An Introduction to C++

What is C++? What are the differences between C and C++?

Ans. The C programming language was designed by Dennis Pitcher in the early 1970's (1972 A.D.) at Bell Laboratories. It was first used system implementation language for the nascent Unix Operating System.

C++ was devised by Bjarne Stroustrup in early 1980's (1983 A.D.) at Bell Laboratories. It is an extension of C by adding some enhancements specially addition of class into C language. So, it is also called as superset of C. Initial it was called as C with class. The main difference between C and C++ is that C++ is an object oriented while C is function or procedure oriented. Object oriented programming paradigm is focused on writing programs that are more readable and maintainable. It also helps the reuse of code by packaging a group of similar objects or using the concept of component programming model. It helps thinking in a logical way by using the concept of real world concept of objects, inheritance and polymorphism. It should be noted that there are also some drawbacks of such features. For example, using polymorphism in a program can slow down the performance of that program.

On the other hand, functional and procedural programming focus primarily on the action and events, and the programming model focus on the logical assertions that trigger execution of program code.

Comparison of POP & OOP

Pure OO	Pure Procedural
methods	functions
objects	modules
message	argument
attribute	variable

The first program in C++ is hello.cpp

```
#include <isostream.h>
#main : generate some simple output
 void main( )
 {
      count << "Hello, world" << endl ;
 }
// comment // << operator
// void data type
// output statement ends with a semi colon (;)
// endl manipulator, causes a linefeed to be inserted.
```

"The workers and professionals of the world will soon be divided into two distinct groups: those who will control computers and those who will be controlled by the computers. It would be best for you to be in the former group." – Lewis D. Eigen

Chapter – 1
Overview

1.1 Comparing Procedural Programming & Object Oriented Programming Paradigm

The machine language rest on the concept of electricity being turned 'on' or 'off'. From this on/off, yes/no, two state system, binary system is constructed.

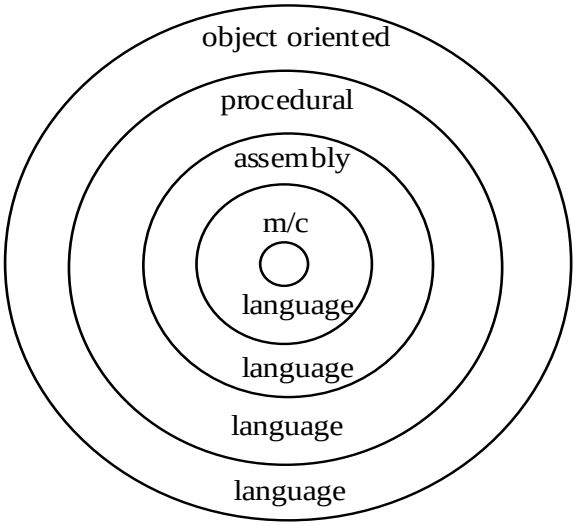


Fig: Language Paradigm

The binary system is based on two digits viz. 0 and 1. M/C language consists of a stream of 0 and 1 having different combination and different meaning. eg. 11110010 01110011 11010010 00010000 01110000 00101011. Assembly language was abbreviation and mnemonic code (codes more easily memorized) to replace the 0s and 1s of machine language. This could be expressed in assembly language (LDA, ADD) statement as

PACK 210(8, 13), 02B(4, 7)

Procedure-Oriented Programming:

Conventional Programming, using high level languages such as COBOL (Common Business Oriented Language), FORTAN (Formula Translation) and C, is commonly called as procedural oriented programming (POP). POP basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as function. A typical program structure for procedural programming is shown

in the figure below. The technique of hierarchical decomposition has been used to specify the task to be completed for solving a problem.

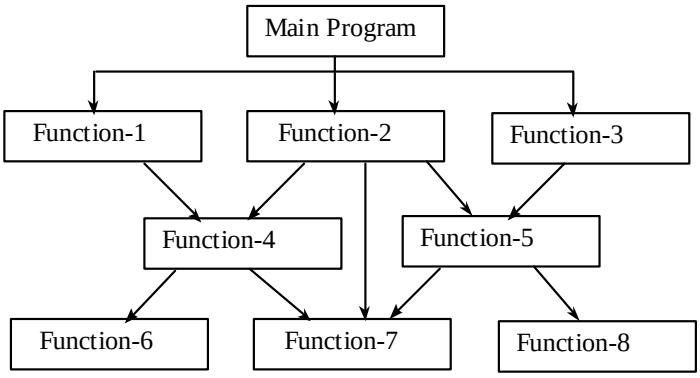


Fig: Typical Structure of Procedure Oriented Programs

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

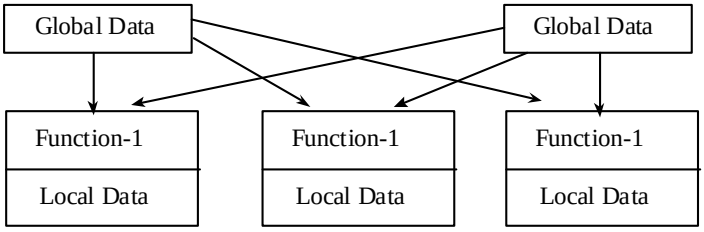


Fig: Relationship of data and functions in procedure programming

Some features of procedure-oriented programming are:

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

Object Oriented Programming Paradigm:

The object oriented approach is to remove some of the flows encountered in the procedure approach. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in the object-oriented programs shown in the figure:

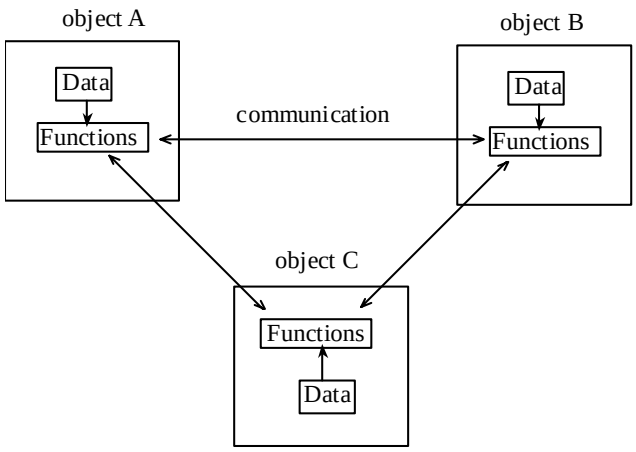


Fig: Organization of data & functions in OOP

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

Some features of object oriented programming are:

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

So, object oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and function that can be used as templates for creating copies of such modules on demand.

1.2 Characteristics of Object-Oriented Language
1.2.1 Objects
1.2.2 Classes
1.2.3 Inheritance
1.2.4 Reusability
1.2.5 Creating new data types
1.2.6 Polymorphism and Overloading

Objects:

Objects are the basic run-time entities in an object-oriented language. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Program

objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory. E.g.

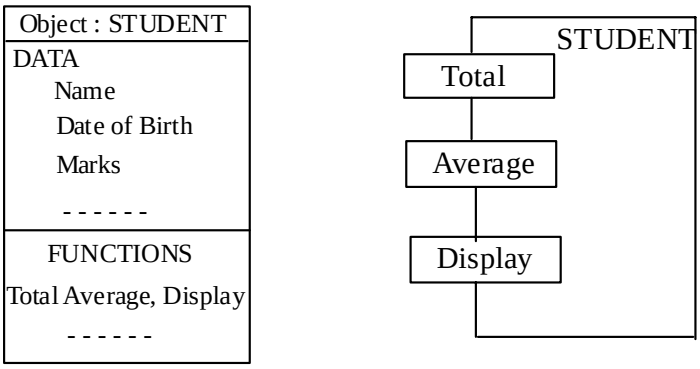


Fig: Two way of representing an object

Classes:

A class is a collection of objects of similar types. Classes are user-defined data types and behaves like the built-in types of a programming language. A class also consists method (i.e. function). So, both data and functions are wrapped into a single class.

Inheritance:

It is the process by which objects of one class acquire the properties of another class. It supports the concept of hierarchical classification. In OOL, the concept of inheritance provides idea of reusability. Eg.

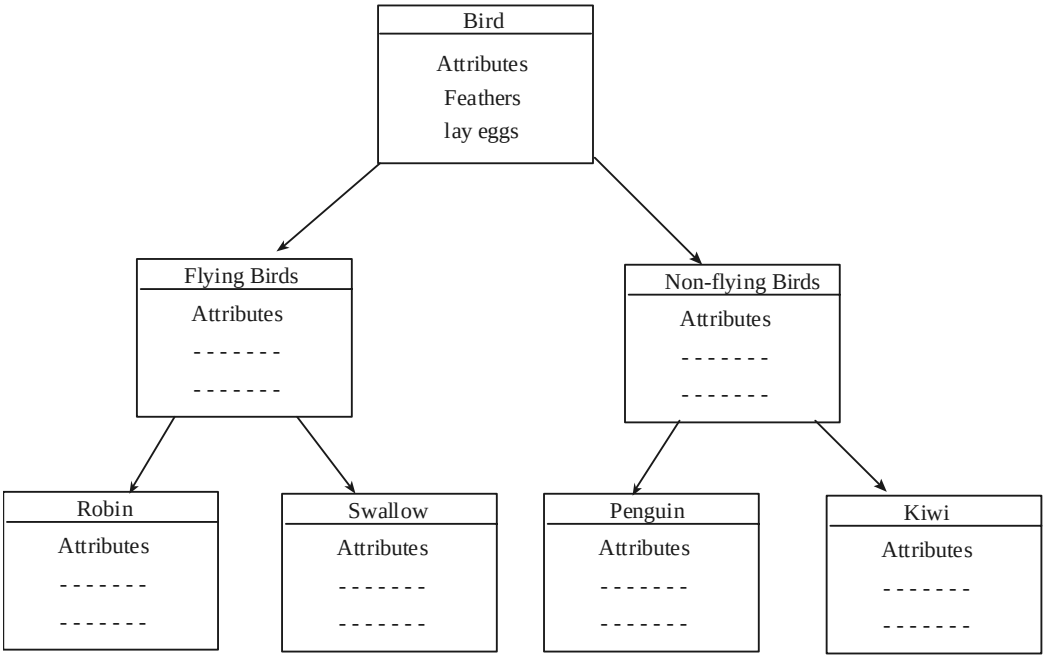


Fig: Property Inheritance

Reusability:

Object-oriented programs are built from reusable software components. Once a class is completed and tested, it can be distributed to other programmers for use in their own programs. This is called reusability. If those programmers want to add new features or change the existing ones, new classes can be derived from existing one. Reusability reduces the time of software development.

Creating new data types:

There are other problems with procedural language. One is the difficulty of creating new data types. Computer languages typically have several built-in data types: integers, floating point number, characters and so on. If you want to invent your own data types, you can. You want to work with complex number, two dimensional co-ordinates or dates, you can create data type complex, date, co-ordinate, etc.

Creating New data types:

One of the benefits of objects is that they give the programmer a convenient way to construct new data types. Suppose, you work with two-dimensional position (such as X & Y co-ordinates, or latitude and longitude) in your program. You would like to express operation on these positional values with normal arithmetic operations, such as

position1=position2+origin ;

where, the variables position1, position2 and origin each represent a pair of independent numerical quantities. By creating a class that incorporates these two values and declaring position1, position2 and origin to be objects of this class, we can, in effect, create a new data types in this manner.

Polymorphism and Overloading:

Polymorphism is another important characteristics of OOL. Polymorphism, a Greek term, means the ability to take more than one form. The process of making an operator to exhibit different behaviours in different instances is known as operator overloading. Using a single function name to perform different types of tasks is known as function overloading. For example,

consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

A single function name can be used to handle different number and different types of arguments as in figure.

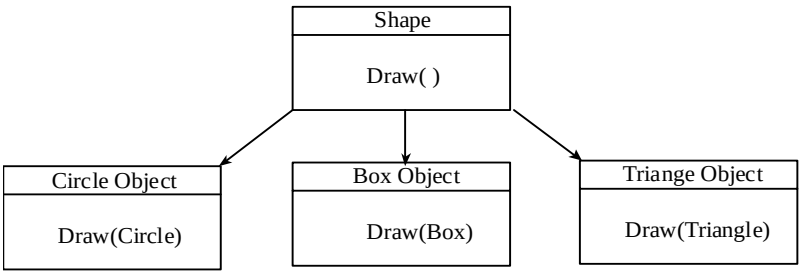


Fig: Polymorphism

All operator overloading and function overloading are examples of polymorphism. Polymorphism is extensively used in implementing inheritance.

1.3 Application & benefits of using OOP

Application of using OOP:

- Real time system
- Simulation and modeling
- Object oriented databases
- Hypertext, hypermedia n/w containing interlinked information and expertext units.
- AI and expert system
- Neural networks and parallel programming.
- Decision support and office automation systems.
- CIM/CAM/CAD system

Notes: CIM stands for Computer Integrated Manufacturing
CAM stands for Computer Aided Manufacturing
CAD stands for Computer Aided Design

Benefits of using OOP:

OOP offers several benefits to both the program designer and the user. Object orientation programming promises greater programmer productivity, better quality of software and lesser maintenance cost. The principle advantages are:

- Through inheritance, we can eliminate redundant code & extend the use of existing classes.
- Reusability saves the development time and helps in higher productivity.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- Object oriented systems can be easily upgraded from small to large system.
- Software complexity can be easily managed.

Chapter – 2
C++ Language Basic Syntax

C++ Language Basic consists of following things:

Token:

The smallest individual units in a program are called as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

C++ tokens are basically almost similar to C tokens except few differences.

Keywords:

The keywords implement specifies C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Many of the keywords are common to both C and C++.

Identifiers and Constants:

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. Identifiers are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and Lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

Example of C++ keywords are

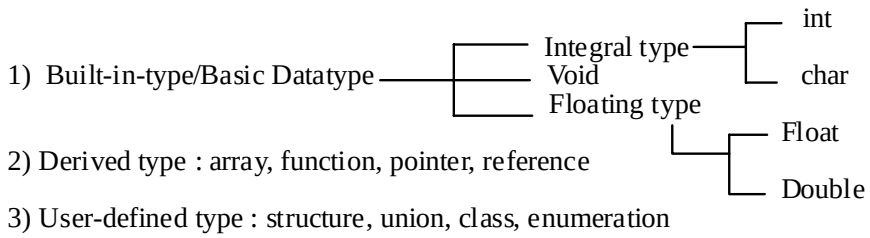
char int float for while switch else new delete etc.

Constants refer to fixed values that do not change during. They include integers, characters, floating point numbers and strings. Constants do not have memory location. Examples are:

```
123      //      decimal
12.37    //      floating point
"C++"    //      string constant
```

Data Types:

Data types in C++ can be classified as:



Built-in-type is also known as basic or fundamental data type. The basic data type may have several modifiers preceding them to serve the needs of various situations except void. The modifier signed, unsigned, long and short may be applied to character and integer basic data types. Long modifier may also be applied to double data type representation in memory in terms of size (Bytes)

Type	Bytes		
char	1	long int	4
unsigned char	1	signed int	4
signed char	1	unsigned int	4
int	2	float	4
unsigned int	2	double	8
signed int	2	long-double	10
short int	2		
unsigned int	2		
signed int	2		

void is a basic data type. It is used mainly for two purposes:

1. To specify the return type of function when it is not returning any value and
2. To indicate an empty argument list to a function. Example:

```
void function1(void) ;
```

User-defined Data types:

Structure and Union are same as in C. Class is a user defined data type takes the keyword class and declaration is as:

```
class class_name
{

}       ;
```

detain in chapter (4)

2.0 Enumeration Data Type:

An enumerated data type is another user-defined type which provides a way for attacking names to numbers, thereby, increasing comprehensibility of the code. The enum keyword automatically enumerates a list of words by assigning them values 0, 1, 2 and so on. This facility provides an alternative means of creating symbolic constants. Eg.

```
enum shape {circle, square, triangle} ;
enum color {red, blue, green, yellow} ;
enum position {off, on} ;
```

In C++, the tag names shape, color, and position becomes new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse ;          //    ellipse is of type shape
color background ;       //    background is of type color
```

Each enumerated data type retains its own separate. This means that C++ doesn't permit an int value to be automatically converted to an enum value. e.g.

```
color background = blue ;       //    allowed
color background = 7 ;          //    Error in C++
color background = color(7) ;   //    ok
```

By default, the enumerators are assigned integers values starting with 0 for the first enumerator, 1 for the second and so on. e.g.

```
enum color(red, blue = 4, green=8) ;
```

enum color(red=5, blue, green) ; are valid definitions. In first case, red is 0 by default. On second case, blue is 6 and green is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

2.1 Derived Data Types

- Arrays, Pointers, function ⟶ Study in chapter-3

2.3 Arrays:

The application of arrays in C++ is similar to that in C. The only except is the way character arrays are initialized. When initializing a character array in C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string[3] = "xyz" is valid in C.
```

It assumes the programmer intends to leave out the null character '\0' in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = "xyz" // o.k.  for C++
data_type array_name[size]
e.g. int marks[30] ;
```

2.4 Pointers:

Pointers are declared and initialized as in C. Examples:

```
int * ip ; // integer pointer
ip = &x ; // address of x assigned to ip
ip = 10 ; // 10 assigned to x through indirection.
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char *const ptr1 = "Good" // constant pointer
```

We cannot modify the address that pointer1 is initialized to

```
int const *ptr2 = &m // pointer to a constant
```

ptr 2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it pointers to not be changed.

We can also declare both the pointer and the variable as constants in the following way: pointers are extensively used in C++ for memory management & achieve polymorphism.

2.5 Const:

The keyword const(for constant), if present, precedes the data type of a variable. It specifies that the value of the variable will not change throughout the program. Any attempt to alter the value of the variable defined with this qualifier will result into an error message from the compiler. Const is usually used to replace defined constant. Variables with this qualifier are often named in all uppercase, as a reminder that they are constants. The following program shows the usage of const.

```
#include <iostream.h>
 void main( )
 {
      float r, a ;
      const float PI=3.14 ;
      cout<<"Enter r:"<< endl ;
      cin>>r ;
      a = PI*r*r ;
      cout<<endl<< "Area of circle="<<a ;
}
```

Scope Resolution Operator (: :)

This operator allows access to the global version of variable. For e.g. it also declare global variable at local place.

```
#include <iostream.h>
 int a=10 ; // global a
void main( )
{
      int a=15 ; // a redeclared, local to main
```

```
    cout << "\n Local a=" <<a<<
      "Global a =" << : : a ;
      : : a=20 ;
    cout<< "\n Local a="<<a
        << "Global a<<: : a ; }
```

Output:

```
    Local a=15      Global a=10
    Local a=15      Global a=20
```

Suppose in a C program there are two variables with the same name a. Assume that one has become declared outside all functions (global) and another is declared locally inside a function. Now, if we attempt to access the variable a in the function we always access the local variable. This is because the rule says that whenever there is a conflict between a local and a global variable, local variable gets the priority. C++ allows you the flexibility of accessing both the variables. It achieves through a scope resolution operator (: :).

2.2 Standard Conversions and Promotions:

Type conversion (often called type casting) refers to changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies. For instance values from a more limited set, such as integers, can be stored in a more compact format and later converted to different format enabling operations not previously possible, such as division with several decimal places, worth of accuracy. In the object-oriented programming paradigm, type conversion allows programs also to treat objects of one type as one of another.

Automatic Type Conversion (or Standard Type Conversion):

Whenever the compiler expects data of a particular type, but the data is given as a different type, it will try to automatically covert. For e.g.

```
int a=5.6 ;
float b=7 ;
```

In the example above, in the first case an expression of type float is given and automatically interpreted as an integer. In the second case, an integer is given and automatically interpreted as a float. There are two types of standard conversion between numeric type promotion and demotion. Demotion is not normally used in C++ community.

Promotion:

Promotion occurs whenever a variable or expression of a smaller type is converted to a larger type.

```
// Promoting float to double
```

```
float a=4 ; //4 is a int constant, gets promoted to float
long b=7 ; // 7 is an int constant, gets promoted to long
double c=a ; //a is a float, gets promoted to double
```

There is generally no problem with automatic promotion. Programmers should just be aware that it happens.

Demotion:

Demotion occurs whenever a variable or expression of a larger type gets converted to smaller type. Bye default, a floating point number is considered as a double number in C++.

```
int a=7.5 // double gets down – converted to int ;
int b=7.0f ; // float gets down – converted to int
char c=b ; // int gets down – converted to char
```

Standard Automatic demotion can result in the loss of information. In the first example the variable 'a' will contain the value 7, since int variable cannot handle floating point values.

2.3 New & Delete Operators:

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

An object can be created by using new and destroyed by using delete as and when required.

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

```
pointer_variable=new data-type ;
```

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

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

where p is a pointer of type int and q is a pointer of type float. Here, p and q must have already declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

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

subsequently, the statements

```
*p=25 ;
```

```
*q=7.5 ;
```

assign 25 to the newly created int object and 7.5 to the float object.

We can also initialize the memory using new operator. This is done as

```
pointer_variable=new data-type(value)
```

For e.g.

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

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

```
pointer_variable=new data_type[size] ;
```

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

```
int *p=new int[10] ; creates a memory space for an array of 10 integers.
```

Delete:

When a data object is no longer needed, it is destroyed to release the memory space for reuse. For this purpose, we use delete unary operator. The general syntax is

```
delete pointer_variable
```

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

```
delete p ;
delete q ;
```

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

```
delete [size] pointer_variable ;
```

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

Control Flow:

1) Conditional Statement : if, if else, nested if else
2) Repetitive Statement : for, loop while, do while
3) Breaking Control Statement : break statement, continue, go to

Manipulators:

Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

The endl manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the new line character "\n".

```
cout<< "m=" <<m <<endl ;
```

2.7 Comments:

C++ introduces a new comment symbol // [double slash]. Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line and whatever follows till the end of the line ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiple comments can be written as

```
// This is an example of
// C++ program to illustrate
// some of its features
```

The C comments symbols / * , * / are still valid and are more suitable for multiple line comments. The above comment is written as:

```
/ *     This is an example of C++ program to illustrate some of its features */
```

cout:

The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++.

The operator << is called insertion (or put to) operator. It inserts the contents of the variable on its right to the object on its left. For e.g.

```
cout << "Number" ;
```

"<<" is the bit-wise left-shift operator. The above concept is called as operator overloading.

cin:

The identifier cin (pronounced as 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyword.

The ">>" operator is known as extraction (or get from) operator. It extracts (or takes) the value form the keyboard and assigns it to the variable on its right. This corresponds to the familiar scanf( ) operation. ">>" is bit-wise right shift operator.

Chapter – 3
Function

Function Definition:

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name (parameter1, parameter2, ..... )
{
        statements
}
```

where,

- type is the data type specifier of data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is block of statements surrounded by braces { }.

e.g.

```
// function example
# include <iostream.h>
 int addition (int a, int b)
 {
       int r ;
       r=a+b ;
       return(r) ;
 }
 int main( )
 {
       int z ;
       z = addition (5,3) ;
       cout << "The result is" <<z ;
       return 0 ;
}
```

Output:

The result is 8.

Function in C++:

An Introduction

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

Syntax of function

```
void show( ) ;  / * Function declaration * /
main ( )
{
      - - - - - -
      - - - - - -
      show( ) ; / * Function call * /
}
void show( ) / * Function definition * /
{
      - - - - - -
      - - - - - - / * Function body * /
      - - - - - -
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and controls return to the main program when the closing brace is encountered. C++ has added many new features to functions to make them more reliable and flexible.

Function Prototyping

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

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

```
type function_name (argument-list) ;
```

The argument_list contains the types and names of arguments that must be passed to the function. E.g.

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

Passing Arguments to Function

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

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

Passing by value:

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

```
#include <iostream.h>
#include<conio.h>
// declaration prototype
void swap(int, int) ;
    void main ( ) {
                    int x,y ; clrscr( ) ;
                    x=10 ;
                    y=20 ;
                    swap(x,y) ;
                    cout << "x =" <<x<<endl ;
                    cout<< "y =" <<y<<endl ;
                    getch( ) ;
    void swap(int a, int b) // function definition
    {
      int t ;
      t=a ;
      a=b ;
      b=t ;
     }
```

Output:

x=10

y=20

Passing by reference:

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

The main advantage of passing by reference is that the function can access the actual variable one value in the calling program. The second advantage is this provides a mechanism for returning more than one value from the function back to the calling program.

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

```
#include <conio.h>
void swap(int &, int &) ;
void main( ) {
                int x,y ;
                x=10 ;
                y=20 ;
                swap(x,y) ;
                cout<< "x=" <<x<<endl ;
                cout << "y="<<y<<endl ;
                getch( ) ;
               }
```

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

Output:

x=20

y=10

Passing by Address or Pointer:

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

```
#include <iostream.h>
#include <conio.h>
void swap(int *, int *) ;
{
      int x, y ;
      x=10 ;
      y=20 ;
swap(&x, &y) ;
cout<< "x="<<x<<endl ;
cout<< "y="<<y<<endl ;
getch( ) ;
}
```

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

Output:

x=20

y=10

Function Overloading

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

To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and / or type of arguments used to call the function.

```
// Program illustrate function overloading
// Function area( ) is overloaded three times
#include <iostream.h>
// Declarations (prototypes)
int area(int) ;
double area(double, int) ;
long area(long, int, int) ;
int main( )
{
        cout<<area(10)<< "\n" ;
        cout<<area(2.5,8)<< "\n" ;
        cout<<area(100L,75,15)<< "\n" ;
        return 0 ;
}
// Function definitions
        int area(int s) // square
          {
                  return(s*s) ;
          }
        double area(double r, int h) // Surface area of cylinder ;
          {
                  return(2*3.14*r*h) ;
          }
        long area(long l, int b, int h) //area of parallelopiped
          {
                  return(2*(l*b+b*h+l*h)) ;
          }
```

Output:

100
125.6
20250

Default Arguments

When declaring a function, we can specify a default value for each parameter. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is

called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead. For e.g.

```
// default values in function
#include <iostream.h>
int main( )
{
        int divide(int a, int b=2) ; //prototype
                                        // b=2 default value
        cout<<divide(12) ;
        cout<<endl ;
        cout<<divide(20,4) ;
        return 0 ;
}
        int divide(int x, int y)
{
        int r ;
        r=x/y ;
        return(r) ;
}
```

In-Line Functions

In C++, it is possible to define functions that are not actually called but, rather are expanded in line, at the point of each call. This is much the same way that a c like parameterized macro works.

The advantage of in-line functions is that they can be executed much faster than normal functions.

The disadvantage of in-line functions is that if they are too large and called to often, your program grows larger. In general, for this reason, only short functions are declared as in-line functions.

To declare an in-line function, simply precede the function's definition with the inline specifier. For e.g.

```
// Example of an in-line fucntion
#include <iostream.h.>
 inline int even(int x) {
              return! (x%2) ; }
          int main( ) {
                            if (even(10)) cout<< "10 is even \n" ;
                            if (even(11)) cout<< "11 is even \n" ;
                            return 0 ;
                        }
```

Output: 10 is even.

Chapter – 4
Classes and Objects

4.1 Introduction

A class is a user-defined data type which holds both the data and function. The data inside the class are called member data and functions are called member function. The binding of data and functions together into a single class type variable is called encapsulation, which is one of the benefits of object-oriented programming.

The general form of declaring a class is

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

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

```
class_name     object_name ;
```

4.2 Class Specification

```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream.h>
class smallobj
{
    private:                //specify a class
      int somedata ;        // class data
    public:
      void setdata(int a)   // member function
```

```
    { some data=d ;}        // to set data
    void showdata( )        //member function to display data
    {
      cout<< "\n Data is" <<somedata ; }
    } ;
void main( )
{
    smallobj s1, s2 ; // define two objects of class smallobj
    s1. setdata(1066) ; // call member function to set data
    s2. setdata(1776) ;
    s1. showdata( ) ;
    s2. showdata( ) ;
}
```

Output:

Data is 1066 ← Object s1 displayed this

Data is 1776 ← Object s2 displayed this

The specifier starts with keyword class, followed by the name smallobj in this example. The body of the class is delimited by braces and terminated by a semicolon. The class smallobj specified in this program contains one data member item and two member functions. The dot operator is also called "class member access operator."

4.2 Data encapsulation (public, protected, private modifiers)

The binding of data and functions together into a single class type variable is called data encapsulation. There are 3 types of encapsulation modifier. They are ;

1. Public 2. Protected 3. Private are also called as visibility labels.

The key feature of object oriented programming is data hiding. The insulation of the data from direct access by the program is data hiding or information hiding.

Public data or functions are accessible from outside the class.

Private data or functions can only be accessed from within the class.

Protected data or functions are accessible from outside the class in limited amount.

(described in inheritance chapter)

e.g.

```
#include <iostream.h>
 class rectangle
    { private:
            int len, br ;
      public:
            void getdata( )
      {
```

```
                cout<<endl<< "Enter length and breadth" ;
                cin>>len>>br ;
          }
 void setdata (int l, int b)
        {
                cout<<endl<< "length=" <<len ;
                cout<<endl<< "breadth="<<br ;
        }
void area_Peri( )
{
        int a, p ;
        a=len*br ;
        p=2*(len+br) ;
        cout<<endl<< "area=" <<a ;
        cout<<endl<< "perimeter"<<p ; }
} ;
void main( )
{
        rectangle r1, r2, r3 ; //define three objects of class rectangle
        r1.setdata(10, 20) ; //setdata in elements of the object.
        r1.displaydata( ) ; //display the data set by setdata( )
        r1.area_Peri( ) ; //calculate and print area and perimeter
        r2.setdata(5, 8) ;
        r2.displaydata( ) ;
        r2.area_Peri( ) ;
        r3.getdata( ) ; //receive data from keyboard
        r3.displaydata( ) ;
        r3.area_Peri( ) ;
}
```

Output:

```
length 10
breadth 20
area 200
perimeter 60
length 5
breadth 8
area 40
perimeter 26
enter length and breadth 2   4
length 2
```

breadth 4
area 8
perimeter 12

4.3 Class Objects

The general syntax:

```
class_name      object_name1, object_name2, ....., object_namen ;
rectangle r1, r2, r3 ;← objects of class rectangle
```

4.4 Accessing Class Members:

The private members cannot be accessed directly from outside of the class. The private data of class can be accessed only by the member functions of that class. The public member can be accessed outside the class from the main function. For e.g.

```
class xyz
    {
      int x ;
      int y ;
      Public:
            int z ;
    } ;
- - - - - -
- - - - - -
void main( )
{
    - - - - - - -
    - - - - - - -
      xyz p ;
      p.x=0 ; // error, x is private
      p.z=10 ; // ok, z is public
      - - - - - -
      - - - - - -
}
```

So,

Format for calling a public member data is

```
object_name.member_variablename ;
```

Format for calling a public member function is

```
object_name.function_name(actual_arguments) ;
```

The dot(.) operator is also called as “class member access operator.”

4.5 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Outside the class definition:

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to. The general form of a member function definition is:

```
return_type class_name : : function name(argument declaration)
{
    // function body
}
```

The membership label class_name : : tells the compiler that the function function_name belongs to the class class_name. That is the scope of the function is restricted to the class_name specified in the header line. The symbol : : is called the scope resolution operator. The member function have some special characteristics in outside the class definition.

- Several different classes can use the same function name.
- Member functions can access the private data of the class. A non-member function cannot do so.
- A member function can call another member function directly, without using the dot operator.

```
#include <iostream.h>
 class rectangle
{
  Private: int len, br ;
  Public:
        void getdata( ) ;
        void setdata(int l, int b) ;
 } ;
void rectangle : : getdata( )
{
cout<<endl<< "enter length and
breadth :" ;
cin>>len>>br ;
}
void rectangle : : area_Peri( )
{
        int a, p ;
        a=len*br ;
        p=2*(len+br) ;
        cout<<endl<< "area:"
                <<a ;
        cout<<endl<< "Perimeter"
                <<P ;
}
}
void main( )
{
        rectangle r1, r2, r3 ;
```

```
void rectangle : : setdata(int l, int b)
{
len=l ;
br=b ;
}
void rectangle: : display( )
{
cout<<endl<< "length:"<<len ;
cout<<endl<< "breadth:"<<br ;
}
```

```
r1.setdata(10, 20) ;
r1.display( ) ;
r1.area_Peri( ) ;
r2.setdata(5, 8) ;
r2.display( ) ;
r2.area_peri( ) ;
r3.getdata( ) ;
r3.display( ) ;
r3.area_peri( ) ;
```

Output:

```
length : 10
breadth : 20
area : 200
Perimeter : 60
length : 5
breadth : 8
area : 40
Perimeter : 26
```

```
enter length and breadth : 2    4
area : 8
Perimeter : 12
```

Inside the Class Definition:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an inline function are also application here. Normally, only small functions are defined inside the class definition.

4.6 'this' pointer

The member functions of every object have access to a pointer named this, which points to the object itself. When we call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object.

Using a this pointer any member function can find out the address of the object of which it is a member. It can also be used to access the data in the object it points to. The following program shows the working of the this pointer. For e.g.

```
#include <iostream.h>
class thisemp
 {
      private:
                int i ;
```

```
public:
        void setdata (int num)
        {
                i=num ; //one way to set data
                this → i=num ; //another way to setdata
        }
        void showdata( )
        {
                cout<< "i is" <<i<<endl ; //one way to display data
                cout<< "my object address is" <<this<<endl ;
                cout<< "num is" this→ i ; //another way to display
        }
        } ;
        void main( )
        {
                thisemp e1 ;
                e1.setdata(10) ;
                e1.showdata( ) ;
        }
```

Output:

i is 10
my object address is θ×121bfff4
num is 10

4.7 Static Member of a Class:

Static Data Member:

If we create the data member of a class is static then the properties of that is

1) It is initialized zero and only once when the first object of its class is created.
2) Only one copy of that member is created for the entire class and is shared by all the objects of that class.
3) It is accessible only within the class, but its lifetime is the entire program.

The static variables are declared as follows:

```
class abc
        {
                static int c ;
                - - - - - - -
                public:
                        - - - - - -
                        - - - - - -
```

```
        } ;
```

The type and scope of each static member must be defined outside the class for example for above class abc C is defined as follows:

```
        int abc: : C ;   or
        int abc : : C=10 ;
```

If we write,

```
        int abc : : C ; then C is automatically assigned to zero. But if we write
        int abc : : C=10 ; C is initialized 10.
```

Static variables are normally used to maintain values common to entire class. For e.g.

```
#include <iostream.h>
 class counter
    {
      int n ;
      static int count ; // static member variable
      public:
            void getdata (int number)
            {
              n=number ;
              count + + ; }
            void showcount( ) {
            cout<< "count:"<<count<<endl ; }
            } ;
      int counter : : count ;
      void main( ) {
      counter c1, c2, c3 ; //count is initialized to zero
      c1.showcount( ) ; // display count
      c2.showcount( ) ;
      c3.showcount( ) ;
      c1.getdata(10) ; //getting data into object c1
      c2.getdata(20) ; //getting data into object c2
      c3.getdata(30) ; //getting data into object c3
    cout<< "value of count after calling" ;
    cout<< "getdata function :" <<endl ;
      c1.showcount( ) ; //showcount
      c2.showcount( ) ;
      c3.showcount( ) ;
    }
```

Output:

```
count : 0
count : 0
```

count : 0
value of count after calling getdata function:
count : 3Zcount :3Zcount : 3

Static Member Function:

If a member function is declared static that has following properties:

1) A static function can access to only other static member (static member data and static member function) declared in the same class.
2) A static member function can be called using the same class name(instead of its objects) as

classname : : function_name ;

The static member function declared as follows:

```
class abc
    {
      int n ;
      static int count ;
      public:
            - - - - -
            - - - - -
      static void output( )
      {
            - - - - -
            - - - - - // body of output
      }
    } ;
      int abc : : count=100 ;
```

static member function can be declared as

class_name : : function_name ;

For above example output is called as follows:

```
      abc :  : output( )  ;
```

For e.g.

```
#include <iostream.h>
 class counter
    {
      int a ;
      static int count ;
      public:
    void assign( )
      {
            ++ count ;
```

```
                a = count ; }
        void outputn( )
          {
                cout<< “A :”<<a<<endl ;
          }
        static void output( )
          {
                cout<< “count :”<<count<<endl ;
          }
          } ;
int counter : : count ;
void main( )
{
        counter c1, c2, c3 ;
        // By above statement 0 is assigned
        // to count
        counter : : output( ) ;
        c1.assign( ) ; c2.assign( ) ; c3.assign( ) ;
        counter c4 ;
        c4.assign( ) ;
        counter : : outputc( ) ;
        c1.outputn( ) ; c2.outputn( ) ; c3.output( ) ;
        c4.output( ) ;
}
```

Output:

```
        count : 0
        count : 4
        A : 1
        A : 2
        A : 3
        A : 4
```

4.8 Pointer within a class

Pointers are not used normally as data member and function member within a class. But they are used in a class for implementing constructor and destruction as well as for inheritance especially in base class and derived class. So, we will discuss them in the next chapter.

4.9 Passing Objects as arguments

An entire object can be passed to a function. There are 3 concepts for passing an object to function.

1. Call by value
2. Call by reference
3. Call by address

If we pass object value to function that is called by value, in this case any change made to the object inside the function do not affect the actual objects. In call by reference, objects are passed through reference. If we pass objects as a call by address, we pass the address of the object. Therefore, any change made to the object inside the function will reflect in the actual object.

The object which is used as an argument when we call the function is known as actual object and the object which is used as an argument within the function header is called formal object. For e.g.

```
#include <iostream.h>
#include <conio.h>
 class complex
        {
          float realP ;
          float imagP ;
          Public:
                  void getdata( ) ;
                  void sum(complex c1, complex c2) ;
                  void output( ) ;
        };
void complex :  : getdata( )
        {
          cout<< “Enter real part:” ;
          cin>>realP ;
          cout<< “Enter imag part:” ;
          cin>>imagP ;
        }
void complex : : sum(complex c1, complex c2)
        {
          realP=c1.realP+c2.realP ;
          imagP=c1.imagP+c2.imagP ;
        }
void main( )
{
        complex x, y, z ;
```

```
    clrscr( ) ;
  cout<< “Enter first complex no. :” <<endl ;
    x.getdata( ) ;
  cout<< “Enter second complex no. :”<<endl ;
    y.getdata( ) ;
    z.sum(x, y) ;
  cout<< “First number :” <<endl ;
    x.output( ) ;
  cout<< “Second number :”<<endl ;
    y.output( ) ;
  cout<< “Sum of two numbers :”<<endl ;
    z.output( ) ;
    getch( ) ;
  }
```

Output:

```
  Enter first complex no. :
  Enter real part : 4
  Enter imag part : 2
  Enter second complex no. :
  Enter real part : 3
  Enter imag part : 4
  First number :
          4+i2
  Second number :
          3+i4
  Sum of two numbers :
          7+i6
```

4.10 Returning Objects from Functions

A function cannot only receive objects as arguments but also can return them. For e.g.

```
#include <iostream.h>
class complex
  {
    float realP ;
    float imagP ;
    Public:
          void getdata( ) ;
          complex sum(complex(1) ;
          void output( ) ;
    } ;
```

```
    void complex : : getdata( )
{
    cout<< "Enter real part:" ;
    cin>>realP ;
    cout<< "Enter imag part:" ;
    cin>>imagP ;
}
complex complex : :sum(complex (1)
{ complex temp ;
    temp.realP=c1.realP+realP ;
    temp.imagP=c1.imagP+imagP ;
    return (temp) ;
void complex : : output( ) {
    cout<<realP<< "+i"<<imagP<,endl ;
}
    void main( )
{ complex x, y, z ;
    cout<< "Enter first complex no:"<<endl ;
    x.getdata( ) ;
    cout<< "Enter second complex no.:"<<endl ;
    y.getdata( ) ;
    z=y.sum(x) ;
    cout<< "First number :"<<endl ;
            x.output( ) ;
    cout<< "Second number:"<<endl ;
            y.output( ) ;
    cout<< "Sum of two numbers :"<<endl ;
            z.output( ) ;
}
```

Output:

```
Enter first complex no :
Enter real part : 5
Enter imag part : 4
Enter second complex no. :
Enter real part : 3
Enter imag part : 2
first number : 5+i4
second number : 3+i2
sum of two numbers : 8+i6
```

4.10 Friend Functions & Friend Classes

The outside functions can't access the private data of a class. But there could be a situation where we could like two classes to share a particular. C++ allows the common function to have access to the private data of the class. Such a function may or may not be a member of any of the classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as

```
class ABC
{
      - - - - - -
      - - - - - -
      public :
                - - - - - -
                - - - - - -
                friend void xyz(void) ; //declaration
} ;
```

We give a keyword friend in front of any members function to make it friendly.

```
      void xyz(void)   //function definition
{
      //function body
}
```

It should be noted that a function definition does not use friend keyword or scope resolution operator (: :). A function can be declared as friend in any number of classes. A friend function, although not a member function has full access right to the private members of the class.

Characteristics of a Friend Function:

- It is not in the scope of the class to which it has been declared as friend. That is why, it cannot be called using object of that class.
- It can be invoked like a normal function without the help of any object.
- It cannot access member names (member data) directly.
- It has to use an object name and dot membership operator with each member name.
- It can be declared in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

e.g. 1

```
#include <iostream.h>
class example
        {
                int a ;
```

```
        int b ;
        public:
            void setvalue( )
            {
                a=25 ;
                b=40 ;
            }
        friend float mean (example e) ;
    } ;
        float mean (example e)
    {
        return float (e.a+e.b)/2.0 ;
    }
    int main( )
    {
        example x ; //object x
        x.setvalue( ) ;
        cout<< "Mean value=" <<mean(x)<< "\n" ;
        return 0 ;
    }
```

Output:

Mean value = 32.5

e.g. 2

```
// A Function Friendly to two classes
#include <iostream.h>
class ABC ;    // Forward declaration
class XYZ
{
    int x ;
    public:
        void setdata (int i)
        {
            x=i ; }
        friend void max (XYZ, ABC) ;
        } ;
    class ABC
        {
            int a ;
            public:
```

```
                    void setdata (int i)
                        { a=i ; }
                    friend void max(XYZ, ABC) ; } ;

                    void max(XYZ m, ABC n)    // Definition of friend
                {
                    if (m.x>=n.a)
                        cout<<m.x ;
                    else
                        cout<<n.a ;
                }
                void main( )
                {
                    ABC P ;
                    P.setdata(10) ;
                    XYZ q ;
                    q. setdata(20) ;
                    max(p, q) ;
                }
```

Output:

20

e.g. 3

```
#include <iostream.h> //Program swapping private data of classes
class class2 ;   // Forward declaration
class class1
{
    int value1 ;
    public:
        void indata (int a)
        { value1=a ; }
        void display(void)
        {    cout<<value1<, "\n" ; }
        friend void exchange (class1 &, class2 &) ;
} ;
class class2
{
    int value2 ;
    Public:
        void indata (int a)
```

```
                { value2=a ; }
                void display(void)
                { cout<<value2<< "\n" ; }
                friend void exchange(class1 &, class2 &) ;
} ;
void exchange(class1 &x, class2 &y)
{
    int temp=x.value1 ;
        x.value1=y.value2 ;
        y.value2=temp ;
}
int main( )
{
    class1 c1 ;
    class2 c2 ;
    c1.indata(100) ;
    c2.indata(200) ;
    cout<< "Value before exchange"<< "\n" ;
    c1.display( ) ;
    c2.display( ) ;
    exchange(c1, c2) ; //swapping
    cout<< "Values after exchange"<< "\n" ;
    c1.display( ) ;
    c2.display ( ) ;
    return 0 ; }
```

Output:

```
    values before exchange
      100
      200
    values after exchange
      200
      100
```

Friend class can be done through same process like friend function and we will discuss in next chapter constructor and destructor.

Chapter – 5
Constructors and Destructors

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

A constructor that accepts no arguments (parameters) is called the default constructor. If there is no default constructor defined, then the compiler supplies default constructor.

A constructor is declared and defined as

```
// class with a constructor
class cons
    { int data ;
      public:
            cons( ) ; // constructor declared
            - - - - - -
            - - - - - -
    } ;
    cons : : cons( )  // constructor defined
    { data=0 ;
    }
```

For above example, if we do not define any default constructor then the statement, cons c1; inside the main( ) function invokes default constructor to create object c1.

Characteristics of constructor

1. Constructor has same name as that of its class name.
2. It should be declared in public section of the class.
3. It is invoked automatically when objects are created.
4. It cannot return any value because it does not have a return type even void.
5. It cannot have default arguments.
6. It cannot be a virtual function and we cannot refer to its address.
7. It cannot be inherited.
8. It makes implicit call to operators new and delete when memory allocation is required.

e.g.

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

```
{
        int a, b ;
        public:
                sample( )
                {
                        cout<< “This is constructor” << endl ;
                        a=100; b=200 ; }
                        int add( )
                        { return(a+b) ; }
                } ;
        void main( )
        {
        clrscr( ) ;
        sample s ; //constructor called
        cout<< “Output is:”<<s.add( )<<endl ;
        getch( ) ;
}
```

Output:

This is constructor

Output is 300.

```
#include <iostream.h>
#include <conio.h>
class myclass {
        int a ;
        public:
                myclass( ) ; // constructor
                void show( ) ;
                } ;
        myclass : : myclass( )  {
        cout<< “In constructor \n” ;
        a=10 ;
        }
        void myclass: : show( ) {
        cout<<a ; }
        int main( ) {
                        clrscr( ) ;
                        myclass ob ;
                        ob.show( ) ;
                        return 0 ;
                        getch( ) ;
```

```
}
```

Output:

In construtor : 10

Constructors that take parameters (Parameterized Constructor):

It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments. For e.g.

```
#include <iostream.h>
#include (conio.h>
    // class declaration
        class myclass {
                int a ;
                public:
                    myclass (int x) ; // Parameterized constructor
                void show( ) ;
                } ;
                myclass : : myclass (int x) {
                cout<< "In constructor \n" ;
                a=x ;
                }
                void myclass: : show( ) {
                cout<<a<< "\n" ;
                }
                int main( ) {
                myclass ob(4) ;
                ob.show( ) ;
                return 0 ;
                }
```

Output:

In constructor

4

```
#include <iostream.h>
#include <conio.h>
class xyz
    { int a,b ;
      public: xyz (int a1, int b1) // Parameterized constructor
            { a=a1 ; b= b1 }
      int mul( )
```

```
    { return (a*b) ; }
    } ;
void main( )
    {
        int i,j ;
        clrscr( ) ;
        cout<< "Enter first no. :" ;
        cin>>i ;
        cout<< "Enter second no." ;
        cin>>j ;
        xyz c1(i, j) ;
        cout<< "Multiplication is:" ;
        cout<<c1.mul( )<<endl ;
        getch( ) ; }
```

Output:

Enter first no. : 5
Enter second no. : 6
Multiplication is : 30

Multiple Constructors in a class (Constructor Overloading):

C++ allows us to declare more than one constructor within a class definition. In this case we say that the constructor is overloaded. All of the constructors can have different arguments as required. For e.g.

```
class abc { int m, n, p ;
            public: abc( ) {m=0; n=0; p=0; }
            abc (int m1, int n1)
            {m=m1 ; n=n1 ; p=0 ;}
            abc (int m1, int n1, int p1)
            {m=m1 ; n=n1 ; p=p1 ; }
          } ;
```

In above example, we have declared three constructors in class abc. The first constructor receives no argument, second receives two arguments and third receives three arguments.

We can create three different types of objects for above class abc like

1. abc c1 ;
2. abc c2(10, 20) ;
3. abc c3 (5, 6, 7) ;

Here, object c1 automatically invokes the constructor which has no argument so, m,n and p of object c1 are initialized by value zero(0).

In object c2, this automatically invokes the constructor which has two arguments, so the value of m, n, p are 10, 20, 0.

In object c3, this automatically invokes the constructor which has three arguments, so the value of m, n, p are 5, 6, 7.

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

For e.g.

```
#include <iostream.h>
#include <conio.h.
class complex
        { int x, y ;
        public: complex (int a)
                {x=a ; y=a  ; )
        complex (int a ; int b)
                {x=a ; y=b ; }
        void add (complex c1, complex c2)
                {x=c1.x+c2.x ;
                  y=c1.y+c2.y ; }
        void show( )
        { cout<<x<< "+i"<<y<<endl ; }
        } ;
void main( )
        { clrscr( ) ;
        complex a(3, 5) ;
        complex b(4) ;
        complex c(0) ; c.add(a, b) ;
        a.show( ) ;
        b.show( ) ;
        cout<< "a+b is"<<endl ;
        c.show( ) ;
        getch( ) ;
        }
```

Output:

```
        3+i5
        4+i4
a+b is  7+i9
```

The value can be passed as arguments to constructor in two different ways:

1. By calling constructor explicitly

   e.g. sample A= A(5) ;

2. By calling constructor implicitly
   eg. sample A(5) ;

```
class sample
{ int a ;
public:
        sample (int x)
        { x=a ; }
        } ;
```

Copy Constructor:

Copy constrcutors are used to copy one object to another one. The general declaration for copy constructor is

```
class abc
        { int x, y ;
        public: abc (abc &c1)
        {x=c1.x ; y=c1.y ; }
        abc( ) {x=10 ; y=20 ; }
        } ;
```

We can create objects for above as

1. abc c2 ;
2. abc c3(c2) ; or abc c3=c2 ;

The statement abc c2 ; calls the no argument constructor while the statement abc c3(c2) ; called first constructor that is copy constructor. The statement abc c3=c2 also called copy constructor.

The general syntax of copy constructor header is clas_name (class_name & object_ref). In above example, we have written copy constructor as

```
        abc (abc &c1)
        {x=c1.x ; y=c1.y ; }
```

where abc is name of class ; c1 is reference of object.

E.g. 1

```
#include <iostream.h>
#include <conio.h>
class complex
        { int r, i ;
        public: complex (int a, int b)
          {r=a ; i=b ; }
        complex (complex &c)
          {r=c.r ; i=c.i ; }
```

```
    void show( )
      {cout<<r<< "+i"<<i<<endl ; }
    } ;
    void main( )
    { int x, y ;
    clrscr( ) ;
    cout<< "Enter real part:" ; cin>>x ;
    cout<< "Enter imag part:" ; cin>>y ;
    complex c1(x, y ) ;        // 1st constructor called
    complex c2(c1) ;           // copy constructor called
    cout<<"First no. is: " ;
      c1.show( ) ;
    cout<< "Second no. is:" ;
      c2.show( ) ;
      getch( ) ; }
```

Output:

```
    Enter real part : 5
    Enter imag part : 6
    First no. is : 5+i6
    Second no. is : 5+i6
```

e.g. 2

```
#include <iostream.h>
#include <conio.h>
class cons
    { int data ;
      public: cons (int c)     // Parameterized constructor
                { data=c ; }
                cons (cons &a)          //copy constructor
                { data=a.data ; }
                void display( )
                {cout<<data ; }
                } ;
      void main( ) {clrscr( ) ;
      cons A(5) ;
      cons B(A) ; // or cons B=A ;
      cout<< "\n data in A:" ;
      A.display( ) ;
      cout<< "\n data in B:" ;
      B.display( ) ;
```

```
    getch( ) ;
    }
```

Output:

```
data in A : 5
data in B : 5
```

Constructor with Default Argument:

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

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

In above example, the default value of b is zero i.e. (0 is assigned to y). We can create the following type of objects for above class:

1. complex c1(5)
2. complex c2(5, 6)

If we create object like(1) then 5 is assigned to x and 0 is assigned to y of c1 because default value of y is zero.

If we create object like(2) then 5 is assigned to x and 6 is assigned to y. (A: : AC) is default constructor. A::A(int i =0) is default argument constructor).

Destructors:

The complement of a constructor is the destructor. This function is called when an object is destroyed. For example, an object that allocates memory when it is created will want to free that memory when it is destroyed.

The name of a destroyer is the name of its class preceded by a ~. For a destructor, it never takes any arguments not returns any value. Destructor will automatically be called by a compiler upon exit from the program to clean up storage taken by objects. The objects are destroyed in the reverse order from their creation in the constructor.

e.g. 1

```
#include <iostream.h>
class myclass { int a ;
            public:
                myclass( ) ;     // constructor
               ~ myclass( ) ;    // destructor
                void show( ) ; } ;
myclass: : myclass( ) {
    cout<< "\n constructor \n" ;
```

```
        a=10 ; }
myclass: :~ myclass( ) {
        cout<< “Destructing .... \n” ;
        }
```

e.g. 2

```
#include <iostream.h>
int count=0 ;
class alpha
{ public: alpha( ) {
        cout ++ ;
        cout << “\n No. of objects created”<<count ; }
~ alpha( )
        { cout<< “\n No. of object destroyed”<< count ;
          count ..... ; }
        } ;
void main( )
{ cout << “\n \n Enter main \n” ;
  alpha A1, A2, A3, A4 ;
{ cout<< “\n \n Enter Block1\n” ;
  alpha A5 ; }
{ cout<< “\n \n Enter Block2\n” ;
  alpha A6 ; }
  cout<< “\n \n Re-enter main\n” ; }
```

Output:

```
        Enter main                                  Re- Enter main
No. of objects created1                     No. of objects destroyed4
No. of objects created2                     No. of objects destroyed3
No. of objects created3                     No. of objects destroyed2
No. of objects created4                     No. of objects destroyed1
        Enter Block1
No. of object created5
No. of object destroyed5
        Enter Block2
No. of object created5
No of object destroyed5
```

```
// Pointer within a class
#include <iostream.h>
#include <conio.h>
```

```
#include <string.h>
class stringeg
        { char * str ;
          public:
                  stringeg (char * s) // constructor
        { int len=strlen(s) ;
          str=new char[len+1] ; // use of new operator
          strcpy(str, s) ; }
        ! stringeg( ) { cout<< "\n object destroyed" ; //destructor
                  delete str ; }     // use of delete operator
void display( )
{
        cout<<str ; }
} ;
void main( )
{
        clrscr( ) ;
        stringeg s1= "this is example of pointer" ;
        cout<<endl<< "s1=" ; s1.display( ) ;
        getch( ) ;
        }
Output:
        s1=This is example of pointer object destroyed.
// friclass.cpp
// friend class
#include <iostream.h>
class alpha( )
{ Private:
          int data1 ;
  Public:
          alpha( ) {data1=99 ; } // constructor
          friend class beta ;
        } ;
class beta
{                                   // all member function can access private alpha data
        public:
        void func1(alpha a) {cout<< "\n data1="<<a.data1 ; }
        void func2(alpha a) {cout<< "\n data1="<<a.data1 ; }
        void func3(alpha a) {cout<< "\n data1="<<a.data1 ; }
} ;
```

```
void main( )
{
        alpha a ;
        beta b ;
        b.func1(a) ;
        b.func2(a) ;
        b.func3(a) ;
}
```

Output:

```
data1 = 99
data1 = 99
data1 = 99
```

Chapter – 6
Operator Overloading

6.1 Introduction:

Operator overloading is one of the feature of C++ language. The concept by which we can give special meaning to an operator of C++ language is known as operator overloading. For example, + operator in C++ work only with basic type like int and float means c=a+b is calculated by compiler if a, b and c are basic types, suppose a, b and c are objects of user defined class, compiler give error. However, using operator overloading we can make this statement legal even if a, b and c are objects of class.

Actually, when we write statement c=a+b (and suppose a, b and c are objects of class), the compiler call a member function of class. If a, b and c are basic type then compiler calculates a+b and assigns that to c.

When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.

We can overload all the C++ operators except the following:

1. Scope resolution operator (: :)
2. Membership operator (.)
3. Size of operator (size of)
4. Conditional operator(? :)
5. Pointer to member operator(.*)

Declaration of Operator Overloading:

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

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

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

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

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

when we write

obj ++ ;

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

6.2 Operator Overloading Restrictions

There are two important restrictions to remember when we are overloading an operator:

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

6.3 Overloading Unary and Binary Operators:

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

Types of Operator Overloading:

There are two types of operator overloading:

1. Unary operator overloading
2. Binary operator overloading

Unary Operator Overloading:

As we know, an unary operator acts on only one operand. Examples of unary operators are the increment and decrement operators ++ and --. For e.g.

-- a where 'a' is the only one operand.

```
// Unary operator overloading for pre-fix increment (++) op
#include <iostream.h>
#include <conio.h>
class index
{
    int count ;
    public: index( ) { count = 0 ; }
            void getdata (int i)
            {  count=i ; }
            void showdata( )
            { cout<< "count="<<cout<<endl ; }
```

```
                void operator++( ) ; // for prefix
        } ;
void index: : operator++( )
{ ++ count ; }
void main( )
{ clrscr( ) ;
  index 11
 11.getdata(3) ;
 11.showdata( ) ;
 ++ 11 ;
 11.showdata( ) ; getch( ) ;
```

Output:

count=3

count=4

```
// Unary operator overloading for prefix increment returning
// value through object
#include <iostream.h>
class index
{ int count ;
 public:
        index( ) {count=0 ;}
        void getdata (int i)
        {
                count=i ; }
        void showdata( )
        {
                cout<<count<<endl ; }
        index operator++( ) ;
} ;
index index: : operator++( )
{
      index temp ;
      temp.count=++count ;
      return temp ;
}
int main( )
{     index i1, i2 ;
      i1.getdata(5) ;
      cout<< "count in i1=" ;
```

```
        i1.showdata( ) ;
        i2=++i1 ;
        cout<< “count in i2=” ; i2.showdata( ) ;
        cout<< “count in i1=” ; i1.showdata( ) ;
        return 0 ;
}
```

Output:

```
count in i1 = 5
count in i2 = 6
count in i1 = 6
```

```
// Unary operator overloading for post-fix increment operator
#include <iostream.h>
class index
        { int count ;
        public:
                index( ) {        // default constructor
                count=0 ; }
                void getdata (int c)
                {
                        count=c ; }
                void showdata( )
                { cout<< “count=”<<count<<endl ; }
                void opeator++(int) ; // for post-fix notation
                } ;
void index: : operator ++(int)
        { count++ ; }
void main( )
{
        index a ;
        a.getdata(5) ;
        a.showdata( ) ;
        a++ ;
        a.showdata( ) ;
}
```

Output:

```
count=5
count=6
```

Binary Operator Overloading:

This takes two operands while overloading. For example c=a+b where a and b are two operands. Following program illustrate overloading the + operator.

```
// Program for binary operator overloading for +
#include <iostream.h>
#include<conio.h>
class complex
        { int real ;
          int imag ;
          public: complex( ) { } //default constructor
          complex (int a, int b)  // parameterized constructor
          { real=a ; imag=b ; }
          void show( )
        { cout<<real<< "+i"<<imag<<endl ; }
        complex operator+ (complex C)
        {
          complex temp ;
          temp.real=real+c.real ;
          temp.imag=imag+c.imag ;
          return(temp) ;
        } // the above . function overload binary + operator
        } ;
void main( )
{
        complex c1(5, 4) ;
        complex c2(3, 2) ;
        complex c3(4, 4) ;
        clrscr( ) ;
        cout<< "c1 is:"<<endl ;
          c1.show( ) ;
        cout<< "c2 is:"<<endl ;
          c2.show( ) ;
        cout<< "c3 is:"<<endl ;
          c3.show( ) ;
          c3=c1+c2 ;
        cout<< "Now c3=c1+c2 is:"<<endl ;
          c3.show( ) ;
          getch( ) ;
}
```

Output:
c1 is:
5+i4
c2 is:
3+i2
c3 is:
4+i4
Now, c3=c1+c2 is:
8+i6

Here, complex operator + (complex C) is operator function. When the statement c3=c1+c2 is executed the operator function is called. The operator function is called by object c1 and object c2 is passed as argument to operator function i.e. the c2 object is copied into object c which is written in the header of operator function.

Inside the operator function temp.real=real+c.real ; calculate the sum of real member of object c2 and real member of object c1 and assign this to real of temp object. In above statement real is member of c1 because this function is called by c1 and c.real is member of c2 because c2 is copied into c at the time of call.

Similarly, the statement temp.imag=imag+c.imag is calculaged. The temp is returned to c3 by return(temp) ; statement.

#Q# WAP to concatenate two strings “Sita” and “Ram” and display “SitaRam” as output.

```
#include <iostream.h>
#include <string.h>
class string
    {
      char s[20] ;
      public:
            void get (char *c)
            { strcpy (s, c) ; }
            void show( )
            { cout<<s; }
            string operator+(string x)
            {
                  string temp ;
                  strcpy (temp.s, s) ;
                  strcat (temp.s, x.s) ;
                  return temp ;
            }
            } ;
```

```
void main( )
{ string s1, s2, s3 ;
  s1.get ("Sita") ;
  s2.get ("Ram") ;
  s3.show( ) ;
}
```

Output:

SitaRam

6.4 Operator Overloading using a Friend Function

[When the overloaded operator function is a friend function, it takes two arguments for binary operator and takes one argument for the unary operator.] Already Studied.

```
// Program to add two complex numbers and display the result
#include <iostream.h>
#include <conio.h>
class complex
    { int real ;
      int imag ;
    public:
      complex ( ) { }
      complex (int r, int i)
      { real =r ; imag=i ; }
    void display( )
    { cout<<real<< "+i"<<imag ; }
    Friend complex operator+ (complex x, complex y) ;
    } ;
    complex operator + (complex x, complex y)
    {
      complex temp ;
      temp.real=x.real+y.real ;
      temp.imag=x.imag+y.imag ;
      return temp ; }
void main( )
    { clrscr( ) ;
      complex a(7, 9), b(6, 4) ,c ;
      c=a+b ;
      getch( ) ;
    }
```

Output:

13+i13

```
//assignment operator(= =) operator using friend function
#include <iostream.h>
#include <conio.h>
#include <string.h>
class string
    { char s[10] ;
      int len ;
      public: void get(char c[10]) ;
      { strcpy(s, c) ; }
      void displaylen (char * c)
      { len=strlen(c)  ;
        cout<< "\n Length of"<<c<< "is"<<len ;
      }
        friend int operator = = (string, string) ;
      } ;
    int operator = = (string a, string b)
    { int len1, len2 ;
      len1=strlen(a.s) ;
      len2=strlen(b.s) ;
    if (len1= = len2)
      return1 ;
    else
      return 0 ;
    }
    void main( )
    { string x, y ; clrscr( ) ;
      char s1[10], s2[10] ;
      cout<< "/n Enter two strings:" ;
      cin>>s1>>s2 ;
      x.get(s1) ;
      y.get(s2) ;
    if (x = = y) cout<< "\n Both are equal \n" ;
    else
      cout<< "\n strings re not equal \n" ;
      x.displaylen(s1) ;
      y.displaylen(s2) ;
      getch( ) ;
    }
```

6.5 Data Conversion:

We have already studied standard conversion of different type of basic data type. For example

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

In above example, we are assigning the value of variable y to variable x. To achieve this, the compiler first converts y into integer and then assign it to x. But the compiler does not support automatic(standard) conversion for the user-defined data types like classes. For this, we need to design our own data conversion routines. There are 3 types of data conversion available for user defined classes:

1. Conversion from basic type to class type (object type)
2. Conversion from class type to basic type
3. Conversion from one class type to another class type

Conversion from Basic type to Class type:

The conversion from basic to class type can be done by using constructor. For e.g.

```
#include <iostream.h>
#include <conio.h>
class distance
    { int feet ;
      float inches ;
    public: distance (float mtr)
    { float f=3.28*mtr ;
      feet=int(f) ;
      inches=12*(f-feet) ; }
      void show( )
      { cout<< "distance is"<<endl ;
        cout<<feet<< "\'-"<<inches<<'\"'<<endl ; }
    } ;
void main( )
    { clrscr( ) ;
      float meter ;
      cout<< "Enter a distance:" ;
      cin>>meter ;
      distance d1=meter ; //This call the constructor which convert floating point data
                          // to class type
      d1.show( ) ;
      getch( ) ;
    }
```

Output:

Enter a distance : 2.3
distance is
7’ – 6.527996”

Conversion between objects(class) and basic types:

By the constructor we cannot convert class to basic type. For converting class type to basic type, we can define a overloaded casting operator in C++. The general format of a overloaded casting operator function.

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

```
// class distance to floating type data meter
#include <iostream.h>
#include <conio.h>
class distance
    { int feet ;
      float inches ;
    public: distance (int f, float i)
    { feet=f ; inches=i ; }
    void show( )
    { cout<<feet<< “\’=”<<inches<<’\”’;}
    operator float( )
    { float ft=inches12 ;
      ft=ft+feet ;
      return(ft/3.28) ; }
      } ;
void main( )
{ distance d1(3, 3.36) ;
    float m=d1 ;
    clrscr( ) ;
cout<< “distance in meter:”<<m<<endl ;
cout<< “distance in feet and inches:”<<endl ;
    d1.show( ) ;
    getch( ) ; }
```

Output:

distance in meter : 1.0
distance in feet and inches:
3’ – 3.36”

3. Conversions between objects of different classes:

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

object of X = object of Y

X and Y both are different type of classes. The conversion takes place from class Y to Class A, therefore, Y is source class and X is destination class.

Class type one to another class can be converted by following way:

1.By constructor

2.By conversion function, i.e. the overloaded casting operator function.

By constructor:

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

```
//convert polar co-ordinate to rectangle co-ordinate
#include <math.h>
#include <iostream.h>
#include <conio.h>
class polar
    { float rd ;
      float ang ;
    public: polar( )
            { rd=0.0 ; ang=0.0 }
            polar (float r, float a)
            { ra=r ; ang=a ; }
            float getrd( )
            { return(rd) ; }
            float getang( )
            { return(ang); }
            void showpolar( )
            { cout<<rd<< ","<<ang<<endl ; }
            } ;
    class rec
            { float x ;
              float y ;
            public: rec( ) {x=0.0 ; y=0.0 ;}
                    rec (float xco, float yco)
                    {x=xco ; y=yco ; }
            rec (polar p)
            { float r=p.getrd( ) ;
              float a=p.getamg( ) ;
```

```
    x=r*cos(a)  ;
    y=r*sin(a) ; }
  void showrec( )
  { cout<<x<< ","<<y<<endl ; }
  } ;
  void main( )
  { rec r1 ;
    polar p1(2.0, 90.0) ; clrscr( ) ;
    r1=p1 ; //convert polar to rec by calling one argument to constructor
  cout<< "polar co."<<endl ;
  p1.showpolar( ) ;
  cout<< "rec co. :"<<endl ;
  r1.showrec( ) ;
  getch( ) ;
  }
```

Output:

polar co. : 2.0, 90.0

rec co. : 0.0, 2.0

By Conversion Function:

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

```
//convert polar co-ordinate to rec co-ordinate
#include <math.h>
#include <iostream.h>
#include <conio.h>
class rec
    { float x ;
      float y ;
    public: rec( )
      {x=0.0 ; y=0.0 ; }
    rec (float xw, float yw)
      {x=xw ; y=yw ; }
```

Chapter – 7
Inheritance

7.1 Introduction:

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

Visibility Modifier (Access Specifier):

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

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

7.3 Defining Derived Class (Specifying Derived Class):

A derived class can be defined by specifying its relationship with the base class in addition to its own detail.

The general syntax is

```
class derived_class_name : visibility_mode base_class_name
{     //members of derived class } ;
```

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

Examples:

```
class ABC : private XYZ       // private derivation
{
```

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

While any derived_class is inherited from a base_class, following things should be understood:

1. When a base class is privately inherited by a derived class, only the public and protected members of base class can be accessed by the member functions of derived class. This means no private member of the base class can be accessed by the objects of the derived class. Public and protected member of base class becomes private in derived class.
2. When a base class is publicly inherited by a derived class the private members are not inherited, the public and protected are inherited. The public members of base class becomes public in derived class where as protected members of base class becomes protected in derived class.
3. When a base class is protectly inherited by a derived class, then public members of base class becomes protected in derived class ; protected members of base class becomes protected in the derived class, the private members of the base class are not inherited to derived class but note that we can access private member through inherited member function of the base class.

Types of Inheritance:

Inheritance are classified into following types:

1. Single inheritance
2. Multiple inheritance
3. Multiple level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Single inheritance:

If a class is derived from only one base class, then that is called single inheritance.

Base Class

Derived Class

Fig : Single Inheritance

examples:

```
#include <iostream.h>
class B
{       private: int x ;
        protected: int y ;
        public: int z ;
        void getdata( )
        { cout<< “Enter 3 numbers=” ;
          cin>>x>>y>>z ; }
        void showdata( )
        { cout<< “x=”<<x<<endl ;
          cout<< “y=”<<y<<endl ;
          cout<< “z=”<<z<<endl ;
        } ;
class D : public B
        { private : int k ;
          public : void getk( )
                   { cout<< “Enter k=” ; cin>>k ; }
          void output( )
          { int s=y+z+k ;
                   cout<< “y+z+k=”<<s<<endl ; } } ;
        void main( )
        {
          D d1 ;
          d1.getdata( ) ;
          d1.getk( ) ;
          d1.showdata( ) ;
          d1.output( ) ;
        }
```

Output:

Enter 3 numbers = 5 6 7

```
        Enter k=8
        x=5
        y=6
        z=7
        y+z+k=21
//protected inherited
#include <iostream.h>
 class B{
                    private : int x ;
                    protected : int y ;
                    public : void getdata( ) {
          cout<< "enter x=" ; cin>>x ;
          cout<< "enter y=" ; cin>>y ;  }
          void showdata( ) {
          cout<< "x="<<x<<endl ;
          cout<< "y="<<y<<endl ;
class D : protected B
          {         private : int z ;
                    public : void getz( ) {
                    cout<< "enter z=" <<endl ; }
          void showz( )
          { cout<< "z="<<z<<endl ; }
          } ;
        void main( )
        { D d1 ;
          d1.getz( )
          d1.showz( )  ;
        }
The output of the above program is
Enter z=4
z=4
//private inherited
#include <iostream.h>
#include<conio.h>
 class B{
                    private : int x ;
                    protected : int y ;
                    public : void getdata( ) {
          cout<< "enter x and y:" ;
                    cin>>x>>y ; }
```

```
    void showdata( )
{ cout<<x<<y<<endl ; }
} ; //end of class B
class D : private B
    {       private : int z ;
            public : void assign( ) {
                    { z=30 ; y=40 ;
    void output( )
    { int f=y+z
      cout<< "s"<<f<<endl ; }
    } ; //end of class D
  void main( )
  { D d1 ;
    clrscr( ) ;
    d1.assign( ) ;
    d1.output( ) ;
    getch( ) ;  }
```

Output:

s=70

Multiple Inheritance:

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

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

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

For example:

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

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

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

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

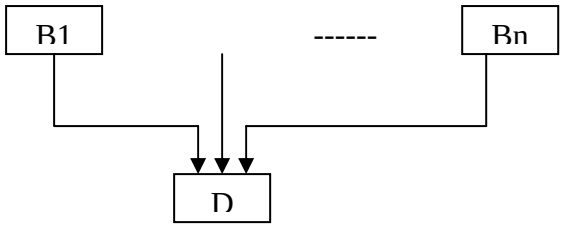


Fig: Multiple inheritance

```
// Multiple inheritance
#include <iostream.h>
#include <conio.h>
class biodata { char name[20] ;
                char semester[20] ;
                int age ;
                int rn ;
                public: void getbiodata( ) ;
                        void showbiodata( ) ;
                        } ;
class marks {  char sub[10] ;
                float total ;
                public:
                        void getrm( ) ;
                        void showm( ) ; } ;
class final: public biodata, public marks
            {   char fteacher[20] ;
                public: void getf( ) ;
                void showf( ) ; } ;
void biodata: : getbiodata( )
{
        cout<< "Enter name:" ; cin>>name ;
        cout<< "Enter semester:" ; cin>>semester ;
        cout<< "Enter age:" ; cin>>age ;
        cout<< "Enter rn:" ; cin>>rn ; }
void biodata: : showbiodata( )
{
        cout<< "Name:"<<name<<endl ;
        cout<< "Semester:"<<semester<<endl ;
        cout<< "Age:"<<age<<endl ;
        cout<< "Rn:"<<rn<<endl ;
void marks: : getm( )
```

```
{
        cout<< "Enter subject name:" ; cin>>sub ;
        cout<< "Enter marks:" ; cin>>total ; }
void marks: : showm( )
{
        cout<<  "Subject name:"<<sub<<endl ;
        cout<< "Marks are:"<<total<<endl ; }
void final: : getf( )
{       cout<< "Enter your favourite teacher" ; cin>>teacher ; }
void final: : showf( )
{       cout<< "Favourite teacher:"<<fteacehr<<endl ; }
void main( )
{       final f ; clrscr( ) ;
        f.getbiodata( ) ;
        f.getm( ) ;
        f.getf( ) ;
        f.showbiodata( ) ;
        f.shown( ) ;
        f.showf( ) ;
        getch( ) ;
}
```

Output:

```
        Enter name : archana
        Enter semester : six
        Enter age : 20
        Enter rn : 10
        Enter subject name : C++
        Enter marks : 85
        Enter favourite teacher : Ram

        Name : archana
        Semester : six
        Age : 20
        Rn : 10
        Subject name : C++
        Marks are : 85
        Favourite teacher : Ram
```

Multilevel inheritance:

The mechanism of deriving a class from another derived class is called multilevel. In the figure, class B1 derived from class B and class D is derived from class B1. Thus class B1 provides a link for inheritance between Band D and hence it is called intermediate base class.

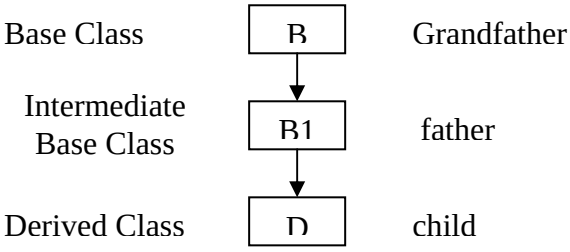


fig : multilevel inheritance

```
#include <iostream.h>
class std
{ protected : char name[20] ;
        int rn ;
public : void.getdata( )
{ cout<< “Student=” ; cin>>name ;
  cout<< “Roll no.=” ; cin>>rn ; }
        void showdata( )
{ cout<< “Student=”<<name<<endl ;
  cout<< “Roll no=”<<rn<<endl ;
} ; // end of class std
class marks : public std {
        protected : int m1, m2 ;
        public:
                void getm( )
        { cout<< “enter marks in Maths:”
          cin>>m1 ;
        cout<< “enter marks in English=” ; cin>>m2 ; }
void showm( ) {
        cout<< “Maths”<<m1<<endl ; cout<< “English=”<<m2<<endl ; }
} ; //end of class marks
class result : public marks
{ int total ;
  public: void calculate( )
        { total=m1+m2 ; }
        void show( )
```

```
        { cout<< "Total marks="<<total ; }
} ; //end of class result
void main( )
{ result s1 ;
s1.getdata( ) ;
s1.getm( ) ;
s1.calculate( ) ;
s1.showdata( ) ;
s1.shown( ) ;
s1.show( ) ;
}
```

Output:

Student=ram
Roll no=4
Enter marks in maths=56
Enter marks in english=45

Student=ram
Roll no=4
Maths=56
English=45
Total marks=101

Hierarchical Inheritance:

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

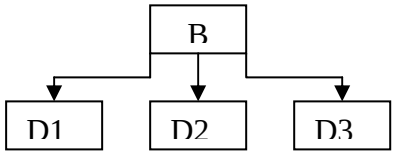


Fig: Hierarchical Inheritance

The general format:

```
class B { ------ }
class D1 : derivation B { ----- } ;
class D2 : derivation B { ----- } ;
class D3 : derivation B { ------} ; where
```

Derivation is public, protected or private type.

With the help of hierarchical inheritance, we can distribute the property of one class into many classes. For e.g

```
#include <iostream.h>
class B
        {       protected : int x, y ;

                public: void assign( )
                        { x=10 ; y=20 ; }
                        } ; //end of class B
class D1: public B
        {       int s ;
                public: void add( )
                { s=x+y ;
                  cout<< "x+y="<<s<<endl ; }
                } ; //end of class D1
class D2: public B
        {       int t ;
                public: void sub( )
                        { t=x-y ;
                        cout<< "x-y"<<t<<endl ; }
        } ; //end of class D2
class D3: public B
        { int m ;
        public: vod mul( ) {
                m=x*y ;
        cout<< "x+y"<<m<<endl ;
void main()
{       D1 d1 ;
        D2 d2 ;
        D3 d3 ;
        d1.assign( ) ;
        d1.add( ) ;
        d2.asign( ) ;
        d2.sub( ) ;
        d3.assign( ) ;
        d3.mul( ) ;
}
```

Output:

```
x+y=30
x-y=-*10
```

x*y=200

Hybrid Inheritance:

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

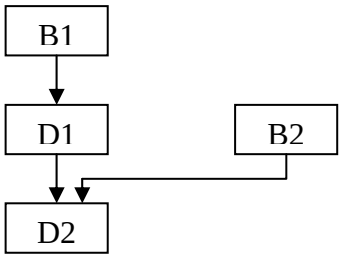


Fig : Hybrid Inheritance

The above diagram is combination of multilevel and multiple inheritance. We can combine other inheritance also.

```
#include<iostream.h>
class B1
    {  protected : int x ;
       public: void assignx( )
                { x=20 ; }
    } ; //end of class B1
class D1: public B1
    {  protected : int y ;
       public : void assigny( )
                { y=40 ; } } ; //end of class D1
class D2 : public D1
    {  protected : int z ;
       public : void assigz( )
                { z=60 ; }
    } ;
class B2
    { protected : int k ;
    public : void assignk( )
                { k=80 ; }
    } ;
class D3 : public B2, public D2
    { private : int total ;
       public : void output( )
```

```
    { total=x+y+z+k ;
    cout<< "x+y+z+k=" <<total<<endll ; }
    } ;
void main( )
{
D3.s ;
s.assignx( ) ;
s.assigny( ) ;
s.assignz( ) ;
s.assignk( ) ;
s.output( ) ;
}
```

Output:

x+y+z+k=200

7.4 Casting Base Class Pointer to Derived Class Pointers:

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

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

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

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

This is vali1d in C++ because d is an object derived from the class B.

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

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

```
#include <iosteram.h>
class BC
```

```
{       public :   int b;
                   void show( )
                   { cout<< "b="<<b<< "\n" ; }
} ;
class DC: public BC
        { public : int d ;
                    void show( )
                    { cout<< "b="<<b<< "\n"<< "d="<<d<< "\n" ;
                   }
                   } ;
void main( )
{
        BC*bptr ;          //base pointer
        BC base ;
        bptr=&base ;       //base address
        bptr→ b=100 ;    //access BC via base pointer
        cout<< "bptr points to base object \n" ;
        bptr→ show( ) ;
        // derived class
        DC derived ;
        bptr=&derived    //address of derived class's object
        bptr→ b=200 ;     // access DC via base pointer
        /* bptr     d=300 ; /         //won't work
        cout<< "bptr now points to derived object \n" ;
        bptr→ show( ) ;   //bptr now point to derived object
        /*accessing d using a pointer to type derived class DC */
        DC*dptr ;          //derived type pointer
        dptr=&derived ;
        dptr→ d=300 ;
        cout<< "dptr is derived type pointer \n" ;
        dptr→ show( ) ;
        cout<< "using(DC*) bptr)\n" ;
          ((DC*)bptr→ )⟶ d=400 ;     // cast bptr to DC type
          ((DC*)bptr)⟶show( ) ;
        }
```

Output:

```
        bptr points base object
          b=100
        bptr now points to derived object
          b=200
```

dptr is derived type pointer
b=200
d=300
using ((DC*)bptr)
b=200
d=400

7.5 Using Constructors and Destructors in inheritance (Derived Classes):

It is possible for the base class, the derived class or both to have constructor and / or destructor.

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

So far, we have passed arguments to either the derived class or base class constructor. When only the derived class takes an initialization, arguments are passed to the derived class constructor in the normal fashion. However if we need to pass an argument to the constructor of the base class, a little more effort is needed:

1. All necessary arguments to both the class and derived class are passed to the derived class constructor.
2. Using an expanded form of the derived class' constructor declaration, we then pass the appropriate arguments along to the base class.

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

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

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

```
// Illustrate when base and derived class constructor and destructor functions are executed
#include <iostream.h>
class base {public:
                base( ) { cout<< "Constructing base \n" ; }
                base( ) {cout<< "Destructing base \n" ; }
            } ;
class derived : public base {
```

```
        public:
          derived( ) { cout<< "Constructing  derived \n" ; }
        ~ derived( ) { cout<< "Destructing derived \n" ; }
        } ;
int main( )
        derived obj ;
        return 0 ;
        }
```

Output:

```
        Constructing base
        Constructing derived
        Destructing derived
        Destructing base
```

```
// Program shows how to pass argument to both base class and derived class
#include <iostream.h>
class base {int i ;
        public :
          base (int n) {
          cout<< "Constructing base \n" ;
                    i=n ; }
        ~ base( ) { cout<< "Destructing base \n" ; }
                    void showi( ) { cout<<i<< "\n" ; }
        } ;
class derived: public base {int j ;
        public:
          derived (int n): base(n)          // pass argument to the base class
        { cout<< "Constructing derived \n" ;
          j=n ; }
        ~ derived( ) {cout<< "Destructing derived \n";}
          void show( ) { cout<<j<< "\n" ; }
        } ;
        void main( )  {derived o(10) ;
                          o.showi( ) o.showj( ) ;  }
```

Output:

```
        Constructing base
        Constructing derived
        10
        10
        Destructing derived
```

Destructing base

7.6 Benefits and cost of inheritance:

The benefits of inheritance are listed below:

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

The cost of inheritance are listed below:

1. The base and derived class get tightly coupled. This means one cannot be used independent of each other that is to say they are interconnected to each other.
2. We know that inheritance uses the concept of reusability of program codes so the defects in the original code module might be transferred to the new module there by resulting in defective modules.

Chapter – 8
Virtual Function and Polymorphism

Polymorphism:

Polymorphism is one of the crucial features of OOP. Polymorphism means one name, multiple form. We have already studied function overloading, constructor overloading, operator overloading, all these are examples of polymorphism .

For e.g.

```
#include <iostream.h>
#include <conio.h>
void area(float r) ;
void area(float l, float b) ;
void main( )
{       float r1, l1, b1 ;
          clrscr( ) ;
        cout<< "enter value of r1:" ; cin>>r1 ;
        cout<< "enter value of l1:" ; cin>>l1 ;
        cout<< "enter value of b1:" ; cin>>b1 ;
        cout<< "Area of circle is"<<endl; area(r1) ;
        cout<< "Area of rectangle is"<<endl ; area(l1, b1) ;
        getch( ) ; }
void area (float r)
{       float a=3.14*r*r ;
        cout<< "Area="<<a<<endl ; }
void area(float l , float b)
{
        float a1=l*b ;
        cout<< "Area="<<a1<<endl ; }
```

Output:

```
        enter value of r1 : 2
        enter value of l1 : 4
        enter value of b1 : 6
        Area of circle is
                Area = 12.56
        Area of rectangle is
                Area = 24
```

In above program two functions have same name but question is how compile differentiates these two. Actually, C++ compile differentiates these two by argument. In one

area function, there is only one argument which is float type but in second area function there are two arguments both are float type.

If any program has two functions both have same name and number of argument is also same then compiler differentiates these by type of arguments.

Classification of Polymorphism:

1. Compile time polymorphism
2. Run time polymorphism

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

a) function overloading
b) operator overloading

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

a) virtual function

8.2 Virtual Function:

Virtual means existing in effect but not in reality. A function is declared virtual by writing keyword 'virtual' in front of function header. A virtual function uses a single pointer to base class pointer to refer to all the derived objects. Virtual functions are useful when we have number of objects of different classes but want to put them all on a single list and perform operation on them using same function call.

When we use the same function name in both the base and derived classes, the function in base class declared as virtual function. The virtual function is invoked at run time based on the type of pointer.

```
#include <iostream.h>
class B
    { public :
            virtual void show( )
      { cout<< "This is in class B"<<endl ; }
    } ;        // end of class B
class D1 : public B
```

```
        { public : void show( )
{ cout<< “This is in class D1”<<endl ; }
        } ;         // end of class D1
class D2 : public B
        { public: void show( )
          { cout<< “This is in class D2”<<endl ; }
        } ;         //end of class D2
void main( )
{       B *P  ;
        D1 obj1 ;
        D2 obj2 ;
        B objbase ;
        p=&objbase ;
        p→ show( ) ;
        p=&obj1 ;
        p→ show( ) ;
        p=&obj2 ;
        p→ show( ) ;
}
```

Output:

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

```
#include <iostream.h>
class polygon
{       protected : int width, height ;
        public :
                void setdata(int a, int b)
                { width=a; height=b; }
                virtual int area( )
                        {return 0 ; }
                } ;       //end of class polygon
class rectangle: public polygon
        { public:
                int area( )
                { return (width*height) ; }
        } ;         //end of class rectangle
class triangle: public polygon
        { public:
```

```
                int area( )
                        { return (width*height /z) ; }
} ;     //end of class triangle
void main( )
{       rectangle r ;
        triangle t ;
        polygon p ;
        polygon *p1=&c ; polygon *p2=&t ;
        polygon *p3=&p ;
        p1→ setdata(4,5) ;
        p2→ setdata(4,5) ;
        p3→ setdata(4,5) ;
        cout<<p1→ area( )<<endl ;
        cout<<p2→ area( )<<endl  ;
        cout<<p3→ area( )<<endl ;
}
```

Output:

20

10

0

Virtual inside the base class and redefine it in the derived class. But the function defined inside the base class B is seldom (rarely) used for performing any task. These functions only serve as placeholder. Such functions are called do-nothing function or pure virtual function.

In above edxample, '=' sign has nothing to do with assignment operation and value 0 is not assigned anything. This simply informs the compiler that function will be pure i.e. it will have no definition relative to base class.

It should be noted that class containing pure virtual function cannot be used to create objects of its own. Such classes are known as Abstract classes.

```
//virtpure.cpp
//pure virtual function
#include <iostream.h>
#include <conio.h>
class B         //Base class
        { public:
                virtual void show( )=0 ; // pure virtual function
        } ;
class D : public B //derived class
        {
```

```
        public:
                void show( )
        { cout<< "Inside derived class"; } } ;
int main( )
{       O d ;
        B *P ;
        P=&d ;
        P→show( ) ;
        return 0 ;
        getch( ) ;
}
```

Output:

Inside derived class

Rules for Virtual Functions:

If a function is made virtual function, following things should be considered:

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

Pure Virtual Functions and Abstract Class:

A pure virtual function is a virtual function with no function body. If we delete the body of virtual function then it becomes pure virtual function. For example, suppose void show( ) is a virtual function in a class base class, then

virtual void show( ) =0 ; becomes pure virtual function.

In general, we declare a function

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

```
        boolean {false, true} ;
class person     //person class
{ protected:
        char name[40] ;
public:
        void setName( )
        { cout<< "Enter name:" ; cin>>name; }
        void printName( )
{ cout<< "Name is:"<<name<<endl ; }
boolean virtual is outstanding( )=0 ; //pure virtual function
} ;
class student: public person    // student class
{ private:
        float gpa ;          //grade point average
  public:
        void setGpa( )    //set GPA
{cout<< "Enter student's GPA:"; cin>>gpa ; }
boolean is outstanding( )
{ return(gpa>3.5)> true:false ; }
} ;
class professor : public person          //professor class
{ private : int numPubs ;        //number of papers published
  public:
        void setNumPubs( )       //set number of paper published
{ cout<< "Enter number of professor's publication:" ;
  cin>>numPubs ;
}
boolean is outstanding( )
{
        return(numPubs>100)? true:false ; }
} ;
void main(void)
{ person*persptr[100] ;          // list of pointers to persons
  student*strptr ;               // pointer to student
  professor*proptr ;             // pointer to professor
  int n=0 ;                      // number of persons on list
  char choice ;
//virtpure1.cpp
//pure virtual function
#include <iostream.h>
```

```
class Base      //base class
{
      public:
            virtual void show( )=0 ; //pure virtual function
} ;
class Derv1: public Base //derived class1
{     public:
            void show( )
            { cout<< "\n Derv2" ; }
} ;
void main( )
{
      Base*list[2] ;    // list of pointers to base class
      Derv1 dv1 ;       // object of derived class1
      Derv2 dv2 ;       // object of derived class2
      list[0]=&dv1 ;    // put address of dv1 in list
      list[1]=&dv2 ;    // put address of dv2 in list
      list[1]   show( ) ; // execute show( ) in both objs
}
```

8.4 Using Virtual Functions:

Imagine a class person that has two derived classes, student and professor. These derived classes each contain a function called is outstanding( ) used for the school administrators to create a list of outstanding students and professors for award day ceremony.

```
do
      { cout<< "Enter student or professor(s/p):" ;
      cin>>choice ;
      if (choice = = 's')         // it's a student
      {
            stuPtr=new student ;   // make new student
            stuPtr→ setName( ) ;   // set student name
            stuPtr → setGpa( ) ;     // set GPA
            PersPtr[n++]=stuPtr ;  // put point in list
      }
      else
      {                                 // it's a professor
            proPtr=new professor ;  // make new professor
            proPtr→ setName( ) ;  // set professor name
            proPtr→ setNamePUbs( ) ; // set number of pubs.
            persPtr[n++]=proPtr ; // Put pointer in list
```

```
        }
        cout<< "Enter another(y/n)? ;    // do another person?
        cin>>choice ;
        } while (choice = = 'y') ;          // cycle until not 'y'
        for (int j=0 ; j<n ; j++)           // print names of all persons and
        {
           persPtr[j]→printName( ) ;     // say if outstanding
        if (PersPtr[j]→is outstanding( ) = = true)
                cout<< "This person is outstanding \n" ;
        }
                // end main( )
        }
#include <iostream.h>
#include <conio.h>
class publication
{       char title[80] ;
        float price ;
        public:
                virtual void getdata( )
                {
                        cout<< "\n Enter title:"  ; cin>>title ;
                        cout<< "\n Enter price:" ; cin>>price ;
                }
                virtual void putdata( )
                {
                        cout<< "\n Title:"<<title ;
                        cout<< "\n Price:"<<price ;
                } } ;
class book : public publication
{       int pages ;
        public : void getdata( )
        {       publication: : getdata( ) ;
                cout<< "\n Enter number of pages:" ;
                cin>>pages ;
        }
        void putdata( )
        {
                publication: : putdata( ) ;
                cout<< "\n Pages:"<<pages ;
        } } ;
```

```
class tape: public publication
{           float time ;
public : void getdata( )
           {        publication : : getdata( )
           cout<< “\n Enter time in minutes:” ;
           cin>>time ;
           }
           void putdata( )
           {        publication : : putdata( ) ;
           cout<< “\n Playing time in minute:” ; << time ;
           }
           } ;
           void main( )
           {
           publication*p[100] ;
           int n=0 ;
           char ch ;
           clrscr( ) ;
           do
           {        cout<< “enter book or tape(b/t)?” ;
                    cin>>ch ;
           if (ch = = ‘b’)
           p[n]=new book ;
           else
           p[n]=new tape ;
           p[n++]    getdata( ) ;
           cout<< “\n Enter another(y/n)>” ; cin>>ch ;
           }        while (ch = = ‘y’ || ch= ‘Y’),
           cout<< “\n You have entered following information: \n” ;
           for (int j=o ; j<n ; j++)
           p[j]→ putdata( ) ;
           getch( ) ;
}
```

8.5 Early & Late Binding:

The differences between early and late binding are given below:

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

Chapter – 9
Input/Output

Introduction:

The main objective of the computer program is to take some data as input, do some process on that data and generate desired output. Thus input/output operations are the most essential part of any program. In C++, there are several I/O function which help to control the input and output operations.

We have already studied, cin and cout in combination with >> and << operators for console input/output operations. Here, we discuss about various I/O functions, including stream and stream classes that implement the I/O operation.

9.1 Stream based input/output:

A stream is an interface provided by I/O system to the programmer. A stream, in general, is a name given to flow of data. In other words, it is a sequence of bytes.

The stream acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called INPUT stream and the destination stream that receives output from the program is called OUTPUT stream. Thus, a program extracts the bytes from an input stream and inserts them into the output stream.

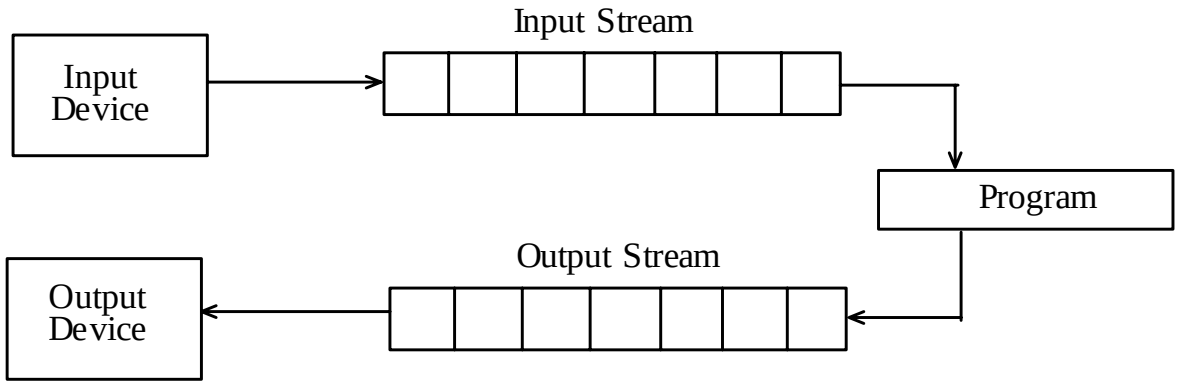


Fig : Stream based I/O

Different input devices like keyboard can send data to the input stream. Also, data in output stream can go the output device like screen (monitor) or any other storage device. In C++, there are predefined I/O stream like cin and cout which are automatically opened where a program begins its execution.

9.2 Input/Output Class Hierarchy

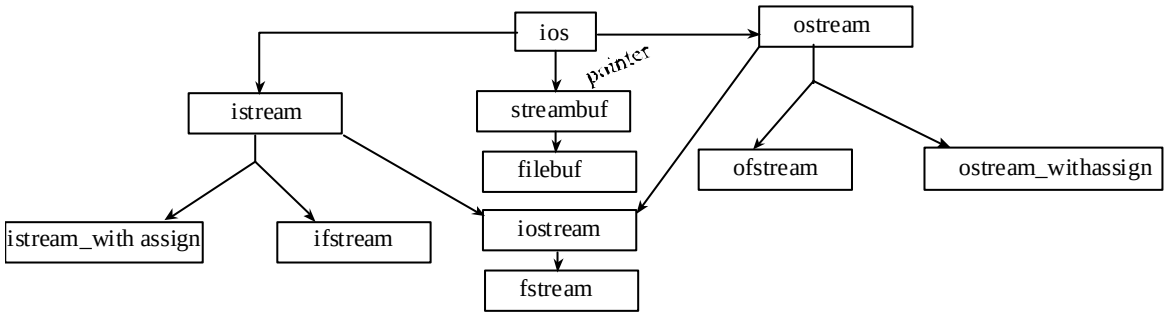


Fig : I/O Class Hierarchy

The above figure shows a hierarchy of stream classes in C++ used to define various stream in order to deal with both the console and disk files. From the figure, it is clear that ios is a base class. This ios class is declared as virtual base class so that only one copy of its members is inherited by its derived classes ; thereby avoiding ambiguity. The ios class comprises of basic functions and constants required for input and output operations. It also comprises of functions related with flag strings.

istream and ostream classes are derived from ios and are dedicated to input and output streams respectively. Their member functions perform both formatted and unformatted operations. istream contains functions like get( ), getline, read( ) and overloaded extraction(>>) operators. ostream comprises of functions like put( ), write( ) and overloaded insertion(<<) operators.

The iostream class is derived from istream and ostream using multiple inheritance. Thus, it provides the facilities for handling both input and output streams. The class istream_withassign and ostream_withassign add assignment operator to these classes. Again the classes ifstream and ofstream are concerned with file I/O function. ifstream is used for input files and ofstream for output files. Also there is another class of stream which will be used both for input and output. All the classes ifstream, ofstream and fstream are derived from classes istream, ostream and iostream respectively.

streambuf is also derived from ios base class. filebuf is derived from streambuf. It is used to set the file buffers to read and write. It also contains open( ) and close( ) used to open and close the file respectively.

Unformatted I/O Operations:

1. Overloaded Operators >> and << :

We know that in C++, cin and cout objects (defined in iostream) in combination with overloaded >> and << operators.

The general format for reading data from keyboard is : cin>>var1>>var2>>.....>>varn,

This statement will cause the computer to stop the execution and look for input data from the keyboard. While entering the data from keyboard, the whitespace, newline and tabs

will be skipped. The operator >> reads the data character by character basis and assigns it to the indicated locations.

Again, the general format for displaying data on the computer screen is :

cout<<item1<<item2<<.....<<itemn,

Here, item1, item2, ..., itemn may be character or variable of any built-in data type.

2. **Put( ) and get( ) functions:**

These are another kind of input/output functions defined in classes istream and ostream to perform single character input/output operations.

get( ):

There are 2 types of get functions i.e. get(char*) and get(void) which help to fetch a character including the blank space, tab and a new line character. get(char*) assigns input character to its argument and get(void) returns the input character.

For e.g.

```
char c;
cin.get(c) ;        // obtain single character from keyboard and assign it to char c
OR
c=cin.get( ) ;      // this will skip white spaces and newline
OR
cin>>c ;
```

Put( ):

It is used to output a line of text character by character basis.

for e.g.

```
cout.put ('T') ;   // prints T
cout.put(ch) ;     // prints the value of variable ch
cout.put(65) ;     // displays a character whose ASCII value is 65 that is A
```

Program

```
#include <iostream.h>
void main( )
{
    int count=0 ;
    char c ;
    cout<< "INPUT TEXT \n" ;
    cin.get(c) ;
    while (c!= '\n')
    { cout.put(c) ;
      count++ ;
      cin.get(c) ;
```

```
    }
    cout<< "\n Number of characters="<<count<<\n" ;
}
```

Output:

INPUT TEXT
Object Oriented Programming
Object Oriented Programming
Number of characters = 27

getline( ) and write( ) Function:

The getline( ) function reads a whole line of text that ends with a newline character. The general syntax is

```
cin.getline( line, size) ;
```

where, line is a variable, size is maximum number of characters to be placed. Consider the following code:

```
char name[30] ;
```

cin.getline(name, 30) or cin>>name

If we input the following string from keyboard:

This is test string

cin.getline(name, 30) inputs 29 character taking white spaces and one left null character. so, it will take whole line but in case of cin it takes only this as it doesn't consider white space.

If we again put

This is test string to demonstrate getline, it takes only. This is test string to demonstrate

For e.g.

```
#include <iostream.h>
void main( )
{   int char city[20] ;
    cout<< "Enter city name:\n" ;
    cin>>city ;
    cout<< "city name:"<<city<<"\n\n" ;
    cout<< "Enter city name again:\n" ;
    cin.getline (city, 20) ;
    cout<< "New city name:"<<city<< "\n\n" ;
}
```

Output:

First Run:

Enter city name: Kathmandu
City name : Kathmandu
Enter city name again : Lalitpur
New city name : Lalitpur

Second Run:
Enter city name : New Baneshwor
City name : New Baneshwor
Enter city name again : Old Baneshwor
New city name : Old Baneshwor

write( ):

This function displays an entire line of text in output string.

General syntax is cout.write(line, size)

where, line represents the name of string to be displayed and second argument size indicates number of characters to be displayed.

For e.g.

```
#include <iosteram.h>
#include <string.h>
void main( )
{
    char *s1= “C++” ;
    char *s2= “Programming” ;
    int m=strleng(s1) ;
    int n=strlen(s2) ;
    for (int i=1 ; i<n ; i++)
      { cout.write(string2, i) ;
            cout<< “\n” ;
      }
    for (i=n ; i>0 ; i.....)
    { cout.write(s2, i ) ;
      cout<< “\n” ;
    }
// concating strings
cout.write (s1, m).write (s2, n),
cout<< “\n” ;
// crossing the boundary
cout.write(s1, 10) ;
}
```

Formatted Console I/O Operations:

C++ supports a number of features that could be used for formatting the output. These features include

- ios class functions and flags
- manipulators

ios class functions and flags:

This ios class contains a large number of member functions that would help to format the output, are listed below:

Function	Task
Width( )	To specify the required field size for displaying an output value
Precision( )	To specify the no. of digits to be displayed after a decimal point of a float value
fill( )	To specify a character that is used to fill the unused portion of a field
setf( )	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf( )	To clear flags specified

Table : ios format function

Manipulators:

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Following are the important manipulators functions:

Manipulator	Equivalent ios function
setw( )	width( )
setprecision( )	precision( )
setfill( )	fill( )
setioflags( )	setf( )
resetioflags( )	unsetf( )

Table : Manipulator Functions

To access all these manipulator, the file iomanip.h should be included in the program. Following section describes different ios class functions:

width( ):

We can use the width( ) function to define the width of a field necessary for the output of an item. The general syntax is

```
cout.width(w) ;
```

where, 'w' is the field width (total number of columns)

for e.g. cout.width(5) ;

```
cout<<123<<45<< "\n" ;
```

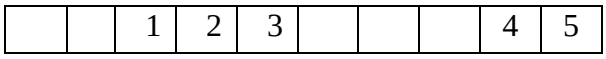
will produce the following output:

		1	2	3	4	5

The value 123 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 45. This can be improved as follows:

```
cout.width(5) ;
cout<<123 ;
cout.width(5) ;
cout<<45<< "\n" ;
```

This produces the following output:

If the 

		1	2	3				4	5

field width specified is smaller than the size of value to be printed, C++ expands the field to fit the value.

precision( ):

This function helps to specify the number of digits to be displayed after the decimal point for printing floating point numbers. By default the floating numbers will be six digit after decimal point. The general syntax is

```
cout.precision (d) ;
```

where, d is the number of digits to be displayed after decimal point.

This function has effect until it is reset. Consider the following statement.

```
cout.precision(3) ;
cout<<sqrt(2)<< "\n" ;
cout<<3.14159<< "\n" ;
cout<<2.50032<< "\n" ;
```

The above program produces the following output:

1.141 (truncated)
3.142 (round to the nearest number)
2.5 (no trailing zero or trailing zero truncated)

This is seen above, we can set the precision only once and this is used by all three outputs.

fill( ):

We use this function for filling and padding the unused positions (filled with white spaces by default) with desired symbol or characters.

syntax : cout.fill(ch) ;

where ch is character used to fill the unused space

For e.g. cout.fill('#')

```
cout.width(10) ;
cout<< "filling" ;
```

#	#	#	f	i	l	l	i	n	g

setf( ) and bit-fields:

setf( ) member function is used to print the output in the desired format like left justified, right justified, etc.

syntax: cout.setf (arg1, arg2) ;

where, arg1 is one of the following flags defined in the ios class which specifies format action required for the output. arg2 is a bit-field which specify the group to which the following flag belongs. There are 3 bit fields namely adjust_field, float_field and base_field.

Table:

Flags and bit fields for setf( ) function

Format Required	Flag (arg1)	Bit- Field (arg2)
• Left-justified output • Right-justified output • Padding after sign or base indicactor (like +##20)	ios: : left ios: : right ios: : internal	ios: : adjustfield ios: : adjustfield ios: : adjustfield
• Scientific notation • Fixed point notation	ios: : scientific ios: : fixed	ios: : floatfield ios: : floatfield
• Decimal base • Octal base • Hexadecimal base	ios: : dec ios: : oct ios: : hex	ios: : basefield ios: : basefield ios: : basefield

```
#include <iostream.h>
void main( )
{       cout.fill('*') ;
        cout.setf(ios: : left, ios: : adjustfield) ;
        cout.width(15) ;
        cout<< "Test Left"<<endl ;
}
```

Output:

T	e	s	t	L	e	f	t	*	*	*	*	*	*	*

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
void main( )
{
        clrscr( ) ;
        cout.fill(#)
        cout.precision(3) ;
        cout.setf(ios: : internal, ios: : adjustfield) ;
        cout.setf(ios: : scientific, ios: : floatfield) ;
        cout.width(15) ;
        cout<<-12.34567<< "\n" ;
        getch( ) ;
```

```
}
```

Output:

-	#	#	#	#	#	1	.	2	3	e	+	0	1

The setf( ) can also be used to display the trailing zero and plus sign in the output setf( ) can also be used in combination with flag ios: : showpoint as a single argument to display trailing zeros and trailing decimal point.

For e.g.

```
#include <iostream.h>
void main( )
{       cout.precision(7) ;
        cout.setf(ios: : showpoint) ;
        cout<<3.14 ; }
```

Output:

3	.	1	4	0	0	0	0	0

setf( ) with ios: : showpos flag as an argument can be used to print a plus (+) sign before number.

```
#include <iostream.h>
        void main( )
        {
        cout.width(5) ;
        cout.setf(ios: : showpos) ;
        cout<<3.14 ;
}
```

Output:

+	3	.	1	4

Managing Output with Manipulators:

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output formats. They provide the features as that of the ios member functions and flags. Some manipulators are more convenient than their counterparts in the class ios. For example, two or more manipulators can be used as a chain in one statement as shown below:

```
cout<<mainp1<<mainp2<<mainp3<<item ;
cout<<mainp1<<item1<<mainp2<<item2
```

Table : Manipulators and their meaning

Manipulator	Meaning	Equivalent
• setw (int w)	set the field with to w	width( )
• setprecision (int d)	set the floating point precision to d	precision( )

• setfill (int c) • setiosflags (long f) • resetiosflags (long f) • endl	set the fill character to c set the format flag f insert new line and flush stream	fill( ) setf( ) unsetf( ) "\n"

For example

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
#include <math.h>
void main( )
{
        cout<<setw(5)<<setprecision(2)<<1.2345 ;
        cout<<setw(10)<<setprecision(4)<<sqrt(2) ;
        cout<<set(15)<<setiosflag(ios: :scientific)<<sqrt(3) ;
        getch( ) ;
}
```

Output:

In above example, we print all the three values in one line with the field size of 5, 10 and 15 respectively. Setprecision (4) is used to output the value of sqrt(2) and setflags (ios: : scientific) used for sqrt(3).

The basic difference between manipulators and ios class function is that ios member functions return the previous format state which can be used later. But manipulators do not return the previous format state. Thus ios functions are useful when we need to save the old format states.

For e.g.

```
        cout previous(2) ;          // previous state
int p=cout.precision(4) ;           // current format state
cout.precision(p)  ;                // p=2
#include <iostream.h>
void main(  )
{       cout.precision(4) ;
        cout<<2.33309<<endl ;
        int p=cout.precision(4) ;
        cout.precision(2) ;
        cout<<endl<<2.33309 ;
        cout.precision(p) ;
        cout<<endl<<2.33309 ;
```

}
Output:
2.3331
2.33
2.3331

9.2 File Input/Output:

Many real_life problems handle large volumes of data and in such situation, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both the following kinds of data communication:

1. Data transfer between the console unit and the program
2. Data transfer between the program and a disk file.

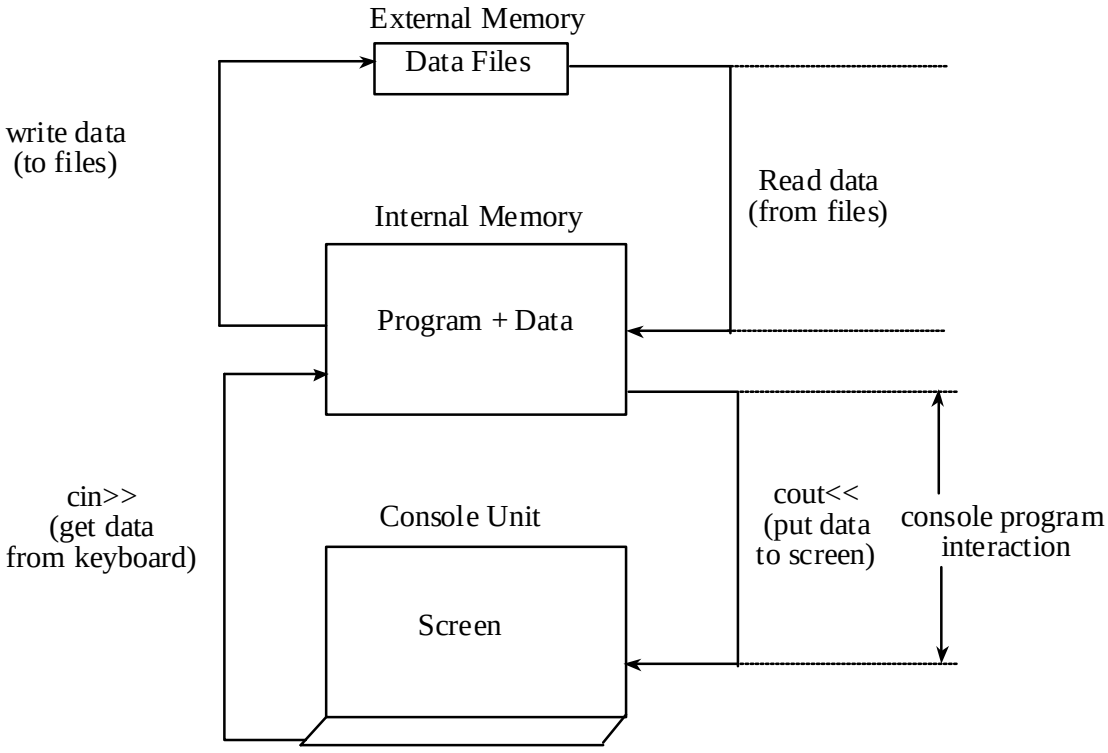


Fig : console-program-file interaction

The above figure demonstrates two types of data communication.

File stream:

The I/O system of C++ handles file operations which are very much similar to the

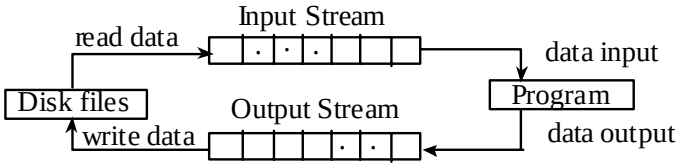


Fig : File Input and Output Stream

console input and output operations. A file stream is an interface between the programs and the files. The stream which supplies data to the program is called input stream and that which receives data from the program is called output stream. That is. the input stream reads or receives data from the file and supplies it to program while the output stream writes or inserts data to the file. This is illustrated in the figure.

Class Hierarchy for File Stream Operation:

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class as shown in figure. These classes, designed to manage the disk files, are declared in fstream.h and therefore we must include this file in any program that uses files.

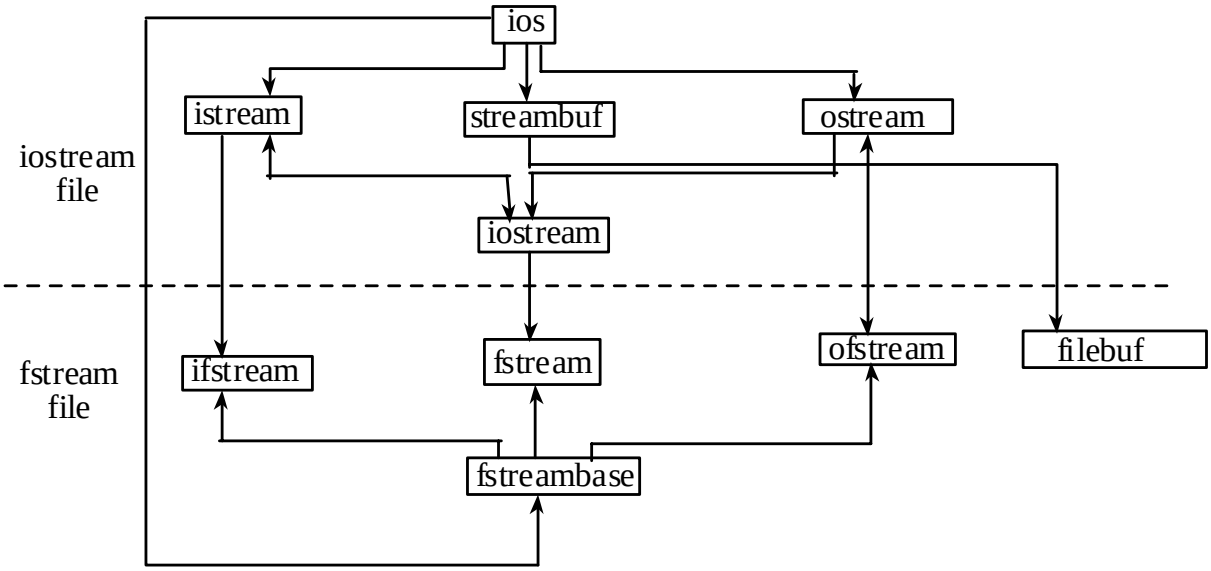


fig : File I/O Class Hierarchy

a) Opening a file using constructor:

This method is used when we use only one file in a stream. A file can be created using a constructor for output or writing as

```
ofstream fout("results.txt") ;  // output only
                                // fout or outfile or o_file or myfile
```

This statement creates an object fout (which can be any valid name like myfile, outfile, o_file, etc) of ofstream class and attaches this object with a file "reacts" which can be any valid name. In this results file, we can only write data because it is created by ofstream class. A filename is used to initialize the file stream object as fout.

A file can be created for input or reading as

```
ifstream fin("test") ;
```

This creates an object fin(it may be any valid name like infile, myfile, etc) of ifstream and attaches a file "test" with it. In file "test" we can perform only input operation i.e. we can read data because it is created by ifstream class.

b) Opening a file using open( ) function:

An open( ) function can be used to open multiple files that use same stream object.

General syntax is

```
file_stream_class stream_object ;
stream_object.open ("filename") ;
```

examples:

```
ofstream fout ;    // for writing
fout.open ("results") ;
ifstream fin ;
fin.open("test") ; // for reading
fstream finout ;  // for both reading and writing
finout.open("Data") ;
```

Table : Details of file stream classes

Class	Contents
filebut	Helps to set the file buffers to read and write and contains open function
fstreambase	Serves as a base for fstream, ifstream and ofstream class. Contains open( ) and close( ) functions
ifstream	Provides input operations contains open( ) with default input mode. Inherits the function get( ) , getline( ), read( ), seekg( ) and tellg( ) function from ifstream.
ofstream	Provides output operations. Contains open( ) with default output mode. Inherits put( ), seekp( ), tellp( ) and write( ) functoin from ofstream.
fstream	Provides support for simultaneous input amd output operations. Contains open( ) with default input mode. Inherits all the functions from istream and ostream classes through iostream.

File Operations:

There are different types of operations that can be performed on a file, like opening, reading, writing, closing, etc. A file can be defined by class ifstream header file fstream.h. If a file is declared by ifstream class then it can be used for reading from file purpose. If a file is declared by ofstream then that can be used for writing purpose. If file is declard by fstream

then it can be used for both reading and writing. There are different file operations that are mentioned below:

1. Opening a file

Before performing read/write operation to a file, we need to open it. A file can be opened in two ways:

a) Using a constructor function of a class
b) Using a member function open( ) of a class

Files Modes [Modes of Opening a File]

Constructors and function open( ) both can be used to create new files as well as to open the existing files. The open ( ) can be used to open file in different modes like read only, write only, append mode, etc.

Basic syntax:

stream_object.open ("filename", mode) ;

where, mode specifies the purpose for which a file is opened.

Table : File mode parameters

Modes	Functions
ios: :app	Start reading or writing at end of file (APPend)
ios: : ate	Go to end-of-file on opening, that is, erase file before reading or writing (trunCATE)
ios: : in	Open file for reading only (default for ifstream)
ios: : out	open file for writing only (default for ofstream)
ios: : binary	open file in binary (not text) mode
ios: : nocreate	error when opening if file does not exist
ios: : noreplace	error when opening for output if file already exists, unless ate or app is set
ios: : trunc	Delete the contents of the file if it exists

Examples:

1) fstream file ;

file.open("Person.Dat", ios: : app) ios: : out(ios: : in) ;

This creates Person file for input and output in append mode.

2) ifstream infile ;

infile.open("Test.txt", ios: : in) ;

This opens Test.txt file for reading only.

3) ofstream outfile ;

outfile.open("Test.txt", ios: : out) ;

This opens Test.txt for writing only.

4) ofstream fout

fout.open("data", ios: : app/ios: : nocreate) ;

This opens the file in append mode but fails to open if it doesn't exist.

2) Closing a File:

After opening any file, it is necessary that it must be closed. The general syntax for closing a file is

stream_class_object.close( ) ;

Examples :

fout.close( ) ;
fin.close( ) ;

File Pointer and their Manipulators:

Each file has two associated pointers called as file pointers. One of them is called the input pointer or get pointer and other is called the output pointer.

When input and output operation takes place, the appropriate pointer is automatically set according to mode. For example when we open a file in reading mode file pointer is automatically set to start of file. And when we open in appended mode the file pointer is automatically set at the end of file.

In C++ there are some manipulators by which we can control the movement of pointer. The available manipulator in C++ are:

1. seekg( )
2. seekp( )
3. tellg( )
4. tellp( )

1. seekg( ):

This move gets pointer i.e. input pointer to a specified location. For example:

infile.seekg(5) ;

move the file pointer to the byte number 5 from starting.

2. seekp( ):

This move puts pointer (output pointer) to a specified location. For example:

outfile.seekp(5) ;

3. tellg( ):

This gives the current position of get pointer (i.e. input pointer)

4. tellp( ):

This gives the current position of put pointer (i.e. output pointer). For example:

```
ofstream fileout ;
fileout.open ("test", ios: : app) ;
int length=fileout.tellp( ) ;
```

By the above statement in length, the total number byte of files are assigned. Because file opened in append mode that means the file pointer is at last of file.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void main( )
{       ofstream fout(“Test.txt”) ;
        fout<< “I am Ram” ;
        fout.seekp(3) ;
        fout<< “is” ;
        int p=fout.tellp( ) ;
        cout<<p ;
        fout.close( ) ;
        getch( )
}
```

Output: 5

Specifying the offset:

seekp( ) and seekg( ) can also be used with two arguments as follows:

```
seekg (offset, refposition) ;
seekp (offset, refposition) ;
```

where,

the paramemter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition is three types:

1. ios: : end – means from end of file
2. ios: : beg – means from beginning of file
3. ios: : cur – means from current file position

The offset is negative also as follows:

```
fileseekg (-5, ios: : cur) ;
```

This statements means file pointer move five byte back from current position.

```
file.seekg(0, ios: : beg) ;        go to start
fout.seekg(0, ios: : cur) ;        stay at current position
fout.seekg(0, ios: : end) ;        go to end
```

Similarly, all above concepts are same for seekp( ) ;

Error Handling in File:

In C++ for handling error in file operation following functions are available:

1. eof( )
2. fail( )
3. bad( )

4. good( )

1.eof( ): This function returns true if end of file encountered while reading otherwise return false.
2.fail( ): This function returns true when an input and output operation has failed otherwise return false.
3.bad( ): This function returns true if an invalid operation is attempted or any unrecoverable error has occurred otherwise return false.
4.good( ): This function returns true if no error occurred otherwise false.

These functions are used as follows:

```
fstream file ;
file.open (“Add”, ios: : in) ;
if (file.eof( ) ) {------}
if (file.fail( ) ) {------}
if (file.bad( ) ) {------}
if (file.good( ) ) {------}
```

String Input/Output:

C++ provides us with the facility to read and write the character strings.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void main( )
{       char str[80] ;
        ofstream outfile (“Test.Dat”) ;  // creates file for output
        outfile<< “This is Testing\n” l  // send text to file
        outfile<< “This demonstrate string I/O” ;
        outfile.close( ) ;
        ifstream infile(“Test.Dat”) ;       // create file for input
        shile (infile)                      // until end of file reached
        { infile.getline(str, 80) ;         // read content of file
          cout<<str<<endl ;                 // and display it
        }
        getch( ) ;
}
```

Output:

Characer I/O:

We can use put( ) [member of ostream] and get( ) [member of istream] to output and input a single character at a file.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main( )
{       char s[80] ;
        cout<< "Enter a string\n" ; cin>>s ;
        int len=strlen=strlen(s) ;
        fstream file ;              // input and output stream
        file.open ("Text", ios: : in/ios: : out) ;
        for (int i=0; i<len; i++)
        file.put(s[i]) ;            // put a character to file
        file.seekg(0) ;             // go to the start
        while (file)
          { file.get(ch) ;          // get a character from file
            cout<<ch ;              // display it on screen
          }
            return 0 ;
          }
```

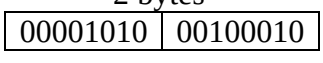
Output:

```
        Enter a string
        cplusplus_programming          input
        cplusplus_programming          output
```

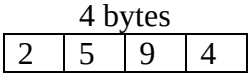
Binary I/O and Class Object I/O:

The functions write( ) and read( ) handle the data in binary form. This means that the values are stored in the disk file in the same format in the internal memory. How an int value 2594 is stored in the binary and character format. An int takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form.

Binary Format



Character Format



These function treats the data stored in the object as the sequence of bytes and make no assumptions about how these bytes should be handled.

The general syntax for input and output function is:

```
infile.read (cchar*)&v, sizeof(v)) ;
```

```
outfile.write((char*)&v, sizeof(v)) ;
```

Address of variable Length of variable v in bytes

The read( ) and write( ) function both takes two arguments. One to hold address of object and another sizeof( ) operator to hold length of objects in bytes. The address of the object must be cast to type pointer to char i.e. char*.

```
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
void main( )
{       clrscr( ) ;                    // simple character example
        ofstream fout("abc.txt") ;
        cout<< "Enter any integer Number" ;
        cin>>a ;
        fout<<a ;
        ifstream fin("abc.txt") ;
        fin>>b ;
        cout<< "content abc.txt is" ;
        cout<<b ;
        getch( ) ;
}
```

Output:

```
        Enter any Integer Number 23
        Content of abc.txt is 23
```

```
//eg of class object I/O or binary I/O
#include <iostream.h>
#include <fstream.h>
class info
{       char name[20 ] ;
        int roll ;
        public:
          void getdata( )
        { cout<< "Enter name and roll" ;
          cin>>name>>roll ; }
          void show( )
          { cout<<name<<roll ; }
        } ;
```

```
    int main( )
    { info X, Y ;
      x.getdata( ) ;
      ofstream fout("abc.txt")  ;
      fout.write ((char*) &X, size of (Y) ) ;
      fout.close( ) ;
      ifstream fin("abc.txt") ;
      fin.red ((char*) &Y, size of (Y)) ;
      Y.show( ) ;
      fin.close( ) ;
      return 0 ;
}
```

Output:

```
    Enter name and roll
      abc 12
      abc 12
```

```
// Program to read and write class object
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
class item
{   char name[10] ;  //item name
    int code ;          // item code
    float cost ;        // cost of each item
    public:
      void readdata(void) ;
      void writedata(void) ;
    } ;
    void item: : readdata(void)       // read from keyboard
    {
      cout<< "Enter name:" ; cin>>name ;
      cout<< "Enter code:" ;  cin>>code ;
      cout<< "Enter cost:" ; cin>>cost ;
    }
    void item: : writedata(void)      // formatted display on screen
    {
        cout<<setiosflags (ios: : left)
            <<setw(10)<<name
            <<setioflags(ios: :right)
```

```
                <<setw(10)<<code
                <<setprecision(2)
                <<setw(10)<<cost
                <<endl ;
}
int main( )
{       item i[3] ;          // Declare array of 3 objects
        fstream file ;       // input and output file
        file.open (“stock.dat”, ios: : in/ios: : out) ;
        cout<< “Enter Detail for Three Items \n” ;
        for (int j=0 ; j<3 ; j++)
        { i[j].readdata( ) ;
        file.write((char*) &i[j], sizeof (i[j])) ; }
        file.seekg(0) ;     // reset to start
        cout<< “\n Output: \n\n” ;
        for (j=0 ; j<3 ; j++)
        { file.read((char*) & i[j] , size of (i[j]) ;
          i[j].writedata( )
        }
        file.close( ) ;
        return 0 ;

}
```

Output:

```
        Enter Detail for Three items
        Enter name : Mango
        Enter code : 1001
        Enter cost : 100.00
        Enter name : Orange
        Enter code : 1002
        Enter cost : 150.00
        Enter name : Apple
        Enter code : 1003
        Enter cost : 200.00
```

Output:

```
        Mango   1001    100.00
        Orange  1002    150.00
        Apple   1003    200.00
```

Chapter – 10
Advanced C++Topics

10.1 Templates:

10.1.1 Introduction to Templates:

Template is one of the important features of C++ which enables us to define generic (generalized) classes and function. Thus, it helps us to perform generic programming. Generic programming is an approach where generic types are used a parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be considered as a macro which helps to create a family of classes or functions. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

There are two types of templates:

1. Function Templates
2. Class Templates

10.1.2 Class Templates:

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

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

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

P.U Question No. 01

```
#include <iostream.h>
class vector
{       int *v ;
        int size ;
```

```
        public:
          vector (int m)  // create a null vector
        { v=new int[size=m] ;
          for (int i=0 ; i<size ; i++)
                    v[i]=0 ;
         }
        vector (int *a)    // create a vector from an array
        { for (int i=0 ; i<size ; i++)
                    v[i]=a[i] ;
          }
        int operator*(vector &y)          // scalar product
        { int sum=0 ;
          for (int i=0 ; i<size ; i++)
                    sum+=this→v[i]*y-v[i] ;
                    return sum ;
        }
        } ;
int main( )
{       int x[3]={1, 2, 3) ;
        int y[3]={4, 5, 6) ;
        vector v1(3) ;     // creates a null vector of 3 integers
        vector v2(3) ;
        v1=x ;
        v2=y ;
        int R=v1*v2 ;
        cout<< "R="<<R ;
        return 0 ;
}
```

Output:

R=32

Example No. 2

```
#include <iostream.h>
const size=3
template <class T>
 class vector
        { T*V ;   // type T vector
        public:
          vector( )
        { v=new T[size] ;
```

```
        for (int i=0 ; i<size ; i++)
                v[i]=a[i] ;
    T operator*(vector &y)
    { T sum=0 ;
      for (int i=0 ; i<size ; i++)
      sum+=this→v[i]*y.v[i] ;
      return sum ;
    } } ;
    int main( )
    { int x[3]={1, 2, 3} ;
      int y[3]={4, 5, 6) ;
      vector <int>v1 ;
      vector<int>v2 ;
      int R=v1*v2 ;
    cout<< "R="<<R<< "\n" ;
    return 0 ;
    }
```

Output: R=32

Example No. 3

```
#include <iostream.h>
template <class T>
class test
{ T   a,b ;
    public:
            void getdata( )
      {cin>>a>>b ;}
      void putdata( )
    {cout<< "You Entered:"<<a<< " "<<b ;}
} ;
int main( )
{     test<int>t1 ;
    t1.getdata( ) ; t1.putdata( ) ;
    test<float>t2 ;
    t2.getdata( ) ; t2.putdata( ) ;
    test(char>t3 ;
    t3.getdata( ) ; t3.putdata( )
    return 0 ; }
```

Output:

    1 2

```
You Entered : 1  2
1.5  2.5
You Entered : 1.5  2.5
x y
You Entered : x  y
```

Class template with multiple parameter:

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

```
template<class T1, class T2, .....>
class classname
{
      - - - - -
      - - - - -  (Body of the class)
      - - - - -
} ;
#include <iostream.h>
template <class T1, class T2>
class Test
{     T1 a ;
      T2 b ;
      public:
                  Test(T1 x, T2 y)
      { a=x ;
        b=y ;
      }
      void show( )
      {
        cout<<a<< “and” <<b<< “\n” ;
      }
      } ;
      int main( )
      {
      Test<float, int> test1(1.54, 154) ;
      Test<int, char> test2 (100, ‘w’) ;
      test1.show( ) ;
      test2.show( ) ;
      return 0 ;
}
```

Output:

1.54 154
100 w

10.1.1 Function Templates:

Similar to class template, the function template will be used to create a family of function with different argument type. The general syntax of a function template is

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

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

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

Output:

```
m and n before swap : 100   200
m and n after swap :  200   100
```

a and b before swap : 11.22 33.44
a and b after swap : 33.44 11.22

Function templates with Multiple Parameters:

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

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

```
#include <iostream.h>
template <class T1, class T2>
void show(T1x, T2 y)
{
    cout<<x<< " "<<y<< "\n" ;
}
int main( )
{
    show(1981, "C++") ;
    show(12.34, 1234) ;
    return 0 ;
}
```

Output:
1981 C++
12.34 1234

10.1.3 Standard Template Library:

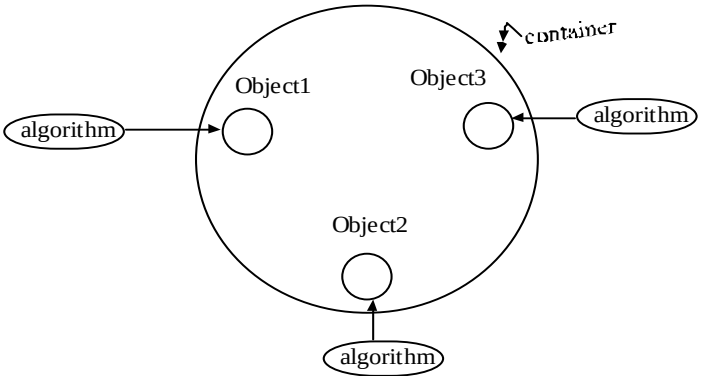
Standard Template Library (STL) is a collection of general-purpose templatized classes and functions (algorithms) that could be used as a standard approach for storing and processing of data. STL was developed by Alexander Stepanov and Meng Lee of Hewlett Packward.

It is an integrated part of ANSI (American National Standard Institute) standard C++ class library. It helps to save C++ users' time and effort there by helping to produce high quality programs. STL components are defined in the namespace std. So, we need to use "using namespace" directive in order to use STL.

Components of STL:

STL has 3 most important components:

a) containers
b) Algorithms
c) Iterators



a) Container:
It is an object that stores data. It is the way of organizating data into memory. STL containers are implemented by template class and therefore can be easily customized to hold different types of data like vector, set, queue, stack, etc.

b) Algorithms:
It is a procedure that is used to process data contained in the containers. STL consists of different algorithms for different tasks like initializing, searching, storing, copying, merging, etc. These algorithms are implemented by template functions.

c) Iterators:
It is an object (like a pointer) that points to elements in a container. It can be used to move the contents of container. As it works just as a pointer, it can be incremented or decremented. They help to connect algorithms with containers and manipulate data stored in containers. There can be different types of iterators like bidirectional, random access iterators, etc.

Table : Containers supported by the STL

Containers	Descriptions	Header Files	Iterators
vector	A dynamic array. Allows instertions and deletions at back. Permits direct access to any element.	<vector>	Random access
List	A bidirectional, linear list. Allows insertions and deletion anywhere.	<list>	Bidirectional
set ⋮ More	An associate container for storing unique sets. Allows rapid lookup. No duplicates allowed	<set>	Bidirectional

10.3 Namespaces

10.3.1 Introduction:

Namespace is a new concept introduced by teh ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive, like

```
using namespace std ;
```

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. Using and namespace are the new keywords of C++.

10.3.2 Declaring a Namespace:

We can define our own namespace in programs. The general sytanx is:

```
namespace      namespace_name
{
```

```
        // Declaration of
        // variable, function, classes, etc
    }
```

For example

```
#include <iostream.h>
using namespace std ;
namespace name
    { int m=20 ;
      void display(int n)
      { cout<< "n="n ;
      }
    void main( )
    { using namespace name ;        // using directive
      cout<< "m="<<m<<endl ;
      display(22) ;
      }
```

Output:

```
    m=20
    n=22
```

Thus, we have seen in above exmaple that we could access all the members declared within the specified namespace by using the "using directive" If we want to access only the particular member within the namespace, then we have to use "using declaration" like if we want to use member 'm' only:

```
void main( )
    { using name: : m         // using declaration
      cout<< "m="<<m ;
    }
```

So, general syntax for using directive as

```
      using namespace namespace_name ;
```

Nesting of Namespaces and unnamed Namespaces:

A namespace can be nested within another namespace. For example:

```
namespace NS1
{
    - - - - - -
    - - - - - -
      namespace NS2
      {
          int m=100 ;
      }
```

```
        - - - - - -
        - - - - - -
    }
```

In order to access m we should write

```
    cout<<NS1: : NS2: :m ; OR
using namespace NS1 ;
    cout<<NS: : m ;
```

An unnamed namespace is one that does not have a name. Unnamed namespace member occupy global scope and are accessible in all scopes. For example:

```
// using unnamed Namescope with Nesting
#include <iostream.h>
using namespace std ;
namespace Name1
    { double x=5.67 ;
      int m=10 ;
    namespace Name2        // Nesting namespace
    {
      doubly y=4.56 ;
    }
    namespace              // unnamed namespace
    { int m=20 ;
    }
void main( )
    { cout<< "x="<<Name1: : x<< "\n" ;  // x is qualified
      cout<< "m="<<Name1: :m<< "\n" ;
      cout<< "y="<<Name1: : Name2: : y<< "\n" ; // ys is fully qualified
      cout<< "m="<<m<< "\n" ;     // m is global
}
```

Output:

```
    x=5.67
    m=10
    y=4.56
    m=20
```

10.4 Exceptions:

10.4.1 Introduction:

The most common types of error(also known as bugs) occured while programming in C++ are Logic error and Syntactic error. The logic errors occur due to poor understanding of

the problem and solution procedure. The syntactic errors arise due to poor understanding of the language. These errors are detected by using exhaustive debugging and testing.

We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions. Exceptions are ru time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of membory or disk space. ANSI C++ provides built-in language features to detect and handle exception which are basically run time errors. Exception handling was added to ANSII C++, provides a type_safe approach for copying with the unusual predictable problems athat arise while executing a program. So, exception handling is the mechanism by using which we can identify and deal such unusual conditions.

10.4.2 Exception Handling Mechanism:

Exceptions are of two kinds, namely, synchronous exception and asynchronous exception. Errors such as “out of range index” and “overflow” belong to synchronous type exceptions whereas errors that caused by events beyond the control of the program (such as keyboard interrupts) are known as asynchronous exceptions.

The purpose of the exception hanlding mechanism is to provide means to detect and reprt an “exceptional circumstances” os that appropriate action can be taken. The exception handling mechanism in C++ can handle only synchronous exceptions. The exception handling mechanism performs the following tasks:

a) Hit the exception i.e. find the unusual condition
b) Throw the exception i.e. inform that error has occured
c) Catch the exception i.e. receive the error information
d) Handle the exception i.e. take the aciotn for correction of problem

In C++, the following keywords are used for exception handling.

a) try
b) catch
c) throw

10.4.3 Exception Handling Construct: try, throw, catch

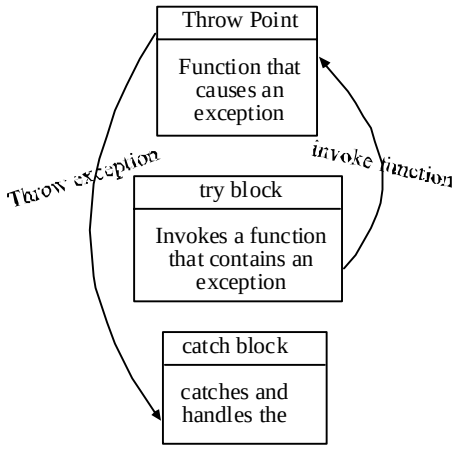


Fig : Function invoked by
try block throwing exception

The keyword 'try' is used to preface a block of statements surrounded by braces which may generate exceptions. This block of statement is known as try block. When an exception is detected, it is thrown using a throw statement in the try block.

A catch block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in the try block, and handlies it appropriately. Fig(a) shows try-catch relationship. The catch block must immediately follow the try block that throws the exception. The general syntax is

```
 - - - -
try
{ - - - -
  - - - -                    // block of statement which detects and throws exception.
  throw exception ;
   - - - -
}
catch(type arg)
{
  - - - -          // block of statements that handles the exception
  - - - -
}
```

Throwing Mechanism:

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

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

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

Catching Mechanism:

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

```
catch(type arg)
{
  // statements for
  // managing exceptions
}
```

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

```
#include <iostream.h>
void main( )
{   int m, n ;
    cout<< "Enter first no.:" ; cin>>m ;
    cout<< "Enter second no." ; cin>>n ;
try
    {
    if (n!=0)
      {     cout<< "division="<<m/n<<endl ; }
    else
      { throw(n) ; }  // throw object of int
    } //end of try block
    catch (int a)
    { cout<< "there is an exception division by zero \n" ;
      << "second no."<<n<<endl ;
    } // end of catch block
    }
```

The output: (1st run)

```
    Enter first no. : 6
    Enter second no. : 3
    division=2
```

(2nd run):

```
    Enter first no. : 6
    Enter second no. : 0
There is an exception divison by zero
    second no. = 0
```

```
// Testing throw restriction
#include <iostream.h>
void test(int x) throw(int, double)
{   if (x = =0) throw 'x' ;    // char
    else
      if (x = = 1) throwx ;    // int
      if (x = = -1) throw 1.0 ; // double
    cout<< "End of function block \n" ;
```

```
}
int main( )
{       try
          { cout<< “Testing Throw Restriction \n” ;
            cout<< “x = = 0 \n” ;
            test(0) ;
            cout<< “x = = 1 \n” ;
            test(1) ;
            cout<< “x = = -1 \n” ;
            test(-1) ;
            cout<< “x = = 2 \n” ;
            test(2) ;
          }
        catch (char c)
        { cout<< “caught a character \n” ;
        }
        catch (int m)
        { cout<< “caught a integer \n” ;
        }
        catch (double d)
          {cout << “caught a double \n” ; }
          cout<< “End of try-catch system \n\n” ;
          return 0 ;
        }
```

Output:

```
        Testing Throw Restriction
        x = = 0
        caught a character
        End of try-catch system
```