:Ram
Age:22.5
Elder person in p1 and p3 is:
Name:ganesh
Age:25
```

6 Template and generic Programming

6.1 Generic and Templates

- Generic is a new concept which enables us to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameter in algorithm so that they work for a variety of suitable data types and data structure. In C++ generic programming is achieved by the means of template.
- Template is one of the important features of C++ which enables us to define generic (generalized) classes and function.
- A template in C++ can be used to create a family of classes or functions. e.g.: A class template for an array of various data types such as int array and float array. Similarly, we can define a template for a function, say mul(), that would help us create various versions of mul()for multiplying int,float and double type values.
- A template can be considered as a macro which helps to create a family of classes or functions. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.
- There are two types of templates:
  - Function Templates
  - Class Templates

6.2 Class Templates

- Class template is one of the kinds of template that helps us to create generic classes. It is a simple process to create a generic class using a template with an anonymous type. The general syntax of a class template is:

```
template <class T>
class class_name
{
        // - - -
        // class member specification
        // with anonymous type T
        // wherever appropriate
        // - - - - -
};
```

This syntax shows that the class template definition is very similar to an ordinary class definition except the use of prefix template **<class T>** & use of type **T**. These tell the compiler that we are going to declare a template and use "**T**" as a type name in the declaration. This **T** can be replaced by any built-in data type (**int, float, char**, etc) or a user-defined data type.

- Body of member function is written Outside the class as follows:

```
template<class T>
return_type class_name <T>::function_name(arg)
{
        //body of function
}
```

- The syntax for defining an object of template class is:

```
class_name<data type> object_name(arg);
```

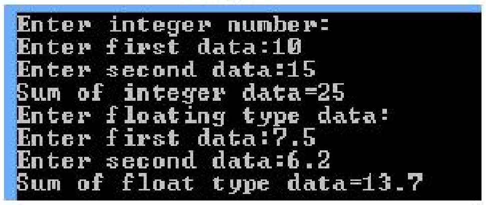
The **arg** are used when constructor are in template class otherwise skipped. Note that template is keyword in C++

Example 6-1: Write a program which generates a template class, by which we can perform integer type data addition and float type data addition also.

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T>
class add
{
```

```
          T a,b;
          public:
                    void getdata()
                    {
                              cout<<"Enter first data:";
                              cin>>a;
                              cout<<"Enter second data:";
                              cin>>b;
                    }
                    T sum()
                    {
                              T c;
                              c=a+b;
                              return (c);
                    }
};
int main()
{
          add<int>obj1;
          add<float>obj2;
          cout<<"Enter integer number:"<<endl;
          obj1.getdata();
          cout<<"Sum of integer data="<<obj1.sum();
          cout<<"\nEnter floating type data:"<<endl;
          obj2.getdata();
          cout<<"Sum of float type data="<<obj2.sum();
          getch();
          return 0;
}
```

Output:



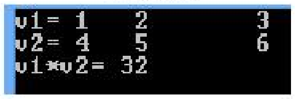
```
Enter integer number:
Enter first data:10
Enter second data:15
Sum of integer data=25
Enter floating type data:
Enter first data:7.5
Enter second data:6.2
Sum of float type data=13.7
```

Example 6-2:

```
#include<iostream>
#include<conio.h>
#define size 3
using namespace std;
template<class T>
class vector
{
          T v[size];
          public:
                    vector(){          }
                    vector(T a[])
                    {
                              for(int i=0;i<size;i++)
                                        v[i]=a[i];
                    }
          T operator*(vector &y)
          {
                    T sum =0;
                    for(int i=0;i<size;i++)
                              sum+=this->v[i]*y.v[i];
                    return sum;
          }
```

```
        void display()
        {
                for(int i=0;i<size;i++)
                        cout<<v[i]<<"\t";
        }
};
int 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<<"v1= ";
        v1.display();
        cout<<"\nv2= ";
        v2.display();
        cout<<"\nv1*v2= "<<r<<endl;
        getch();
        return 0;
}
```

Output:



```
v1= 1   2       3
v2= 4   5       6
v1*v2= 32
```

6.2.1 Class template with multiple parameter

- We can use more than one generic data type in a class template. They are declared as a comma separated list within the template specification.
- Syntax:

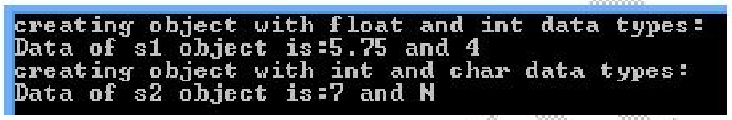
```
template<class T1, class T2,....>
class class_name
{
        //body of the class
};
```

Example 6-3: Two generic data types in a class definition

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T1,class T2>
class sample
{
        T1 a;
        T2 b;
        public:
                sample(T1 x, T2 y)
                {
                        a=x;
                        b=y;
                }
                void show()
                {
                        cout<<a<<" and "<<b<<endl;
                }
};
```

```
int main()
{
        cout<<"creating object with float and int data types:"<<endl;
        sample<float,int>s1(5.75,4);
        cout<<"Data of s1 object is:";
        s1.show();
        cout<<"creating object with int and char data types:"<<endl;
        sample<int,char>s2(7,'N');
        cout<<"Data of s2 object is:";
        s2.show();
        getch();
        return 0;
}
```

Output:

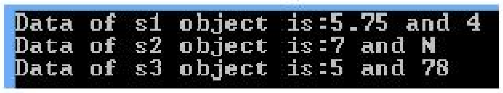


```
creating object with float and int data types:
Data of s1 object is:5.75 and 4
creating object with int and char data types:
Data of s2 object is:7 and N
```

Example 6-4: Using Default Data types in a class definition

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T1=int,class T2=int>                    //default data types specified as int
class sample
{
        T1 a;
        T2 b;
        public:
                sample(T1 x, T2 y)
                {
                        a=x;
                        b=y;
                }
                void show()
                {
                        cout<<a<<" and "<<b<<endl;
                }
};
int main()
{
        sample<float,int>s1(5.75,4);
        cout<<"Data of s1 object is:";
        s1.show();
        sample<int,char>s2(7,'N');
        cout<<"Data of s2 object is:";
        s2.show();
        sample<>s3(5.75,'N');          //declaration of class object without type specification
        cout<<"Data of s3 object is:";
        s3.show();
        getch();
        return 0;
}
```

Output:



```
Data of s1 object is:5.75 and 4
Data of s2 object is:7 and N
Data of s3 object is:5 and 78
```

➢ The above program declares **s3** object without any type specification, thus the default data type for **T1** and **T2** is considered as int. The parameter values passed by **s3** are type casted to **int** and displayed as output.

6.3 Function Templates

- Similar to class template, the function template will be used to create a family of function with different argument type.
- Function template helps in working with any type of data for the function. Any function argument is accepted by the function
- The general syntax of a function template is

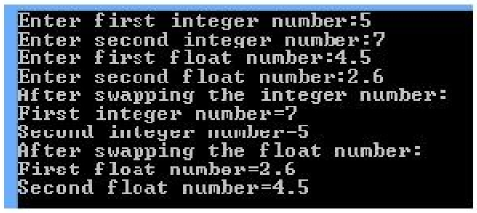
```
template <class T>
return_type function_name (arguments of type T)
{
        // - - - -
        // Body of function
        // with type T
        // wherever appropriate
        // - - - -
}
```

- The function template syntax is similar to that of the class template except that we are defining function instead of classes. We must use the template parameter T as and when necessary in the function body and its argument list.

Example 6-5:

```
#include<iostream>
#include<conio.h>
using namespace std;
template <class T>
void swapfun(T &a,T &b)
{
        T temp;
        temp=a;
        a=b;
        b=temp;
}
int main()
{
        int i1,i2;
        float f1,f2;
        cout<<"Enter first integer number:";
        cin>>i1;
        cout<<"Enter second integer number:";
        cin>>i2;
        cout<<"Enter first float number:";
        cin>>f1;
        cout<<"Enter second float number:";
        cin>>f2;
        cout<<"After swapping the integer number:"<<endl;
        swapfun(i1,i2);
        cout<<"First integer number="<<i1<<endl;
        cout<<"Second integer number="<<i2<<endl;
        cout<<"After swapping the float number:"<<endl;
        swapfun(f1,f2);
        cout<<"First float number="<<f1<<endl;
        cout<<"Second float number="<<f2<<endl;
        getch();
        return 0;
}
```

Output:



```
Enter first integer number:5
Enter second integer number:7
Enter first float number:4.5
Enter second float number:2.6
After swapping the integer number:
First integer number=7
Second integer number=5
After swapping the float number:
First float number=2.6
Second float number=4.5
```

6.3.1 Function Template with multiple parameter

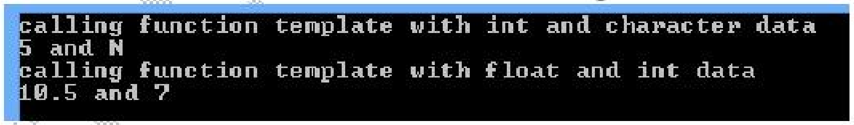
- Similar to template classes, we can use more than one generic data type in the template statement using a comma-separated list as shown below:

```
template<class T1, class T2, ....>
return_type function_name(arguments of types T1,T2, .....)
{
        //Body of function

}
```

Example 6-6:

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T1, class T2>
void display(T1 x,T2 y)
{
        cout<<x<<" and "<<y<<endl;
}
int main()
{
        cout<<"calling function template with int and character data"<<endl;
        display(5,'N');
        cout<<"calling function template with float and int data"<<endl;
        display(10.5,7);
        getch();
        return 0;
}
```

Output:



```
calling function template with int and character data
5 and N
calling function template with float and int data
10.5 and 7
```

6.3.2 Overloading of Template Functions

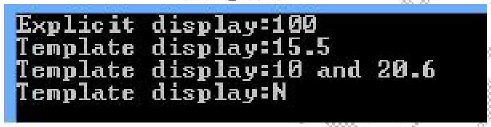
- A template function may be overloaded either by template function or ordinary function of its name. In such cases, the overloading resolution is accomplished as follows:
  1. Call an ordinary function that has an exact match.
  2. Call a template function that could be created with an exact match.
  3. Try normal overloading resolution to ordinary functions and call the one that matches.
- An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions.

Example 6-7:

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T>
void display(T x)
{
        cout<<"Template display:"<<x<<endl;
}
template<class T1, class T2>
void display(T1 m,T2 n)
{
        cout<<"Template display:"<<m<<" and "<<n<<endl;
}
```

```
void display(int x)
{
        cout<<"Explicit display:"<<x<<endl;
}
int main()
{
        display(100);
        display(15.5);
        display(10,20.6);
        display('N');
        getch();
        return 0;
}
```

Output:



```
Explicit display:100
Template display:15.5
Template display:10 and 20.6
Template display:N
```

6.4 Non-type Template argument

- Template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument **T,** we can also use other arguments such as built-in type (int, float etc), constant expression, function name and strings.
- Consider the following example:

```
template<class T, int size>
class sample
{
        T a[size];
        ………
        ………
};
```

In above the template has two argument one **T** type and other is built-in type (int).
If in main we write

```
sample<int, 30> obj1;
sample<char, 20> obj2;
```

Then **obj1** is array of 30 integer and **obj2** is array of 20 characters.

6.5 Standard Template Library (STL)

In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett Packard developed a site of general purpose templatized class (data structure) and functions (algorithms) that could be used as a standard approach of storing and processing of data. The collection of these generic classes and functions is called the Standard Template Library (STL). STL has now become a part of ANSI standard C++ class library.

Using STL we can save considerable time and effort and lead to high quality programs, all these benefits are possible because we are basically reusing the well written and well tested components defined in the STL. STL components are defined in the namespace std. We must therefore use the using namespace directive.

- The STL provide a readymade set of common classes for C++ that can be used with any built-in type and with any user defined type that supports some elementary operation
- STL algorithms are independent of containers, which significantly reduces the complexity of the library.
- It set out general purpose template classes (data structure) and functions that could be used as 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 as its core there are three components these three components works in conjunction with one another to provide support to a variety of programming solutions

  Syntax:

  ```
  using namespace std;
  ```

6.5.1 Components of STL:

The STL contains several components and they are

1. Containers
2. Algorithms
3. Iterators

These three components work in conjunction with one another to provide support to a variety of programming solutions. Figure below shows algorithms employ iterators to perform operation stored in containers.

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

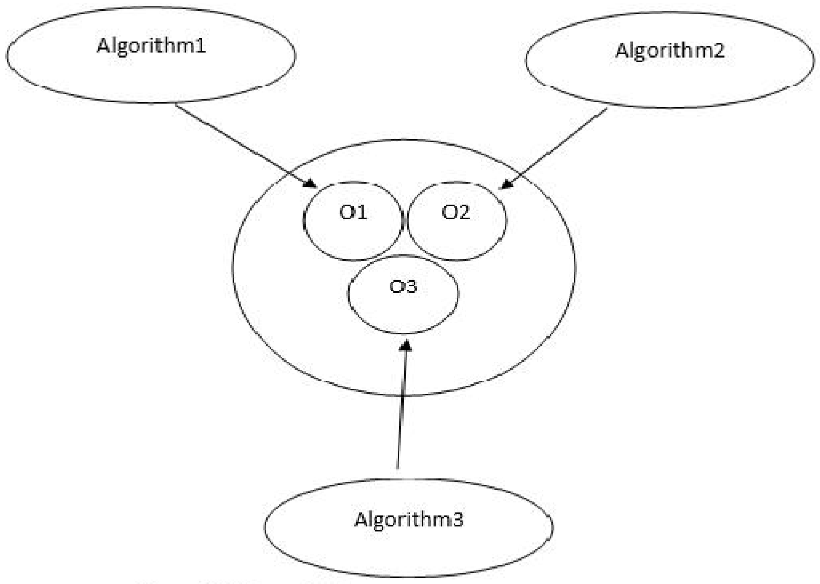


Figure 6-1: Relationship between three STL components

An **algorithm** is a procedure used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, coping, sorting and merging. Algorithms are implemented by templates functions. The algorithm is implemented by template.

An **iterator** is an object (like pointer) that points to an element in a container. We can use iterators to move through the contents of containers. It is handled just like pointer and can be incremented and decremented. Iterator connect algorithm with containers and play a key role in the manipulation of data stored in the containers.

6.5.2 Features of STL

It helps in saving time, efforts, load fast, high quality programming because STL provides well written and tested components, which can be reuse in our program to make our program more robustness.

6.6 Exception Handling

- The most common types of error (also known as bugs) occurred while programming in C++ are Logic error and Syntactic error. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language. These errors are detected by using exhaustive debugging and testing.
- We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions. Exceptions are run-time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. ANSI C++ provides built-in language features to detect and handle exception which are basically run time errors.
- Exception handling was added to ANSII C++, provides a type safe approach for copying with the unusual predictable problems that arise while executing a program. So, exception handling is the mechanism by using which we can identify and deal such unusual conditions.

6.6.1 Basics of Exception Handling

- Exceptions are of two kinds, namely, synchronous exception and asynchronous exception. Errors such as "out of range index" and "overflow" belong to synchronous type exceptions whereas errors that caused by events beyond the control of the program (such as keyboard interrupts) are known as asynchronous exceptions.
- The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstances", so that appropriate action can be taken. The exception handling mechanism in C++ can handle only synchronous exceptions. The exception handling mechanism performs the following tasks:
  1. Hit the exception i.e. find the unusual condition
  2. Throw the exception i.e. inform that error has occurred
  3. Catch the exception i.e. receive the error information
  4. Handle the exception i.e. take the action for correction of problem

6.6.2 Exception Handling Mechanism

- C++ exception handling mechanism is basically built upon three keywords namely **try, throw** and **catch**.
  - The keyword **try** is used to preface a block of statements (surrounded by braces) which may generate exception. This block of statement is known as try block.

    Syntax:
    ```
    try
    {
            //statements
    }
    ```
  - When as exception is detected it is thrown using a **throw** statement in the try block.

    Syntax:
    ```
    throw(exception);
    ```
  - A catch block defined by the keyword **catch**, catches the exception thrown by the throw statement in the try block and handles it appropriately.

    Syntax:
    ```
    catch(exception)
    {
            //exception handling statements
            //or user notification statements
    }
    ```
- The catch block must immediately follow the try block that throws the exception. The general syntax is

```
- - - -
try
{
        - - - -
        - - - - // block of statement which detects and throws exception.
        throw exception ;
```

```
        - - - -
}
catch(type arg)
{
        - - - - // block of statements that handles the exception
        - - - -
}
```

Throwing Mechanism:
The exception is thrown by the use of throw statement in one of the following ways:

```
throw(exception);
throw exception;
throw;
```

The object "exception" may be of any type or a constant. We can also throw an object not intended for error handling.

Catching Mechanism:
Code for handling exceptions is included in catch blocks. A catch block looks like a function definition and is of the form:

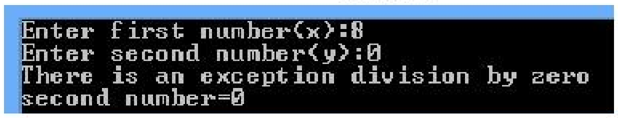
```
catch(type arg)

{

        // statements for managing exceptions

}
```

The "type" indicates the type of exception that catch block handles. The parameter arg is an optional parameter name. The exception handling code is placed between two braces. The **catch** statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

Example 6-8: Write a program which reads two numbers and then divide first number by second number. Raise exception if second number is zero.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
        int x,y;
        cout<<"Enter first number(x):";
        cin>>x;
        cout<<"Enter second number(y):";
        cin>>y;
        try
        {
        if(y!=0)
                cout<<"Division x/y="<<(x/y)<<endl;
        else
                throw(y);
        }
        catch(int n)
        {
                cout<<"There is an exception division by zero"<<endl;
                cout<<"second number="<<n;
        }
        getch();
        return 0;
}
```

Output:



```
Enter first number(x):8
Enter second number(y):0
There is an exception division by zero
second number=0
```

- Note: In example 6-8, the argument within catch is **int** type (i.e. **catch(int n)**) because exception object is an **int** type (i.e. in statement **throw(y)**, y is **int** type).

6.6.3 Exception generated by a function

- When we call a function which generate the exception then the **throw** statement is written within function body.

```
return-type function_name(arg)
{
        ……..
        throw exception;
        ………
}
```

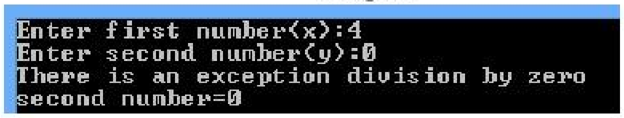
- We can call the function within **try** block

```
try
{
        …………..
        function_name(actual parameter);
        …………….
}
catch(arg)
{
        …………….
        …………….
}
```

Example 6-9: Write a function which divide one number by another number. Raise exception if there is division by zero condition. Call that thin try block.

```
#include<iostream>
#include<conio.h>
using namespace std;
void div(int f,int s)
{
        if(s!=0)
                cout<<"Division x/y="<<(f/s)<<endl;
        else
                throw(s);
}
int main()
{
        int x,y;
        cout<<"Enter first number(x):";
        cin>>x;
        cout<<"Enter second number(y):";
        cin>>y;
        try
        {
                div(x,y);
        }
        catch(int n)
        {
                cout<<"There is an exception division by zero"<<endl;
                cout<<"second number="<<n;
        }
        getch();
        return 0;
}
```

Output:



```
Enter first number(x):4
Enter second number(y):0
There is an exception division by zero
second number=0
```

6.6.4 Multiple catch statement

- If a program has more than one condition to throw an exception then we can use a multiple catch statement.
- Syntax of multiple catch statement

```
try
{
        .....
}
catch(type1 arg)
{
        //block1
}
catch(type2 arg)
{
        //block2
}
.
.
.
catch(typen arg)
{
        //blockn
}
```

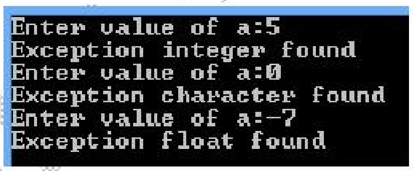
- When an exception is thrown from try block exception handlers are searched in order for an appropriate match i.e. first the type of exception thrown is checked with the type of arg of first catch (with type1) if both has same type then the corresponding block is execute (block1) if both has not same type then that checked with type of arg of second catch (with type2) if there is a matched then body of second catch (block2) execute. If the type has not same as type of arg of second catch then that checked with type of third catch and so on.

Example 6-10: Write a function which take an integer as an argument and raise an exception integer found if value of argument is > 0, raise an exception character found is value of argument is = 0, raise an exception float if value of argument is < 0.

```
#include<iostream>
#include<conio.h>
using namespace std;
void sample(int n)
{
        try
        {
                if(n>0)
                        throw n;
                else if(n==0)
                        throw 'n';
                else
                        throw float(n);
        }
        catch(int x)
        {
                cout<<"Exception integer found"<<endl;
        }
        catch(char y)
        {
                cout<<"Exception character found"<<endl;
        }
```

```
          catch(float z)
          {
                    cout<<"Exception float found"<<endl;
          }
}
int main()
{
          int a;
          cout<<"Enter value of a:";
          cin>>a;
          sample(a);
          cout<<"Enter value of a:";
          cin>>a;
          sample(a);
          cout<<"Enter value of a:";
          cin>>a;
          sample(a);
          getch();
          return 0;
}
```

Output:



```
Enter value of a:5
Exception integer found
Enter value of a:0
Exception character found
Enter value of a:-7
Exception float found
```

- **Note:** From above example we can say that function which generate exception can be called from main without within try block if try and catch block are part of function.

6.6.5 Catch all exception

- If we have a situation, we are not able to anticipate all possible types of exceptions and therefore not able to design independent catch handlers to catch them. In such case, we can force a catch statement to catch all exceptions instead of a certain type.
- For this we can use the following type of catch block:

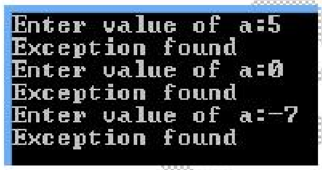
```
catch(…)
{
          …….
}
```

Example 6-11: Write a program using function which take a parameter if the value of parameter>0 then throw integer type, if parameter = 0 then throw character type, if parameter < 0 then throw float type exception but for all design only one catch block.

```
#include<iostream>
#include<conio.h>
using namespace std;
void sample(int n)
{
          try
          {
                    if(n>0)
                              throw n;
                    else if(n==0)
                              throw 'n';
                    else
                              throw float(n);
          }
          catch(...)
          {
                    cout<<"Exception found"<<endl;
          }
}
```

```
int main()
{
        int a;
        cout<<"Enter value of a:";
        cin>>a;
        sample(a);
        cout<<"Enter value of a:";
        cin>>a;
        sample(a);
        cout<<"Enter value of a:";
        cin>>a;
        sample(a);
        getch();
        return 0;
}
```

Output:



```
Enter value of a:5
Exception found
Enter value of a:0
Exception found
Enter value of a:-7
Exception found
```

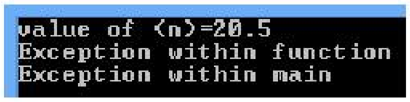
6.6.6 Rethrowing exception

- For rethrowing exception we can write a statement like follows:
  throw;
- This causes the current exception to be thrown to next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

Example 6-12:

```
#include<iostream>
#include<conio.h>
using namespace std;
void sample(float n)
{
        try
        {
                if(n==0.0)
                        throw n;
                else
                        cout<<"value of (n)="<<n<<endl;
        }
        catch(float x)
        {
                cout<<"Exception within function"<<endl;
                throw;
        }
}
int main()
{
        try
        {
                sample(20.5);
                sample(0.0);
        }
        catch(float m)
        {
                cout<<"Exception within main"<<endl;
        }
        getch();
        return 0;
}
```

Output:



```
value of (n)=20.5
Exception within function
Exception within main
```

7 Object Oriented Design

A cursory explanation of object-oriented programming tends to emphasize the syntactic features of languages such as C++ or Delphi, as opposed to their older, non object-oriented versions, C or Pascal. Thus, an explanation usually turns rather quickly to issues such as classes and inheritance, message passing, and virtual and static methods. But such a description will miss the most important point of object-oriented programming, which has nothing to do with syntax. Working in an object-oriented language (that is, one that supports inheritance, message passing, and classes) is neither a necessary nor sufficient condition for doing object-oriented programming. As we emphasized in Chapters 1 and 2, the most important aspect of OOP is the creation of a universe of largely autonomous interacting agents. But how does one come up with such a system? The answer is a design technique driven by the determination and delegation of responsibilities. The technique described in this chapter is termed responsibility-driven design.

7.1 Responsibility Implies Noninterference

As anyone can attest who can remember being a child, or who has raised children, responsibility is a sword that cuts both ways. When you make an object (be it a child or a software system) responsible for specific actions, you expect a certain behavior, at least when the rules are observed. But just as important, responsibility implies a degree of independence or noninterference. If you tell a child that she is responsible for cleaning her room, you do not normally stand over her and watch while that task is being performed-that is not the nature of responsibility. Instead, you expect that, having issued a directive in the correct fashion, the desired outcome will be produced.

The difference between conventional programming and object-oriented programming is in many ways the difference between actively supervising a child while she performs a task, and delegating to the child responsibility for that performance. Conventional programming proceeds largely by doing something to something else-modifying a record or updating an array, for example. Thus, one portion of code in a software system is often intimately tied, by control and data connections, to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values, or simply through inappropriate use of and dependence on implementation details of other portions of code. A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible.

This notion might at first seem no more subtle than the concepts of information hiding and modularity, which are important to programming even in conventional languages. But responsibility-driven design elevates information hiding from a technique to an art. This principle of information hiding becomes vitally important when one moves from programming in the small to programming in the large.

One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. For example, a simulation manager might work for both a simulation of balls on a billiards table and a simulation of fish in a fish tank. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behavior to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned-it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express. In subsequent chapters, we will present several such examples.

7.2 Programming in the Small and in the Large

The difference between the development of individual projects and of more sizable software systems is often described as programming in the small versus programming in the large.

Programming in the small characterizes projects with the following attributes:

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

Programming in the large, on the other hand, characterizes software projects with features such as the following:

- The software system is developed by a large team, often consisting of people with many different skills. There may be graphic artists, design experts, as well as programmers. Individuals involved in the specification or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.

- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

While the beginning student will usually be acquainted with programming in the small, aspects of many object-oriented languages are best understood as responses to the problems encountered while programming in the large. Thus, some appreciation of the difficulties involved in developing large systems is a helpful prerequisite to understanding OOP.

7.3 Why Begin with Behavior?

Why begin the design process with an analysis of behavior? The simple answer is that the behavior of a system is usually understood long before any other aspect.

Earlier software development methodologies (those popular before the advent of object-oriented techniques) concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application. But structural elements of the application can be identified only after a considerable amount of problem analysis. Similarly, a formal specification often ended up as a document understood by neither programmer nor client. But behavior is something that can be described almost from the moment an idea is conceived, and (often unlike a formal specification) can be described in terms meaningful to both the programmers and the client.

Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development. It is but one of many alternative object-oriented design techniques. We will illustrate the application of Responsibility-Driven Design with a case study

7.4 A Case Study in RDD

Imagine you are the chief software architect in a major computer firm. One day your boss walks into your office with an idea that, it is hoped, will be the next major success in your product line. Your assignment is to develop the Interactive *Intelligent Kitchen Helper* (Figure 7.1).

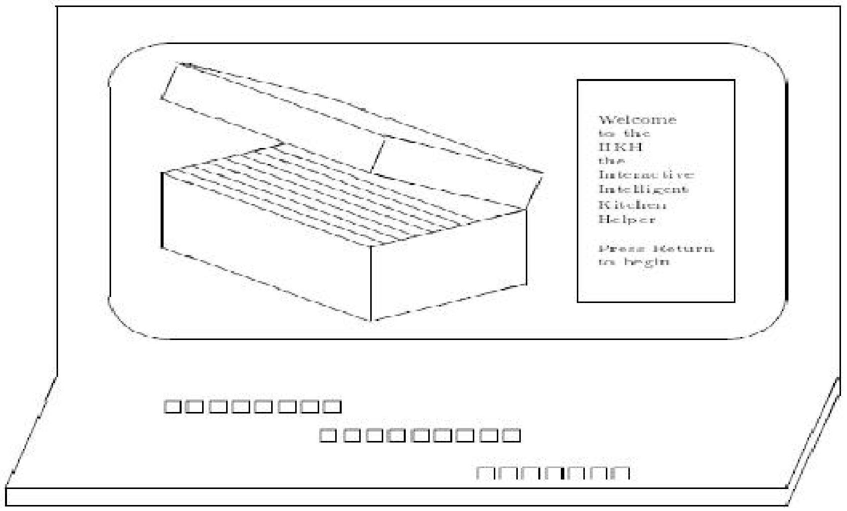


Figure 7-1: View of the Interactive Intelligent Kitchen Helper.

The task given to your software team is stated in very few words (written on what appears to be the back of a slightly-used dinner napkin, in handwriting that appears to be your boss's).

7.4.1 The interactive Intelligent Kitchen Helper

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

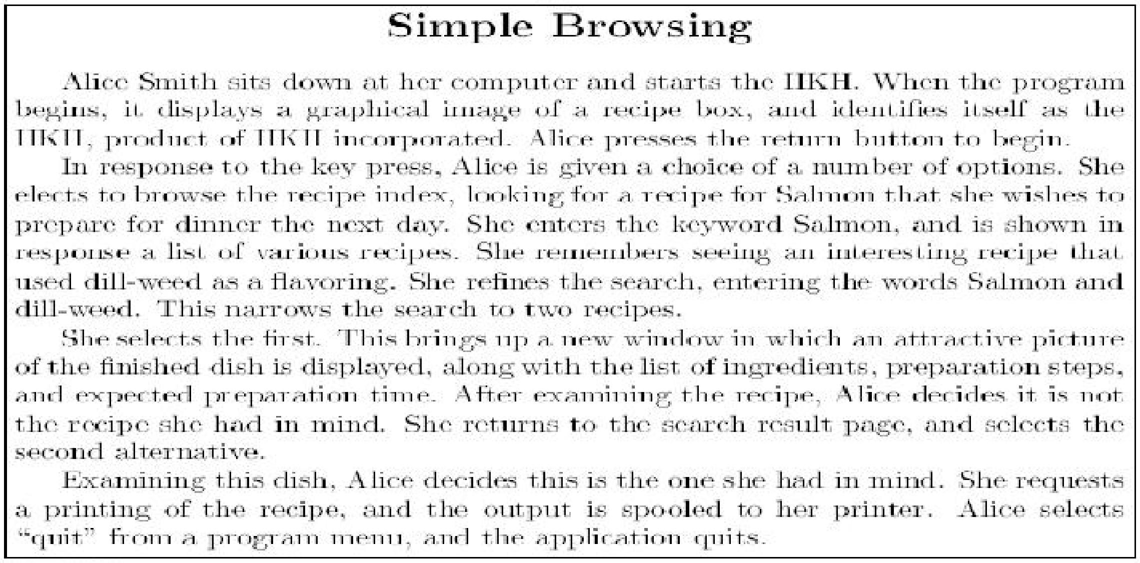
As is usually true with the initial descriptions of most software systems, the specification for the IIKH is highly ambiguous on a number of important points. It is also true that, in all likelihood, the eventual design and development of the software system to support the IIKH will require the efforts of several programmers working together. Thus, the initial goal of the design team must be to clarify the ambiguities in the description and to outline how the project can be divided into components to be assigned for development to individual team members.

The fundamental cornerstone of object-oriented programming is to characterize software in terms of behavior; that is, actions to be performed. We will see this repeated on many levels in the development of the IIKH. Initially, the team will try to characterize, at a very high level of abstraction, the behavior of the entire application. This then leads to a description of the behavior of various software subsystems. Only when all behavior has been identified and described will the software design team proceed to the coding step. In the next several sections we will trace the tasks the software design team will perform in producing this application.

7.4.2 Working through Scenarios

The first task is to refine the specification. As we have already noted, initial specifications are almost always ambiguous and unclear on anything except the most general points. There are several goals for this step. One objective is to get a better handle on the \look and feel" of the eventual product. This information can then be carried back to the client (in this case, your boss) to see if it is in agreement with the original conception. It is likely, perhaps inevitable, that the specifications for the final application will change during the creation of the software system, and it is important that the design be developed to easily accommodate change and that potential changes be noted as early as possible. Equally important, at this point very high level decisions can be made concerning the structure of the eventual software system. In particular, the activities to be performed can be mapped onto components.

In order to uncover the fundamental behavior of the system, the design team first creates a number of scenarios. That is, the team acts out the running of the application just as if it already possessed a working system. An example scenario is shown in Figure 7.2.



Simple Browsing

Alice Smith sits down at her computer and starts the IIKH. When the program begins, it displays a graphical image of a recipe box, and identifies itself as the IIKH, product of IIKH incorporated. Alice presses the return button to begin.

In response to the key press, Alice is given a choice of a number of options. She elects to browse the recipe index, looking for a recipe for Salmon that she wishes to prepare for dinner the next day. She enters the keyword Salmon, and is shown in response a list of various recipes. She remembers seeing an interesting recipe that used dill-weed as a flavoring. She refines the search, entering the words Salmon and dill-weed. This narrows the search to two recipes.

She selects the first. This brings up a new window in which an attractive picture of the finished dish is displayed, along with the list of ingredients, preparation steps, and expected preparation time. After examining the recipe, Alice decides it is not the recipe she had in mind. She returns to the search result page, and selects the second alternative.

Examining this dish, Alice decides this is the one she had in mind. She requests a printing of the recipe, and the output is spooled to her printer. Alice selects "quit" from a program menu, and the application quits.

Figure 7-2: An Example Scenario.

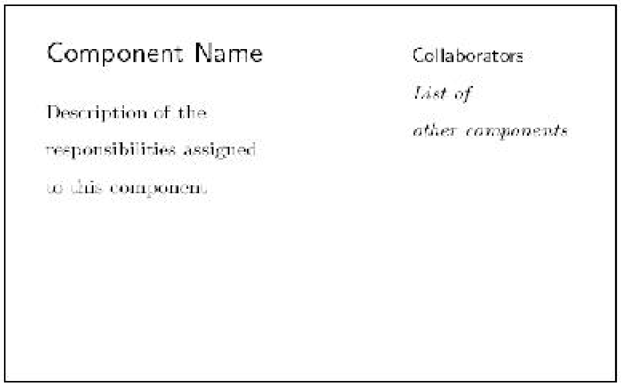
7.4.3 Identification of Components

The engineering of a complex physical system, such as a building or an auto-mobile engine, is simplified by dividing the design into smaller units. So, too, the engineering of software is simplified by the identification and development of software components. A component is simply an abstract entity that can perform tasks-that is, fulfill some responsibilities. At this point, it is not necessary to know exactly the eventual representation for a component or how a component will perform a task. A component may ultimately be turned into a function, a structure or class, or a collection of other components. At this level of development there are just two important characteristics:

- A component must have a small well-defined set of responsibilities.
- A component should interact with other components to the minimal extent possible.

7.5 CRC Cards-Recording Responsibility

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



As part of this process, it is often useful to represent components using small index cards. Written on the face of the card is the name of the software component, the responsibilities of the component, and the names of other components with which the component must interact. Such cards are sometimes known as CRC (Component, Responsibility, Collaborator) cards, and are associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the CRC card.

7.5.1 Give Components a Physical Representation

While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the "surrogate" for the software during the scenario simulation. He or she describes the activities of the software system, passing "control" to another member when the software system requires the services of another component.

An advantage of CRC cards is that they are widely available, inexpensive, and erasable. This encourages experimentation, since alternative designs can be tried, explored, or abandoned with little investment. The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling (which we will describe shortly). The constraints of an index card are also a good measure of approximate complexity-a component that is expected to perform more tasks than can fit easily in this space is probably too complex, and the team should find a simpler solution, perhaps by moving some responsibilities elsewhere to divide a task between two or more new components.

7.5.2 The What/Who Cycle

As we noted at the beginning of this discussion, the identification of components takes place during the process of imagining the execution of a working system. Often this proceeds as a cycle of what/who questions. First, the design team identifies what activity needs to be performed next. This is immediately followed by answering the question of who performs the action. In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

A popular bumper sticker states that phenomena can and will spontaneously occur. (The bumper sticker uses a slightly shorter phrase.) We know, however, that in real life this is seldom true. If any action is to take place, there must be an agent assigned to perform it. Just as in the running of a club any action to be performed must be assigned to some individual, in organizing an object-oriented program all actions must be the responsibility of some component. The secret to good object-oriented design is to first establish an agent for each action.

7.5.3 Documentation

At this point the development of documentation should begin. Two documents should be essential parts of any software system: the user manual and the system design documentation. Work on both of these can commence even before the first line of code has been written.

The user manual describes the interaction with the system from the user's point of view; it is an excellent means of verifying that the development team's conception of the application matches the client's. Since the decisions made in creating the scenarios will closely match the decisions the user will be required to make in the eventual application, the development of the user manual naturally dovetails with the process of walking through scenarios.

Before any actual code has been written, the mindset of the software team is most similar to that of the eventual users. Thus, it is at this point that the developers can most easily anticipate the sort of questions to which a novice user will need answers. A user manual is also an excellent tool to verify that the programming team is looking at the problem in the same way that the client intended. A client seldom presents the programming team with a complete and formal specification, and thus some reassurance and two-way communication early in the process, before actual programming has begun, can prevent major misunderstandings.

The second essential document is the design documentation. The design documentation records the major decisions made during software design, and should thus be produced when these decisions are fresh in the minds of the creators, and not after the fact when many of the relevant details will have been forgotten. It is often far easier to write a general global description of the software system early in the development. Too soon, the focus will move to the level of individual components or modules. While it is also important to document the module level, too much concern with the details of each module will make it difficult for subsequent software maintainers to form an initial picture of the larger structure.

CRC cards are one aspect of the design documentation, but many other important decisions are not reflected in them. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions. A log or diary of the project schedule should be maintained. Both the user manual and the design documents are refined and evolve over time in exactly the same way the software is refined and evolves.

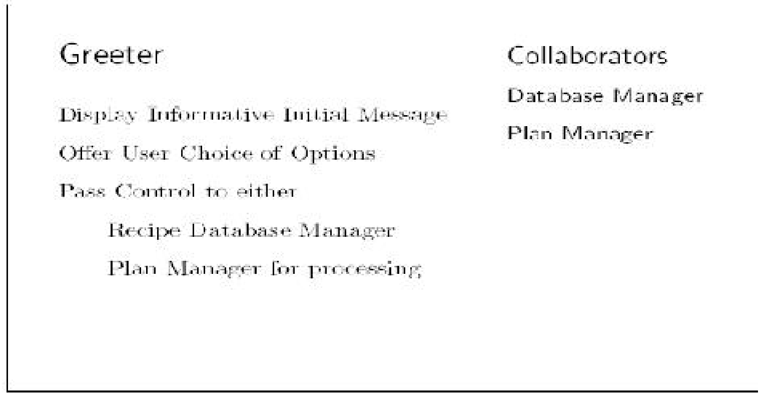
7.6 Components and Behavior

To return to the IIKH, the team decides that when the system begins, the user will be presented with an attractive informative window (see Figure 7.1). The responsibility for displaying this window is assigned to a component called the Greeter. In some as yet unspecified manner (perhaps by pull-down menus, button or key presses, or use of a pressure-sensitive screen), the user can select one of several actions. Initially, the team identifies just five actions:

1. Casually browse the database of existing recipes, but without reference to any particular meal plan.
2. Add a new recipe to the database.
3. Edit or annotate an existing recipe.
4. Review an existing plan for several meals.
5. Create a new plan of meals

These activities seem to divide themselves naturally into two groups. The first three are associated with the recipe database; the latter two are associated with menu plans. As a result, the team next decides to create components corresponding to these two responsibilities. Continuing with the scenario, the team elects to ignore the meal plan management for the moment and move on to refine the activities of the Recipe Database component. Figure 7.3 shows the initial CRC card representation of the Greeter.

Broadly speaking, the responsibility of the recipe database component is simply to maintain a collection of recipes. We have already identified three elements of this task: The recipe component database must facilitate browsing the library of existing recipes, editing the recipes, and including new recipes in the database.



Greeter

Display Informative Initial Message

Offer User Choice of Options

Pass Control to either

Recipe Database Manager

Plan Manager for processing

Collaborators

Database Manager

Plan Manager

Figure 7-3: CRC card for the Greeter.

7.6.1 Postponing Decisions

There are a number of decisions that must eventually be made concerning how best to let the user browse the database. For example, should the user first be presented with a list of categories, such as “soups," “salads," “main meals," and “desserts"? Alternatively, should the user be able to describe keywords to narrow a search, perhaps by providing a list of ingredients, and then see all the recipes that contain those items (\Almonds, Strawberries, Cheese"), or a list of previously inserted keywords (“Bob's favorite cake")? Should scroll bars be used or simulated thumb holes in a virtual book? These are fun to think about, but the important point is that such decisions do not need to be made at this point (see Section 7.6.2, “Preparing for Change"). Since they affect only a single component, and do not affect the functioning of any other system, all that is necessary to continue the scenario is to assert that by some means the user can select a specific recipe.

7.6.2 Preparing for Change

It has been said that all that is constant in life is the inevitability of uncertainty and change. The same is true of software. No matter how carefully one tries to develop the initial specification and design of a software system, it is almost certain that changes in the user's needs or requirements will, sometime during the life of the system, force changes to be made in the software. Programmers and software designers need to anticipate this and plan accordingly.

- The primary objective is that changes should affect as few components as possible. Even major changes in the appearance or functioning of an application should be possible with alterations to only one or two sections of code.
- Try to predict the most likely sources of change and isolate the effects of such changes to as few software components as possible. The most likely sources of change are interfaces, communication formats, and output formats.
- Try to isolate and reduce the dependency of software on hardware. For example, the interface for recipe browsing in our application may depend in part on the hardware on which the system is running. Future releases may be ported to different platforms. A good design will anticipate this change.
- Reducing coupling between software components will reduce the dependence of one upon another, and increase the likelihood that one can be changed with minimal effect on the other.
- In the design documentation maintain careful records of the design process and the discussions surrounding all major decisions. It is almost certain that the individuals responsible for maintaining the software and designing future releases will be at least partially different from the team producing the initial release. The design documentation will allow future teams to know the important factors behind a decision and help them avoid spending time discussing issues that have already been resolved.

7.6.3 Continuing the Scenario

Each recipe will be identified with a specific recipe component. Once a recipe is selected, control is passed to the associated recipe object. A recipe must contain certain information. Basically, it consists of a list of ingredients and the steps needed to transform the ingredients into the final product. In our scenario, the recipe component must also perform other activities. For example, it will display the recipe interactively on the terminal screen. The user may be given the ability to annotate or change either the list of ingredients or the instruction portion. Alternatively, the user may request a printed copy of the recipe. All of these actions are the responsibility of the Recipe component. (For the moment, we will continue to describe the Recipe in singular form. During design we can think of this as a prototypical recipe that stands in place of a multitude of actual recipes. We will later return to a discussion of singular versus multiple components.)

Having outlined the actions that must take place to permit the user to browse the database, we return to the recipe database manager and pretend the user has indicated a desire to add a new recipe. The database manager somehow decides in which category to place the new recipe (again, the details of how this is done are unimportant for our development at this point), requests the name of the new recipe, and then creates a new recipe component, permitting the user to edit this new blank entry. Thus, the responsibilities of performing this new task are a subset of those we already identified in permitting users to edit existing recipes. Having explored the browsing and creation of new recipes, we return to the Greeter and investigate the development of daily menu plans, which is the Plan Manager's task. In some way (again, the details are unimportant here) the user can save existing plans. Thus, the Plan Manager can either be started by retrieving an already developed plan or by creating a new plan. In the latter case, the user is prompted for a list of dates for the plan. Each date is associated with a separate Date component. The user can select a specific date for further investigation, in which case control is passed to the corresponding Date component. Another activity of the Plan Manager is printing out the recipes for the planning period. Finally, the user can instruct the Plan Manager to produce a grocery list for the period.

The Date component maintains a collection of meals as well as any other an-notations provided by the user (birthday celebrations, anniversaries, reminders, and so on). It prints information on the display concerning the specified date. By some means (again unspecified), the user can indicate a desire to print all the information concerning a specific date or choose to explore in more detail a specific meal. In the latter case, control is passed to a Meal component.

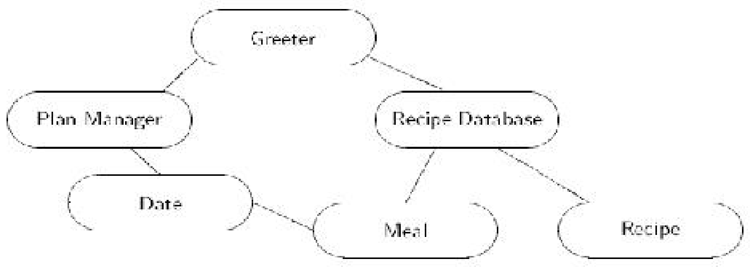


Figure 7-4: Communication between the six components in the IIKH.

The Meal component maintains a collection of augmented recipes, where the augmentation refers to the user's desire to double, triple, or otherwise increase a recipe. The Meal component displays information about the meal. The user can add or remove recipes from the meal, or can instruct that information about the meal be printed. In order to discover new recipes, the user must be permitted at this point to browse the recipe database. Thus, the Meal component must interact with the recipe database component. The design team will continue in this fashion, investigating every possible scenario. The major category of scenarios we have not developed here is exceptional cases. For example, what happens if a user selects a number of keywords for a recipe and no matching recipe is found? How can the user cancel an activity, such as entering a new recipe, if he or she decides not to continue? Each possibility must be explored, and the responsibilities for handling the situation assigned to one or more components.

Having walked through the various scenarios, the software design team eventually decides that all activities can be adequately handled by six components (Figure 7.4). The Greeter needs to communicate only with the Plan Manager and the Recipe Database components. The Plan Manager needs to communicate only with the Date component; and the Date agent, only with the Meal component. The Meal component communicates with the Recipe Manager and, through this agent, with individual recipes.

7.6.4 Interaction Diagrams

While a description such as that shown in Figure 7.4 may describe the static relationships between components, it is not very good for describing their dynamic interactions during the execution of a scenario. A better tool for this purpose is an interaction diagram. Figure 7.5 shows the beginning of an interaction diagram for the interactive kitchen helper. In the diagram, time moves forward from the top to the bottom. Each component is represented by a labeled vertical line. A component sending a message to another component is represented by a horizontal arrow from one line to another. Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow. (Some authors use two different arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.) The commentary on the right side of the figure explains more fully the interaction taking place.

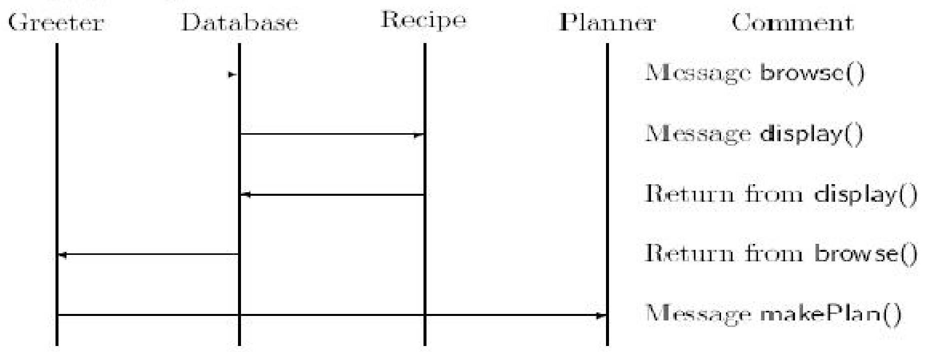


Figure 7-5: An Example interaction diagram.

With a time axis, the interaction diagram is able to describe better the sequencing of events during a scenario. For this reason, interaction diagrams can be a useful documentation tool for complex software systems.

7.7 Software Components

In this section we will explore a software component in more detail. As is true of all but the most trivial ideas, there are many aspects to this seemingly simple concept.

7.7.1 Behavior and State

We have already seen how components are characterized by their behavior, that is, by what they can do. But components may also hold certain information. Let us take as our prototypical component a Recipe structure from the IIKH. One way to view such a component is as a pair consisting of behavior and state.

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

It is not necessary that all components maintain state information. For example, it is possible that the Greeter component will not have any state since it does not need to remember any information during the course of execution. However, most components will consist of a combination of behavior and state.

7.7.2 Instances and Classes

The separation of state and behavior permits us to clarify a point we avoided in our earlier discussion. Note that in the real application there will probably be many different recipes. However, all of these recipes will perform in the same manner. That is, the behavior of each recipe is the same; it is only the state-the individual lists of ingredients and instructions for preparation-that differs between individual recipes. In the early stages of development our interest is in characterizing the behavior common to all recipes; the details particular to any one recipe are unimportant.

The term class is used to describe a set of objects with similar behavior. We will see in later chapters that a class is also used as a syntactic mechanism in almost all object-oriented languages. An individual representative of a class is known as an instance. Note that behavior is associated with a class, not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner. On the other hand, state is a property of an individual. We see this in the various instances of the class Recipe. They can all perform the same actions (editing, displaying, printing) but use different data values.

7.7.3 Coupling and Cohesion

Two important concepts in the design of software components are coupling and cohesion. Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.

Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse.

In particular, coupling is increased when one software component must access data values-the state-held by another component. Such situations should almost always be avoided in favor of moving a task into the list of responsibilities of the component that holds the necessary data. For example, one might conceivably first assign responsibility for editing a recipe to the Recipe Database component, since it is while performing tasks associated with this component that the need to edit a recipe first occurs. But if we did so, the Recipe Database agent would need the ability to directly manipulate the state (the internal data values representing the list of ingredients and the preparation instructions) of an individual recipe. It is better to avoid this tight connection by moving the responsibility for editing to the recipe itself.

7.7.4 Interface and Implementation-Parnas's Principles

The emphasis on characterizing a software component by its behavior has one extremely important consequence. It is possible for one programmer to know how to use a component developed by another programmer, without needing to know how the component is implemented. For example, suppose each of the six components in the IIKH is assigned to a different programmer. The programmer developing the Meal component needs to allow the IIKH user to browse the database of recipes and select a single recipe for inclusion in the meal. To do this, the Meal component can simply invoke the browse behavior associated with the Recipe Database component, which is defined to return an individual Recipe. This description is valid regardless of the particular implementation used by the Recipe Database component to perform the actual browsing action.

The purposeful omission of implementation details behind a simple interface is known as information hiding. We say the component encapsulates the behavior, showing only how the component can be used, not the detailed actions it performs. This naturally leads to two different views of a software system. The interface view is the face seen by other programmers. It describes what a software component can perform. The implementation view is the face seen by the programmer working on a particular component. It describes how a component goes about completing a task.

The separation of interface and implementation is perhaps the most important concept in software engineering. Yet it is difficult for students to understand, or to motivate. Information hiding is largely meaningful only in the context of multiperson programming projects. In such efforts, the limiting factor is often not the amount of coding involved, but the amount of communication required between the various programmers and between their respective software systems. As we will describe shortly, software components are often developed in parallel by different programmers, and in isolation from each other.

There is also an increasing emphasis on the reuse of general-purpose software components in multiple projects. For this to be successful, there must be minimal and well-understood interconnections between the various portions of the system. As we noted in the previous chapter, these ideas were captured by computer scientist David Parnas in a pair of rules, known as Parnas's principles:

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

A consequence of the separation of interface from implementation is that a programmer can experiment with several different implementations of the same structure without affecting other software components.

7.8 Formalize the Interface

We continue with the description of the IIKH development. In the next several steps the descriptions of the components will be refined. The first step in this process is to formalize the patterns and channels of communication.

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

7.8.1 Coming up with Names

Careful thought should be given to the names associated with various activities. Shakespeare said that a name change does not alter the object being described, but certainly not all names will conjure up the same mental images in the listener. As government bureaucrats have long known, obscure and idiomatic names can make even the simplest operation sound intimidating. The selection of useful names is extremely important, as names create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem. Often a considerable amount of time is spent finding just the right set of terms to describe the tasks performed and the objects manipulated. Far from being a barren and useless exercise, proper naming early in the design process greatly simplifies and facilitates later steps.

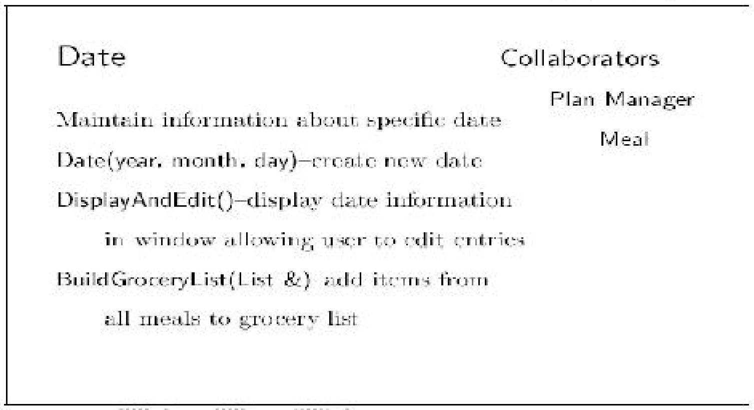
The following general guidelines have been suggested:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.

- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as "CardReader" or "Card_reader," rather than the less readable "cardreader."
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a "TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations. Does the empty function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O, 1 as l, 2 as Z, 5 as S).
- Name functions and variables that yield Boolean values so they describe clearly the interpretation of a true or false value. For example, "Printer-IsReady" clearly indicates that a true value means the printer is working, whereas "PrinterStatus" is much less precise.
- Take extra care in the selection of names for operations that are costly and infrequently used. By doing so, errors caused by using the wrong function can be avoided.

Once names have been developed for each activity, the CRC cards for each component are redrawn, with the name and formal arguments of the function used to elicit each behavior identified. An example of a CRC card for the Date is shown in Figure 7.6. What is not yet specified is how each component will perform the associated tasks.

Once more, scenarios or role playing should be carried out at a more detailed level to ensure that all activities are accounted for, and that all necessary information is maintained and made available to the responsible components.


Date

Maintain information about specific date

Date(year, month, day)–create new date

DisplayAndEdit()–display date information
in window allowing user to edit entries

BuildGroceryList(List &)–add items from
all meals to grocery list

Collaborators

Plan Manager

Meal

Figure 7-6: Revised CRC card for the Date component.

7.9 Designing the Representation

At this point, if not before, the design team can be divided into groups, each responsible for one or more software components. The task now is to transform the description of a component into a software system implementation. The major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.

It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they have been chosen, the code used by a component in the fulfillment of a responsibility is often almost self-evident. But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

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

7.10 Implementing Components

Once the design of each software subsystem is laid out, the next step is to implement each component's desired behavior. If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language. In a later section we will describe some of the more common heuristics used in this process.

If they were not determined earlier (say, as part of the specification of the system), then decisions can now be made on issues that are entirely self-contained within a single component. A decision we saw in our example problem was how best to let the user browse the database of recipes.

As multiperson programming projects become the norm, it becomes increasingly rare that any one programmer will work on all aspects of a system. More often, the skills a programmer will need to master are understanding how one section of code fits into a larger framework and working well with other members of a team.

Often, in the implementation of one component it will become clear that certain information or actions might be assigned to yet another component that will act "behind the scene," with little or no visibility to users of the software abstraction. Such components are sometimes known as *facilitators*. We will see examples of facilitators in some of the later case studies.

An important part of analysis and coding at this point is characterizing and documenting the necessary preconditions a software component requires to complete a task, and verifying that the software component will perform correctly when presented with legal input values.

7.11 Integration of Components

Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using stubs-simple dummy routines with no behavior or with very limited behavior-for the as yet unimplemented parts.

For example, in the development of the IIKH, it would be reasonable to start integration with the Greeter component. To test the Greeter in isolation, stubs are written for the Recipe Database manager and the daily Meal Plan manager. These stubs need not do any more than print an informative message and return. With these, the component development team can test various aspects of the Greeter system (for example, that button presses elicit the correct response). Testing of an individual component is often referred to as *unit testing*.

Next, one or the other of the stubs can be replaced by more complete code. For example, the team might decide to replace the stub for the Recipe Database component with the actual system, maintaining the stub for the other portion. Further testing can be performed until it appears that the system is working as desired. (This is sometimes referred to as integration testing.)

The application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is greatly facilitated by the conscious design goal of reducing connections between components, since this reduces the need for extensive stubbing.

During integration it is not uncommon for an error to be manifested in one software system, and yet to be caused by a coding mistake in another system. Thus, testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software, once more, into the larger system. Reexecuting previously developed test cases following a change to a software component is sometimes referred to as *regression testing*.

7.12 Maintenance and Evolution

It is tempting to think that once a working version of an application has been delivered the task of the software development team is finished. Unfortunately, that is almost never true. The term software maintenance describes activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall into this category.

- Errors, or bugs, can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing releases or in subsequent releases.
- Requirements may change, perhaps as a result of government regulations or standardization among similar products.
- Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure-sensitive touch screen, may become available. Output technology may change-for example, from a text-based system to a graphical window-based arrangement.
- User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products. Better documentation may be requested by users.

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.