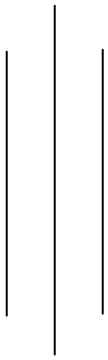
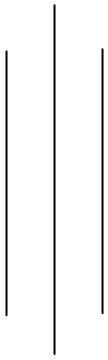
Kathford College of Engineering and Management

Balkumari, Lalitpur
Affiliated to **Tribhuvan University**



Lecture Notes
On
Object Oriented Programming C++



Bachelor's Degree in Computer and Electronic Engineering

Prepared by- **Prem Raj Bhatta**
[M.Sc. IT]

1. Introduction to Object Oriented Programming

Language Paradigms:

- Imperatives – Procedural Programming : C, Pascal, COBOL, FORTRAN etc.
- Applicative – Functional Programming – LISP, ML.
- Rule-based – Logic Programming :– PROLOG
- Object-Oriented Programming: – C++, JAVA, SMALLTALK

Our focus here is on procedural and Object oriented Programming approach.

Procedural programming Language

In procedural programming programs are organized in the form of subroutines. The subroutines do not let code duplication. This technique is only suitable for medium sized software applications. Conventional programming, using HLL e.g. COBOL, FORTRAN, C etc is commonly known as procedural programming. A program in Procedural Language is a list of instructions each statement in the language tells the computer to do something involving – reading, calculating, writing output. A number of functions are written to accomplish such tasks. Program become larger, it is broken into smaller units – functions. The primary focus of procedural oriented programming is on functions rather than data.

Procedure oriented programming basically consists of writing a list of instructions for computer to follow, and organize these instructions into groups known as functions.

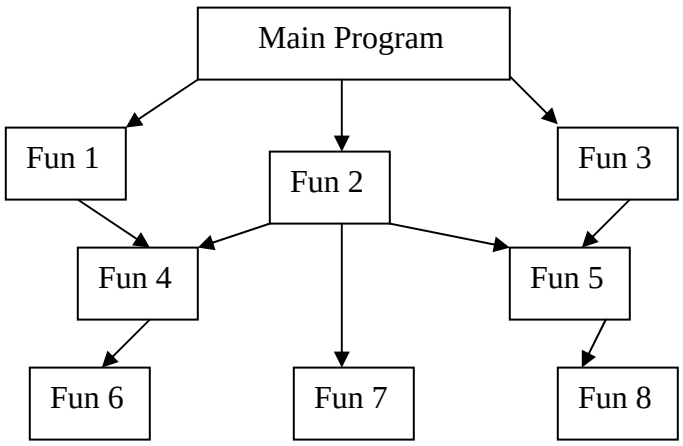
In procedural approach,

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

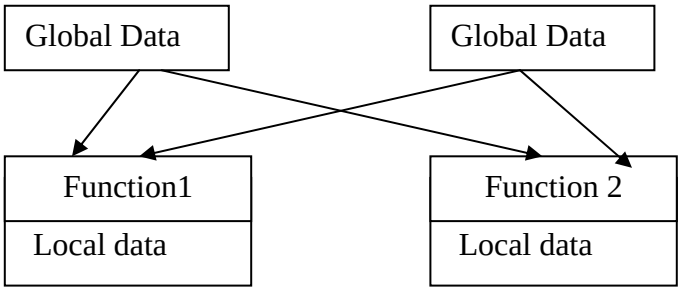
Dividing the program into functions and modules is a task in structured programming. In structured programming there are certain structures as

Sequences control
Selection Structures
Iteration
Functions
Modules

A pictorial views



– In a multi function program, two types of date are used: local and global.
  – Data items are placed as global so that they can be accessed by all the functions freely.
  – Each function may have their own data also called as local data.



Characteristics of Procedural approach:
  – Emphasis is on doing things, (Algorithms)
  – Large programs are divided into smaller program – function
  – Most of functions share global data.
  – Data more openly around system from function to function.
  – Function transform data from one form to another
  – Employs top-down approach for program design

Limitation of Procedural language
  – In large program, it is difficult to identify which data is used for which function.
  – Global variable overcome the local variable.
  – To revise an external data structure, all functions that access the data should also be revised.
  – Maintaining and enhancing program code is still difficult because of global data.
  – Focus on functions rather than data.
  – It does not model real world problem very well. Since functions are action oriented and do not really correspond to the elements of problem.

The Object Oriented Approach:

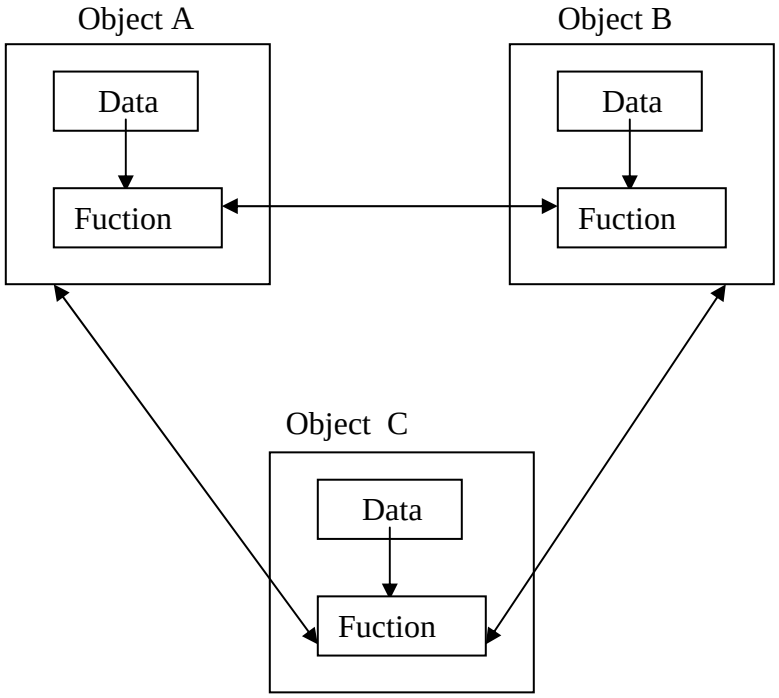
The fundamental idea behind object-oriented programming is to combine or *encapsulate* both *data* (or *instance variables*) and *functions* (or *methods*) that operate on that data into a single unit. This unit is called an *object*. The data is hidden, so it is safe from accidental alteration. An object's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

An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called *sending a message* to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called *message*.

In addition, object-oriented programming supports *encapsulation*, *abstraction*, *inheritance*, and *polymorphism* to write programs efficiently. Examples of object-oriented languages include *Simula*, *Smalltalk*, *C++*, *Python*, *C#*, *Visual Basic .NET* and *Java* etc.

In Object Oriented programming
  – Emphasis is on data rather than procedures.
  – Programs are divided into objects.
  – Data structures are designed such that they characterize the objects .

- Functions & data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
- Data is hidden & can't be accessed by external functions.
- Object can communicate with each other through function.
- New data & functions can be easily added.
- Follows Bottom up approach.



Features of Object Oriented Language:

1. **Objects**: Objects are the entities in an object oriented system through which we perceive the world around us. We naturally see our environment as being composed of things which have recognizable identities & behavior. The entities are then represented as objects in the program. They may represent a person, a place, a bank account, or any item that the program must handle. For example Automobiles are objects as they have size, weight, color etc as attributes (ie data) and starting, pressing the brake, turning the wheel, pressing accelerator pedal etc as operation (that is functions).

.

Object: Student
Data Name Date of birth Marks

Object: Account
Data Account number Account Type Name

Example of objects:

Physical Objects:

- Automobiles in traffic flow. simulation
- Countries in Economic model
- Air craft in traffic – control system .

Computer user environment objects.

-Window, menus, icons etc

Data storage constructs.

-Stacks, Trees etc

Human entities:

-employees, student, teacher etc.

Geometric objects:

-point, line, Triangle etc.

Objects mainly serve the following purposes:

- Understanding the real world and a practical base for designers
- Decomposition of a problem into objects depends on the nature of problem.

2. **Classes** : A class is a collection of objects of similar type. For example manager, peon, secretary, clerk are member of the class employee and class vehicle includes objects car, bus, etc.

   It defines a data type, much like a **struct** in C programming language and built in data type(int char, float etc). It specifies what data and functions will be included in objects of that class. Defining class doesn't create an object but class is the description of object's attributes and behaviors. Person Class : Attributes: Name, Age, Sex etc.

   Behaviors: Speak(), Listen(), Walk()

   Vehicle Class: Attributes: Name, model, color, height etc

   Behaviors: Start(), Stop(), Accelerate() etc.

   When class is defined, objects are created as

   <classname> <objectname> ;

   If employee has been defined as a class, then the statement

   employee manager;

   Will create an object manager belonging to the class employee.

Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class. The following points gives the idea of class:

- A class is a template that unites data and operations.
- A class is an abstraction of the real world entities with similar properties.
- Ideally, the class is an implementation of abstract data type.

3. Encapsulation and Data Abstraction:

The wrapping up of data and function into a single unit is called encapsulation. Encapsulation is most striking feature of a class. The data is not accessible from outside of class. Only member function can access data on that class. The insulation of data from direct access by the program is called data hiding. That is data can be hidden making them private so that it is safe from accidental alteration.

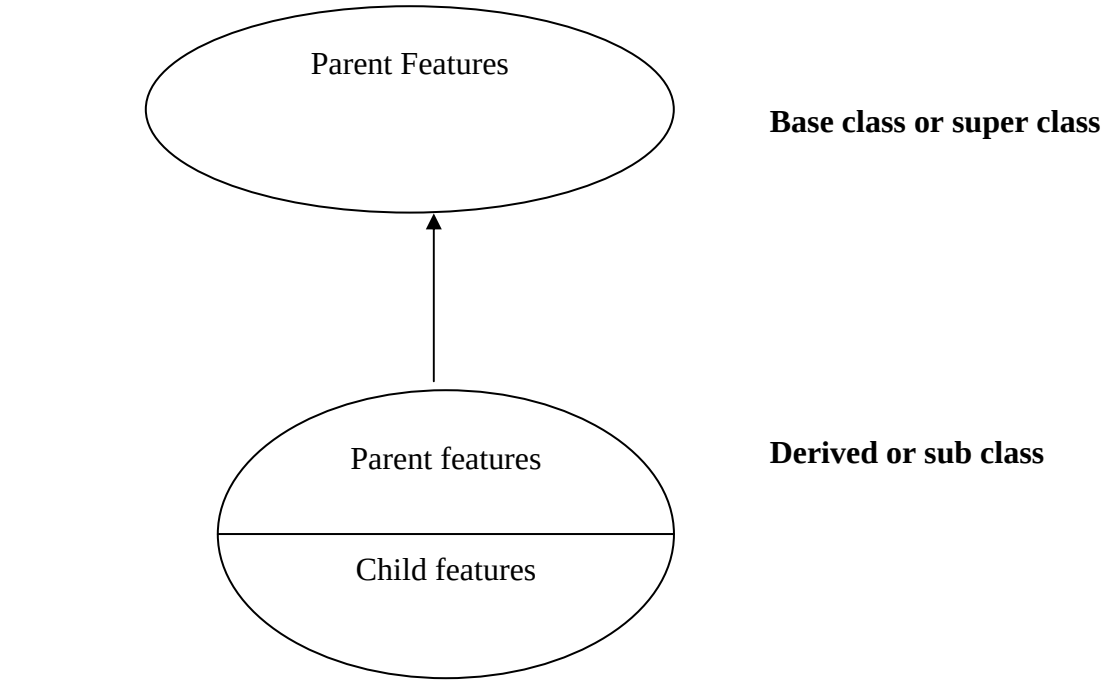
Abstraction is representing essential features of an object without including the background details or explanation. It focuses the outside view of an object, separating its essential behavior from its implementation.

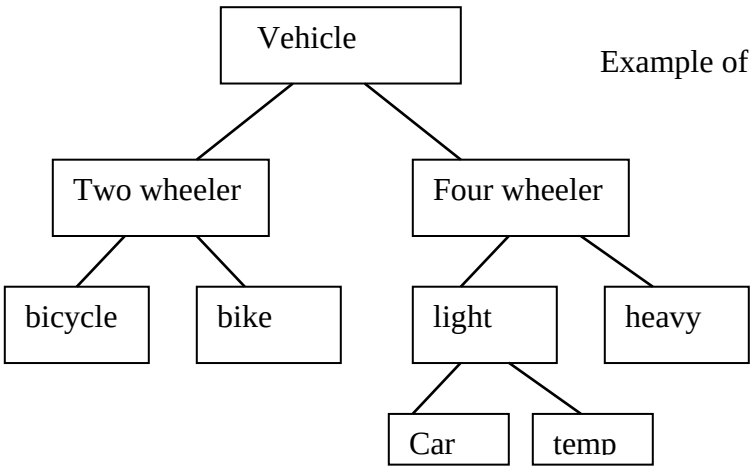
The class is a construct in C++ for creating user-defined data types called Abstract Data Types (ADT)

4. Inheritance:

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

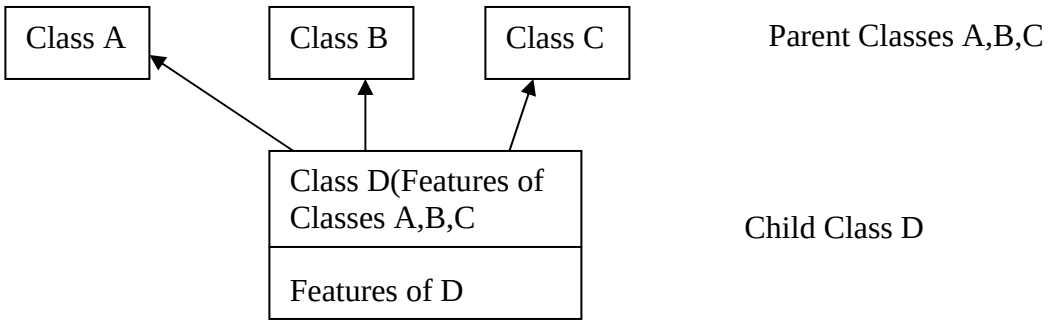
It supports the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the code.





Example of Inheritance class hierarchy

Multiple Inheritance: If derived class inherits the features of more than one base class it is called multiple inheritance.

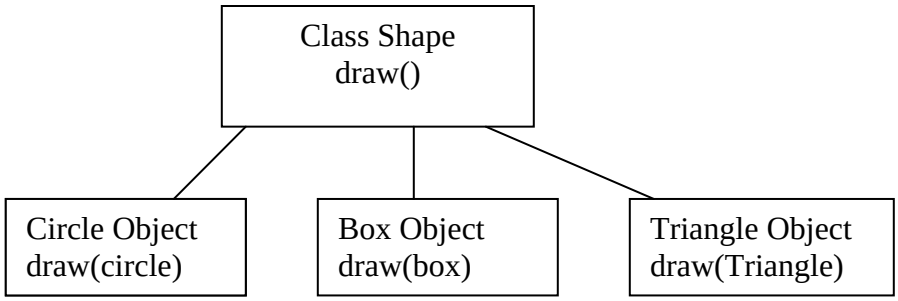


Multiple Inheritance

5. **Polymorphism:**

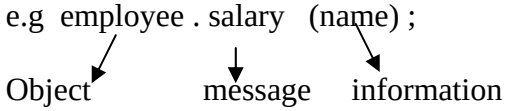
Polymorphism means "having many forms". The polymorphism allows different objects to respond to the same message in different ways, the response specific to the type of object. Polymorphism is important when object oriented programs dynamically creating and destroying the objects in runtime. Example of polymorphism in OOP is operator overloading, function overloading.

For example operator symbol '+' is used for arithmetic operation between two numbers, however by overloading (means given additional job) it can be used over Complex Object like currency that has Rs and Paisa as its attributes, complex number that has real part and imaginary part as attributes. By overloading same operator '+' can be used for different purpose like concatenation of strings.



Dynamic Biding: Binding refers to the linking a function call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given function call is not known until the time of the call at run time. It is associated with polymorphism & inheritance.

6. **Message Passing**: An Object-Oriented program consists of set of objects that communicate with each other. Object communicates with each other by sending and receiving message (information). A message for an object is a request for execution of a procedure and therefore will invoke a function or procedure in receiving object that generate the desired result. Message passing involves specifying the name of the object name of the function (message) and the information to be sent.



Advantages of OOPs

Object oriented programming contributes greater programmer productivity, better quality of software and lesser maintenance cost. The main advantages are:

- Making the use of inheritance, redundant code is eliminated and the existing class is extended.
- Through data hiding, programmer can build secure programs that cannot be invaded by code in other pats of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- System can be easily upgraded from small to large systems.
- Software complexity can be easily managed.
- Message passing technique for communication between objects makes the interface description with external system much simpler.
- Aids trapping in an existing pattern of human thought into programming.
- Code reusability is much easier than conventional programming languages.

Disadvantages of OOPs

- Compiler and runtime overhead. Object oriented program required greater processing overhead – demands more resources.
- An object's natural environment is in RAM as a dynamic entity but traditional data storage in files or databases
- Re-orientation of software developer to object-oriented thinking.
- Requires the mastery in software engineering and programming methodology.
- Benefits only in long run while managing large software projects.
- The message passing between many objects in a complex application can be difficult to trace & debug.

Object oriented Languages:

The language should support several of OOPs concepts to claim that they are Object-Oriented. Depending upon the features they support, they can be classified as

1. Object–Based Languages
2. Object-Oriented Languages

Object Based Language: Supports encapsulation & object identity without supporting the important features of OOP languages such as Inheritance and Dynamic Bindings. Major features are

- Data Encapsulation
- Data hiding & access mechanisms
- Automatic initialization & clear-up of objects
- Operator overloading
- Don't support inheritance & Dynamic binding . e.g Ada.

Object-Based Language : Encapsulation + Object Identity
Object – Oriented language : Incorporates all features of object-based language plus inheritance and dynamic bindings

Object-Oriented L : Object based feature + inheritance + dynamic bindings
e.g SMALLTALK , C++ , JAVA,EIFFEL etc.

Application of OOP

Applications of OOP are beginning to gain importance in many areas.

- The most popular application of OOP, up to now. Has been area of user interface design such as Windows .
- Real Business systems are often much more complex attributes & behaviors .
- OOP can simplify such complex problem. The areas of application of OOP include
- Real time systems
- Simulation & modeling
- Object-Oriented databases
- Hypertext, hypermedia
- AI & expert system
- Neural Networks & parallel programming
- Decision support & Office automation system
- CAM/CAD systems .
- Computer Based training and Educational Systems

Object-Oriented Programming Languages :

SmallTalk :

- Developed by Alen Kay at Xerox Palo Alto Research center (PARC) in 1970's
- 100% OO Language
- The syntax is very unusual and this leads to learning difficulties for programmers who are used to conventional language syntax .
- Language of this type allocate memory space for object on the heap and dynamic garbage collector required .

Eiffel :

- Eiffel was designed by a Frenchman named Bertrand Meyer in the late 1980's .
- Syntax is very elegant & simple, fewer reserved world than Pascal .
- Compiler normally generates "C source which is than compiled using c compiler which can lead long compile time.
- All Eiffel object are created on the heap storage
- Pure object oriented
- Not very popular as a language for mainstream application development .

Java

- Designed by SUN(Stanford University Net) Microsystems, released in 1996 and is a pure O-O language.
- Syntax is taken from C++ but many differences .
- All object are represented by references and there is are pointer type.
- The compiler generates platform independent byte code which is executed at run time by interpreter.
- A very large library of classes for creating event driven GUI is included with JAVA compiler
- JAVA is a logical successor to C++ can also be called as C++--++(C-Plus-Plus-Minus-Minus-Plus-Plus i.e. remove some difficult to use features of C++ and Add some good features)

C++

- Developed by Bjarne Stroustrup at Bell Lab in New jersey in early 1980 s as extension of C
- Employs the basic syntax of the earlier C language which was developed in Bell Lab by Kernigan & Ritchie.
- One of most popular language used for s/w development .
- By default, c++ creates objects on the systems stack in the same way as the fundamental data type.
-

Unlike Java, SmallTalk and Eiffel, C++ is not pure O-O language. i.e it can be written in a conventional 'C'.

2. Migrating from C to C++

History of C++

C++ is an object oriented programming language. It was called “C with class”. C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both languages and create a more powerful language that could support object oriented programming features and still remain power of C. The result was C++. The major addition was class to original C language, so it was also called “C with classes”. Then in 1993, the name was changed to C++. The increment operator ++ to C suggest an enhancement in C language.

C++ can be regarded as superset of C (or C is subset of C++). Almost all C programs are also true in C++ programs.

C++ program Construction.

Before looking at how to write C++ programs consider the following simple example program.

// Sample program
// Reads values for the length and width of a rectangle
// and returns the perimeter and area of the rectangle.

```
#include <iostream.h> //for cout and cin
#include <conio.h> //for getch()
void main()
{
  int length, width;
  int perimeter, area;              // declarations
  cout <<  "Length = ";             // prompt user
  cin >> length;                    // enter length
  cout << "Width = ";               // prompt user
  cin >> width;                     // input width
  perimeter = 2*(length + width);     // compute perimeter
  area = length*width;              // compute area
  cout << endl
       << "Perimeter is " << perimeter;
  cout << endl
       << "Area is " << area
       << endl;                    // output results
  getch();
} // end of main program
```

The following points should be noted in the above program:

1. **Single line comment** (// ……)

   Any text from the symbols // until the end of the line is ignored by the compiler. This facility allows the programmer to insert **Comments** in the program. Any program that is not very simple should also have further comments

indicating the major steps carried out and explaining any particularly complex piece of programming. This is essential if the program has to be extended or corrected at a later date. This is a kind of documentation. Also C comment type /*----------*/ is also a valid comment type in C++.

2. The line

   #include <iostream.h>

   must start in column one. It causes the compiler to include the text of the named file (in this case iostream.h) in the program at this point. The file iostream.h is a system supplied file which has definitions in it which are required if the program is going to use stream input or output. All programs will include this file. This statement is a **preprocessor directive** -- that is it gives information to the compiler but does not cause any executable code to be produced.
3. The actual program consists of the **function** main() which commences at the line
   void main( ).
   All programs must have a function main( ). Note that the opening brace ({) marks the beginning of the body of the function, while the closing brace (}) indicates the end of the body of the function. The word void indicates that main() does not return a value. Running the program consists of obeying the statements in the body of the function main().
4. The body of the function main contains the actual code which is executed by the computer and is enclosed, as noted above, in braces {}.
5. Every statement which instructs the computer to do something is terminated by a semi-colon. Symbols such as main(), { } etc. are not instructions to do something and hence are not followed by a semi-colon. Preprocessor directives are instruction to the compiler itself but program statements are instruction to the computer.
6. Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as
   cout << "Length = "
   send the quoted characters to the output stream cout. The special identifier endl when sent to an output stream will cause a newline to be taken on output.
7. All variables that are used in a program must be declared and given a type. In this case all the variables are of type int, i.e. whole numbers. Thus the statement
   int length, width;
   declares to the compiler that integer variables length and width are going to be used by the program. The compiler reserves space in memory for these variables.

Variable Definition at the point of use.

In C, local variables can only be defined at the top of a function, or at the beginning of a nested block. In C++, local variables can be created at any position in the code, even between statements. Also local variables can be defined in some statements, just before their usage.

```
//to find sum of first natural number to display for loop
#include<iostream.h>
#include<conio.h>
void main()
{
```

```
        int n;
        cout<<"\nEnter Number";
        cin>>n;
        int sum=0;

        for( int i=0;i<=n;i++)
               sum+=i;
        cout<<"Sum of first n natural number:"<<sum;
        getch();
}
```

8. Values can be given to variables by the **assignment** statement, e.g. the statement
   area = length*width;
   evaluates the expression on the right-hand side of the equals sign using the current values of length and width and assigns the resulting value to the variable area.
9. Layout of the program is quite arbitrary, i.e. new lines, spaces etc. can be inserted wherever desired and will be ignored by the compiler. The prime aim of additional spaces, new lines, etc. is to make the program more readable. However superfluous spaces or new lines must not be inserted in words like main, cout, in variable names or in strings.

Input and Output:

C++ Supports rich set of functions for performing input and output operations. The syntax using these I/O functions is totally consistent of the device with I/O operations are performed. C++'s new features for handling I/O operations are called streams. Streams are abstractions that refer to data flow. Streams in C++ are :

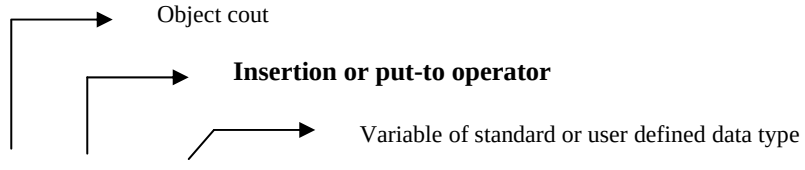
- Output Stream
- Input Stream.

Output Stream:

The output stream allows us to write operations on output devices such as screen, disk etc. Output on the standard stream is performed using the cout object. C++ uses the bit-wise-left-shift operator for performing console output operation. The syntax for the standard output stream operation is as follows:

cout<<variable;

The word cout is followed by the symbol << , called the insertion or put to operator , and then with the items (variables, constants, expressions) that are to be output. Variables can e of any basic data types. The us of *cout* to perform an output operation is as shown :



cout << variable;

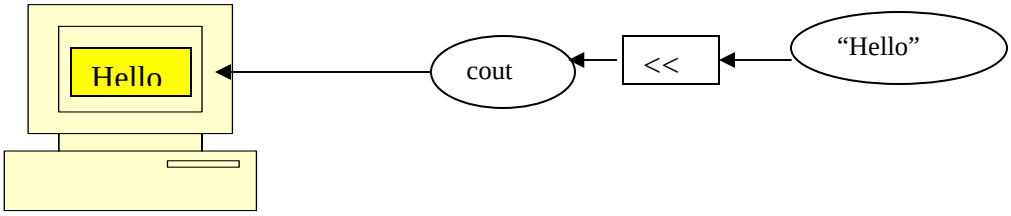


Figure: output with cout

The following are example of stream output operations:

1. cout << "hello world";
2. int age;
   cout<< age;
3. float weight;
   cout<< weight;

etc.

More than one item can be displayed using a single cout output stream object. Such out put operations in C++ are called cascaded output operations. For example

cout<<"Age is: "<<age<<"years";

This cout object will display all the items from left to right. If value of age is 30 then this stream prints

Age is : 30 years

C++ does not restricts the maximum number of items to output. The complete syntax of the standard output streams operation is as follows:

cout<<variable1<<vaariable2<<……<<variableN;

The object cout must be associated with at least one argument. Like printf in C, A constant value can be sent as an argument to the cout object.

e.g.

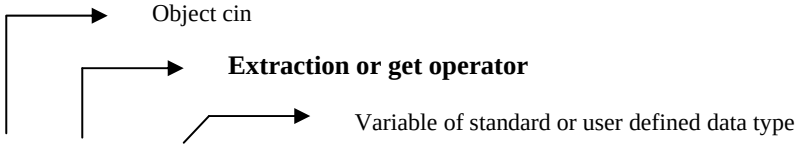
cout<<'A'; //prints a constant character A
cout<<10.99; //Prints constant 10.99
cout<<" "; //prints blanks
cout<<"\n", //prints new line

Input Streams:

The input stream allows us to perform read operations with input devices such as keyboard, disk etc. Input from the standard stream is performed using the cin object. C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for the standard output stream operation is as follows:

cin<<variable;

The word cin is followed by the symbol >> and then with variable, into which input data is to be stored. The use of *cin* to perform an input operation is as shown :



cin>> variable;

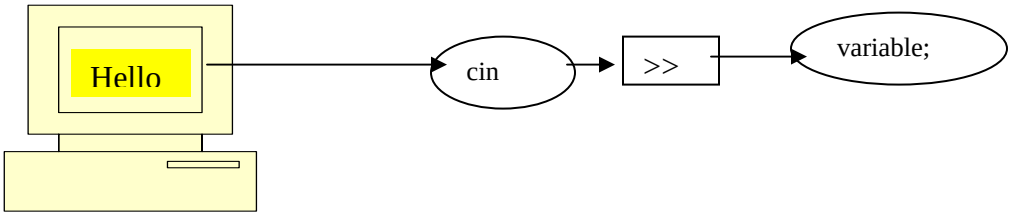


Figure: Input with cin

Following Examples show the stream input operations;

1. int amount;
   cin>>amount;
2. float weight;
   cin>>weight;
3. char name[20];
   cin>>name; etc

Input of more than one item can also be performed using the cin input stream object. Such input operation in C++ are called cascaded input operations. For example, reading the name of a person his address, age can be performed by the cin as:

cin>> name>>address>>age;

The cin object reads the items from left to right. The complete syntax of the standard input streams operations is as follows:

cin>>var1>>var2>>var3>>……>>varN;

e.g. cin>>i>>j>>k>>l;

The following are two important points to be noted about the stream operations.

- Streams do not require explicit data type specification in I/O statement.
- Streams do not require explicit address operator prior to the variable in the input statement.

In C printf and scanf functions, format strings (%d,%s,%c etc) and address operator (&) are necessary but in cin stream format specification is not necessary and in the cout stream format specification is optional. **Format-free I/O is special features of C++ which make I/O operation comfortable.**

Note: The operator << and >> are the bit-wise left-shift and right-shift operators that are used in C and C++. In C++ the operator can be overloaded i.e. same operator can perform different activities depending on the context.

Manipulators:

Manipulators are instructions to the output stream that modify the output in various ways. For example endl , setw etc.

The endl Manipulator:

The endl manipulator causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line. It has same effect as sending the '\n' character but somewhat different. It is a manipulator that sends a newline to the stream

and *flushes* the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output). Unlike '\n' it also causes the output buffer to be flushed but this happens invisibly.
e.g.

```
cout << endl<< "Perimeter is " << perimeter;
 cout << endl << "Area is " << area << endl;
```

The setw Manupulator:

To use setw manipulator the "iomanip.h" header file must be included. The setw manipulator causes the number or string that follows it in the stream to be printed within a field n characters wide , where n is the argument used with setw as stew(n). The value is right justified within the field.
e.g.

```
//demonstrates setw manipulator
#include<iostream.h>
#include<iomanip.h>
void	main()
{
	 long pop1=5425678,pop2=47000;pop3=76890;
	  cout<<setw(8)<<"LOCATION"<<setw(12)<< "POPULATION"<<endl
	<<setw(8)<< "Patan"<<setw(12)<<pop1<<endl
	<<setw(8)<< "Khotang" <<setw(12)<<pop2<<endl
	<<setw(8)<< "Butwal" <<setw(12) <<pop3<<endl;
}
```

The output of this program is:

```
LOCATION    POPULATION
  Patan              5425678
 Khotang              47000
  Butwal              76890
```

Manipulators come in two flavors: those that take and arguments and those that don't take arguments. Followings are the some important non-argument manipulators .
Table: No-argument Manipulators

Manipulators	Purpose
Ws	Turn on whitespace skipping on input
Dec	convert to decimal
Oct	Convert to octal
Hex	convert to Hexadecimal
Endl	insert newling and flush the output stream
Ends	Insert null character to terminate an output string
Flush	flush the output stream
Lock	lock the file handle
Unlock	Unlock the file handle

```
//demonstrates  manipulators
#include<iostream.h>
#include<iomanip.h>
void main()
{
  long pop1=2425678;
  cout<<setw(8)<<"LOCATION"<<setw(12)<< "POPULATION"<<endl
          <<setw(8)<< "Patan"<<setw(12)<<hex<<pop1<<endl
          <<setw(8)<< "Khotang" <<setw(12)<<oct<<pop1<<endl
          <<setw(8)<< "Butwal" <<setw(12) <<dec<<pop1<<endl;
}
```

Output:

```
LOCATION  POPULATION
   Patan      25034e
 Khotang    11201516
  Butwal     2425678
```

There are manipulators that take arguments for example setw(), setfill() setprecision() etc e.g.

```
cout<<setw(12)<<setfill(64)<<"string";
```

In this statement setw(12) describes the field width 12 and setfill(64) fills the blanks with character specified with integer arguments. The output of this statement is :

@@@@@@string where @ is the character corresponding to int 64

```
cout<<setprecision(5)<<12.456789;
```

The output of this statement is : 12.456 i.e. setprecision() describes the no of digits to be displayed.

Data Types:

Fundamental Data Types

Data type defines the amount of memory that will be used by a variable and the valid range of values that a variable can represent. Both C and C++ compilers support all the fundamental (also known as basic or built-in) data types. C++ defines five fundamental data types: *int*, *char*, *float*, *double*, and *void*. The modifiers *signed*, *unsigned*, *long*, and *short* may be applied to character and integer basic types except *void*. However, the modifier *long* may also be applied to *double*. The following table shows the memory required for all combinations of the basic data types and modifiers and the valid range of values it can represent.

Data Type	Memory Required	Range of Values
char	1 byte (8 bits)	- 128 to 127 (- 2^7 to 2^7 – 1)
unsigned char	1 byte (8 bits)	0 to 255
signed char	1 byte (8 bits)	- 128 to 127 (- 2^7 to 2^7 – 1)
int	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
unsigned int (unsigned)	2 bytes (16 bits)	0 to 65535
signed int (signed)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
short int (short)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
long int (long)	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
unsigned short int (unsigned short)	2 bytes (16 bits)	0 to 65535
signed short int (signed short)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
unsigned long int (unsigned long)	4 bytes (32 bits)	0 to 4,294,967,295
signed long int (signed long)	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
float	4 bytes (32 bits)	3.4×10^{-38} to 3.4×10^{38}
double	8 bytes (64 bits)	1.7×10^{-308} to 1.7×10^{308}
long double	10 bytes (80 bits)	3.4×10^{-4932} to 1.1×10^{4932}

The type *void* can be used to specify the return type of a function when it is not returning any value, to indicate an empty argument list to a function, and in the declaration of generic pointers. For example,

- To specify return type and empty argument list

```
void anyfunction(void);
```

- To declare generic pointers

A generic pointer can be assigned a pointer value of any basic data type as follows:

```
void *gp;
int *ip;
gp = ip;
```

We can also assign void pointer to other type pointers as follows:

```
ip = (int*)gp;
```

Note: Two more data types, **bool** and **wchar_t** has also added in ANSI C++.

User Defined Data Types

In C++, we can use the concepts of **struct**, **union** and **class** (discussed later) to define user-defined data types.

Another way is to use *enumerated data type*. An enumeration, introduced by the keyword **enum** and followed by a *type name*, is a set of integer constants represented by identifiers (also called *enumeration constants*). The values on these identifiers start at 0, unless specified, and incremented by 1. The identifiers must be unique, but can have the same integer value. Some examples are given below:

enum color {red, blue, green, yellow};
color background;
background = blue;

enum boolean {false, true};
boolean b = true;

enum gender {male, female};
gender g =male;

Some valid definitions are:
enum suit {clubs = 1, diamonds, hearts, spades};
enum color {red, blue = 4, green = 8};

Variables

A variable is the name used for the quantities which are manipulated by a computer program. i.e. it is a named storage location in memory. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.
In order to distinguish between different variables, they must be given **identifiers**, names which distinguish them from all other variables. The rules of C++ for valid identifiers state that:
An identifier must:

- start with a letter
- consist only of letters, the digits 0-9, or the underscore symbol _
- not be a **reserved word**

For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers.
The following are valid identifiers
Length days_in_year DataSet1 Profit95
Int _Pressure first_one first_1
although using _Pressure is not recommended.
The following are invalid:
days-in-year 1data int first.val throw
Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word main in a program is quite different from the word Main or the word MAIN.

Reserved words

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

C++ Reserved Words

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	const	const_cast	continue
default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

Some of these reserved words may not be treated as reserved by older compilers. However it is better to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the PC, which add near, far, huge, cdecl, and pascal.

Declaration of variables

In C++ (as in many other programming languages) all the variables that a program is going to use must be **declared** prior to use. Declaration of a variable serves two purposes:

- It associates a **type** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.
- It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

Expressions and Statements

An expression is any arrangement of operands and operators that specifies a computation. For example, a + 13, a++, and (a – 5) * b / 2 are expressions.

Expressions and statements are not same. Statements tell the compiler to do something and they must terminate with a semicolon (also known as the statement terminator), while expressions specify a computation. There can be several expressions in a statement.

The const Qualifier

The keyword **const** (for constant) precedes the data type and specifies that the value will not change throughout the program. Any attempt to alter the value will give error message from the compiler. C++ requires a **const** to be initialized. For example,

const int size = 10;

A **const** in C++ is local to the file where it is declared. To make it visible to other files, we must explicitely define it as an **extern**. For example,

extern const int size = 10;

Type Conversion

There are two types of type conversion: *automatic conversion* and *type casting*.

- **Automatic Conversion:** When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is also called type promotion. The order of types is given below:

Data Type	**Order**
long double	(highest)
double	
float	
long	
int	
char	(lowest)

- **Type Casting:** Sometimes, a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically. For this C++ permits explicit type conversion of variables or expressions as follows:

  (type-name) expression //C notation
  type-name (expression) //C++ notation

  For example,
  int a = 10000;
  *int b = long(a) * 5 / 2; //correct*
  *int b = a * 5/2; //incorrect* (can you think how?)

Scope Resolution Operator(::)

C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the scope resolution operator.

:: GloabalVariableName

The global variable to be accessed must be preceded by the scope resolution operator. It directs the compiler to access a global variable, instead of one defined as a local variable. The scope resolution operator permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

```
//An example of use of scoperesolution operator ::
#include<iostream.h>
#include<conio.h>
int x=5;
void main()
{
      int x=15;
      cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
      {
            int x=25;
            cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
      }
      cout<<"Local data x="<<x<<"Global data x="<<::x<<endl;
      cout<<”Global +Local=”<<::x +x;

   getch();
}
```

Reference Variables:

C++ introduces a new kind of variable known as reference variable. A reference variable provides an alias (Alternative name) of the variable that is previously defined. For example, if we make the variable **sum** a reference to the variable **total**, the **sum** and **total** can be used interchangeably to represent that variable.

Syntax for defining reference variable

Data_type & reference_name = variable_nane

Example:

```
   int total=100 ;
   int &sum=total;
```

Here total is int variable already declared. Sum is the alias for variable total. Both the variable refer to the same data 100 in the memory.

```
cout<<total;  and cout<<sum; gives the same output 100.
 And total= total+100;
Cout<<sum;    //gives output 200
```

A `reference` variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it means. The initialization of reference variable is completely different from assignment to it.

A major application of the reference variables is in passing arguments to function.

```
//An example of reference
#include<iostream.h>
#include<conio.h>

void main()
{
      int x=5;
      int &y=x;
      //y is alias of x
      cout<<"x="<<x<<"and y="<<y<<endl;
      y++;
      //y is reference of x;
      cout<<"x="<<x<<"and y="<<y<<endl;
      getch();
}
```

Passing by reference

We can pass parameters in function in C++ by reference .When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function i.e. when function is working with its own arguments , it is actually working on the original data.
Example

```
void fun(int &a)
//a is reference variable
{
     a=a+10;
}
 int main()
 {
     int x=100;
     fun(x);  //CALL
     cout<<x;  //prints 110
}
```

when function call fun(x) is executed, the following initialization occurs:
`int &a=x;` i.e. a is an alias for x and represent the same data in memory. So updating a in function causes the update the data represented by x. this type of function call is known as *Call by rererence*.

```
//pass by reference
#include<iostream.h>
#include<conio.h>
void swap(int &, int &);
void main()
{
     int a=5,b=9;
     cout<<"Before Swapping: a="<<a<<" and b="<<b<<endl;
     swap(a,b);//call by reference
```

```
      cout<<"After Swapping: a="<<a<<" and b="<<b<<endl;
      getch();
}

void swap(int &x, int &y)
{
      int temp;
      temp=x;
      x=y;
      y=temp;
}
```

Return by reference

A function can return a value by reference. This is a unique feature of C++. Normally function is invoked only on the right hand side of the equal sign. But we can use it on the left side of equal sign and the value returned is put on the right side.

```
//returning by reference from a function as a parameter
#include<iostream.h>
#include<conio.h>
int x=5,y=15;//globel variable
int &setx();
void main()
{
      setx()=y;
   //assign value of y to the variable
   //returned by the function
   cout<<"x="<<x<<endl;
   getch();
}
int &setx()
{
   //display global value of x
      cout<<"x="<<x<<endl;
      return x;
}
```

Inline Function:

A inline function is a short-code function written and placed before main function and compiled as inline code. The prototyping is not required for inline function. It starts with keyword **inline** . In ordinary functions, when function is invoked the control is passed to the calling function and after executing the function the control is passed back to the calling program.

But , when inline function is called, the inline code of the function is inserted at the place of call and compiled with the other source code together. That is the main feature of inline function and different from the ordinary function. So using inline function executing time is reduced because there is no transfer and return back to control. But if function has long code inline function is not suitable because it increases the length of source code due to inline compilation.

```
// Inline Function
//saves memory, the call to function cause the same code to be
//executed;the function need not be duplicated in memory
#include<iostream.h>
#include<conio.h>
inline float lbstokg(float pound)
{
     return (0.453592*pound);
}

void main()
{
     float lbs1=50,lbs2=100;
     cout<<"Weinght in Kg:"<<lbstokg(lbs1)<<endl;
     cout<<"Weinght in Kg:"<<lbstokg(lbs2)<<endl;
     getch();
}
```

Default Arguments

In C++ a function can be called without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. The default value are specified when function is declared.

The default value is specified similar to the variable initialization . The prototype for the declaration of default value of an argument looks like

```
float  amount(float p, int  time, float rate=0.10);
```

// declares a default value of 0.10 to the argument rate.

The call of this function as

```
value = amount(4000,5);// one argument missing for rate
```

passes the value 4000 to p , 5 to time and the function looks the prototype for missing argument that is declared as default value 0.10 the the function uses the default value 0.10 for the third argument. But the call

```
 value = amount(4000,5,0.15);
```

no argument is missing , in this case function uses this value 0.15 for rate.

Note : only the trailing arguments can have default value. We must add default from right to left. E.g.

```
int add( int  a,  int b =9, int  c= 10 );  // legal
int add(int a=8, int b, int c); // illegal
int add(int a, int b = 9, int c);  //illegal
int add( int a=8, int b=9,int c=10) // legal

//default arguments in function
//define default values for arguments that are not passed when
//a function call is made
```

```
#include<iostream.h>
#include<conio.h>
void marks_tot(int m1=40,int m2=40, int m3=40 );
void main()
{
   //imagine 40 is given if absent in exam.
     marks_tot();
   marks_tot(55);
   marks_tot(66,77);
   marks_tot(75,85,92);
   getch();
}

void marks_tot(int m1, int m2, int m3)
{
     cout<<"Total marks"<<(m1+m2+m3)<<endl;
}
```

Const Arguments

When arguments are passed by reference to the function, the function can modify the variables in the calling program. Using the constant arguments in the function , the variables in the calling program can not be modified. const qualifier is used for it.

e.g.

```
void func(int&,const int&);
 void  main()
{
     int a=10, b=20;
     func(a,b);
}
void func(int& x, int &y)
{
     x=100;
     y=200; // error  since y is constant argument
}
```

Function overloading

Overloading refers to the use of same thing for different purpose. When same function name is used for different tasks this is known as function overloading. Function overloading is one of the important feature of C++ and any other OO languages.

When an overloaded function is called the function with matching arguments and return type is invoked.
e.g.
void border(); //function with no arguments
void border(int); // function with one int argument
void border(float); //function with one float argument
void border(int, float);// function with one int and one float arguments

For overloading a function prototype for each and the definition for each function that share same name is compulsory.

```
//function overloading
//multiple function with same name
#include<iostream.h>
#include<conio.h>
int max(int ,int);
long max(long, long);
float max(float,float);
char max(char,char);
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}

int max(int i1, int i2)
{
     return(i1>i2?i1:i2);
}

long max(long l1, long l2)
{
     return(l1>l2?l1:l2);
}

float max(float f1, float f2)
{
     return(f1>f2?f1:f2);
}

char max(char c1, char c2)
{
     return(c1>c2?c1:c2);
}
```

Structure review

An structure is an user defined data type which contains the collection of different types of data under the same name. A structure is compared with records in other languages.

Syntax:

```
struct  tag-name
{
        data-type var-name;
        data-type var-name;
        ……..
} ; //end of structure
```

e.g. struct currency

```
{
        int rs;
        float ps;
};
```

declaring variable of structure

tag-name st-var-name; e.g. currency c1,c2;

Initialization of structure variable: structure variable can be initialized at the time of declaraction as

currency c1 ={ 144,56.7};

However each member data of structure variable can be initialized separately after declaration as

currency c1;
c1.rs=144;
c1.ps=56.7;

But initialization all member data at once listing is illegal after declaration as

currency c1;
c1={144,56.7} // error:

Computation using structure

A simple Example.

```
#include<iostream.h>
 struct currency
{
      int rupees;
      float paise;
};          // currency is name for struct currency

void main()
{
      currency c1,c3;
      currency c2 ={123,56.4};
```

```
      cout<<"Enter Rupees:"; cin>> c1.rupees;
      cout<<"Enter paises"; cin>> c1. paise;
      c3.paise = c1.paise+ c2.paise;
      if(c3.paise>=100.0)
      {
       c3.paise-=100.0 ;
       c3.rupees++;
      }
      c3.rupees+=c2.rupees+c1.rupees;
      cout<<"Rs." <<c1.rupees<<" Ps. " <<c1.paise<<" + ";
      cout<<"Rs." <<c2.rupees<<" Ps. "<<c2.paise<<" = ";
      cout<< "Rs."<<c3.rupees<<" Ps."<<c3.paise<<endl;
}
```

Passing structure as function arguments

A function can receive the structure as parameter and is able to access and operate on the individual elements of the structure. When passing the structure variable as the function argument the whole variable is not passed but only the reference (address) of the structure variable is passed. The following example shows the passing structure as function argument.

```
//passing structure as function argument
#include<iostream.h>
#include<conio.h>
struct currency
{
      int rs;
      float ps;
};

currency addcurr( currency,currency );
void main()
{
      currency c3;
      currency c1={100,58.5};
      currency c2={200,62.8};
      c3 = addcurr(c1,c2);
      cout<<"Sum is Rs."<<c3.rs<<"Ps."<<c3.ps<<endl;
   getch();
}
currency addcurr(currency cc1, currency cc2)
{
      currency cc3={0,0.0};
      cc3.ps=cc2.ps+cc1.ps;
      if(cc3.ps>=100.0)
      {
            cc3.ps-=100.0;
            cc3.rs++;
      }
      cc3.rs+=cc1.rs+cc2.rs;
 return cc3;
 }
```

3. C++ Operators and Control Structures

The following table shows decreasing order of precedence from top to bottom.

Operator	**Type**	**Associativity**
:: ::	binary scope resolution unary scope resolution	left to right
() [] . –> ++ --	parentheses array subscript member selection via object member selection via pointer unary postincremet unary postdecrement	left to right
++ -- + - ! ~ (type) sizeof & * new new[] delete delete[]	unary preincrement unary predecrement unary plus unary minus unary logical negation unary bitwise complement unary cast determine size in bytes address dereference dynamic memory allocation dynamic array allocation dynamic memory deallocation dynamic array deallocation	right to left
.* –>*	pointer to member via object pointer to member via pointer	left to right
* / %	multiplication division modulus	left to right
+ -	addition subtraction	left to right
<< >>	bitwise shift left bitwise shift right	left to right
< <= > >=	relational less than relational less than or equal to relational greater than relational greater than or equal to	left to right
== !=	relational is equal to relational is not equal to	left to right
&	bitwise AND	left to right
∧	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right

?:	ternary conditional	right to left
= += –= *= /= %= &= ^= |= <<= >>=	assignment addition assignment subtraction assignment multiplication assignment division assignment modulus assignment bitwise AND assignment bitwise exclusive OR assignment bitwise inclusive OR assignment bitwise left-shift assignment bitwise right-shift assignment	right to left
,	Comma	left to right

Control Structures

Like C, C++ also supports the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching or decision)
3. Loop structure (iteration or repetition)

1. Sequence structure: In this structure, the sequence statements are executed one after another from top to bottom.

statement 1;
statement 2;
statement 3;
……..
statement n;

In this case *statement 1* is executed before *statement 2*, *statement 2* is executed before *statement 3*, and so on.

2. Selection structure: This structure makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression. The two structures of this type are: **if** and **switch**.

A. The *if* statement: There are three forms of *if* statement.

a. Simple *if*: The syntax is:

```
if(expression) {
    statement (s);
}
```

This statement lets your program to execute *statement(s)* if the value of the *expression* is true (non-zero). If there is only one statement, it is not necessary to use braces.

Example piece of code:

```
int x = 5,  y = 10;
if(x<10)
   cout<<“X is less than ten”;
if(y > 10)
```

```
   cout<<“wow y is greater than ten”;
```

Output:
Here since only the condition at the first if statement is true i.e. x<10, the output is
X is less than ten

b. The *if-else* statement: The syntax is:

```
if (expression) {
    statement(s);
} else {
    statement(s);
}
```

If the value of the *expression* is true, the program executes the *statement(s)* following *if* otherwise the *statement(s)* following *else*. If there is only one statement, it is not necessary to use braces.

Example piece of code:

```
int x = 3;
if(x>2)
   cout<<“condition true”;
else
   cout<<“condition false”;
```

Output:
Since x = 3, x>2 is true so the output is - condition true. If we had x = 1, then x>2 would have been false giving us an output as – condition false.

c. The *if-else-if* ladder: The syntax is:

```
if (expression) {
    Statement(s);
} else if (expression) {
    Statement(s);
} else if (expression) {
    statement(s);
}
…
} else {
    Statement(s);
}
```

In this case, the program goes down until one of the expressions is true. It then executes the following *statement(s)* and exits. If none of the expressions are true, the program executes the *statement(s)* following *else*. If there is only one statement, it is not necessary to use braces.

Example piece of code:

```
int age // this is taken as input
if( age < 18 )
  cout<< "You are a child";
else if( age < 55 )
  cout<< "You are an adult";
else
```

```
    cout<< "You are a senior";
```

Output:

Determine the output if age = 24. (Is it, You are an adult??)

B. The *switch* statement: The syntax form is:

```
switch (expression) {
        case value 1:
                statement(s);
                break;
        case value 2:
                statement(s)
                break;
        ……
        case value n:
                statement(s);
                break;
        default:
                statement(s);
}
```

In this case, the value of the *expression* is compared with each of the constant values in the *case* statements. If a match is found, the program executes the *statement(s)* following the matched *case* statement. If none of the constants matches the value of the *expression*, then the program executes the *statement(s)* following *default* statement. However, the default statement is optional.

The value of the *expression* must be of type *integer* or *character* and each of the values specified in the case statement must be of a type compatible with the *expression*. Each *case value* must be a unique literal. Duplicate *case values* are not allowed. The *break* statement is used to terminate the entire *switch* statement.

Consider the following example:

```
switch( month ) {
        case 4:
        case 6:
        case 9:
        case 11:
                 cout<< "30 days";
                break;
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
                cout<< "31 days";
                break;
        case 2:
                 cout<< "if leap year 29 days else 28 days";
```

```
            break;
        default:
            cout<< "not a number for months must be 1-12";
    }
```

If `month` is 4, 6, 9, or 11, then “30 days” is printed. If `month` is 1, 3, 5, 7, 8, 10, or 12, then “31 days” is printed. If month is 2, then “if leap year 29 days else 28 days” is printed. Lastly, if the month value is not between 1 and 12 no case is matched, so the default statement “not a number for months must be 1-12” is printed.

3. **Loop Structure:** These structures repeatedly execute a section of your program a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statement following the loop. C++ provides three kinds of loops: the **for** loop, the **while** loop, and the **do** loop.

   A. The *for* loop: The general form is:

```
for(initialization; test expression; iteration) {
        Statement(s);
}
```

   First before loop starts, the *initialization* statement is executed that initializes the loop control variable or variables in the loop. Second the *test expression* is evaluated, if it is true, the program executes the *statement(s)*. Finally, the *iteration* (usually an expression that increments or decrements the loop control variables) statement will be executed, and the *test expression* will be evaluated, this continues until the *test expression* is false.

   The curly braces are unnecessary if only a single statement is being repeated. When we have the fixed number of the iteration known then we usually (although not always) use for loop.

   Example piece of code:

```
for( int i=10; i > 1; i-- )
      cout<<i;
```

   Output:

   10 9 8 7 6 5 4 3 2

   B. The *while* loop: The general form is:

```
while(test expression) {
        statement(s);
}
```

   Here the program executes the *statement(s)* as long as the *test expression* is true. When the *test expression* becomes false, the *while* loop stops executing the *statement(s)*.

   The curly braces are unnecessary if only a single statement is being repeated. This loop structure is usually used if we don’t know the number of iterations before the loop starts.

   Example piece of code:

```
int i=1;
while( i <= 10 ) {
    cout<< "i="<< + i;
```

```
    i++;
}
```

Output:
The above code prints the number 1 to 10 as i = 1, i = 2, …, i = 10, when i reaches 11 the condition i<=10 becomes false and the loop terminates.

Remember: Do not forget to increment i, otherwise loop will never terminate i.e. code goes into infinite loop. So when using loop, remember to guarantee its termination.

C. The *do-while* loop: The general form is:

```
do{
        statement(s);
} while(test expression);
```

In this case, the program executes the *statement(s)* first and then evaluates the *test expression*. If this expression is true, the loop will repeat. Otherwise, the loop terminates.
The curly braces are unnecessary if only a single statement is being repeated. This loop always executes *statement(s)* at least once, because its *test expression* is at the bottom of the loop.

Example piece of code:

```
int i = 10;
do{
  cout<< "Hello";
   i++;
} while( i < 10 );
```

Output:
"Hello" is printed, though i<10 is false.

Jump Statements

For the transfer of control from one part of the program to another, C++ supports three jump statements: **break**, **continue**, and **return**.

A. The *break* Statement: The use of break statement causes the immediate termination of the *switch* statement and the loop (all type of loops) from the point of break statement. The control then passes to the statements following the switch statement and the loop.

Example piece of code:

```
for( int i = 1;  i <= 10;  i++ ) {
    if( i == 5 )
            break;
    cout<< " i = "<< i;
    }
}
```

Output:
i = 1 i = 2 i = 3 i = 4
In the code above the loop is terminated immediately after the value of i is 5.

Remember: In case of nested loop if break statement is in inner loop only the inner loop is terminated.

B. The *continue* Statement: Continue statement causes the execution of the current iteration of the loop to stop, and then continue at the next iteration of the loop.

Example:

```
for( int i=1; i<= 10; i++ ) {
        if( i == 5 || i == 7 ) {
                continue;
        }
        cout<<i<<" ";
}
```

Output:

1 2 3 4 6 8 9 10

C. The *return* Statement: It is used to transfer the program control back to the caller of the method. There are two forms of return statement. First with general form ***return expression;*** returns the value whereas the second with the form ***return;*** returns no value but only the control to the caller.

4. CLASSES AND OBJECTS

Structure in C and Structures in C++

Structures in C

- One of the unique features of C language
- Provides method for packing together the data of different types
- Convenient tool for handling a group of logically related data items.

Consider:

```
struct  complex
{
   int real;
   int img;
};
struct complex c1, c2, c3;
```

The complex number c1, c2, c3 are assigned values using a dot or period operator.

But—

- We cannot add or subtract two structure variables in C.
- Also data hiding is not permitted in C

Structures in C++

- C++, Structure supports all features of Structures in C.
- We can easily add or subtract two structure variables.
- C++, Structure can have both data and function. Data are called data member while functions are called the member function.
- In C++, structure names are Stand-alone i.e. Keyword struct may be omitted in the declaration of structure variable.
  E.g.

```
 struct student
{
      char name[20];
      int roll;
};
student s1,s2;  //C++ declaration
                //Invalid in C
```

- In C++, a structure member can be declared as *"Private"* so that they cannot be accessed directly by the external function.

Example:

Simple program shows the use of structures with member data and function both.

```
#include<iostream.h>
#include<conio.h >
 struct  coordinate
{
      int x;     //x coordinate
      int y;     //y coordinate
      void read()
      {
            cout<<"x coordinate";   cin>>x;
            cout<<"y coordinate";  cin>>y;
      }
      void print()
      {
            cout<<"("<<x<<","<<y<<")";

      }

      void add(coordinate p1, coordinate p2)
      {
            x=p1.x+p2.x;
            y=p1.x+p2.y;
      }
};  //end of structure definition
void main()
{
      coordinate p1,p2,p3;
      cout<<"enter co-ordinates of first point"<<endl;
      p1.read();
      cout<<"enter co-ordinate of second points"<<endl;
      p2.read();
      p1.print();
      p2.print();
      p3.add(p1,p2);//p3=p1+p2
      p3.print();
      getch();
}
```

- In main(),p1.read();executes the member function read() defined in the structure co-ordinate
- The data values for p1 are assigned with input values.
- The statement p1.print( ) ; displays data member with message passed as function argument.
- p3.add(p1,p2) adds the contents of corresponding data members of p1 and p2 and assign the sum to p3
- Structure members are public by default
- The structures are extended in C++ and the new type *class* is defined for OOP.

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 form the external use.

Class Specification

Class Specification has the two parts:

1. Class declaration
2. Class function definition

Class declaration

Class declaration describes the type and scope of its member

The general form of class declaration

```
class  class-name
{
      private:
            variable declaration
            function declaration
      public:
            variable declaration
            function declaration
};
```

- Class declaration is similar to a structure declaration.
- The class body contains the declaration of variables and functions. These are collectively called the members.
- Members are usually grouped under two sections, private and public to denote which of the members are private and which of them are public.
- The keyword public and private are known as *visibility labels* and are followed by colon.
- The members that have been declared as private can be accessed only from within the class. Where as the members that have been declared as public can be accessed from the outside the class also.
- The data hiding using private keyword is the key feature of the OOP.
- The use of keyword private is optional, by default the member of a class are private. If both of levels are missing, such a class is completely hidden from the outside world and does not have any purpose.
- **In OOP, generally data are made private and functions are made public.**

Creating the Objects

The Object of class are defined as,

class-name object-name;

e.g.

```
class test
{
      -------
      -------
};
test t1, t2;   //two object of class test
```

How Class provides the data hiding

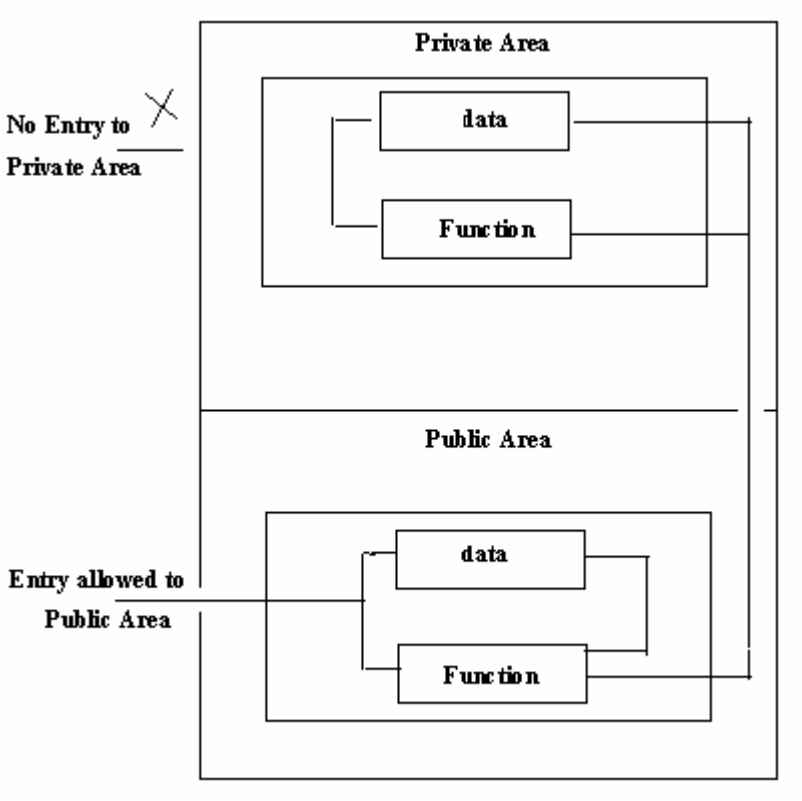


fig. data hiding in class

The data declared inside the class are called the data member and the functions are called the member function. Only the member function can have access to the private data members and private functions. The private members have no access to the external function i.e. private members are hidden from the external function so provide the data encapsulation.

The binding of data and function together into a single class-type variable is called encapsulation.

Example to show the data encapsulation:

```
class test
      {
            int x;
            public:
            int y;
      };

      void main()
      {
            test t;          //creating the object
            t.x=10;          //error, x is a private
            t.y=15;          //ok, y is public
      }
```

Difference between the Structure in C++ and class in C++

There is very little syntactical difference between the structures and class in C++, therefore they can be used interchangeably with minor modification.

The only difference between a Structure and Class in C++ is that, by default the members of a class are private while the default members of Structure are public.

Example:

```
//prog 1
#include <iostrem.h>
struct example
{
      float a;
};
int main()
{
      example ex;
      cout<<"Enter the floating no";
      cin>> ex.a;
      cout<<"the no is:"<<ex.a<<"\n";
      return 0;
}
```

```
//prog 2
      #include <iostream.h>
class example
{
      float a;
};
int main()
{
 example ex;
  cout<<"Enter the floating no";
  cin>>ex.a;
  cout<<"the no is:"   <<ex.a;
  return 0;
}
```

The first program prog1 is compiled and we get the required output because in structure all member are public and it can be used by function main. While in second program we get the error message like:"Float 'a' is not accessible in main ()". Because member in class are by default private and cannot be accessible in main function.

ACCESSING CLASS MEMBER: We can access the class member by using dot operator as,

Accessing data member

Syntax:

Object-name.Datamember

Accessing Member Functions

Objectname.function-name (actual arguments);

Example:

```
#include<iostream.h>
#include<string.h>
class student
{
    private:
          int roll_no;
          char name[20];

    public:
          void getdata();
          {
                cout <<"enter roll no:";
                cin>>roll_no;
                cout<<"enter name";
                cin>>name;
          }

          void showdata()
```

```
            {
                  cout<<"roll no"<<roll_no;
                  cout<<"Name:"<<name;
            }
};  //end class

void main()
{
    student s1,s2,s3;
    cout<<enter records for student1";
    s1.getdata();
    cout<<"enter record for student2";
    s2.getdata();
    cout<<"enter record for student3";
    s3.getdata();
    //display
    s1.showdata();
    s2.showdata();
    S3.showdata();
}
```

Defining member function:

- Member function definition describes how the class functions are implemented.
- Member functions can be defined in two places
  - outside the class definition
  - inside the class definition

1. **Inside the class definition**:

   The member function defines inside the class are considered as an inline automatically and no need of keyword inline.

   The example of member functions showdata(), getdata() etc in above examples are defined inside the class definition.

2. **Outside the class definition:**

   The member functions that are declared inside the class can be defined outside the class. The function definition outside a class definition consists of the function header with associated class label to represent the membership of that class and contains the body of the function.

Syntax of the function definition outside class definition is as follows:

```
return-type classname::function name(arguments..)
  {
        function Body;
  }
```

The :: operator tells the compiler that function is member of that class.

An example

```
//an example of function
```

```
//definition outside the class
#include<iostream.h>
#include<conio.h>
class student
{
   private:
            int roll;
            char name[20];
   public:
            void getdata();//function declaration
            void showdata();

};   // end of class

// definition of functions outside class
void student::getdata()
 {
      cout<<"\nEnter Roll No:";
      cin>>roll;
      cout<<"\nEnter Name:";
      cin>>name;
}
void student::showdata()
 {
            cout<<"name:"<<name<<endl;
            cout<<"roll no:"<<roll<<endl;
 }
void main()
{
      student s1,s2;
      s1.getdata();
      s2.getdata();
      cout<<"first student"<<endl;
      s1.showdata();
      cout<<"second student"<<endl;
      s2.showdata();
      getch();
}
```

Example of class to find the roots of a Quadratic equation: $ax^2 + bx + c = 0$

```
#include <iostream.h>
#include <math.h>
class quadratic
{
      float a, b, c;
      public:
      void set_data(float l, float m, float n)
      {
            a=l;
            b=m;
            c=n;
      }
      void equal_root()
      {
```

```
            float r;
            r= -b/(2*a);
            cout<<”The roots are equal”;
            cout<<”X=” <<r<<”\n”;
      }
      void real_root(float dis)
      {
            float r1,r2,temp;
            temp = sqrt(dis);
            r1 = (-b+ temp)/(2*a);
            r2 = (-b-temp)/(2*a);
            cout<<”Roots are real”;
            cout<<”Root1=”<<r1<<”\n”;
            cout<<”Root2=”<<r2<<”\n”;
      }
};
void main()
{
      quadratic eq;
      float aa,bb,cc;
      cout<<”Enter three floating point number:”;
      cin>>aa>>bb>>cc;
      eq.set_data(aa,bb,cc);
      if(aa==0 && bb!=0)
      {
            float temp;
            temp=cc/bb;
            cout<<”Roots are linear”<<endl;
            cout<<”X=”<<temp<<”\n”;
      }
      else
      {
            float disc;
            disc= bb*bb-4*aa*cc;
            if (disc = = 0)
                  eq.equal_root();
            else if (disc > 0)
                  eq.real_root(disc);
            else
                  cout<<”Roots are imaginary:”;
      }
}
```

Another Example of class by using string function

```
//string as class member
#include<iostream.h>
#include<conio.h>
#include<string.h>

class student
```

```
{
      private:
            char name[25];
            int age;
            float weight;
      public:
            void setdata(char sname[],int sage, int sweight)
            {
                  strcpy(name,sname);
                  age=sage;
                  weight=sweight;
            }

            void showdata()
            {
                  cout<<"\nName="<<name;
                  cout<<"\nAge="<<age;
                  cout<<"\nWeight="<<weight;
            }
};

void main()
{
      student s1,s2;
      s1.setdata("Ram Bdr",18,65.5);
      s2.setdata("Sita Devi",28,55.09);

      s1.showdata();
      cout<<endl;
      s2.showdata();
      getch();
}
```

Nested of member function

- Since a member function of a class can only be called by an object of that class using a dot operator.
- However, a member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

Example:

```
//nesting member function
#include<iostream.h>
#include<conio.h>
class set
{
      private:
            int m,n;
      public:
            void input();
            int largest();
            void display();
```

```
};

inline int set::largest()
{
      if(m>=n)
            return m;
      else
            return n;
}

inline void set::input()
{
      cout<<"input values of m & n"<<endl;
      cin>>m>>n;
}
void set::display()
{
      cout<<"largest value="<<largest()<<endl;
}
void main()
{
      set set1;
      set1.input();
      set1.display();
}
```

An Array of Objects

As we know an array is a collection of similar data types. We can have an array of user defined data types class. Such variables are called arrays of objects. Like a structure we can use array of a class.

e.g.

```
class Test
{
      float a;
};
```

the array of object can be declared as:

Test T[n]; //arrays of n Test i.e. arrays of object

To access member data by objects we use array index as:

T[i].input_data( ); T[i].display( ); where i may be 0 to n-1 in array of size n.

Example:

```
//Example to show arrays of Objects
#include<iostream.h>
#include<conio.h>
class test
{
      float a;
      public:


void set_data(float no)
      {
            a=no;
```

```
      }
      void input_data()
      {
            cout<<"Enter the floating no:"<<endl;
            cin>>a;
      }
      void display()
      {
            cout<<"The no is:"<<a<<"\n";
      }
};
void main()
{
      test t[4];
      t[0].set_data(10.55);
      t[1].input_data();
      t[2].input_data();
      t[3].set_data(11.66);
      for(int i=0;i<4;i++)
            t[i].display();
      getch();
}
```

Private member function and its Access

Although it is normal to place all the data items in a private section and all the function in public, in some situation certain functions require to be hidden (like private data) from outside calls. Eg. Tasks such as deleting an account in a customer file. We can place these functions in the public 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.

```
      class account
      {
            int a;
            void read(void);
            public:
            void update(void);
            void write(void);
      };
```

if acc1 is an object of account, then

```
      acc1.read()     //error, because object cannot
                      // access private member
```

However, the function **read ()** can be called by the function **update ()** to update the value of **a**

```
//accessing private member function
#include<conio.h>
#include<iostream.h>
```

```
class account
{
      private:
            int a;
            void read()
            {
                  cout<<"a="<<endl;
                  cin>>a;
            }
      public:
            void update()
            {
                  read();//without dot operator
            }

            void write()
            {
                  cout<<a;
            }
};


void main()
{
      account acc1;
      acc1.write();  //legal
      //if we write s1.read();// it will be illegal
      s1.update(); //legal
      getch();
}
```

Some special characteristic of member functions:

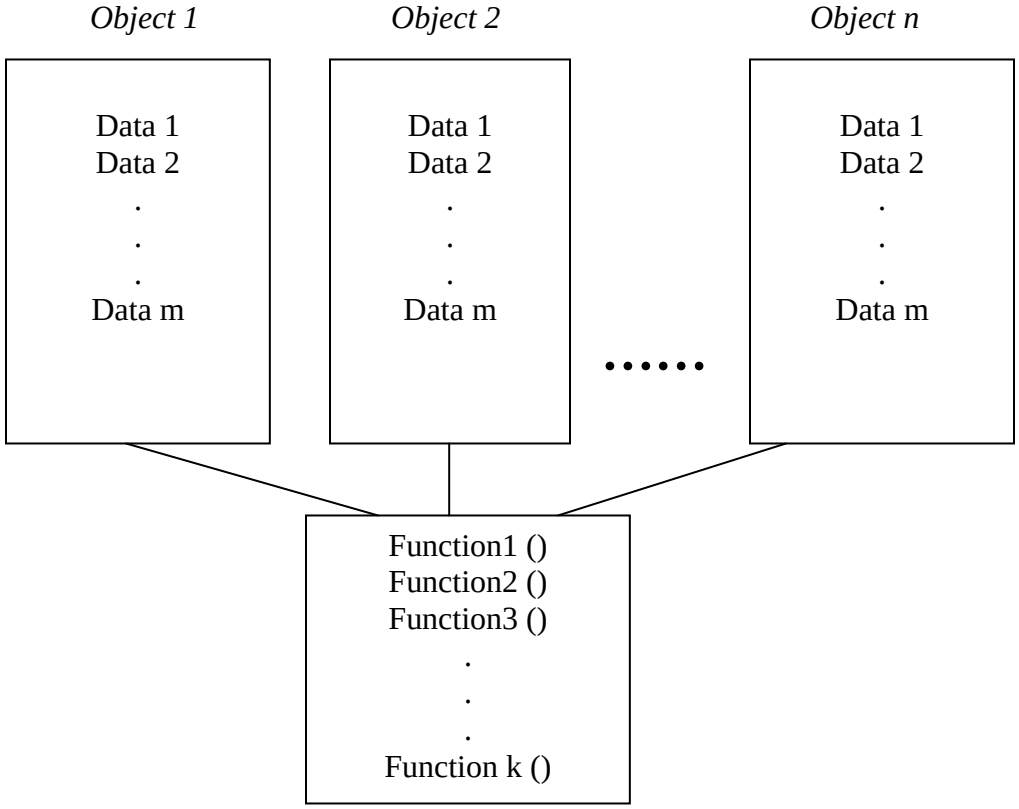
- Member functions can access the private data of the class. A non-member function cannot do so, except the **friend function**
- A member function can call another member function directly without using the dot operator.
- The private member functions and data cannot be accessed by the object of the class directly.
- A function definition inside class definition as default behaves as inline but if it is defined outside class definition, we should use keyword **inline** to make it inline function.

Class and Object and Memory

When a class is specified, memory space of all the member function is allocated but memory allocation is not done for the data member.

When an object is created, memory allocation for its data members is done. The logic behind separate memory allocation for member functions is quite obvious. All instances of a particular class would be using the same member functions but they may be storing different data in their data members.

Memory allocation for objects is illustrated in fig below



It can be observed that “n” objects of the same class are created and data members of those objects are stored in distinct memory location, whereas the member functions of object 1 to object n are stored in the same memory area. Therefore, each object has a separate copy of data members and the different objects share the member functions among them.

Note: *try to answer the following question:*
What are the commonality and differences between objects of same class? Justify your answer.

ASSIGNMENT

Long:

1. What is encapsulation and how does it help data hiding in Object-Oriented Programming.
2. Create a class called employee that contains a empname (array of string) and an empid (type long). Include a member function called getdata( ) to get data from the user for inserting into the object and another function called putdata( ) to display the data.

   Write a main ( ) program to exercise this class. It should create an array of the type employee and then invite the user to input data for up to 100 employees. Finally it should print out the data for all the employees. You need to make the array of employees an external variable.

Short

1. How do Structure in C and C++ differ?
2. What is a class? How does it accomplish data hiding?
3. Define the term Data and Class and what the relationship between a Class and Object is.
4. Explain how Class provides data hiding with example.
5. What is meant by data Encapsulation?
6. Explain a member function? How is a member function of a class defined?
7. Differentiate between public and private definition.
8. How does a C++ structure differ from C++ class?
9. What are Objects and how they are created?

Constructors and Destructors:

Constructors:

Constructors are member functions that are executed automatically when an object is created. Thus no explicit call is necessary to invoke them through the dot operator.

The constructors are invoked whenever an object of its associated class is created. It is called constructor because *it construct the value of data members* of the class. Constructors are *mainly used for data initialization*.

Syntax:

```
class class-name
{
private:
--------------
--------------
public:
            class-name() //constructor
            {
                  //Body of constructor
            }
};
```

To understand how constructors work, let's consider the following **example**:

```
//constructor
#include <iostrem.h>
class test
{
      public:
      test()
      {
            cout<<"In the constructor:";
      }
};
void main( )
{
      test t;           //creation of object
}
```

We have designed a class called "test" which has no data members and has only one member function test( ). In main( ), an object of type test is defined.
When this program is run, the output is:

```
In the constructor:
```

This implies that when an instance of an object is created, the member function test( ) is automatically invoked. Such member functions which are executed automatically when an instance of a particular class is created are known as constructors.

The constructors have the same name as the class_name.
When an object is created, the following process will take place.

- The object occupies space at a particular time. Instantiation of an object always involves reserving enough memory space for the data of that object.
- The instantiation does not reserve the memory for the methods. They exist only once for class, not once for every object.

- In addition to the reservation of space, the constructor may also be extended to other processes for initialization of data of that object.
- Constructor is a function (member) which is called automatically when object is created.

```
//constructor
//automatic initialization is carried out using a special
//member function.
#include<iostream.h>
#include<conio.h>

class counter
{
      private:
            int count;
      public:
            counter() {count=0;}
                  //constructor initialize value of count 0
                  //same name as class name
            void inc_count(){count++;}
            int get_count(){return count;}
};

void main()
{
      counter c1,c2;  //automatically count=0
      cout<<"\n c1="<<c1.get_count();
      cout<<"\n c2="<<c2.get_count();
      c1.inc_count();
      c2.inc_count();
      c2.inc_count();
      cout<<"\n c1="<<c1.get_count();     //c1=1
      cout<<"\n c2="<<c2.get_count();     //c2=2
      getch();
}
```

In this sample example, there is a constructor whose task is to create objects and initialize them by value count=0. The constructor counter(); is automatically invoked when object c1 and c2 of class measure are created and both objects are initialized to given value as defined by constructor.

Characteristics of constructors:

- Constructors name is the same as the class name.
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore they cannot return value.
- They cannot be inherited.
- Like other functions, they can have default arguments.
- Constructors of a class are the first member function to be executed.

Types of constructor

There are three types of constructors:

1. The default constructor
2. User-defined constructor
3. Copy constructor

1. **Default constructor:** (also called implicit constructor.)

The constructor that accepts no parameter is called the default constructor. The default constructor of class A is A::A( ). If no such constructor is defined, then the compiler supplies a default constructor.

Therefore, a statement such as,

A a;

Invokes the default constructor of the compiler to create the object "a".

- The compiler provides a (hidden) default constructor that has no arguments.
- The default constructor takes no arguments and performs no processing other than reservation of memory.
- This constructor is always called by compiler if no user-defined constructor is provided.
- This constructor is automatically called while creating the object.

The main **advantages** of default constructor is to allocate memory for objects

Example:

```
class student
{
      private:
            -----------
            -----------
      public:
            ----------
            ----------
};
void main( )
{
      student s1,s2     //default constructor student( ),
                        //(hidden) is automatically called.
}
```

2. **User-defined constructor**:

If initialization of data of objects is required while creating an object, then the programmer has to define his own constructor for that purpose. The code of a user defined constructor does not actually cause memory to be reserved for the data because it is still done by default constructor automatically.

The user defined constructor take arguments.

The main **advantages** of user defined constructor are to initialize the object while it is created.

Example:

```
class test
{
      float a;
      public:
            test (float number)
            {
                  a= number;
            }
            void display( )
            {
                  cout<<"The number is:"<<a;
            }
};
void main( )
{
      test t1(2.99);
      test t2=3.99;
      test t3= test(8.99);
      t1.display();
      t2.display();
      t3.display();
}
```

3. Copy Constructor:

Constructor having a reference parameter is known as copy constructor. The copy constructor creates an object as an exact copy of another object in terms of its attributes. In copy constructors, one newly instantiated object equals another exiting object of the same class.

Copy constructor are called using assignment operator or object as arguments automatically.

Example:

```
Class test
{
      int id;
      public:
            test()
            {   }
            test( int a)
            {
                  id=a;
            }
            test (test &x)
            {
                  id= x.id;
            }
            void display( )
            {
                  cout<<id;
            }
};
void main( )
{
      test a(100);
```

```
	test b(a);
	test c=a;
	test d;
	d=a;
	cout<<"id of a:"<<a.display( );
	cout<<"id of b:"<<b.display( );
	cout<<"id of c:"<<c.display( );
	cout<<"id of d:"<<d.display( );
}
```

Parameterized Constructor:

The constructors that can take arguments are called parameterized constructor.

Example:

```
Class test
{
	float a;
	public:
		test(float number)
		{
			a=number;
		}
		void display( )
		{
			cout<<"the number is:"<<a;
		}
};
void main( )
{
	test t1(4);
	test t2=5;
	t1.display( );
	t2.display( );
}
```

Constructor with default arguments:

It is possible to define constructor with default arguments.

Example, if we have class like:

```
	class complex
	{
		float x,y;
		public:
			complex(float real, float imag =0)
			{
				x=real;
				y=imag;
			}
	};
```

The default value of the argument imag is zero. Then, the statement-

```
	complex c (5.0);
```

assign the value 5.0 to the real variable and 0.0 to the imag(by default). However, the statement—

```
	complex c (2.0,3.0);
```

assign 2.0 to real and 3.0 to imag.

Example:

```
Class test
{
      float a;
      public:
            test(float number=0.0)
            {
                  a=number;
            }
            void display( )
            {
                  cout<<"The no. is:"<<a;
            }
};
void main( )
{
      test t1;
      test t2(2.99);
      test t3=test(3.99);
      test t4=4.99;
      t1.display( );
      t2.display( );
      t3.display( );
      t4.dispaly( );
}
```

***Note**: Don't confuse with default constructor with default argument constructor. Try to find difference between them.*

Constructor overloading:

A class can have multiple constructors. If more than one constructor is used in a class, it is known as constructor overloading. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

Example:

```
//constructor overloading
#include <iostream.h>
#include <conio.h>

class Account
{
       private:
            int accno;
            float balance;
      public:
            Account()   //constructor1
            {
                  accno=1024;
                  balance=5000.55;
            }
            Account(int acc)   //constructor2 with one argument
            {
                  accno=acc;
```

```
                balance=0.0;
        }

        Account(int acc, float bal)   //constructor3,with two
                                      //arguments
        {
                accno=acc;
                balance=bal;
        }

        void display()
        {
                cout<<"Account no.="<<accno<<endl;
                cout<<"Balance="<<balance<<endl;
        }
};  //end of class definition
void main()
{
     Account acc1;  //constructor1
     Account acc2(100);  //constructor2
     Account acc3(200, 8000.50);  //constructor3
     cout<<endl<<"Account information"<<endl;
     acc1.display();
     acc2.display();
     acc3.display();
     getch();
} //end of main()
```

Destructors:

A destructor is a member function which is automatically called when a program finishes execution. As name implies, destructor is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~). Example for the **class** test can be defied as-

```
     ~test( )
     {

     }
```

Example:

```
Class test
{
     public:
     test( )
     {
          cout<<”First in the constructor:”;
     }
     ~test( )
     {
          cout<<”Now in the destructor:”;
     }
};
void main( )
{
     test t;
}
```

Objects as Function Arguments

Like any other data types, an object may be used as function arguments. This can be done in two ways:

1. A copy of the entire object is passed to the function.
2. Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual objects used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

The following program illustrates the object as the function arguments:

```
#include<iostream.h>
#include<conio.h>
class Distance
{
      int feet;
      float inches;
      public:
      Distance()
      {
            feet=0;
            inches=0.0;
      }
      Distance(int f, float i)
      {
            feet=f;
            inches=i;
      }
      void getdata()
      {
            cout<<"Enter the feet:";cin>>feet;
            cout<<"\nEnter the inches:";cin>>inches;
      }
      void display()
      {
            cout<<feet<<"\'"<<inches<<"\"";
      }
      void add_distance(Distance, Distance);
}; //End of class

void Distance::add_distance(Distance d1,Distance d2)
{
      inches=d1.inches+d2.inches;
      while(inches>=12.0)

      {
            inches-=12.0;
            feet +=1;
      }
```

```
      feet+=d1.feet+d2.feet;
}


int main()
{
      Distance dist1,dist3;
      Distance dist2(11,8.5);
      dist1.getdata();
      cout<<"\nDistance1=";dist1.display();
      cout<<"\nDistance2=";dist2.display();
      dist3.add_distance(dist1,dist2);
      cout<<"\nDistance3=";dist3.display();
      getch();
      return 0;
}
```

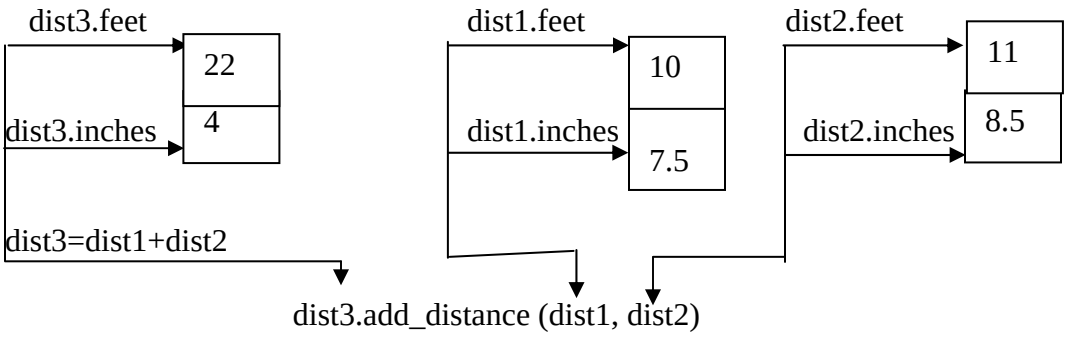


fig accessing members of objects within a class function

Passing objects by reference: Passing objects as reference is same as the passing variable function arguments. Just pass the reference of an object defining reference object as arguments.

The following program illustrates this idea

```
//object as function argument
//pass by reference

#include<iostream.h>
#include<conio.h>
class account
{
      private:
            int accno;
            float balance;
      public:
            void getdata()
            {
                  cout<<"enter account no";
                  cin>>accno;
                  cout<<"enter balnce";
                  cin>>balance;
            }
            void display()
            {
                  cout<<"account number is "<<accno<<endl;
```

```
                cout<<"Balance is"<<balance<<endl;
            }
            void MoneyTransfer(account & acc,float amt);
            //function for transfer amount to some object passed
}; //end class
            void account::MoneyTransfer(account &acc, float amt)
            {
                balance=balance-amt;//deduct money from balance
                acc.balance=acc.balance+amt;//add money to
                                                destination
            }

void main()
{
      int money;
      account acc1,acc2;
      acc1.getdata();
      acc2.getdata();
      cout<<"A/C info: "<<endl;
      acc1.display();
      acc2.display();
      cout<<"how much money is to be transferred from acc2 to acc1";
      cin>>money;
      acc2.MoneyTransfer(acc1,money);//transfers money
               //from acc2 to acc1;
      cout<<"updated information of account: "<<endl;
      acc1.display();
      acc2.display();
      getch();
}
```

Returning Objects from Functions

In previous example, we passed object as function arguments, now we will see an example of a function that returns an object.

```
#include<iostream.h>
#include<conio.h>
class Distance
{
      int feet;
      float inches;
      public:
      Distance()
      {
            feet=0;
            inches=0.0;
      }
      Distance(int f, float i)
      {
            feet=f;
            inches=i;
      }
      void getdata()
      {
            cout<<"Enter the feet:";cin>>feet;
            cout<<"\nEnter the inches:";cin>>inches;
```

```
      }
      void display()
      {
            cout<<feet<<"\'"<<inches<<"\"";
      }
      Distance add_distance(Distance);
}; //End of class
Distance Distance::add_distance(Distance d2)
{
      Distance temp;
      temp.inches=inches+d2.inches;
      while(temp.inches>=12.0)
      {
            temp.inches-=12.0;
            temp.feet +=1;
      }
      temp.feet+=feet+d2.feet;
      return temp;
}
int main()
{
      Distance dist1,dist3;
      Distance dist2(11,10.25);
      dist1.getdata();
      cout<<"\nDistance1=";dist1.display();
      cout<<"\nDistance2=";dist2.display();
      dist3=dist1.add_distance(dist2);
      cout<<"\nDistance3=";dist3.display();
      getch();
      return 0;
}
```

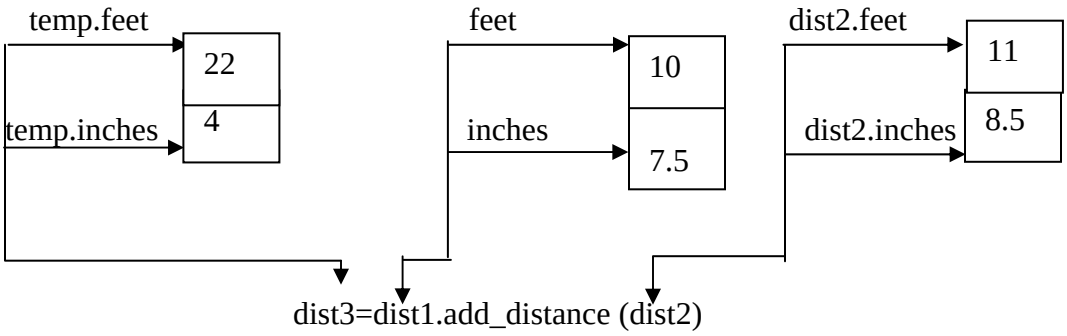


fig accessing members of objects within a called function

Long question:

1. What are constructor and destructor? Explain different types of constructor used in C++.
2. Create a class called Distance with two data members inch and feet. Provide Constructor and different member function with the following operations.
   - To input data for Distance objects.
   - To show the data of Distance objects.
   - Member function to add two Distance objects passed as object as function arguments and then display the result.
3. Create a class called Distance with two data members inch and feet. Provide Constructor and different member function with the following operations.
   - To input data for Distance objects.
   - To show the data of Distance objects.
   - Member function to add two Distance objects passed as object as function arguments leaving the result in the third Distance object and then display the result.
4. Create the class called time that has separate integer member (attribute) data for hours, minutes, and seconds. One constructor should initialize this data to 0, and another should initialize it to a fixed values. A member function should display it in 10:34:55 format. The final member function should add the two objects of type time passed as arguments.
   A main program should create two initialize time objects and one not initialize time object. Then it should add the two initialize value together, leaving the result in the third time variable. Finally it should display the value of this third variable.

Short questions:

1. Point out few important characteristics of constructors.
2. Differentiate between constructors and other member functions.
3. Explain the use of destructors. Point out the main differences between constructors and destructors.
4. Differentiate between parameterized and non-parameterized constructors.
5. Discuss the importance of the constructor.
6. What is the primary role of the constructors?
7. Discuss the primary role of the user-defined constructors with example.

6. Operator Overloading

Operator Overloading is one of the most fascinating features of C++. It is the mechanism of giving special meanings to an operator. By overloading operators we can give additional meanings to operators like +, *, -, <=, >= etc. which by default are supposed to work only on standard data types like ints , floats.

Example, if str1 and str2 are two character arrays holding strings “well” and “come” in them then to store “welcome” in third string str3 in C we need to perform the following operations:

```
char str1[ ]= “wel”;
char str2[ ]= “come”;
char str3[20];
strcpy(str3, str1);
strcat(str3,str2);
```

No doubt this does the desired task but the following have more sense:

```
str3 = str1 + str2;
```

Such a form obviously would not work in C, since we are attempting to apply the + operator on non-standard data types (string) for which addition is not defined. But C++ permits the + operator to be overloaded such that it knows how to add two strings.

Even though the semantics of an operator can be expressed, we cannot change its syntax. When an operator overloaded, its original meaning is not lost. The grammar rules defined by C++ that govern its use such as the number of operands, precedence and associatively of the operator remains the same for overloaded operators.

We can overload (give additional meanings to) all the C++ operators except the following:

- Class member access operators (. , .*)
- Scope resolution operator (::)
- Size of operator (sizeof)
- Conditional operator (?:)

Syntax for operator overloading: The keyword operator is used for overloading.

```
return_type operator operator_symbol (arguments)
{
     //body of the function

}
```

Operator functions must be either member functions or friend functions. A basis difference between them is that a friend function will have only one argument for unary operator and two for binary operator while a member function has no arguments for unary operator and only one for binary operator.

The process of operator overloading generally involves following steps.

1. Declare a class whose objects are to be manipulated using operators.
2. Declare the operator function, in public part of class. It can either normal member function or friend function.
3. Define operator function within the body of a class or outside the body of the class but function prototype must be inside the class body.

Unary Operator Overloading

An unary operators acts on only one operand. Examples of unary operators are the increment and decrement operators ++ and -- and the unary minus as in -45.

Syntax:

```
Return_type operator unary_operator_symbol ()
{

     //body of functions
}
```

Invoking overloading unary operator:

```
Prefix form: unary_operator object_name;
Postfix form: object_name unary_operator;
```

Let us now implement an overloading unary operator-

```
#include<iostream.h>
#include<conio.h>
class index
{
     int count;
     public:
     index()
     {
          count=0;
     }
     void display()
     {
          cout<<count;
     }
     void operator++()
     {
          ++count;
     }
};
int main()
{
     index c;
     cout<<"C=";
     c.display();
     ++c;
     ++c;
     cout<<"C=";
     c.display();
     getch();
     return 0;
}
```

In this program the count of object c is initially set to 0. On encountering the expression ++c it is incremented by 1. The output of the program looks like:

C=0
C=2

Internally, the expression ++c is treated as:
c.operator ++ ( );

While calling this function no value is passed to it and no value is returned from it. The compiler can easily distinguish between the expression ++c and an expression, say ++i, where i might be an integer variable. It can make this distinction by looking at the data types of the operands. If the operand is a basic type like an int, as in ++i, then the compiler will use its build-in routine to increment an int. But if the operand is an index variable (an object), then the compiler will now use our operator ++ () function.

Overloading Unary operator that return a value:

If we have to use overloaded operator function for return a value as:

```
Obj2=obj1++;  //returned object of obj++ is assigned to
              //obj2

//Example: unary operator overloading  with return type.
#include<iostream.h>
#include<conio.h>
class index
{
     int count;
     public:
     index()
     {
          count=0;
     }
     void display()
     {
          cout<<count;
     }
     index operator++()
     {
          ++count;
          index temp;
          temp.count=count;
          return temp;
     }
};
```

```
int main()
{
     clrscr();
     index c,d;
     cout<<endl<<"C=";
     c.display();
     ++c;
     cout<<endl<<"C=";
     c.display();
     d=++c;
     cout<<endl<<"C=";
     c.display();
     cout<<endl<<"D=";
     d.display();
     getch();
     return 0;
}
```

Here the *operator ++ ( )* function increments the *count* in its own objects as before, then creates the new *temp* object and assigns *count* in the new object the same value as in its own object. Finally it returns the *temp* object. This has the desired effect. Expression like *++c* now return a value, so they can be used in other expressions such as
d= ++c;
In this case the value returned from *++c* is assigned to *d.*
Program's output would look like this:

```
C=0
C=1
C=2
D=2
```

In our program we created a temporary object called temp. Its purpose was to provide a return value for the ++ operator. We could have achieved the same effect using the nameless temporary object.

Nameless temporary object : A convenient way to return an object is to create a nameless temporary object in the return statement itself . In the above program, modify class definition by

```
public :
     index() {count=0;}
     …….
     ……
     index operator++()
     {
          ++count;
          return index(count);
     }
```

Note that the operator ++( ) function has changed. In this function the statement,

```
     return index(count);
```

creates an object of type index. This object has no name.

Note : When ++ or – is used in its overloaded role, there is no difference between pre and post operations . i.e. object ++ and ++object has the same role.

i.e. obj2=++obj1; and obj2=obj1++; has exactly same effect

So to distinguish postfix and prefix operation C++ provides additional syntax to express this prefix and postfix operation. The operator function.

Operator++ () above is defined to indicate prefix and postfix operation as

```
      // prefix operation
      index operator ++ ()
      {
            return index(++count);    // object is created with ++count
                                      // i.e. new value of count and
      }

// postfix operation
index operator ++ ()
{
      return index(count++)    // object is created with count++
                              // i.e. old value of count and value is
                              //returned.
}
```

We can give increment role to -- operator and decrement role to ++ operator defining operator function as

```
// decrement role to ++
index operator ++()
{
      count--;                // decrements
      return index(count);
}

// increment role to --
index operator --()
{
      count++;
      return index(count);    // increments

}
```

Binary operator overloading

Binary operators can be overloaded as unary operator. The syntax for overloading the biary operator is:

```
retrun_type operator operator_symbol(arg)
{
     //body of function.
}
```

Invoking binary operator:

```
Object1 operator object2;
```

The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly. The data members of the first object are accessed without using the dot operator whereas, the second argument members can be accessed using the dot operator if argument is an object, and otherwise it can be accessed directly.

The following examples illustrate the overloading of binary operators:

- `Distance operator +( Distance d);`
- `Void Distance operator +( Distance d);`
- `int Distance operator + ( Distance d);`

Example:

```
//Program to illustrate the binary operator overloading.
class Distance
{
     int feet;
     float inches;
     public:
     Distance(){}
     Distance(int ft, float in)
     {
          feet=ft;
          inches=in;
     }
     void  display()
     {
          cout<<feet<<"\'"<<inches<<"\"";
     }
     Distance operator +( Distance d)
     {
          Distance temp;
          temp.inches=inches+d.inches;
          if(temp.inches>=12.0)
          {
               temp.inches-=12.0;
               temp.feet+=1;
          }
          temp.feet+=feet+d.feet;
          return temp;
```

```
	}
};
void main()
{
	Distance d1(11,7.25);
	Distance d2(10,8.75);
	Distance d3;
	d3=d1+d2;
	d3.display();
}
```

We would agree that the statement

d4=d1+d2+d3;

Is more intuitive than the statement:

d4.add_dist (d1, d2.add_dist (d3));

When the operator + ( ) function is called, the object d2 is passed to it and is collected in the object d. As against this the object d1 gets passed to it automatically. This becomes possible because the statement d3= d1 + d2 is internally treated by the compiler as

d3. d1.opeator + (d2);

Concatenating strings with overloaded + operator

In C, + operator can not concatenate two strings. In C++ it is possible to use + operator to concatenate two strings using overloaded + . The original mining of + can not be altered for basic data type but we are giving additional meaning to this +.

```
//String concatenation using + operator
#include<iostream.h>
#include<string.h>
#include<conio.h>
class String
{
	private:
		char str[40];
	public:
		String()
		{
			strcpy(str," ");
		}
		String(char *mystr)
		{
			strcpy(str,mystr);
		}
		void display()
		{
			cout<<str;
		}
		String operator +(String s)
		{
			String temp;
			temp=str;
			strcat(temp.str,s.str);
			return temp;
```

```
            }
};//end of class String

void main()
{
      String s1="Tribhuvan";
   String s2="University";
   String s3;
   s3=s1+s2;
   s1.display();cout<<" + ";
   s2.display(); cout<<" = ";
   s3.display();
   getch();
}
```

Overloaded relational operator

The relational operators can also be overloaded as other binary operators to extend their semantics. The following example shows the '>' operator overloading to use it for comparison of two user defined objects.

Example:

```
//overloading relational operator
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>

class money
{
      private:
            int rs; float ps;
      public:
            money(){rs=0;ps=0.0;} //no argument constructor
            money(int r,float p)
            {
                  rs=r; ps=p;
            }

            void show()
            {
                  cout<<"Rs. : "<<rs<<" Ps. "<<ps;
            }
            void get()
            {
                  cout<<"Enter Rs.:";cin>>rs;
                  cout<<"Enter Ps.:";cin>>ps;
            }
            int operator>(money);
};  //end class

//definition of > outside the class definition
int money::operator>(money m2)
{
            float mm1=rs+ps/100;
            float mm2 = m2.rs+m2.ps/100;
            return(mm1>mm2)?true:false;
```

```
}
void main()
{
            money m1;
            m1.get();
            money m2;
            m2.get();
            cout<<"Amount 1:";m1.show();
            cout<<endl<<"Amount 2:";m2.show();
            if(m1>m2)
                  cout<<endl<<"Amount 1 is greater than amount2";
            else
                  cout<<endl<<"Amount 1 is less than  to  amount2";
            getch();
}
```

Overloading equality operator

```
//overloading == operator
#include<iostream.h>
#include<conio.h>
class ratio
{
      int num, den;
      public:
            ratio(){}
            ratio( int n, int d ) {num =n;den=d;}

            void get()
            {
                  cout<<"Nr:";cin>> num;
                  cout<<"Dr:";cin>>den;
            }
            int operator==(ratio&r)
            {
                  return (num*r.den==den*r.num);
            }
};
void main ()
{
      ratio r1;
      r1.get();
      ratio r2;
      r2.get();
      if(r1==r2)
            cout<<"Equal Ratio";
      else
            cout<<"Unequal Ratio";
      getch();
}
```

Rules for overloading operators

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.

- We cannot change the meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from other.
- Overloaded operators follow the syntax rules of the original operators. That cannot be overridden.
- As described above, all operators cannot be overloaded.
- Unary operators, overloaded by means of a member function take no explicit arguments and return no explicit values. But, those overloaded by means of friend function take one argument.
- Binary operators, overloaded by means of a member function take one explicit argument. But, those overloaded by means of friend function take two argument.

7. Data Conversion

The = operator will assign a value from one variable to another. The type of data to the right of an assignment operator is automatically converted into the type of the variable on the left.

Example:
```
int a;
foat b = 3.1416;
a=  b;
```

Converts "b" to an integer before its value is assigned to "a". So the fractional part is truncated. So for built in data type, data conversion is done automatically.

For user defined data types like:

```
dist3 = dist1 + dist2;   //dist1,   dist2,   dist3   are class type objects
```

When the objects are of the same class type the operation of addition and assignment are carried out smoothly and the compiler does not make any errors. The value of all the data members of the right hand objects are simply copied into the corresponding members of the object on the left hand. What if one of the operand is an object and the other is a built in type variable? Or what if they belong to two different classes?

In case the data items are of different types, data conversion interfaces must be explicitly specified by the user.

Following four types of situations might arise in the data conversion between incompatible types:

1. Conversion between basic data types
2. Conversion between built in types to class types.
3. Conversion between class types to built in data types.
4. Conversion between one class types to another class type.

Conversion between basic data types:

The compiler has several built in routines for the conversion of basic data types such as `char to int, int to float, float to double etc`. this features of the compiler which performs conversion of data without the user intervention is known a implicit type conversion.

Example:
```
float pi;
int a;
pi = a;
```

The compiler calls a special routine to convert the value of `a`, which is integer to a floating point format so it can be assigned to `pi`.

The compiler can be instructed explicitly to perform type conversion operators known as typecast operators. Example to convert `int to float`, the statement is—

```
Pi = (flaot) a;
```

This is C style and also valid in C++.

In C++ the above statement can be written as:

```
Pi = float (a);
```

Conversion form built into class type:

To convert data from a basic type to a user defined type, the conversion function must be defined in user defined object's class in the form of constructor. The constructor function takes a single argument of basic data type.

Syntax:

```
constructor ( Basic type)
{
     //converting statements
}
```

Example:

```
class Hour
{
     int hr;
     public:
     Hour()
     {
          hr = 0;
     }
     Hour (int m)
     {
          hr = m/60;
     }
     Void display()
     {
          cout<<"Hour = "<<hr;
     }
};
Void main()
{
     Hour h1;
     int m;
     cout<<"Enter minutes:";
     cin>>m;
     h1 = m;   //m is basic data and
               // h1 is user defined data
               //convert from basic to user defined data.
     h1.display();
}
```

Conversion form class type into built type:

The conversion function must be defined in user defined objects class in the form of the operator function. The operator function is defined as an overloaded basic data types which takes no arguments. It converts the data members of an object to basic data types and returns a basic data items.

Syntax:

```
Operator basic_data ( )
{
     //conversion statements
}
```

Examples:

```
class Hour
{
     int hr;
     public:
     Hour()
     {
          hr = 0;
     }
     Operator int()
     {
          int m;
          m = hr * 60;
          return (m);
     }
     Void getdata()
     {
          cout<<”Enter the hours:”;
          cin>> hr;
     }
};
Void main()
{
     Hour h1;       // h1 is user defined data
     float m;       //m is basic data
     h1.getdata()
     m = h1    //convert from basic to user defined data.
     Cout<<”Minutes = ”<<m;
}
```

Conversion between objects of different classes:

The C++ compiler does not support automatic data conversion between objects of classes.

```
class classX
{


};
class classY
{

};
```

```
objX = objY;
```

ObjX is an object of class classX and objY is object of class classY. The classY type is converted to classX type data and the converted value is assigned to objX. Since the conversion takes place from classY to classX, classY is called source and classX is called destination class.

The conversions between objects of different classes can be carried out by either one argument constructor or an operator function. The choice depends on whether we put the conversion routine in the destination class or in the source class.

1. Conversion Routine in Source Objects: Operator Functions

The conversion routine in the source object's class is implemented as an operator function.

Syntax:

```
class classX      //destination class
{


};
class classY      //source class
{

   public:
   operator classX()
   {
        //code for conversion from
        //classY to classX
   }
};
```

For assignment statements such as:

```
objx = objY;
```

ObjY is the source object of the classY and objX is the destination object of the classX. The conversion `operator classX()` exist in the source object's class.

Example: *conversion polar coordinate to Rectangular coordinate using routine in polar class (source class)*

```
class Rec
{
   double xco, yco;
   public:
   Rec()
   {
        xco = 0.0;
        yco = 0.0;
   }
   Rec(double x, double y)
   {
        xco = x;
        yco = y;
```

```
   }
   void display()
   {
        cout<<"("<<xco<<","<<yco<<")";
   }
};
class Polar
{
   double radius, angle;
   public:
   polar()
   {
        radius = 0.0;
        angle = 0.0;
   }
   operator Rec ()
   {
        double x = radius*cos(angle);
        double y = radius*sin(angle);
        return Rec(x,y);
   }
   void display()
   {
        Cout<<"("<<radius<<","<<angle<<")";
   }
};
void main()
{
   Rec r;
   Polar p(5.0,.785398);
   r = p;
   cout<<"POLAR COORDINATE:";
   p.display();
   cout<<"RECTANGULAR COORDINATE:";
   r.display();
}
```

2. Conversion Routine in Destination Object: Constructor Function

The conversion routine can be defined in the destination class as a one argument constructor.

Syntax:

```
class classY      //source class
{

};
class classX      //destination class
{
```

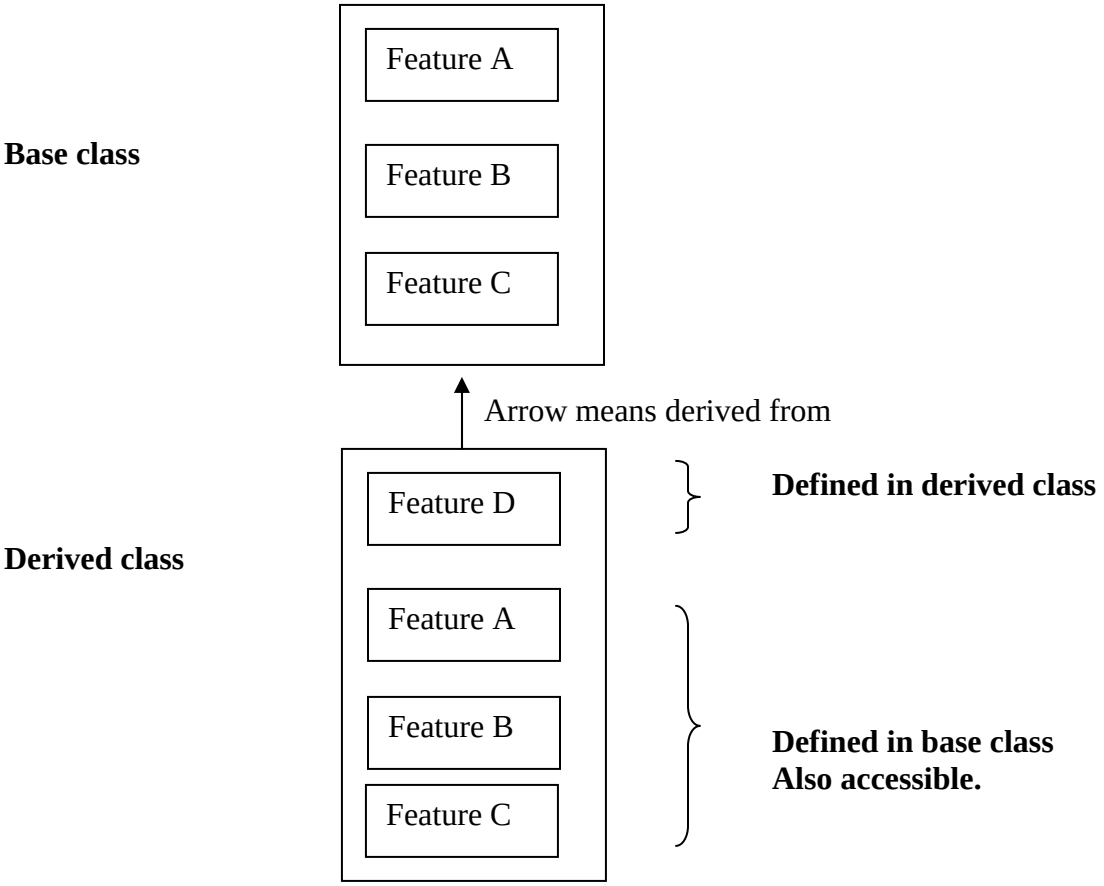
```
   public:
        classx(classY objY)
   {
        //code for conversion function
   }
};
PROGRAM  CONVERTING  POLAR  TO  REC  USING  ROUTINE  IN
REC(DESTINATION CLASS)
class Polar
{
   double radius, angle;
   public:
   polar()
   {
        radius = 0.0;
        angle = 0.0;
   }
   polar (double r, double a)
   {
        radius = r;
        angle = a;
   }
   void display()
   {
        Cout<<”(”<<radius<<”,”<<angle<<”)”;
   }
   double getradian ()
   {
        return radius;
   }
   double getangle()
   {
        return angle;
   }
};
class Rec
{
   double xco, yco;
   public:
   Rec()
   {
        xco = 0.0;
        yco = 0.0;
   }
   Rec(polar p)
   {
```

```
        double r = p.getdata ();
        double a = p.getdata ();
        xcor = r*cos(a);
        ycor = r*sin(a);
   }
   void display()
   {
        cout<<"("<<xco<<","<<yco<<")";
   }
};

void main()
{
   Rec r;
   Polar p(5.0,.785398);
   r = p;
   cout<<"POLAR COORDINATE:";
   p.display();
   cout<<"RECTANGULAR COORDINATE:";
   r.display();
}
```

8. Inheritance

Inheritance is the most powerful feature of OOP. Inheritance is the process of creating new classes, called derived classes, from existing classed. These existing classes are called base classes. The derived classes inherit all the capabilities of the base classes but can add new features and refinements of its own. By adding these refinements the base class remains unchanged.

- Inheritance permits code reusability.
- Inheritance is like a child inheriting the features of its parents.
- It is a technique of organizing information in a hierarchical (tree) form.
- A base class is also called ancestor, parent or super class and derived class is also called as descendent, child, or subclass.



Protected data members: Instead of private and public members there is another type protected member in a class. The purpose of making data member protected in a class is to make such member accessible from the function of the derived class. No other class than derived class function can access the protected data of a base class.
Derived class declaration: The derived class inherits all the featured of its parent class and adds its own new features.

The ***syntax*** of declaration of derived class is:

```
class derive class : <private/ public> base class
{
      //member of derived class
      //other members
}
```

Syntax:

```
   class A
   {
        private :
              // data members ;
        protected :
              //data members  ;
        public ;
              //function members ;

 } ;
```

```
class B : public A   // public derivation
{

      //members of B
} ;

class C : private A // private derivation
{

      //members for c
} ;
```

```
class D : A   // private derivation default
{
   //  member of D
} ;
```

Example:

```
// an example of inheritance
#include<iostream.h>
#include<conio.h>

class parent //base class
{
      protected: //if private count will be inaccessible to derived class
            int count;
      public:
            parent () {count=0;} //zero argument constructor
      void display()
      {
            cout<<"count="<<count<<endl;
      }

      void operator ++()//unary operator overloading
      {
            count++;
      }

};
class child : public parent  //derived class
{
      public:
            void operator --()
            {
                  count--;
```

```
            }
};


void main()
{
      child c;
      ++c;
      ++c;
      c.display();
      --c;
      c.display();
      getch();
}
output:
      count=2
      count=1
```

Here we have first declared a base class called parent and then derived a class called child from it. Child inherits all the features of the base class parent. Child doesn't need a constructor or the `operator ++ ()` function, since they are already present in the base class.

The first line of the child class,

`class child : public parent`

Specifies that the child has been derived from the base class parent.

A protected member (i.e. int count) is introduced in base class. A private member cannot access by objects of a derived class. The protected members can be accessed by the member functions of derived class.

Example: operator -- () is defined in derived class child which can access the protected data "count" of class parent.

It is not possible to define an object of base class that us – overloaded operator: since – is member of derived class.

Inheritance and member Accessibility:

- Private members of a base class cannot be inherited directly to the derived class. The private member of a class is accessible only to the member function of its own.
- Making members of base class protected, they are accessible to the member function of derived class.
- A protected member can be considered as a hybrid of a private and a public member. Like private members, protected are accessible only to its class member function and they are invisible outside the class. Like public members, protected members are inherited by derived classes and also accessible to member function of derived class.
- A public member is accessible to members of its own class, member of derived class and even outside the class.

Public and private inheritance:

The visibility mode in the derivation of new class can be either public or private.

Public inheritance: when a class is derived publicly from its base class, the object of derived class can access public member of base class.

Private inheritance: when a class is derived privately, the object of derived class cannot access public member functions of the base class. Since objects can never access private or protected members of a class, the result is that no member of the base class is accessible to objects of the derived class.

Example

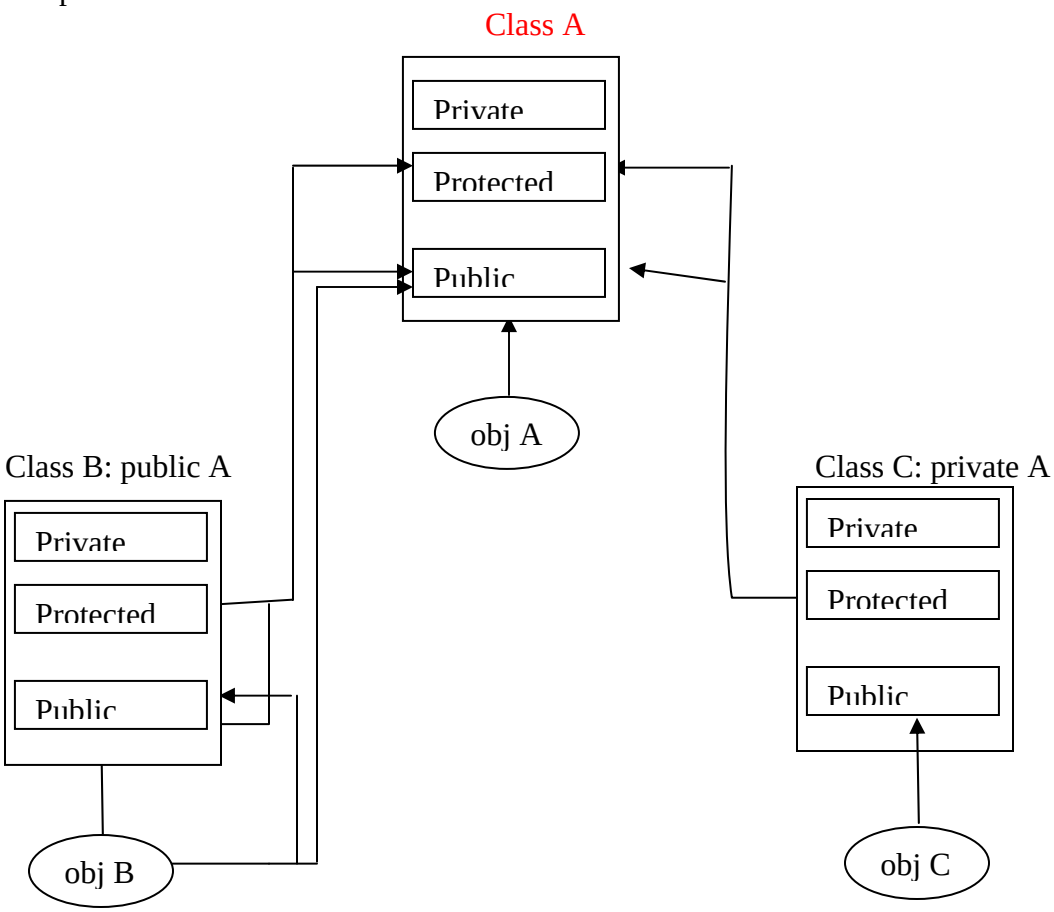
```
/*public or private inheritance*/
#include<iostream.h>
#include<conio.h>

class  A
{
       private :
            int pvtdataA ;
      protected :
            int protdataA ;
      public:
            int  pubdataA ;
} ;

class B  : public A // publicly derived
{
      public :
            void function ( )
            {
                  int a ;
                  //a = pvtdataA ; // error: not accessible.
                  a = protadataA ; // ok
                  a = pubdataA ;  // ok
            }
} ;

class C  : private A // privately derived.
{
      public :
            void funct ( )
            {
                  int  a ;
                  //a = pvtdataA ; // Error : not accessible
                  a = protdataA ; //  ok
                  a = pubdataA ; //   ok
            }
};

void main()
{
      int  a ;
      B objB ;
```

```
        //a = objB.pvtdataA ;  //  error : not accessible
        //a = objB.protdataA ; //  error : not accessible
        a = objB.pubdataA ; //   ok

        C objC ;
        //a = objC.pvtdataA;   // Error; not accessible
        //a = objC.protdataA;  // error; not accessible
        //a = objC.pubdataA;  //  error; not accessible A is private to C
        getch();
}
```

The program specifies a base class, A, with private, protected, and public data items. Two classes, B and C, are derived from A. B is publicly derived and C is privately derived. Functions in the derived classes can access private or protected members of the base class. Objects of the derived classes cannot access private or protected members of the base class.

Objects of the publicly derived class B can access public members of the base class A, while objects of the privately derived class C cannot. They can only access the public members of their own derived class.

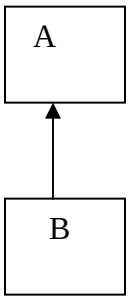
Following figure shows the relationship and accessibility of members in private and public inheritance.



Level of Inheritance

The level of inheritance refers to the length of its path from the root (top base class). A base class itself might have been derived from other classes in the class hierarchy. Inheritance is classified into the following forms based on the levels of inheritance and interrelationship among the classes in their hierarchy.

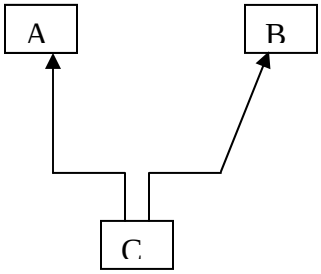
1. Single Inheritance: When a class is derived from only one base class, such derivation is called single inheritance. In single inheritance, base class and derived class exhibits one to one relationship. Following diagram exhibits the point



B is derived from class A.

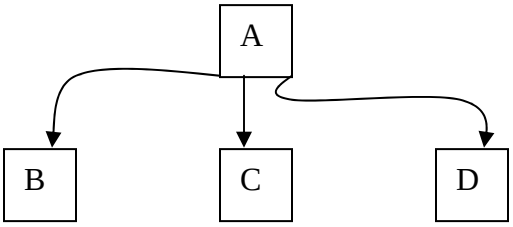
```
class  A          class B : A
{ ...             { ...
} ;               } ;
```

2. Multiple inheritances: Derivation of a class from two or more base class is called multiple in heritance. In multiple inheritances, the derived class inherits some or all the features of base classes from which it is derived. Following figure shows the multiple inheritances.




```
Class  A        Class  B        Class C: A , B
{ ....          { ...           { ...
} ;             } ;             } ;
```

3. Hierarchical Inheritance: When several (more than one) classes are derived from singled base class z-e. Feature of one class may be inherited by more than class, and then it is called hierarchical inheritance. Following figure shows this type of inheritance.



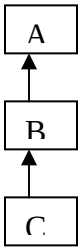
```
Class A
{…..
};
```

```
Class  D : A
{ ...
};
```

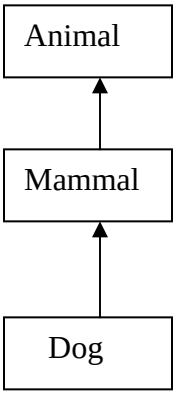
```
Class  B : A
{ ....
} ;
```

```
Class  C : A
{ ....
};
```

4. <u>Multilevel inheritance:</u> The derivation of a class from another derived class called multilevel inheritance. In figure below, we can see multilevel inheritance

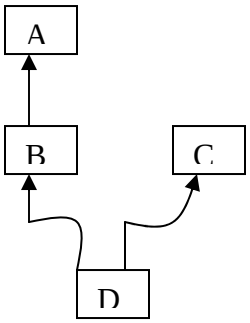


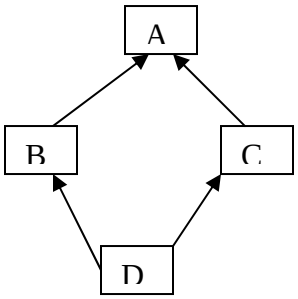
e.g

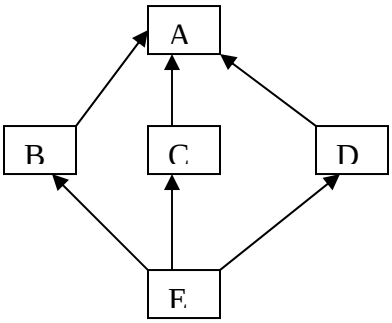


a serves as base class for derived class B, which is intern serves as base class for class C. The class B is known as intermediate base class and the chain ABC is inheritance path.

5. <u>Hybrid inheritance:</u> Derivation of class involving more than one form of inheritance is known as hybrid in heritance see fly.



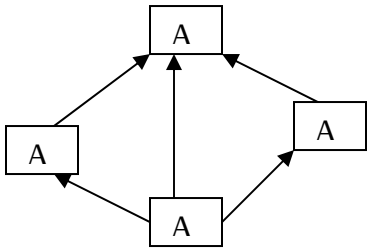




Hybrid (multilevel + multiple)

[Multilevel, Hierarchical, multiple]
→ Hybrid

6. <u>Multi-path inheritance:</u> Derivation of a class from other derived classes, which are derived from same base class, is called multi-path inheritance. Fly.



Derived Class Constructors:

The derived class need not have a constructor as long as the base class has a no-argument constructor. If base class has constructor with arguments (one or more) then derived class must have a constructor explicitly defined such that it passes arguments to the base class constructor.

In the application of inheritance, objects of derived class are usually created instead of the base class. So derived class should have constructor and pass arguments to the constructor of base class. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class is executed.

Constructor only in base class:

```
#include <iostream.h >
class B
{
      public:
      B( )
      {
            cout <<" No argument constructor of base class executed";
      }
};

class D : public B    // publicly derived
{
       public:
};


void main ( )
{
      D obj1 ;   // accesses base constrictor
}
```

```
Run : No argument constructor of base class executed.
```

Similarly, when constructor is present only derived class it is invoked at the object instantiation of derived class

Constructor in base and derived class:

```
class  B
{
      public:
      B(int a) { cout <<" One argument constructor in base class B";}
};

class D : public B
{
      D(int a)
       {cout <<" One constructor in derived class D";}
};
```

```
void main ( )
{
 D objd (3);
}
```
The compilation generates error like:
Cannot find "default" constructor to initialize base class 'B'

To overcome this error, explicit invocation of a constructor of base class in derived class constructor is needed.

```
class D : public B
{
       public:
                  D( int a ) : B (a)
                  {cout <<" One arg .Constrictor in derived class D ";}
};
```

Run : one – arg constructor in base class 'B'
one – arg constructor in derived class D.

Example:

```
//derived class constructor
#include<iostream.h>
#include<conio.h>

class one //base class
{
      protected:
            int count;
      public:
            one() {count=0;} //zero argument constructor
            one(int i) {count=i;}

   void display()
   {
      cout<<"count: "<<count<<endl;
   }
   void operator ++()//unary operator overloading
   {
      count++;
   }

};
class two: public one  //derived class
{
      public:
            two(): one()
            { }
            two(int i): one(i)
            { }
            void operator --()
            {
                  count--;
            }
};
```

```
void main()
{
      two i1;
   two i2(200);
   i1.display();  //displays 0
   i2.display(); //displays 200
   ++i1;
   ++i1;
   i1.display();   //displays 2
   --i2;
   i2.display();   //displays 199
   getch();
}
```

Constructor in multiple inherited classes with explicit invocation

```
#include<iostream.h >
class base1 // base class.
{
      public :
            base1()
            { cout << " No arg constructor in abase1 " ; }
};

class base2
{
      public :
            base2()
            {cout <<" No a rrg  Constructor in base2 "; }
};

class derived : public base1, public base2
{
      public :
      derived():base2(), base1()    //  explicit call
      { cout <<" No arg  constructor in derived class " ;}
};

void main ( )
{
      derived obj ;
}

Run :       No arg constructor in base1
            No arg constructor in base2
            No arg constructor in derived class
```

Here in above program, class derived is derived publicly from class base1 and base2 in order. Constructors of base classes are invoked explicitly in derived class as

derived ( ) : base2 ( ), base1 ( )

The order of execution of constructor is same as order as order of inheritance rather than order of explicit call.

Constructor invocation for data member initialization:

In multiple inheritances, the constructors of base classes are invoked first, in the order in which they appear in the declaration of the derived class.
In multilevel inheritance, constructors are invoked in the order of inheritance. For initialization of data members, the derived class object are created and values are supplied either by object of derived class or a constant value can be mentioned in the definition of the constructor.

The syntax of defining constructor in derived class

derived class (arg – list) : Base2 (arg – list2) , ... , Base N (arg – list N)
{
    // body of derived class constructor

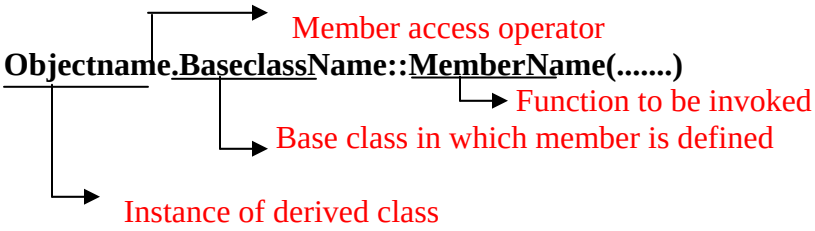
}

Ambiguity Resolution in Multiple inheritances:

Ambiguity is a problem that surfaces in certain situation involving multiple inheritances.

- Base classes having function with the same name.
- The class derived from these base classes is not having a function with the name as those of its base classes.
- Member of a derived class or its objects referring to a member, whose name is the same as those in base classes

The problem of ambiguity is resolved using the scope resolution operator as shown in figure.



Example: of Ambiguity

```
//amboguty in member access
#include<iostream.h>
#include<conio.h>

class  A
{
      public:
            void show() { cout<<" class  A "; }
};
class  B
{
      public:
            void show() { cout<<" class  B "; }
};
class C : public A , public B
{
```

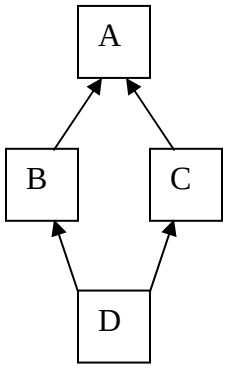
```
};
void main()

{
      C objC;     // object of class C
      //objC.show() ;   //  ambiguous....error
      objC.A::show() ; // ok invokes show() in class A.
      objC.B::show() ;  // ok
      getch();
}
```

The statement objC.show(); is ambiguous as compiler has to choose A::show( ) or B:: show( ). It can be resolved using the scope resolution operator as follows

objc.A::show( );

refers to the version of show ( ) in the class A.

Another example:




```
class  A
{    public :
          void func ( ) ;
};
class B : public A.
{
     //......
};
class C :public A
{
     // .......
};
class D : public B, public C
{
     //……..
};
void main ( )
{
     D obj  D;
     obj D. fun ( ) ; // ambiguous :
}
```

Overriding Member Functions:

Defining a functions in derived class that have same name as those in base class is known as function (Member function) overriding. The member function in derived class overrides the function.

```
#include <iostream.h>
#include <conio.h>
class publication
{
     char title[50];
     float price;
```

```
     public:
     void getdata()
     {
          cout<<"Enter title:";cin>>title;
          cout<<"Enter price:";cin>>price;
     }
     void putdata()
     {
          cout<<"\ntitel:"<<title;
          cout<<"\nprice:"<<price;
     }
};
class book:public publication
{
     private:
     int pages;
     public:
     void getdata()
     {
          publication::getdata();
          cout<<"Enter the pages:";cin>>pages;
     }
     void putdata()
     {
          publication::putdata();
          cout<<"\nPages:"<<pages;
     }
};
class CDROM : public publication
{
     private:
     float time;
     public:
     void getdata()
     {
          publication::getdata();
          cout<<"Enter the time:";cin>>time;
     }
     void putdata()
     {
     publication::putdata();
     cout<<"\nTime is:"<<time;
     }
};
void main()
{
     book b1;
```

```
     CDROM c1;
     b1.getdata();
     c1.getdata();

     b1.putdata();
     c1.putdata();
     getch();
}
```

This is the solution of the Question:

Q: A company that makes both books and CR-ROM version of its multimedia works. Create a class called publication which stores the title and price of publications. From this class derive another two class book, which adds a page count and CD-ROM which adds a playing time in minutes. Each of these classes should have some member functions to get the data from keyboard and display the data.

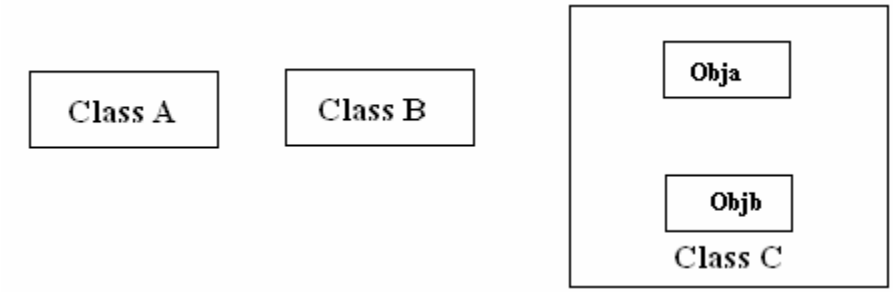
Write a main routine to create objects and storing the data. The main routine also displays the data.

Containership:

Inheritance is the mechanism used to obtain the "a kind of" relation. That is, if class B is derived from a class A, then "B is a kind of A". C++ supports another form of relationship, called containership. E.g. books are made of chapters, ocean have whales, dolphins etc. it allows us to view an object as a collection of another objects.

```
class  A {     };
class B { };
class C
{
     A obja;        //obja is objects of class A
     B objb;   //objb is objects of class B
     --------
     --------
};
```

Since, the class C contains instance of class A and class B, containership can be viewed as nesting of class i.e. class within class. And often referred to as "has a" relationship.



```
//Example of containership:
class A
{
     int a;
     public:
     A(){ a = 0; }
     A(int x) { a = x; }
     Void display()
     {
          cout<<”a = ”<<a;
     }
};
class B
{
     int b;
     public:
     B(){ b = 0; }
     B(int x) { b = x; }
     Void display()
     {
          cout<<”b = ”<<b;
     }
};
class C
{
     A obja;
     B objb;
     int c;
     public:
     C():obja(),objb()
     {
          c = 0;
     }
     C(int x1,int x2,int x3):obja(x1),objb(x2)
     {
          c = x3;
     }
     void display()
     {
          Obja.display();
          Objb.display();
          Cout<<”c = ”<c;
     }
};
void main()
{
     C C1;
```

```
      C C2(2,3,4);
      C1.display();
      C2.display();
}
```

Abstract Classes

The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. It is only designed to be inherited.

```
class One
{
      private:
            ……….
      public:
            ……….
};
class Two
{
      public:
            ……….
};

void main( )
{
      Two t1;
}
```

The class One serves as framework for building the derived class and it is treated as a member of the derived class Two. The instance of class One is not created in the function main( ), however it provides a framework for the class Two, so class One can be regarded as Abstract class.

9. Functions

Abstract Classes

The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. It is only designed to be inherited.

```
class One
{
      private:
            ……….
      public:
            ……….
};

class Two
{
      public:
            ……….
};

void main( )
{
      Two t1;
}
```

The class One serves as framework for building the derived class and it is treated as a member of the derived class Two. The instance of class One is not created in the function main( ), however it provides a framework for the class Two, so class One can be regarded as Abstract class.

Friend function:

The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members. This policy is, if you are not a member you cannot get it. Sometimes this feature leads to considerable inconvenience in programming. If we want to use a function to operate on objects of two different classes, then a function outside a class should be allowed to access and manipulate the private members of the class. In C++, this is achieved by using the concept of friend function.

Private member of a class cannot be accessed from outside the class. Non member function of a class cannot access the member of a class. But using friend function we can achieve this.

The function declaration must be prefixed by the keyword friend whereas the function definition must not. The function could be defined anywhere in the program similar to any normal C++ function. Function definition does not use either the keyword friend or scope resolution operator **::**

A friend function is not a member of any classes but has the full access to the member of class within which it is declared as friend.

```
Class test
 {
      ……
      …….
      …….
  Public:
      ……….
      ……….
      friend void friendfunc();//declaration
};
```

A friend function has the following characteristics.

- It is not in the scope of the class within 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 C++ function without the help of any object.
- Unlike member functions, it cannot access member name directly and has to use an object name and dot operator with each member name. i.e. t1.x.
- It can be declared as public or private part of the class, the meaning is same.
- Normally, it takes objects as arguments.

Friends as bridges

If we want to operate on objects of two different classes, the function may take objects of two classes as arguments, and operate on their private data. For this we have to make use of friend functions that can act as bridge between two classes.

```
// Bridging classes with friend fuctions
#include<iostream.h>
#include<conio.h>
class second; //declaration like function prototype
class first
{
      private:
            int data1;
      public:
            void setdata(int x)
            {     data1=x;}
            friend int sum(first a, second b);//friend function
};
class second
{
      private:
            int data2;
   public:
      void setdata(int x)
      {     data2=x;}
      friend int sum(first a, second b);//friend function
};

int sum(first a, second b)
{
      return (a.data1 + b.data2);
}
```

```
void main()
{
      first a;
      second b;
      a.setdata(15);
      b.setdata(10);
      cout<<"sum of first and second is:"<<sum(a, b);//displays 25
      getch();
}
```

Example

```
// an example of friend function
#include<iostream.h>
#include<conio.h>
class time
{
      private:
            int hrs,min;
      public:
            void gettime(int h,int m)
            {hrs=h;min=m;}
            void puttime()
            {cout<<hrs<<":"<<min;}
            friend time sum(time, time);//return object of time

};
//definition  of sum.
time sum(time t1, time t2)
{
      time t3;
      t3.min=t1.min+t2.min;
      t3.hrs=t3.min/60;
      t3.min=t3.min%60;
      t3.hrs+=t1.hrs+t2.hrs;
      return t3;  //returns object t3;
}
void main()
{
      time T1,T2,T3;
      T1.gettime(2,56);
      T2.gettime(3,45);
      T3=sum(T1, T2);//friend function
      cout<<"now the time value is"<<endl;
      T1.puttime(); cout<<"+";
      T2.puttime(); cout<<"=";
      T3.puttime();
      getch();
}
```

Friend Classes

The member function of a class can all be made friends at the same time when we make the entire class a friend.

```
// an example of friend class
#include<iostream.h>
#include<conio.h>
```

```
class first
{
      private:
            int data1;
      public:
            void setdata(int x)
            {     data1=x;}
            friend class second; //class second can access private data
};
class second
{
   public:
      void func1(first a)
      {cout<<"\n data1="<<a.data1;}

      void func2(first a)
      {cout<<"\n data1="<<a.data1;}
};

void main()
{
      first a;
      second b;
      a.setdata(15);
      b.func1(a);
      b.func2(a);
      getch();
}
```

Abastract classes and pur virtual function:

When the objects of base class are never instantiated, such a class is called abstract base class or simply or simply abstract class. Such a class only exists to act as a parent of derived classes from which objects are instantiated. It may also provide interface for class hierarchy.

To make a class abstract so that object instantiation is not allowed and derivation of child classes are allowed, at least one pure virtual function should be placed in the class.

A pure virtual class is one with the expression = O is added to the declaration of virtual function. The syntax of declaration of pure virtual function and making a class abstract is:

```
class A
{
public:
virtual void show() = 0 ;     // pure virtual function
};
```

Here class A become abstract since there is presence of pure virtual function show (). The expression = 0 has no any other meaning the equal sign = 0 does not assign 0 to function show (). It is only method to tell the compiler that show () is pure virtual function hence class A is abstract class.

All classes with pure virtual function are known as concrete classes.

A pure virtual function has following properties.

1. A pure virtual function has no implementation in the base class.
2. It acts as an empty bucket (virtual function is partially filled bucket) that the derived class are supposed to fill it.
3. A pure virtual function can be invoked by its derived class.

An example:

```
#include<iostream.h>
  class Absperson
  {
     public:
     virtual void service1(int); // normal virtual function
     virtual void service2(int)=0; //pure  virtual function
   };
void Absperson :: service1 (int n)
{    service2 (n);
}
class person: public Absperson
{
     public:
     void service2 (int n) ;
};
void person :: service2 (int n)
{
cout <<" The no.of.years of service:" << (60-n) << endl;
}
void main ( )
{
     person father, son;
     father. service1 (50);
     son. service2 (20);
}
```

output:

```
     The no. of years of service: 10
     The no. of years of service: 40
```

Example 2

```
#include<iostream.h>
  class Base
{
     public:
     virtual void show()=0 ;  // pure virtual function
};
class Derived1 : public Base
{
     public :
     void show() { cout <<" Derived1 \n"; }
```

```
};
class Derived2 : public Base  // derived class 2
{
     public:
     void show() { cout <<" Derived2 \n" ; }
};
void main()
{
     // Base baseobj ;   //  can't  make  object  of  abstract
class
     Base * ptr[2] ;          //  array  of  ptr  of  base
class.
     Derived1 dv1 ;           // object of derived1
     Derived2 dv2 ;           //  object of derived2.
     ptr[0] = &dv1 ;
     ptr[1] = &dv2;

     ptr[0]->show();
     ptr[1]->show();
}    output : Derived1
              Derived2
```

The pure virtual function in the base class must be override in all its derived class from which we want to instantiate objects. If a class doesn't override pure virtual function, it itself becomes abstract and objects cannot be instantiated.

Virtual destructors: Since destructor are member functions, they can be made virtual with placing keyword virtual before it. The syntax is

Virtual ~ classname ( ) ; // virtual destructor.

• The destructor in base class should always be virtual. If we use **delete** with a base class object to destroy the derived class object, then it calls the **delete** calls the member function destructor for base class. This causes the base class object to be destroyed. Hence making destructor of base class virtual, we can prevent such mis-operation.

Example:

```
#include<iostream.h>
class Base
{
 public:
        ~Base ( ) ;      // non virtual
     //virtual ~Base()
     {cout<<"Base Destryed\n";}
};
 class Derv1:public Base
 {
     public:
```

```
    ~Derv1()
    {
    cout<<"Derived1 destroyed\n";
    }
};
class Derv2:public Base
{
    public:
    ~Derv2()
    {
    cout<<"Derived2 destroyed\n";
    }
};

void main()
{
 Base * pBase = new Derv1;
 delete pBase;
}
    The output for it is :
         Base destroyed.
```

- pBase stores address of object of Derv1 class.
- Delete pBase destroy the Base object i.e. calls the destructor of base class.
- If the destructor is made virtual by the line virtual ~Base ( ) ;

then,

```
delete pBase ;
```

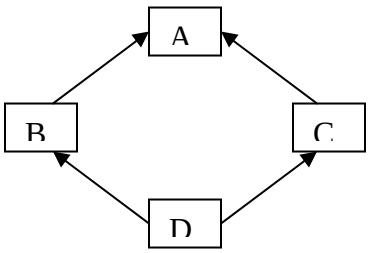
Simply calls the destructor of Derv class first and the output is now

Derv destroyed.
Base destroyed.

}

Virtual Base class:

In multiple inheritance, if a base class parent derives its two child class then another class is derived from two child, as



When member function of class D want to access data member of parent class A, then problem arises due to ambiguity.

To resolve such ambiguity we use virtual base class.

A virtual base class is one from which classes are derived virtually. as

```
class A.
{    // body of class A
};
class B: virtual public A
{
// Body of B
};
```

```
class C:virtual public A
{    // Body of class C
};
class D: public B; public C.
{
};
```

Example:

```
class parent
{ protected:
int basedata;
};
class grandchild: public  child1, child2
{    public:
     int getdata()
     { return basedata; }     // Error: ambiguous
};
```

```
class child1: public parent
{  };
class child2: public parent
{ };
```

When the member function of grandchild attempts to access base data in parent, each $child_1$ and $child_2$ inherits the copy of basedata. Since grandchild class is derived from both $child_1$ and $child_2$, so attempting to access base data becomes ambiguous in grandchild.

This ambiguity is overcomed by making virtual base class as

```
class child1: virtual public parent
{   };
class child2: virtual public parent
{  };
```

The use of virtual in these two class causes them to share a single common copy of base data. So attempt to access base data in grandchild is not ambiguous.

```
#include <iostream.h>
#include <conio.h>
class student
{
     protected:
     int roll;
     public:
     void getno(int a)
     {
          roll=a;
     }
     void putno()
     {
          cout<<"\nRollNumber is:"<<roll;
     }
};
class test:virtual public student
{
     protected:
     float part1,part2;
     public:
     void getmark(float a, float b)
```

```
      {
            part1=a;
            part2=b;
      }
      void putmark()
      {
            cout<<"\nPart1="<<part1;
            cout<<"\nPart2="<<part2;
      }
};
class sport:virtual public student
{
      protected:
      int score;
      public:
      void getscore(int a)
      {
            score=a;
      }
      void putscore()
      {
            cout<<"\nScore:"<<score;
      }
};
class result:public test, public sport
{
      float total;
      public:
      void display()
      {
            total=part1+part2+score;
            putno();
            putmark();
            putscore();
            cout<<"\nTotal Score:"<<total;
      }
};
void main()
{
      clrscr();
      result student1;
      student1.getno(999);
      student1.getmark(25,54);
      student1.getscore(7);
      student1.display();
      getch();
}
```

//Write a program to that uses a new and delete for dynamic

//Memory allocation to calculate the product of two matriceS.

```
#include <iostream.h>
#include <conio.h>
class matrix
{
        int mrow,mcol;
        int *ptr;
        public:
        matrix(int r, int c)
        {
                mrow=r;mcol=c;
                ptr=new int [r*c];
        }
        void getdata()
        {
                int i,j,mat_off,temp;
                cout<<"\nEnter elements matrix:\n";
                for(i=0;i<mrow;i++)
                {
                        for(j=0;j<mcol;j++)
                        {
                                mat_off=i*mcol+j;
                                cin>>ptr[mat_off];
                        }
                }
        }
        void print()
        {
                int i,j,mat_off;
                for(i=0;i<mrow;i++)
                {
                        cout<<"\n";
                        for(j=0;j<mcol;j++)
                        {
                                mat_off=i*mrow+j;
                                cout<<"\t";
                                cout<<ptr[mat_off];
                        }
                }
        }
        matrix operator *(matrix b)
        {
                matrix c(b.mcol,mrow);
                int i,j,k,mat_off1,mat_off2,mat_off3;
                for(i=0;i<c.mrow;i++)
                {
                        for(j=0;j<c.mcol;j++)
                        {
                                mat_off3=i*c.mcol+j;
                                c.ptr[mat_off3]=0;
                                for(k=0;k<b.mrow;k++)
                                {
                                        mat_off2=k*b.mcol+j;
```

```
                                        mat_off1=i*mcol+k;
                                        c.ptr[mat_off3]+=ptr[mat_off1]+b.ptr[mat_off2];
                                }
                        }
                }
                return c;
        }
};
void main()
{
        clrscr();
        int rowa,cola,rowb,colb;
        cout<<"Enter the dimensional of matrix A";
        cin>>rowa;
        cin>>cola;
        matrix a(rowa,cola);
        a.getdata();

        cout<<"Enter the dimensional of matrix B";
        cin>>rowb;
        cin>>colb;
        matrix b(rowb,colb);
        b.getdata();

        matrix c(cola,rowb);
        c=a*b;
        cout<<"\nThe product of two matrix=";
        c.print();
        getch();
}
```

10. Namespace

A program includes many identifiers defined in different scopes. Sometimes an identifier of one scope will overlap (i.e. collide) with an identifier of the same name in a different scope, potentially creating a problem. Identifier overlapping also occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions).

To solve this problem, we use the concept of *namespace*. Each namespace defines a scope where identifiers are placed. The general form namespace is:

```
namespace namespace_name {
        Namespace body (variables, functions, classes, etc.)
}
```

To use a namespace member, either the member's name must be qualified with the namespace name and a binary scope resolution operator (::), as in

namespace_name :: member

or a **using** statement must occur before the name is used. The general form in this case is

using namespace namespace_name;

In this case, all the members of the namespace namespace-name can be used directly by their names. We can also use only the specified members by using "using statement" as follows

using namespace_name :: member;

One example of the namespace is the C++ standard library (**std**). All classes, functions, and templates are declared within the namespace name std.

Example:

```
#include<iostream>
#include<conio>

namespace TestSpace {
        int m;
        void display(int n) {
                std :: cout<<n;
        }
}

void main() {
        clrscr();
        TestSpace :: m = 8;
        TestSpace :: display(TestSpace :: m);
        getch();
}
```

The same program can be written as follows:

```
#include<iostream>
#include<conio>

using namespace std;

namespace TestSpace {
```

```
        int m;
        void display(int n) {
                cout<<n;
        }
}
using namespace TestSpace;

void main() {
        clrscr();
        m = 8;
        display(m);
        getch();
}
```

We can again write the same program as follows:

```
#include<iostream>
#include<conio>

using std :: cout;

namespace TestSpace {
        int m;
        void display(int n) {
                cout<<n;
        }
}

using TestSpace :: m;

using TestSpace :: display;
void main() {
        clrscr();
        m = 8;
        display(m);
        getch();
}
```

Note: Like in class, we can also declare a function inside the namespace and define it outside. For example,

```
namespace TestSpace {
        int m;
        void display(int);
}

void TestSpace :: display(int n) {
        cout<<"You entered "<<n;
}
```

Nesting Namespaces

A namespace can be nested within another namespace as follows:

```
namespace Outer {
        ……
        ……
        namespace Inner {
```

```
                ……
                ……
        }
        ……
        ……
}
```

Example:

```
#include<iostream>
#include<conio>

using namespace std;
namespace TestSpace {
        int m = 100;
        void display() {
                cout<<m;
        }

        namespace InnerSpace {
                int m = 200;
                void display() {
                        cout<<m;
                }
        }
}

void main() {
        clrscr();
        TestSpace :: InnerSpace :: display();
        getch();
}
```

Output:

200

Alternatively, we can write

```
using namespace TestSpace :: InnerSpace;
display();
```

Or,

```
using TestSpace :: InnerSpace :: display;
display();
```

Unnamed Namespace

An unnamed namespace is one that does not have a name. Unnamed namespace members occupy global scope and are accessible in all scopes following the declaration in the file. The members in this case can be accessed without using any qualification. A common use of unnamed namespace is to protect global data between files.

Example:

```
#include<iostream>
#include<conio>

using namespace std;
```

```
namespace {
        int m;
        void display(int n) {
                cout<<"You entered "<<n;
        }
}

void main() {
        clrscr();
        cout<<"m = ";
        cin>>m;
        display(m);
        getch();
}
```

Using Classes in the Namespace

We can also define classes inside the namespace. For example,

```
#include<iostream>
#include<conio>

namespace NS {
        class rectangle {
                private:
                        int length, breadth;
                public:
                        rectangle(int l, int b) {
                                length = l;
                                breadth = b;
                        }

                        int findarea() {
                                return 2 * length * breadth;
                        }
        };
}

void main() {
        clrscr();
        using namespace std;
        NS :: rectangle r1(5, 3);
        cout<<"Area = "<<r1.findarea();
        getche();
}
```

Output:
Area = 30

Alternatively, we can write
using namespace NS;
rectangle r1(5, 3);
Or
using NS :: rectangle;
rectangle r1(5, 3);

11. Templates

Templates are a mechanism that makes it possible to use one function or class to handle many different data types. By using templates, we can design a single class or function that operates on data of many types, instead of having to create a separate class or function for each type. When used with function they are called function templates and when used with class they are called class templates.

Function Templates

The limitation of function is they can operate only on a particular data type. It can be overcome by defining that function as a function template or generic function. A function template specifies how an individual function can be constructed. Consider the following program:

```
//function overloading
//multiple function with same name
//showing need for template

#include<iostream.h>
#include<conio.h>
int max(int ,int);
long max(long, long);
float max(float,float);
char max(char,char);
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}

int max(int i1, int i2)
{
      return(i1>i2?i1:i2);
}

long max(long l1, long l2)
{
      return(l1>l2?l1:l2);
}

float max(float f1, float f2)
{
      return(f1>f2?f1:f2);
}

char max(char c1, char c2)
{
```

```
      return(c1>c2?c1:c2);
}
```

Above program of multiple max functions are used to find greater value among two, for different data types. This illustrates the need for function templates. The program consists of 4 max functions.

```
      int max(int ,int);
      long max(long, long);
      float max(float,float);
      char max(char,char);
```

Whose logic of finding greater value is same and differs only in terms of data types. The C++ template features enables substitution of a single piece of code for all these overloaded function as follows.

```
template <class T>
T max( T a, T b)
{
      return(a>b?a:b);
}
```

Such functions are known as function templates. When max operation is requested on operands of any data types, the compiler creates a function internally without the user intervention and invokes the same.

A function template is prefixed with the keyword template and a list if template type arguments. The template-type arguments are called generic data types, since their memory requirement and data representation is not known in the declaration of the function template. It is known only at the point of a call to a function template.

```
      Template <class T, …..>
      Return_type function_name(arguments)
      {
            ……..//body of template
            ……..
      }
```

```
//find greater using template
#include<iostream.h>
#include<conio.h>

template <class T>
T max( T a, T b)
{
      return(a>b?a:b);
}
void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
```

```
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    getch();
}
```

Function and Function Template

Function templates are not suitable for handling all data types, so it is necessary to override function templates by using normal function for specific data types. If a program has both the function and function template with the same name, first compiler selects the normal function, if it matches with the requested data type, otherwise it creates a function using a function template.

```
//function with function template
#include<iostream.h>
#include<string.h>
#include<conio.h>

template <class T>
T max( T a, T b)
{
      return(a>b?a:b);
}

//for string data types
char *max(char *a, char *b)
{
      if(strcmp(a,b)>0)
      return a;
   else
      return b;
}

void main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<<max(i1,i2)<<endl;
    long l1=40000, l2=38000;
    cout<<"Greater is "<<max(l1,l2)<<endl;
    float f1=55.05, f2=67.777;
    cout<<"Greater is "<<max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<<max(c1,c2)<<endl;
    char str1[]="apple", str2[]="zebra";
    cout<<"greater is "<<max(str1,str2);
    getch();
}
```

Overloaded Function Templates

The function templates can also be overloaded with multiple declarations. Similar to overloading of normal functions, overloaded function templates must differ either in terms of number of parameters or their type.

```
//overloading function template
```

```
//overloaded function templates
#include<iostream.h>
#include<conio.h>

template <class T>
void print(T data)
{
      cout<<data<<endl;
}
template <class T>
void print(T data, int n)
{
      for(int i=0;i<n;i++)
      cout<<data<<endl;
}

void main()
{
      print(1);  // 1
   print(1.5);    //1.5
   print(420,2);       //420 two times
   print("my Nepal my pride",3); //3 times
   getch();
}
```

Class Templates

Similar to functions, classes can also be declared to operate on different data types. Such classes are called class templates. A class template specifies how individual classes can be constructed similar to normal class specification. These classes model a generic class which supports similar operations for different data types. A generic stack like generic function can be created which can be used for storing data of type integer, floating number, character etc.

```
//implementation of stack class as template
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define max 20

template <class T>
class stack
{
      private:
      T s[max];
      int top;
   public:
       stack()   //constructor
       { top=-1;}

       void push(T x)//put number on stack
       {

                  s[++top]=x;
       }
       T pop()//take number from stack
```

```
       {

            return s[top--];
       }
};

void main()
{
      //for integer data type
      stack <int> s1;

      s1.push(11);
      s1.push(22);
      s1.push(33);
      cout<<"\nNumber Popped:"<<s1.pop(); //33
      cout<<"\nNumber Popped:"<<s1.pop(); //22
      s1.push(44);
      cout<<"\nNumber Popped:"<<s1.pop(); //44
      //for floating point data type
      stack <float> s2;
      s2.push(11.11);
      s2.push(22.22);
      s2.push(33.33);
      cout<<"\nNumber Popped:"<<s2.pop(); //33.33
      cout<<"\nNumber Popped:"<<s2.pop(); //22.22
      s2.push(44.44);
      cout<<"\nNumber Popped:"<<s2.pop(); //44.44

      //for character data type
      stack <char> s3;
      s3.push('A');
      s3.push('B');
      s3.push('C');
      cout<<"\nNumber Popped:"<<s3.pop(); //C
      cout<<"\nCharcter Popped:"<<s3.pop(); //B
      s3.push('D');
      cout<<"\nCharcter Popped:"<<s3.pop(); //D
      getch();
}
```

Write a program in C++ using function template to sort different types of data using any algorithms.

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

template <class t>
void bubble(t sort[], int n)
{       t temp;
     int i,j;
     for(i=0;i<n;i++)
     {
          for(j=0;j<n-i-1;j++)
          {
```

```
          if(sort[j]>sort[j+1])
          {
               temp=sort[j];
               sort[j]=sort[j+1];
               sort[j+1]=temp;
          }
          }
     }
}
void main()
{
     clrscr();
     int arr[]={1,8,5,9,10};
     float num[]={1.5,12.5,3.5,5.5,1.5};
     int i;
     cout<<"Number integer:\n";
     bubble(arr,5);
     for(i=0;i<5;i++)
          cout<<arr[i]<<endl;
     cout<<"Number floating\n";
     bubble(arr,5);
     for(i=0;i<5;i++)
          cout<<num[i]<<endl;
     getch();
}
```

12. Exceptions

Introduction

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing. Exceptions might include conditions such as division by zero, access to an array outside of its bounds, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc.

When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. C++ provides built-in language features to detect and handle exceptions, which are basically runtime errors.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

- Find the problem (**Hit the exception**).
- Inform that an error has occurred (**Throw the exception**).
- Receive the error information (**Catch the exception**).
- Take corrective action (**Handle the exception**).

The error handling code basically consists to two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

Exception Handling Mechanism

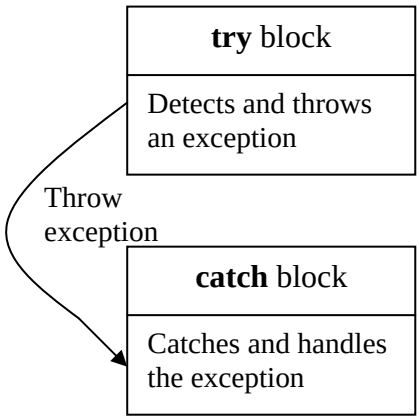
Exception handling mechanism in C++ is basically built upon three keywords: **try**, **throw**, and **catch**. The keyword **try** is used to surround a block of statements, which may generate exceptions. This block of statements is known as *try block*.

When an exception is detected, it is thrown using the **throw** statement situated either in the *try block* or in functions that are invoked from within the *try block*. This is called *throwing an exception* and the point at which the **throw** is executed is called the *throw point*.

A *catch block* defined by the keyword *catch* catches the exception thrown by the **throw** statement and handles it appropriately. This block is also called exception handler. The *catch block* that catches an exception must immediately follow the *try block* that throws an exception.

The figure below shows the exception handling mechanism if try block throws an exception. The general form in this case is:

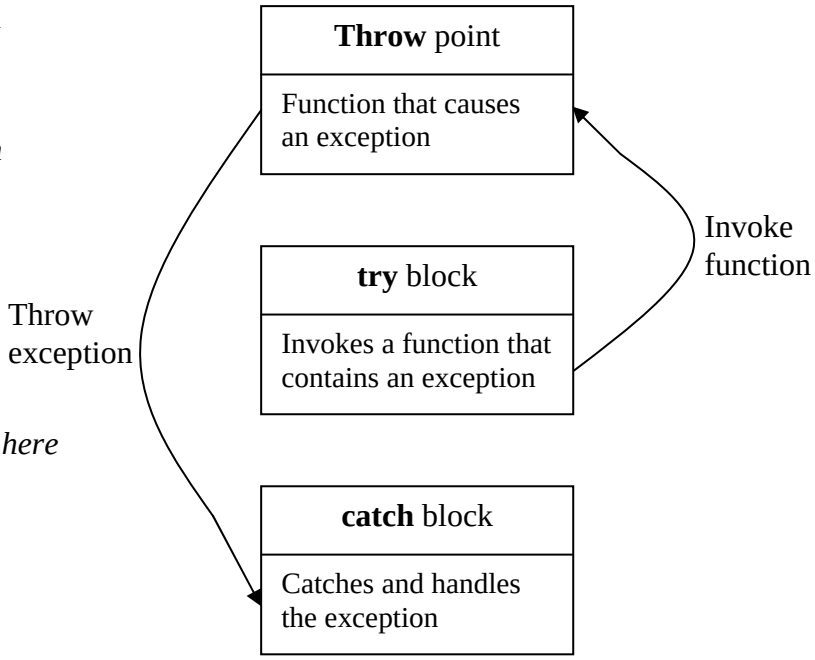
```
……
……
try {
        ……
        throw exception;
        ……
        ……
} catch(type arg) {
        ……
```



```
        ……..
}
……..
……..
```

The figure below shows the exception handling mechanism if function invoked by try block throws an exception.

```
type function(arg list) {
        ……..
        ……..
        throw exception
        ……..
        ……..
}
……..
……..
try {
        ……..
        invoke function here
        ……..
} catch(type arg) {
        ……..
        ……..
}
……..
……..
```



Examples

Example1: Try block throwing exception

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

void main() {
    clrscr();
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
    try {
        if(b == 0)
            throw b;
        else
            cout<<"Result = "<<(float)a/b;
    } catch(int i) {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<"\nEND";
    getche();
}
```

Example2: Function invoked by try block throwing exception

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

void divide(int a, int b) {
    if(b == 0)
        throw b;
  else
        cout<<"Result = "<<(float)a/b;
}

void main() {
    clrscr();
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
     try {
        divide(a, b);
    } catch(int i) {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<"\nEND";
    getche();
}
```

Output:(In both examples)

First Run

Enter the values of a & b:
5
2
Result = 2.5
END

Second Run

Enter the values of a & b:
9
0
Divide by zero exception: b = 0
END

In the first example, if exception occurs in the *try block*, it is thrown and the program control leaves in the *try block* and enters the *catch block*. In the second example, *try block* invokes the function *divide*. If exception occurs in this function, it is thrown and control leaves this function and enters the *catch block*.

Note that, exceptions are used to transmit information about the problem. If the type of exception thrown matches the argument type in the **catch** statement, then only *catch block* is executed for handling the exception. After that, control goes to the statement immediately after the *catch block*.

If they do not match, the program is aborted with the help of **abort()** function, which is invoked by default. In this case, statements following the catch block are not executed.

When no exception is detected and thrown, the control goes to the statement immediately after the *catch block* skipping the *catch block*.

Throwing Mechanism

When an exception is detected, it is thrown using the **throw** statement in one of the following forms:

throw(exception);
throw exception;
throw; //used for rethrowing an exception (discussed later)

The operand *exception* may be of any type (*built-in* and *user-defined*), including constants. When an exception is thrown, the *catch statement* associated with the *try block* will catch it. That is, the control exits the current *try block*, and is transferred to the *catch block* after the *try block*.

Throw point can be in a deeply nested scope within a *try block* or in a deeply nested function call. In any case, control is transferred to the *catch statement*.

Catching Mechanism

Code for handling exceptions is included in *catch blocks*. The general form of catch block is:

```
catch(type arg) {
    Body of catch block
}
```

The *type* indicates the type of exception that *catch block* handles. The parameter *arg* is optional. If it is named, it can be used in the exception handling code. 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. After its execution, the control goes to the statement immediately following the catch block.

If an exception is not caught, abnormal program termination will occur. The catch block is simply skipped if the catch statement does not catch an exception.

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

```
try {
    Try block
} catch(type1 arg) {
    Catch block1
} catch(type2 arg) {
    Catch block 2
}
……
……
} catch(typeN arg) {
    Catch block N
}
```

When an exception is thrown, the exception handlers are searched *in order* for an appropriate match. The first handler that yields a match is executed. After that, the control goes to the first statement after the last **catch** block for that **try** skipping other exception handlers. When no match is found, the program is terminated.

Note: It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Example:

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

void test(int x) {
    try {
            if(x == 0) throw x;
            if(x == 1) throw 1.0;
  }  catch(int m) {
            cout<<"Caught an integer\n";
   } catch(double d) {
            cout<<"Caught a double";
   }
}

void main() {
   clrscr();
   test(0);
   test(1);
   test(2);
   getche();
}
```

Output:

Caught an integer
Caught a double

⇒ **Catch All Exceptions:** If we want to catch all possible types of exceptions in a single **catch** block, we use **catch** in the following way:

```
catch(…) {
    Statements for processing all exceptions
}
```

Example:

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

void test(int x) {
    try {
            if(x == 0) throw x;
            if(x == 1) throw 1.0;
      }  catch(…) {
            cout<<"Caught an exception\n";
      }
}

void main() {
   clrscr();
   test(0);
```

```
    test(1);
    test(2);
    getche();
}
```

Output:
Caught an exception
Caught an exception

Rethrowing an Exception

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block. For example,

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

void divide(int a, int b) {
    try {
        if(b == 0)
                throw b;
        else
                cout<<"Result = "<<(float)a/b;
  }
  catch(int) {
        throw;
  }
}

void main() {
        clrscr();
        int a, b;
        cout<<"Enter values of a & b:\n";
        cin>>a>>b;
        try {
                divide(a, b);
        } catch(int i) {
                cout<<"Divide by zero exception: b = "<<i;
        }
        cout<<"\nEND";
        getche();
}
```

Output:
Similar to first two examples (Try Yourself)

Specifying Exceptions

It is also possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form is as follows:

```
type function(arg-list) throw (type-list) {
        Function body
}
```

The type-list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty. Hence the specification in this case will be

```
type function(arg-list) throw () {
        Function body
}
```

Note: A function can only be restricted in what types of exception it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function (and not within the function).

Example:

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

void test(int x) throw (int, double) {
        if(x == 0) throw x;
         if(x == 1) throw 1.0;
}

void main() {
  clrscr();
  try {
         test(1);
  } catch(int m) {
        cout<<"Caught an integer\n";
  } catch (double d) {
        cout<<"Caught a double";
  }
  getche();
}
```

Output:

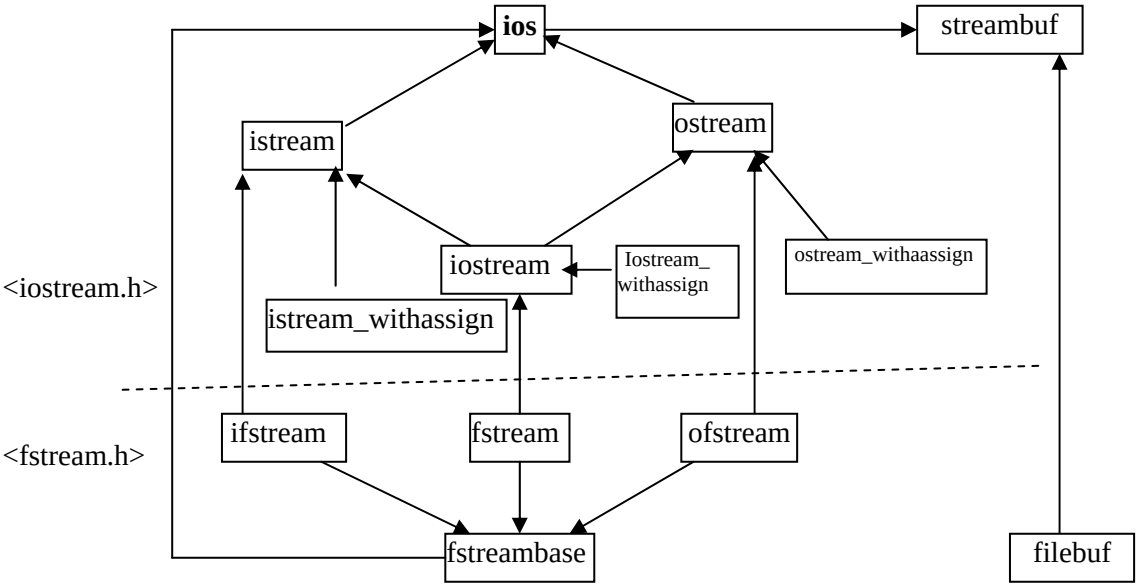
Caught a double

12. Stream Classes (Class hierarchy)

A stream is a name given to flow of data. In C++ stream is represented by an object of a particular class e.g. *cin* and *cout* are input and output stream objects.

There are no any formatting characters in stream like %d, %c etc in C which removes major source of errors.Due to overloading operators and functions, we can make them work with our own classes.

The Stream class hierarchy:



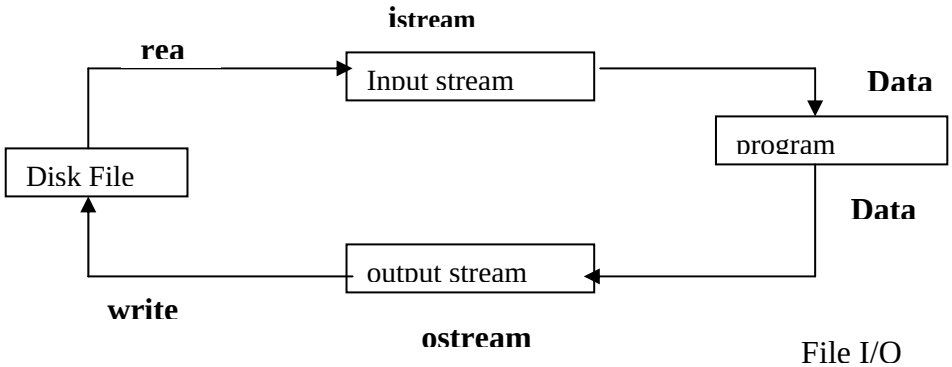
C++ Class Hierarchy

filebuf: The class filebuf sets the file buffer to read and write.

ios: ios class is parent of all stream classes and contains the majority of C++ stream features.

istream class: Derived from ios and perform input specific activities.

ostream class: derived from ios class and perform output specific activities.



File I/O

iostream class: Derived from both istream and ostream classes, it can perform both input and output activities and used to derive iostream_withassign class.

_withassign classes: There are three _withassign classes.

- istream_withassign
- ostream_withassign
- iostream_withasssign

These classes are much like those of their parent but include the overloaded assignment operators.

streambuf: sets stream buffer i.e. an area in memory to hold the objects actual data. Each object of a class associated with the streambuf object so if we copy the stream object it cause the confusion that we are also copying streambuf object. So _withassign classes can be used if we have to copy otherwise not.

fstreambase: Provides operations common to file streams. Serves as a base for fstream, ifstream and ofstream and contains open() and close() functions.

ifstream: Contains input operations in file. Contains open() with default input mode, inherits get(), getline() read(), seekg(), tellg() from istream.

ofstream: Provides output operation in file. Contains open() with default output mode, inherits put(). Seekp(), tellp() and write() from ostream.

fstream: Provides support for simultaneous input and output operations. Contains open() with default input mode: Inherits all the functions of istream and ostream through iostream.

File I/O with stream classes:

In C++,file handling is done by using C++ streams. The classes in c++ for file I/O are ***ifstream*** for input files,***ofstream*** for output files, and ***fstream*** for file used for both input and output operation. These classes are derived classes from ***istream,ostream***,and ***iostream*** respectively and also from fstreambase.

-The header file for *ifstream,ofstream and fstream* classes is <fstream.h>
-To create and write disk file we use ofstream class and create object of it.
e.g. ofstream outf;

The creation and opening file for write operation is done either using its constructor or using ***open()*** member function which had already been defined in ofstream class.

Creating and opening file for write operation is as:

```
ofstream outf("myfile.txt"); //using constructor of ofstream class.
    Or
ofstream outf;
```

```
outf.open(“myfile.txt”);  // using open() member function.
```

Writing text into file:

We use the object of *ofstream* to write text to file created as:

```
outf<<”This is the demonstration of file operation\n”;
outf<<”You can write your text\n”;
outf<<”The text are written to the disk files\n”;
```

An example for writing to disk file.

```
#include<fstream.h>
void main()
     {         //constructor creates file and ready to write
          ofstream outf(“myfile.txt”);

        /* Alternate for above line is
         ofstream outf;     //using open() member function
         outf.open(“myfile.txt”);
       */

          outf<<”File demonstration program\n”;
          //writes strings to file myfile.txt
          outf<<”These strings are written to disk\n”;
    }
```

Writing data to file:

```
int x = 20; float f = 2.5;
char ch = 'c'; char* str = " string";
```

Writing to file is done as

```
Outf<<x<<" " <<f<<' '<<ch<<' '<<str;
```

Example

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

void main()
{
     char ch='c';
     int i= 70;
     float f = 6.5;
     char *str= "Patan";
     ofstream fout("Test.data");
     fout<<ch<<' '<<' '<<f<<' '<<str;
     cout<<"Data written to file\n";
}
```

Reading data from file:

-To read data from file , we use an object of ifstream class File is opened for reading using constructor of ifstream class or open() member function as;

```
ifstream fin("test.txt");           //constructor
or
ifstream fin;
fin.open("test.txt");         // member function open();
Reading data is done as:
Fin>>ch>>i>>f>>str; which is similar as reading data from keyboard
by cin object.
```

String with embedded blanks:

-Require delemeter line \n for each string with embedded blank and read/write operation is easy.

Reading text from file:

To read text from file we use ***ifstream*** class and file is opened for read operation using constructor or open() member function.

```
e.g.  :  ifstream  infile("myfile.txt");  //using constructor
    or
          ifstream infile;
          infile.open("myfile.txt");

  //  Reading from file myfile.txt:
 While(infile) // or while(!infile.eof())   until end of file
 {
   infile.getline(buffer,maxlength); //buffer to be defined as char
                                     //string of length maxlength
   cout<<buffer;            // for display to screen
 }
```

A sample program to read from myfile.txt

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

void main()
{
    const int LEN = 100;
    char  text[LEN];      //for buffer
    ifstream infile("myfile.txt");
    while(infile)  //until end of file Alternate is
                   //while(!infile.eof())
    {
```

```
        infile.getline(text,LEN);    // read a line of text
        cout<<endl<<text;            //display line of text
    }
}
```

Character I/O in file[get() and put() function]

put() and get() functions are members of ostream and istream classes so they are inherited to ofstream and ifstream objects. put() is used to write a single character in file and get is used for reading a character from file.
Example:

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

void main()
{
     char*str="This is a string written to file one char at a time";
     ofstream fout;
     fout.open("myfile.txt");
     for(int i=0;i<strlen(str);i++)]
     {
          fout.put(str[i]);
     }
     cout<<"File write completed";
}
```

// Reading character wise from above file

```
char ch;
ifstream infile;
infile.open("myfile.txt");
while(infile)
{
     infile.get(ch);
     cout<<ch;
}
```

Working with multiple file:

When more than one file is used in a single program for read write operation one file is closed or disconnected from program using close() member function and other file is opened using open().

A sample program:

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

```
void main()
{

ofstream outfile;
//create file district and open for write
outfile.open(“district”);
outfile<<”Kathmandu\n”;
outfile<<”Lalitpur\n”;
outfile<<”Kavreplanchowk\n”;
outfile<<”Dhading\n”;

outfile.close();   // close the file district after writing

outfile.open(“headqtr”);
outfile<<”Kathmandu\n”;
outfile<<”Patan\n”;
outfile<<”Dhulikhel\n”;
outfile<<”Trisuli\n”;
outfile.close();    //closes the file headqtr

//Reading the above files
const int LEN = 80;
char text[LEN];
ifstream infile(“district”);  //opens file district for read
while(infile)
{
  infile.getline(text,LEN);
  cout<<endl<<text;
}
inflie.close();   //closes file district after display

infile.open(“headqtr”);
while(infile)
{
   infile.getline(text,LEN);
   cout<<text;
}
infile.close(); //closes file headqtr
}
```

Writing and reading of user input to the file:

We can also write user-input (values of variables in a program input from keyboard) and read those values by using objects of ofstream and ifstream respectively same as done above . Look at these simple program example.

```
#include<fstream.h>
```

```
void main()
{
    ofstream  fout("test"); //creates and open for writing
    cout<<"Enter the Name:";
    char name[20];
    cin>>name;         //reading from keyboard
    fout<<name<<endl;  //writing to the file "test"
    cout<<"Enter telephohe:";
    int tel;
    cin>>tel;    //reading from keyboard
    fout<<tel;   //writing to file "test"

    fout.close();  //Closes the file "test"
    ifstream fin("test");  //opens the file test for read
    fin>>name;     //reading from file
    fin>>tel;      //reading from file
    cout<<endl<<"The name is: "<<name;
    cout<<endl<<"Telephone no: " <<tel;
    fin.close();
 }
```

Opening file in different mode:

In above example we have used the **ofstream** and **ifstream** constructors or **open()** member function using only one argument i.e. filename e.g. "test" etc. However this can be done by using two argument. One is filename and other is filemode.

Syntax:

Stream-object.open("filename",filemode);

The second argument filemode is the parameter which is used for what purpose the file is opened. If we haven't used any filemode argument and only filename with open() function, the default mode is as:

ios::in for ifstream functions means open for reading only.
i.e. **fin.open("test");** is equivalent to **fin.open("test",ios::in);** as default
ios::out for ofstream functions means open for writing only.
fout.open("test"); is same as **fout.open("test",ios::out);** as default.

Class fstream inherits all features of ifstream and ofstream so we can use fstream object for both input/output operation in file. When fstream class is used , we should mention the second parameter <filemode> with open().
The file mode parameter can take one or more such predefined
constants in **ios** class. The following are such file mode parameters.

Parameters	**Meanings**
ios::app	Append to end of file
ios::ate	Go to end-of-file on opening

ios::binary	Binary file
ios::in	open file for reading only
ios::nocreate	Opens fails if the file does not exists
ios::noreplace	Open files if the file already exists
ios::out	Open file for writing only
ios::trunc	Delete the contents of files if it exits

- Opening file in **ios::out** mode also opens in the **ios::trunc** mode default
- **ios::app** and **ios::ate** takes to the end-of-file when opening but only difference is that **ios::app** allows to add data only end-of-file but **ios::ate** allows us to add or modify data at anywhere in the file. In both case file is created if it does not exists.
- Creating a stream **ofstream** default implies output(write) mode and **ifstream** implies input(read), but **fstream** stream does not provide default parameter so we must provide the mode parameter with **fstream.**
- **The mode can combine two or more parameters using bitwise OR operator (|)**
  e.g. fout.open("test",ios::app|ios::out);

File Pointers:

The file management system associates two types of pointers with each file.

1. get pointer (input pointer)
2. put pointer (output pointer)

These pointers facilitate the movement across the file while reading and writing.

- The get pointer specifies a location from where current read operation initiated.
- The put pointer specifies a location from where current write operation initiated.

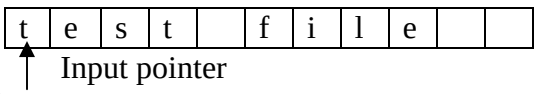
The file pointer is set to a suitable location initially depending upon the mode which it is opened.

• **Read-only Mode**: When a file is opened in read-only mode, the input (get) pointer is initialized to the beginning of the file.

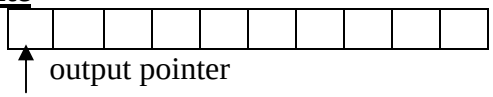
• **Write-only: mode**: In this mode, existing contents are deleted if file exists and put pointer is set to beginning of the file.

• **Append mode**: In this mode, existing contents are unchanged and put pointer is set to the end of file so writing can be done from end of file.

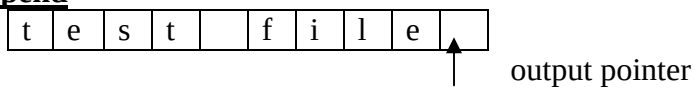
Read



write



Append



Functions manipulating file pointers:

C++ I/O system supports 4 functions for setting a file to any desired position inside the file.

The functions are

Function	member of class	Action
seekg()	ifstream	moves get file pointer to a specific location
seekp()	ofstream	moves put file pointer to a specific location
tellg()	ifstream	Return the current position of the get ptr
tellp()	ofstream	Return the current position of the put ptr

These all four functions are available in fstream class by inheritance. The two seek() functions have following prototypes.

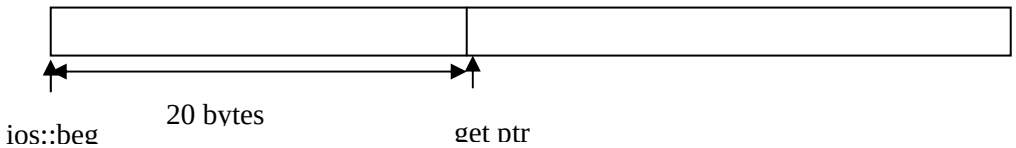
```
istream & seekg (long offset, seek_dir origin =ios::beg);
ostream & seekp (long offset, seek_dir origin=ios::beg);
```

- Both functions set file ptr to a certain offset relative to specified origin. The origin is relative point for offset measurement. The default value for origin is ios::beg.
- (seek_dir) an enumeration declaration given in ios class as

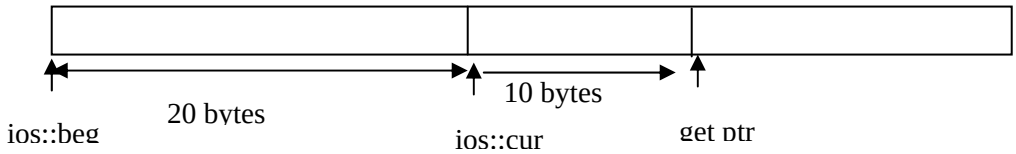
orgin value	seek from
ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

e.g. `ifstream infile;`

`infile.seekg(20,ios::beg);` or `infile.seekg(20);` // default ios::beg move file ptr to 20th byte in the file. The reading start from 21st item [byte start from 0] with file.



Then after, `infile.seekg(10,ios::cur);` moves get pointer 10 bytes further from current position.



Similarly:

```
ofstream  outfile;
outfile.seekp(20,ios::beg);   // out file. seek p (20);
```

moves file put pointer to 20th byte and if write operation is initiated, start writing from 21st item.

Consider following example

```
ofsteam outfile("student",ios::app);
int size=outfile.tellp();
```

Return the size of file in byte to variable size since ios::app takes file put ptr at end of file. The function tellp() returns the takes file put ptr at end of file. The function tellp() returns the current position of put ptr.

Equivalently:

```
ifstream infile("student");
infile.seekg(0,ios::end);
int size=infile.tellg() ;
     This returns the current file pointer position which is at end
of file so we gget he size of fife "student".
```

Some of pointer offset calls and their actions:

Assume: ofstream fout;

Seek	**Action**
fout.seekg(0,ios::beg)	Go to beginning of the file
fout.seekg(0,ios::cur)	Stay at current location
fout.seekg(0,ios::end)	Go to the end of file
fout.seekg(n,ios::beg)	move to (n+1) byte from beginning of file.
fout.seekg (n,ios::cur)	move forword by n bytes from currrent position
fout.seekg(-n,ios:: cur)	move backward by n bytes from currnt position
fout.seekp(n,ios:: beg)	move write pointer (n+1) byte location
fout.seekp(-n,ios:: cur)	move write ptr n bytes backwards.

File I/O with fstrem class

Fstream class supports simultaneous input/output operations. It inherits function from istream and ostream class through iostream.

Following program illustrates this

```
#include<iostream.h>
#include<fstream.h>      //Assume file student.in
#include<conio.h>             //is created with
#include<process.h>           //1. no of student (count)
void main()                   //2. for n students
{
     fstream infile;     // input file name
     fstream outfile;    // output file percentage sane
     int i, count, percentage;
     char name[20];
               //open for read mode
     infile.open("student.in",ios::in);
     if(infile.fail())   // if operation failed.
     {
          cout<<"Error: student.in open fail";
```

```
          exit(1);
     }
          //open next file for write
     outfile.open("student.out",ios::out);
     if(outfile.fail())
     {    cout<<"Error:......"; exit(1);
     }
     infile>>count; // no of student
     outfile<<"     student Information processing" <<endl;
     outfile<<"     ------------------------------" <<endl;
     for(i=0; i<count; i++)
     {         // Read data percentage from input file
          infile>>name;
          infile>>percentage;
               // write in output file.
          outfile<<"Name:"<<name<<endl;
          outfile<<"precentage:"plercentage<<endl;
          outfile<<"passed in:";
          if(percentage>=75)
               outfile<<"first Division/distinction";
          else if(percentage>=45)
               outfile<<" Second Div";
          else if(percentage>=35)
               outfile<<"Passed";
          else
          outfile<<"Failed";
          outfile<<endl;
          outfile<<"....................."<<endl;
     }
                    // close files;
          infile.close();
          outfile.close();
}
```

The put () and get () function:

- The function get() is a member function of the file stream class fstream, and used to read a single character from file.
- The function put() is member function of fstream class and used to write a single character into file.

Example:

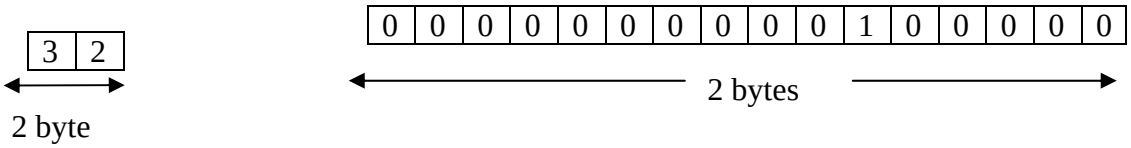
```
#include<fstream.h>
void main()
{
     char c, string[100];
     fstream file("student.txt",ios::in|ios::out);
     cout<<"Enter string:";
```

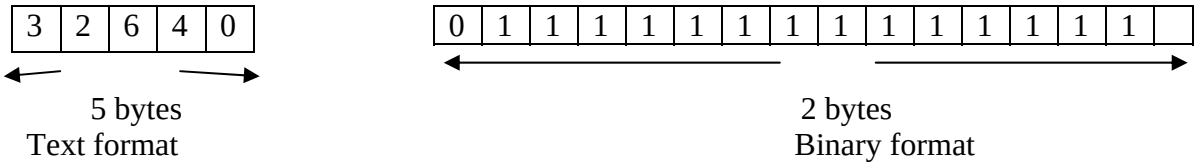
```
     for (int i=0; string[i]!='\0'; i++)
     file.put(string[i]);
     file.seekg(0); // seek to the beging
     cout<<"output string:";
     while(file)
     {
          file.get(c);
          cout<<c;
     }
}
```

The write () and read () function:

- The write () function is a member of stream class fstream and used to write data in file as binary format.
- The read () function is used to read data (binary form) from a file.
- The data representation in binary format in file is same as in system. The no of byte required to store data in text form is proportional to its magnitude but in binary form, the size is fixed.

e.g.





The prototype for read () & write () functions are as:.

```
infile.read((char*)&variable, sizeof(variable));
outfile.write((char*)&variable, sizeof(variable));
```

- The first parameter is a pointer to a memory location at which the data is to be retrieved [read()] or to be written [write()] function.
- The second parameter indicates the number of bytes to be transferred.

Example :writing variable in to files

```
#include<fstream.h>
void main()
{
     int number1=530;
```

```
     float number2=100.50;
//  open file in read binary mode, read integer and class
     ofstream ofile("number.bin",ios::binary);
     ofile.write((char*)&number1, sizeof(number1) );
     ofile.write((char*)&number2, sizeof(float) );
     ofile:close();
//  open file in read binary mode, read integer & close
     ifstream ifile ("number.bin",ios::binary);
     ifile.read((char*)&number1,sizeof(number1));
     ifile.read( (char*) &number2,sizeof(number2));
     cout<<number1<<" "<<number2<<endl;
     ifile.close();

}
```

Object I/O in file

C++ is Object-oriented language so we need objects to be written in file and read from file . Following examples show the I/O operations .

Writing object to disk file:

Generally binary mode is used which writes object in disk in bit configurations.

//example

```
#include<fstream.h>
#include<iostream.h>
class emp
{
     char empname[20];
     int eno;
     float sal;
     public:
     void getdata()
     {
          cout<<"Enter Name:"; cin>>empname;
          cout<<"Enter Emp No:"; cin>>eno;
          cout<<"Enter salary:"; cin>>sal;
     }

};

void main()
{
emp em;
cout<<"Enter the detail of employee"<<endl;
em.getdata();
ofstream fout("emp.dat");
```

```
fout.write((char*)&em,sizeof(em));
cout<<"Object written to file";
}
```

READING FROM FILE:

```
#include<fstream.h>
#include<iostream.h>
class emp
{
     char empname[20];
     int eno;
     float sal;
     public:

     void showdata()
     {
          cout<<"\nName:"<<empname<<endl;
          cout<<"Emp NO:"<<eno<<endl;
          cout<<"Salary:"<<sal<<endl;
     }
};

void main()
{
emp em;
ifstream fin("emp.dat");
fin.read((char*)&em,sizeof(em));
cout<<"Object detail from file";
em.showdata();
}
```

Writing and reading objects:

```
//student.cpp
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class student
{
     char name[20];
     int roll;
     char add[20];
     public:
     void readdata()
     {
          cout<<"Enter name:";cin>>name;
```

```
            cout<<"Enter Roll. no.:";cin>>roll;
            cout<<"Enter address:";cin>>add;
      }
      void showdata()
      {
            cout<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
            <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
      }
};

void main()
{

      student s[5];
      fstream file;
      file.open("record.dat", ios::in|ios::out);
      cout<<"enter detail for 5 students:";
      for(int i=0;i<5;i++)
      {
            s[i].readdata();
            file.write((char*)&s[i],sizeof(s[i]));
      }
      file.seekg(0);//move pointer begining.
      cout<<"Output from file"<<endl;
      cout<<setiosflags(ios::left)<<setw(10)<<"RollNo"
      <<setiosflags(ios::left)<<setw(10)<<"Name"
      <<setiosflags(ios::left)<<setw(10)<<"Address"<<endl;
      for(i=0;i<5;i++)
      {
            file.read((char*)&s[i],sizeof(s[i]));
            s[i].showdata();
      }
      file.close();
}
```

Command Line Arguments

C++ supports the features that facilitates the supply of arguments to the main() function. The arguments are supplied to the main at the time of program execution from command line. The main function takes two arguments. First of which is argument count argc and second is an array of arguments name argv[] as

```
main(int argc, char*argv[] )
```

such program is invoked in command prompt as
C> programname arg1 arg2….

Following example shows the use of command line arguments which reads two different files and display the contents one containing even numbers and another containing odd numbers.

//Program commandline arguments

//evenodd.cpp

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<conio.h>
int main(int argc,char*argv[])
{
    int number[9]={11,22,33,44,55,66,77,88,99};
    if(argc!=3)
    {
        cout<<"argc="<<argc<<endl;
        cout<<"Error in arguments"<<endl;
        getch();
        exit(1);
    }
    ofstream fout1, fout2;
    fout1.open(argv[1]);
    if(fout1.fail())
    {
        cout<<"couldnot open the file"<<argv[1]<<endl;
        getch();
        exit(1);
    }
    fout2.open(argv[2]);
    if(fout1.fail())
    {
        cout<<"couldnot open the file"<<argv[2]<<endl;
        getch();
        exit(1);
    }

    for(int i=0;i<9;i++)
    {
        if(number[i]%2==0)
        fout2<<number[i]<<" ";
        else
        fout1<<number[i]<<" ";
    }
    fout1.close();
    fout2.close();
    ifstream fin;
    char ch;
```

```
     for(i=1;i<argc;i++)
     {
          fin.open(argv[i]);
          cout<<"Contents of "<<argv[i]<<endl;
          while(fin)
          {
               fin.get(ch);
               cout<<ch;
          }
          cout<<endl<<endl;
          fin.close();
     }
     return 0;
 }
```

This program can be invoked in command line as:

C:\evenodd EVEN ODD

The output of program will be;

Contents of EVEN
11 33 55 77 99

Contents of ODD
22 44 66 88

Example 2:

A program that copies contents of a text file to another file

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<conio.h>
int main(int arg,char*argv[])
{
     char *str="This is a test file written and saved into disk for
     copy";
     if(arg!=3)
     {
          cout<<"argc="<<arg<<endl;
          cout<<"Error in arguments"<<endl;
          getch();
          exit(1);
     }
```

```
     ofstream fout;
     fout.open("test");
     fout<<str<<endl;
     fout<<"The contents of string is written into the file"<<endl;
     fout.close();
     fout.open(argv[2]);
     if(fout.fail())
     {
          cout<<"couldnot open the6 file"<<argv[2]<<endl;
          getch();
          exit(1);
     }
     ifstream fin;
     fin.open(argv[1]);
     if(fin.fail())
     {
          cout<<"couldnot open the file"<<argv[1]<<endl;
          getch();
          exit(1);
     }
     while(fin)
     {  char ch;
          fin.get(ch);
          fout<<ch;
     }
     fin.close();
     fout.close();
     cout<<"File "<<argv[1]<<" is "<<"Copied to "<<argv[2]<<endl;
     return 0;
}
```

Execute it as:

```
C:\filecpy test TEST1
```

Output will be

File test is Copied to TEST1;