Object Oriented Programming

Unit -1

Software Evolution

The software evolution occurred in several phases. Since the beginning of the first computer, programming for the computer started to develop software. The earlier electronic computer ENIAC was programmed in machine language by using switches to enter 1 and 0.

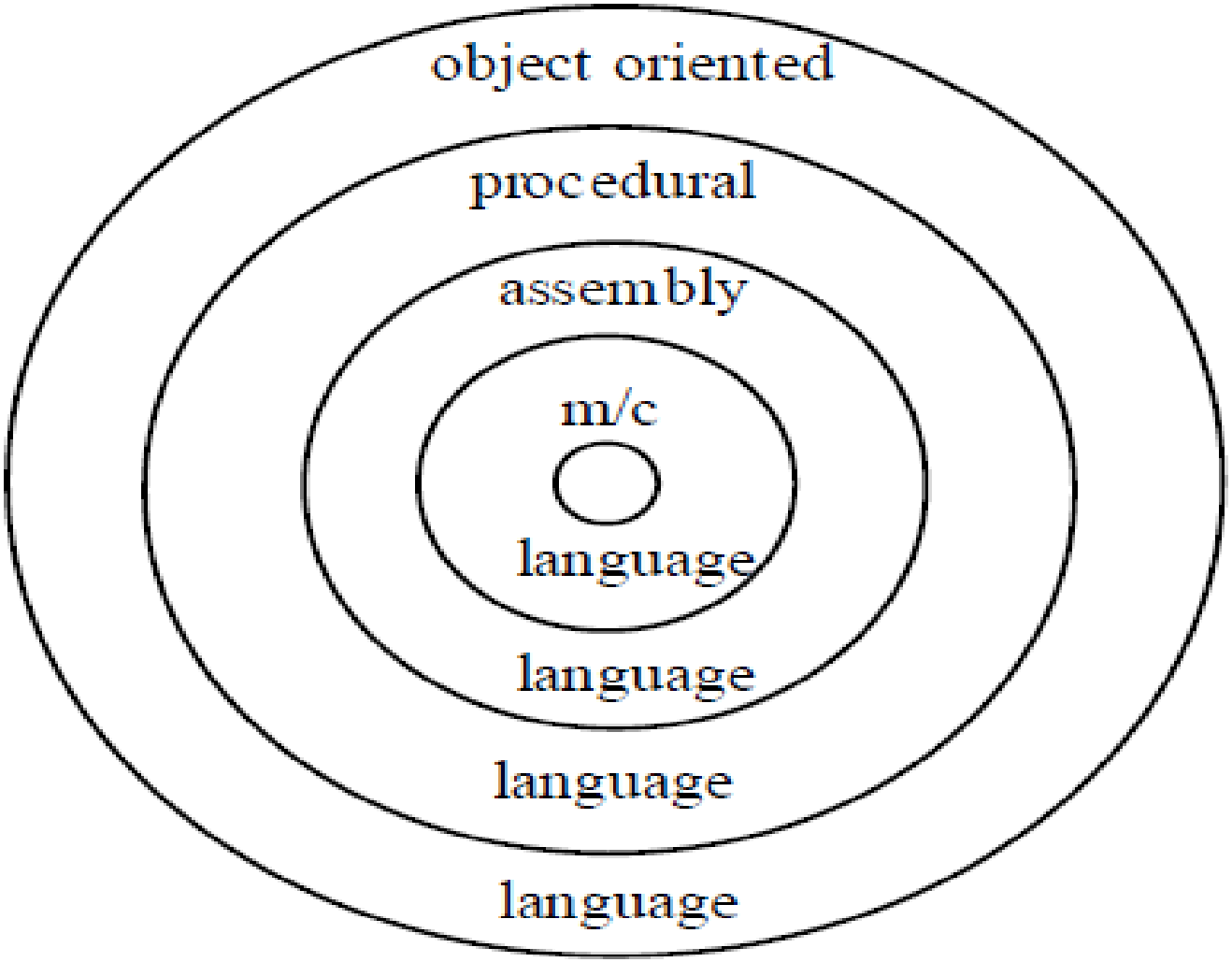


Fig: Language Paradigm

Procedure-Oriented Programming:

In procedure oriented programming a large program is broken down into smaller manageable parts called procedures or functions.

In procedure oriented programming priority is given on function rather than data. In procedure oriented programming language, a program basically consists of sequence of instructions each of which tells the computer to do something such as reading inputs from the user, doing necessary calculation, displaying output.

High Level Programming Languages like COBOL, FORTRAN, Pascal, C are common procedure oriented programming languages.

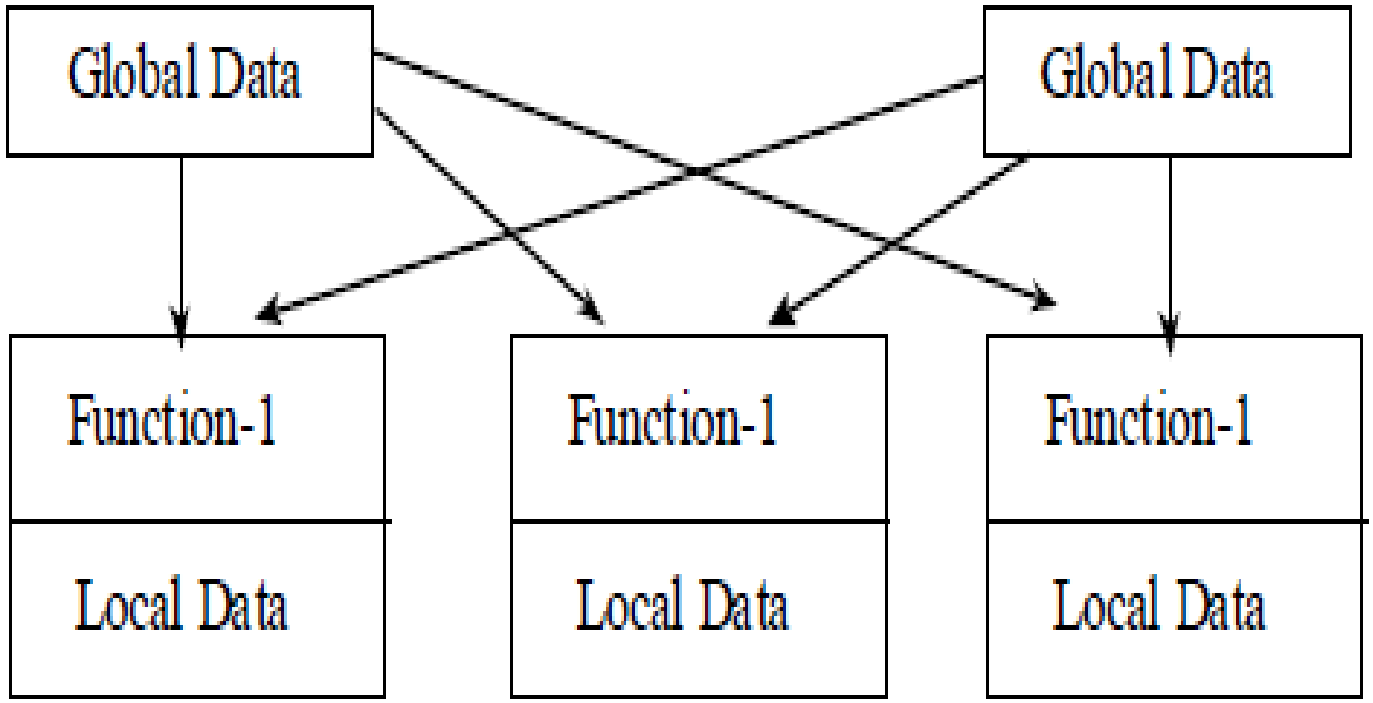


Fig: Relationship of data and functions in procedure programming

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

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

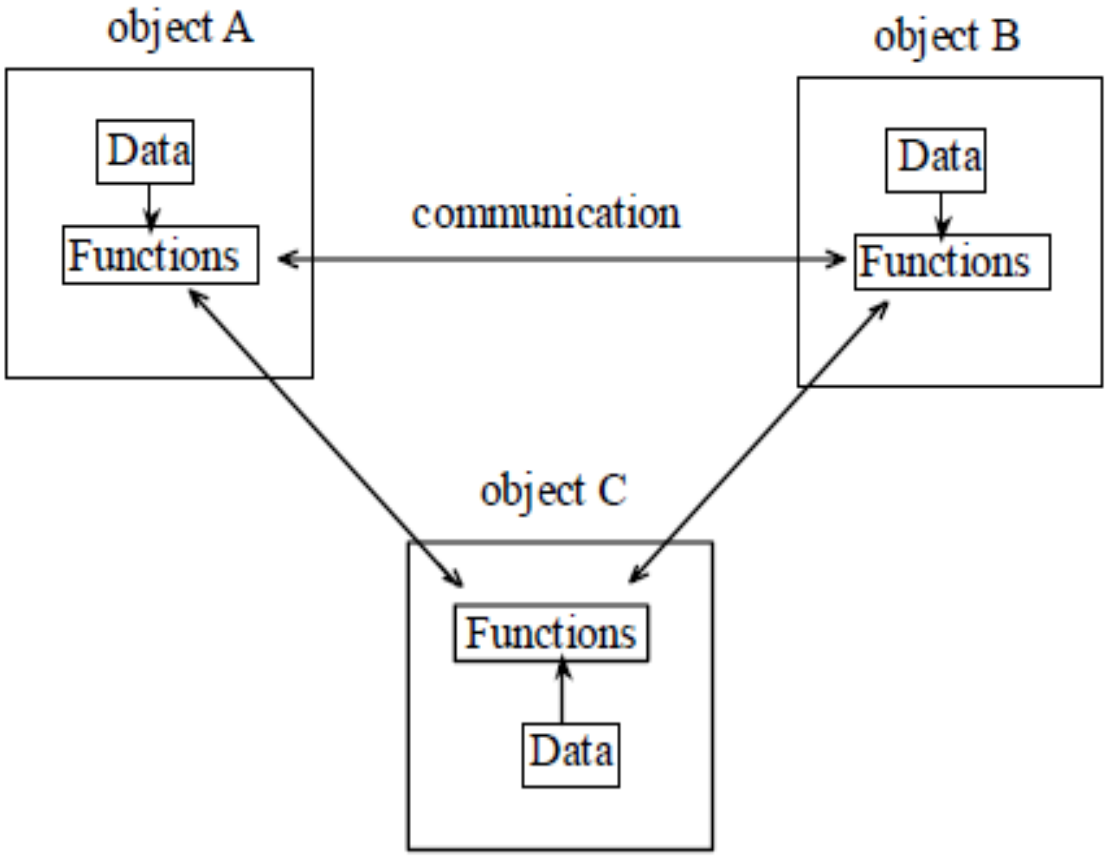


Fig: Organization of data & functions in OOP

Characteristics of OOPs

Some noticeable characteristics of OOP are as follows:

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Function and data are tied together in a single unit.
- Data can be hidden to prevent from accidental alteration from other function or objects.
- Data access is done through the visible functions so that communication between objects is possible.
- Data structures are modelled as objects.
- Follows Bottom up approach of program design methodology.

Procedure oriented versus Object oriented programming

The differences between procedural and Object oriented programming are tabulated below:

Procedure Oriented Programming	Object Oriented Programming
Emphasis is given on procedures.	Emphasis is given on data.
Programs are divided into functions.	Programs are divided into objects.
Follow top-down approach of program design.	Follow bottom-up approach of program design.
Generally data cannot be hidden.	Data can be hidden, so that non-member function cannot access them.
It does not model the real world	It models the real world problem

problem perfectly.	very well.
Data move from function to function.	Data and function are tied together. Only related function can access them
Maintaining and enhancing code is still difficult.	Maintaining and enhancing code is easy.
Code reusability is still difficult.	Code reusability is easy in compare to procedure oriented approach.
Examples: FORTRAN,COBOL, Pascal, C	Example: C++, JAVA, Smalltalk

Characteristics of Object-Oriented Language

- **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.

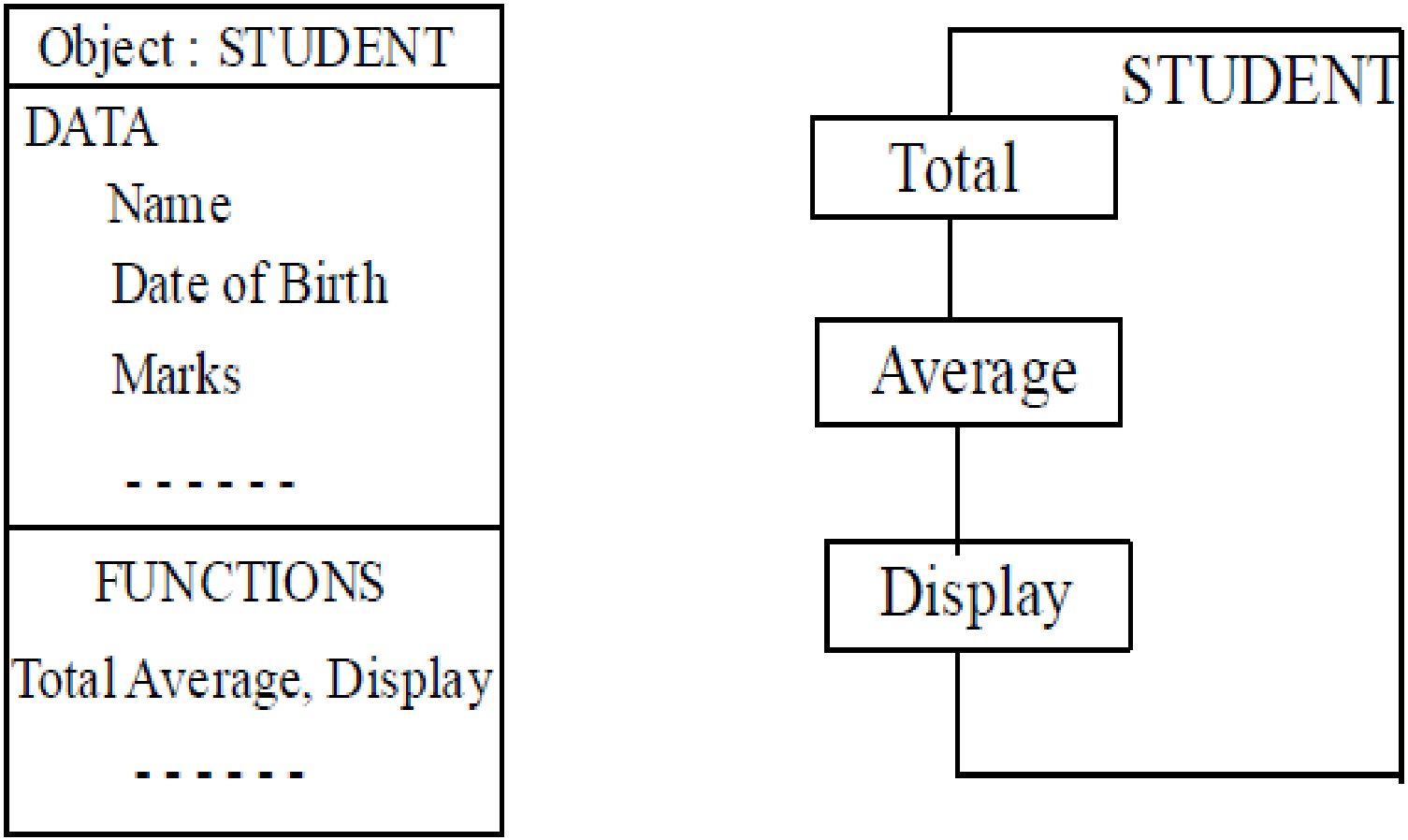


Fig: Two way of representing an object

- **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

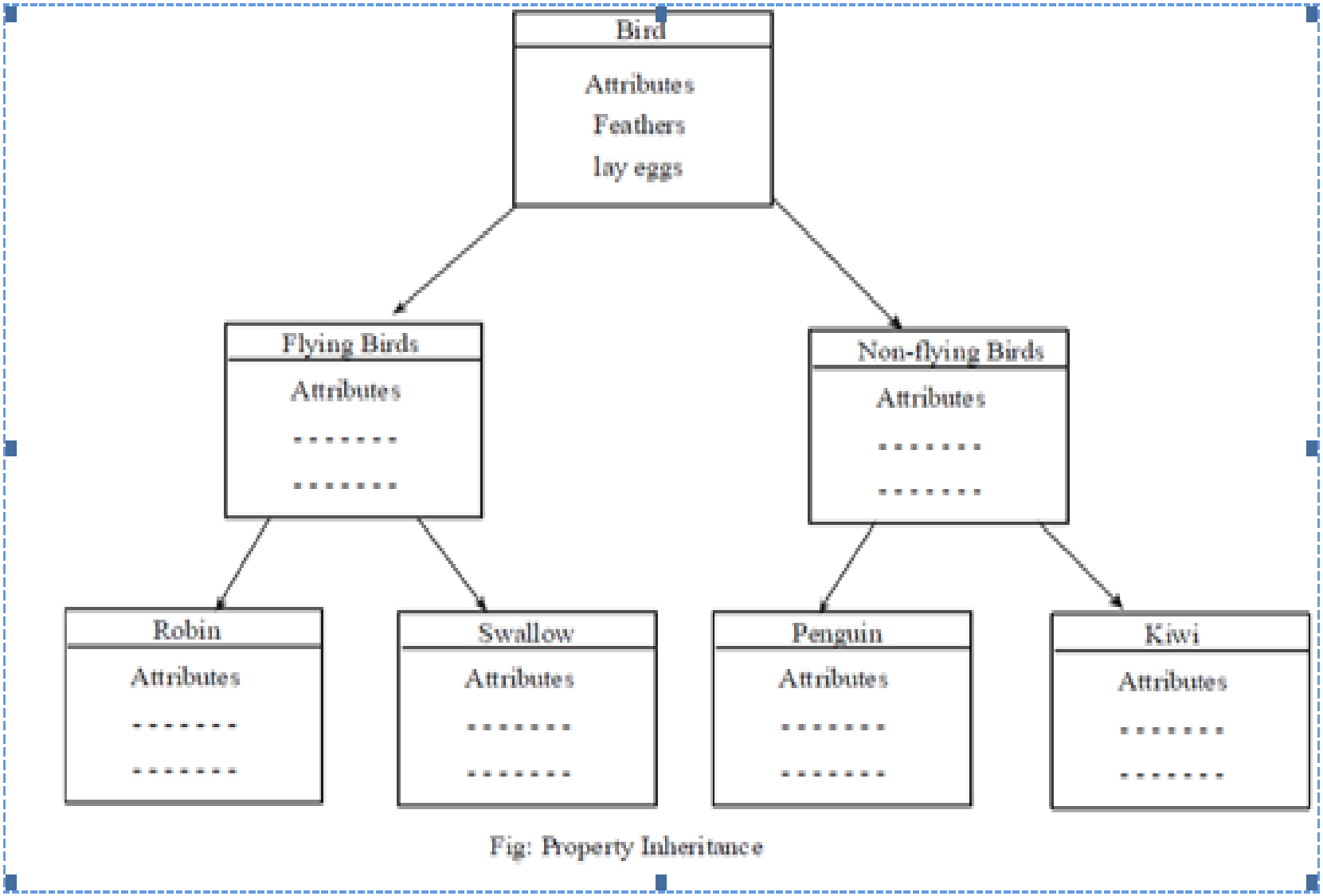


Fig: Property Inheritance

- **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.

- **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.

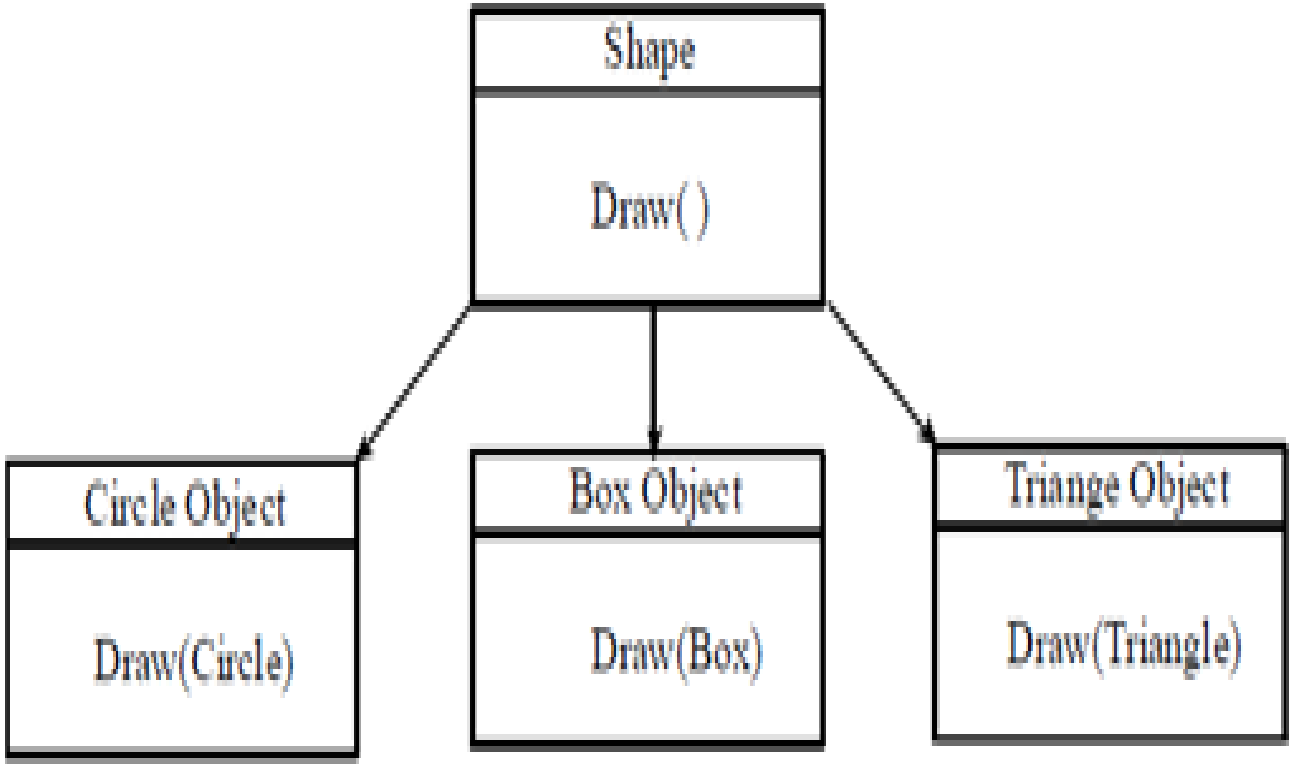


Fig: Polymorphism

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

Benefits of using OOP:

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.

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.

Unit 2: Introduction to c++

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

- 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 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

Unit-3

C++ Language Basic Syntax

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:

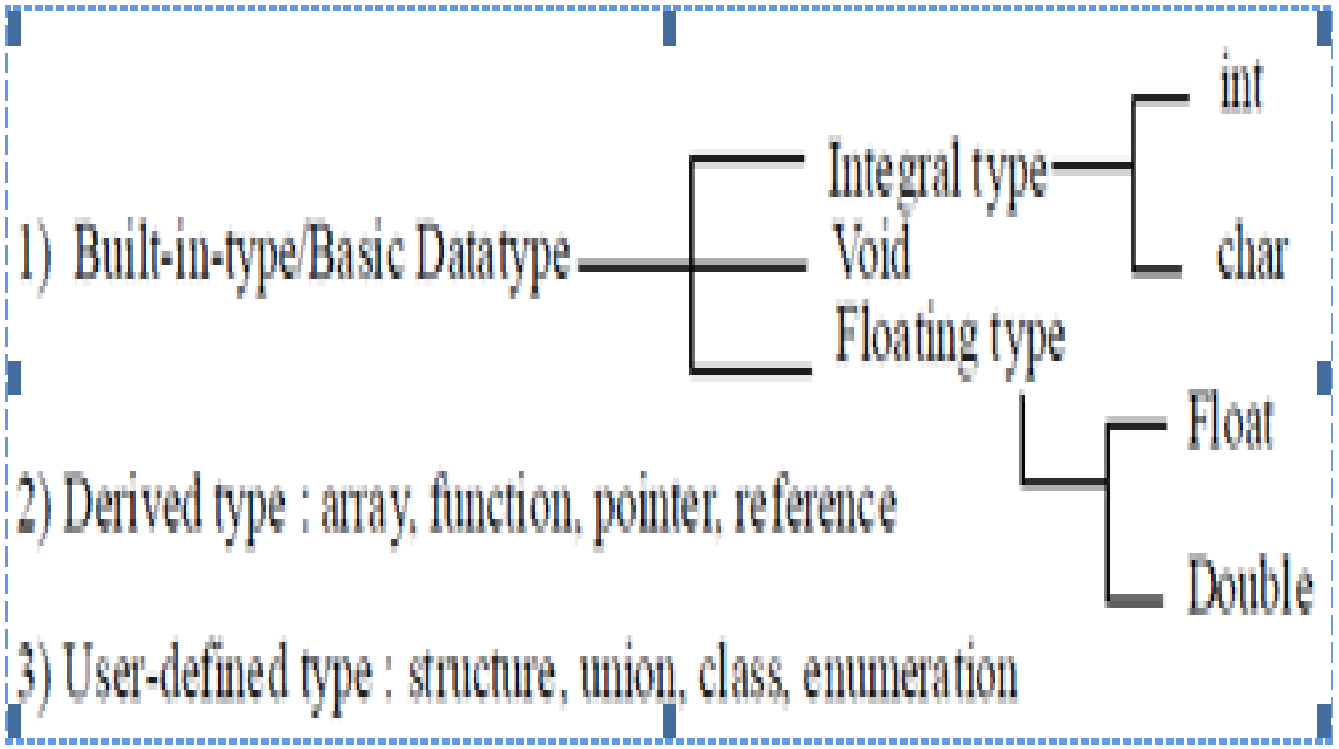
123 // decimal

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

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) ;
```

Derived Data Types

- Derived datatypes are those which depend on built –in data types.That’s Why ,they are called so. For Eaxamples: Arrays, Pointers, function etc.

Arrays and Pointer

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] ;

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.

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
{
};
```

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>
#include<conio.h>
int a=10 ; // global a
void main( )
{
 clrscr( );
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 ;
getch();
}
```

Output:

Local a=15 Global a=10

Local a=15 Global a=20

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.

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.

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.

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.

For e.g.

p = new int ;

q = new float ;

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 ;

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 ;

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.

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

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} ;
```

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.

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 */

Input/output basic

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” ;

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.

Function Definition:

A function is a group of statements that is executed when it is called from some point of the program. 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.

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.

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:

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.

Function prototype is a declaration statement in the calling program and is 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

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 swap(int *a, int *b)
{
int t ;
t=*a ;
*a=*b ;
*b=t ;
}
void main( ) {
clrscr( );
int x,y ;
x=10 ;
y=20 ;
swap(x,y) ;
cout<< “x=” <<x<<endl ;
cout << “y=”<<y<<endl ;
getch( ) ;
}
```

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>
#include<conio.h>
// Declarations (prototypes)
int area(int) ;
double area(double, int) ;
long area(long, int, int) ;
int main( )
{ clrscr( );
cout<<area(10)<< “\n” ;
cout<<area(2.5,8)<< “\n” ;
cout<<area(100L,75,15)<< “\n” ;
return 0 ;
getch( );
}
```

CONTI…..

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

Inline 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 inline functions. To declare an in-line function, simply precede the function's definition with the inline specifier.

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

Unit 4: Classes and Objects

- 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-specifier:
member_data1 ;
member_data2 ;
- - - - - - - -
- - - - - - - -
access-specifier:
member_function1 ;
- - - - - - - -
- - - - - - - -
} ;
```

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 ;

```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream.h>
#include<conio.h>
class smallobj
{
private: //specify a class
int somedata ; // class data
public:
void setdata(int a) // member function
{ somedata=d ;} // to set data
void showdata( ) //member function to display data
{
cout<< “\n Data is” <<somedata ; }
} ;
```

```
void main( )
{ clrscr( );
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( ) ;
getch ( );
}
```

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.”

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 ;

- Public
- Protected
- 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.

e.g.

```
#include <iostream.h>
#include<conio.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( )
{ clrscr( );
rectangle r1, r2, r3 ; //define three objects of class rectangle
r1.getdata(10, 20) ; //setdata in elements of the object.
r1.setdata( ) ; //display the data set by setdata( )
r1.area_Peri( ) ; //calculate and print area and perimeter
r2.getdata(5, 8) ;
r2.setdata( ) ;
r2.area_Peri( ) ;
r3.getdata( ) ; //receive data from keyboard
r3.setdata( ) ;
r3.area_Peri( ) ;
getch ( );
}
```

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

Class Objects

The general syntax:

class_name object_name1, object_name2, ..., object_name n ;

rectangle r1, r2, r3 ; objects of class rectangle

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.”

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>
#include<conio.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 : : setdata(int l, int b)
{
len=l ;
br=b ;
}
void rectangle: : display( )
{
cout<<endl<< “length:”<<len ;
cout<<endl<< “breadth:”<<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( )
{ clrscr( );
rectangle r1, r2, r3 ;
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( ) ;
getch( );
}
```

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.

‘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>
#include<conio.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()
    {
     clrscr( );
    thisemp e1 ;
    e1.setdata(10) ;
    e1.showdata( ) ;
    getch( );
    }
```

Output:

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

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>
#include<conio.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( ) {
clrscr( );
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( ) ;
getch( );
}
```

Output:

```
count : 0
count : 0
count : 0
value of count after calling getdata function:
count : 3 count :3 count : 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>
#include< conio.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( )
{ clrscr( );
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( ) ;
getch( );
}
```

Output:

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

Returning Objects from Functions

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

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

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

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>
#include< conio.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( )
    {
    clrscr();
    example x ; //object x
    x.setvalue( ) ;
    cout<< “Mean value=” <<mean(x)<< “\n” ;
    return 0 ;
    getch( );
     }
```

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

Constructor:

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

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.

```
// 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

- Constructor has same name as that of its class name.
- It should be declared in public section of the class.
- It is invoked automatically when objects are created.
- It cannot return any value because it does not have a return type even void.
- It cannot have default arguments.
- It cannot be a virtual function and we cannot refer to its address.
- It cannot be inherited.
- 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 ;
}
myclass: : void 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 ;}
```

```
    myclass: : void 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 ; }
- } ;

Other Contructors : 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 ;
 }
} ;
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 created6
- No of object destroyed6

```
// 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

Unit 5: Operator Overloading:

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

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.

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 --.

```
// 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.

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.

```
// 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

Data Conversion:

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=inches/12 ;
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"

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 X, therefore, Y is source class and X is destination class.

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

- By constructor:
- By conversion function, i.e. the overloaded casting operator function.

By constructor:

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

By Conversion Function:

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

```
//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
```

Unit 6: Inheritance

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.

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

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.

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.

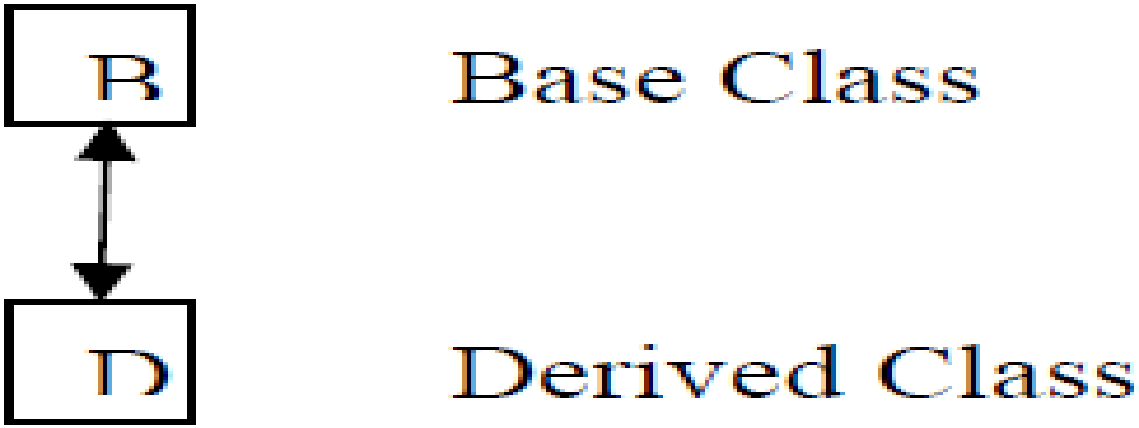


Fig : Single Inheritance

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
```

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
} ;
```

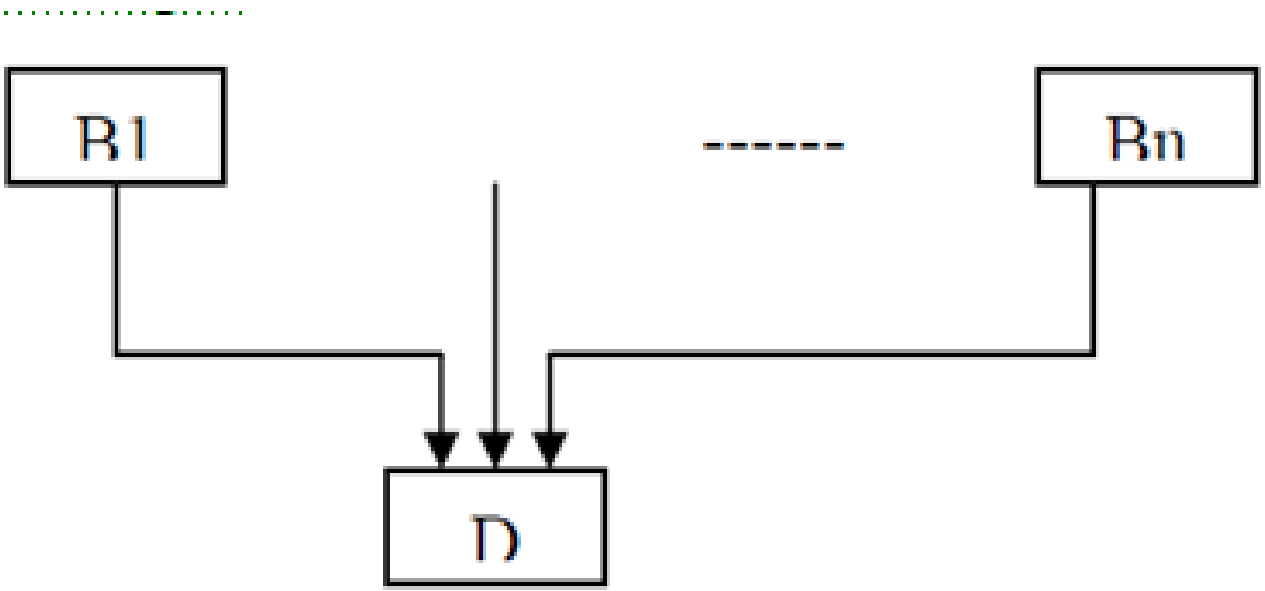


Fig: Multiple inheritance

```
// 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.

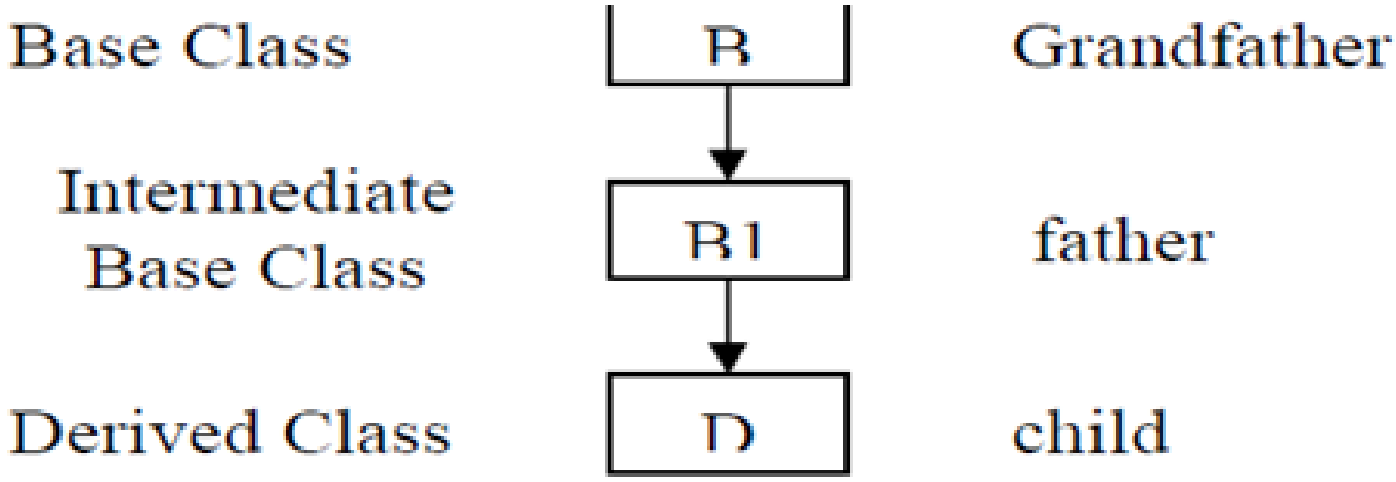


fig : multilevel inheritance

```
#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

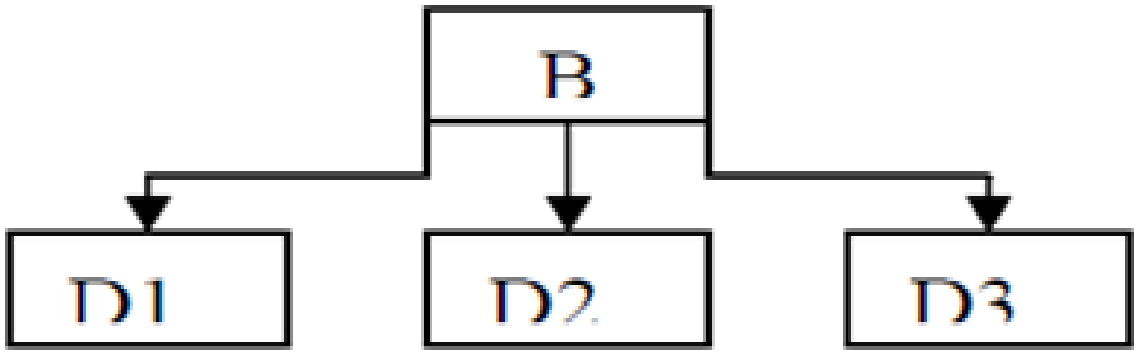


Fig: Hierarchical Inheritance

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

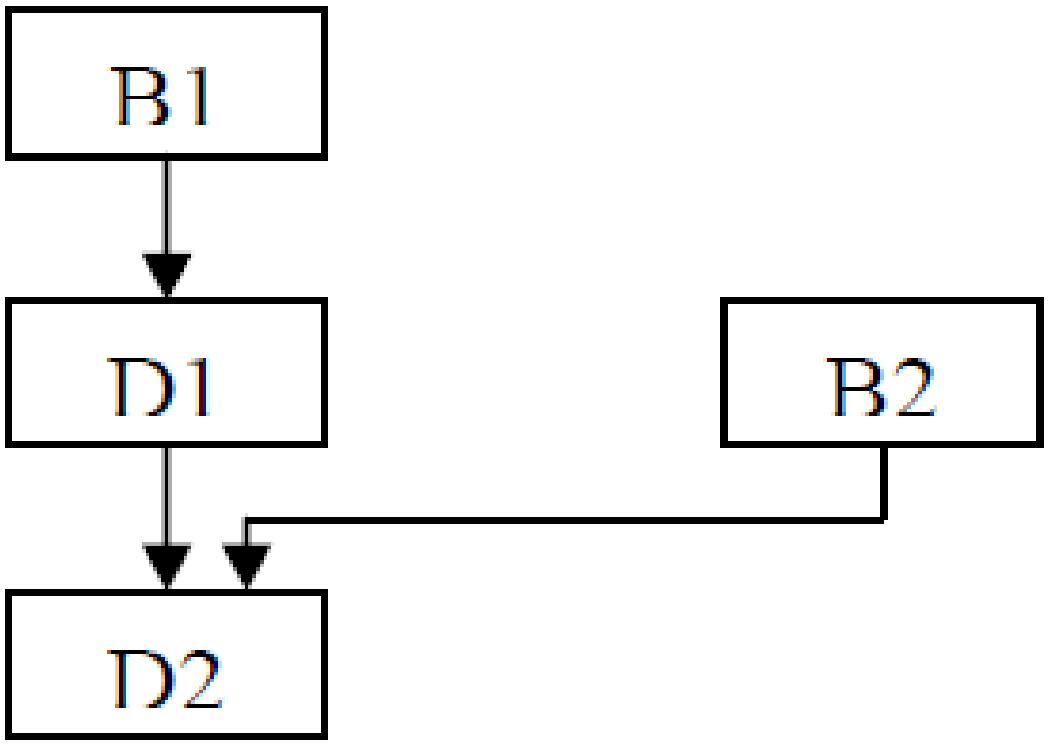


Fig : Hybrid Inheritance

```
#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

Benefits 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.
4. Code sharing can occur at several places.

Unit 7: 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 .

Classification of Polymorphism:

1. **Compile time polymorphism**: The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time, therefore, compiler can select appropriate function. So, this is known as compile time polymorphism. The example of compile time polymorphism are

   a) function overloading

   b) operator overloading

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

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. The virtual function is invoked at run time based on the type of pointer.

For e.g.

```
#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
```

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.

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

```
//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
    }
```

Unit 8: 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. Some input/output functions are : cin (), cout (), get(),put() etc.

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.

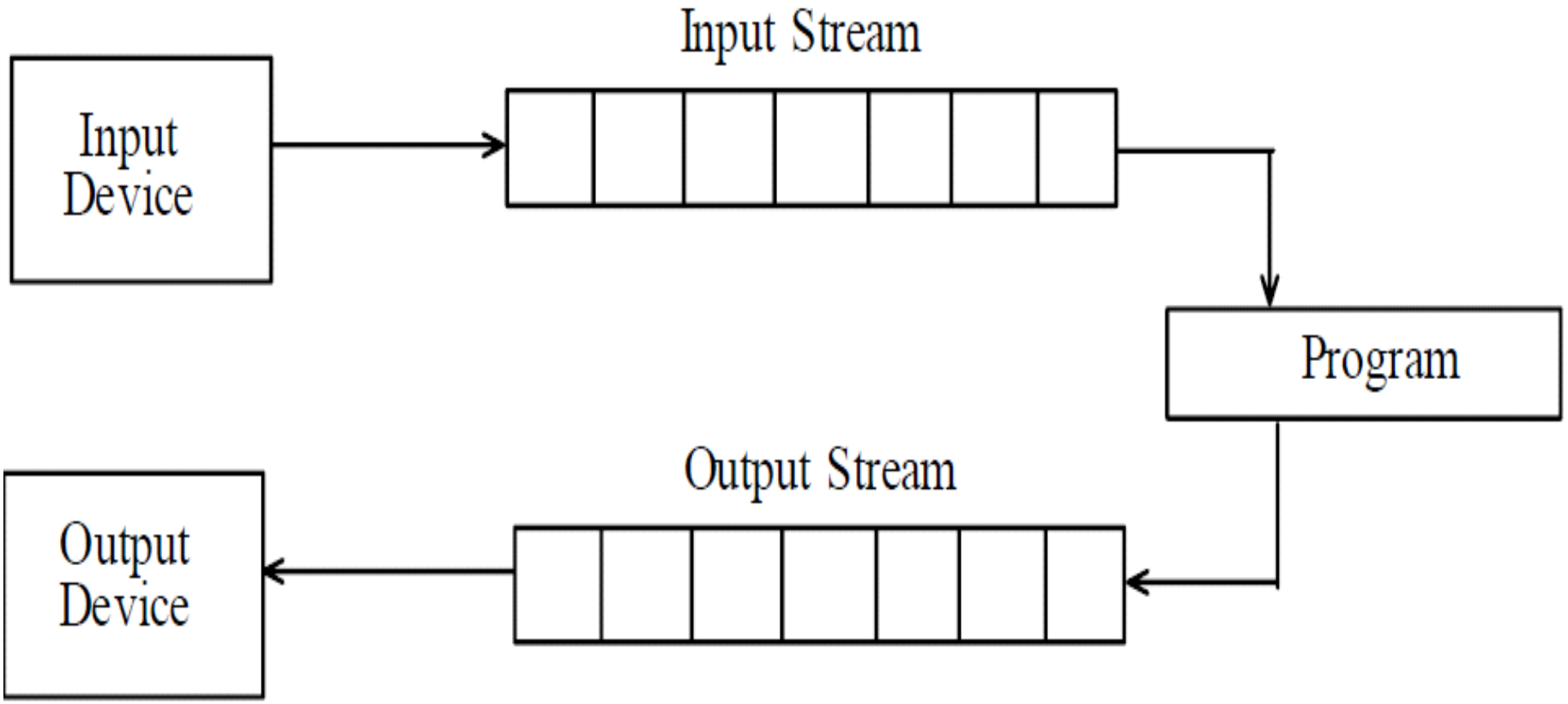


Fig : Stream based I/O

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,

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

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

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.

For e.g.

```
char c;
cin.get(c) ; // obtain single character from keyboard and assign it to char c
```

Put( ):

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

for e.g.

```
cout.put ('T') ; // prints T
```

getline( ): 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

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.

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

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) ;

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” ;
```

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.

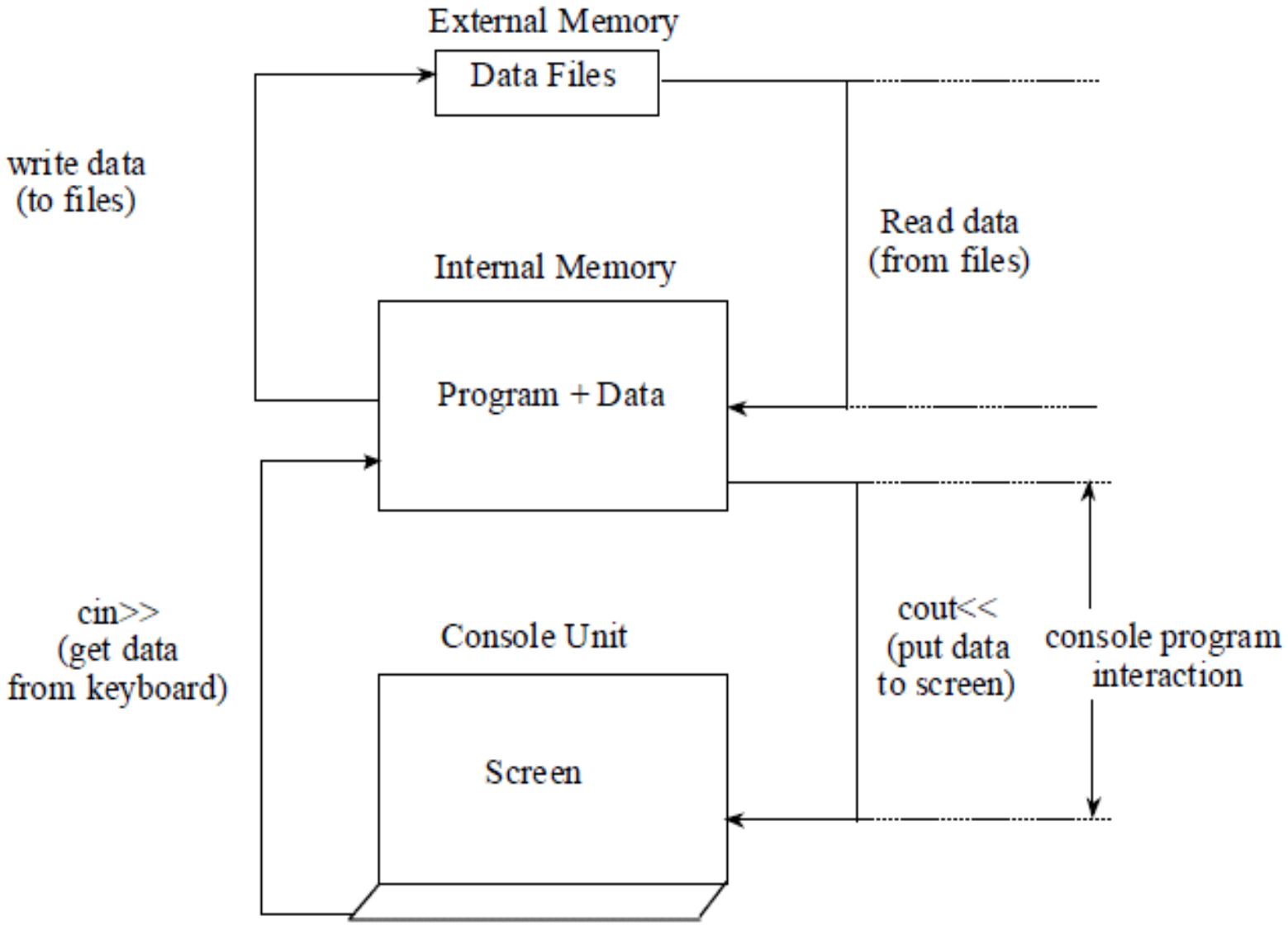


Fig : console program file interaction

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. 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

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.

The available manipulator in C++ are:

1. seekg( ): This move gets pointer i.e. input pointer to a specified location.
2. seekp( ): This move puts pointer (output pointer) to a specified location.
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).

Error Handling in File:

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

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

```
// 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

Unit 9: **Templates**

A template is one of the recently added feature in c++. It supports the generic data types and generic programming. Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types. Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types.

Templates offer several advantages:

- Templates are easier to write.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Templates in c++ comes in two variations

a) function templates

b) class templates

Class template

The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

The general form of a class template is

```
Template <class T>
Class class_name
{
//class member with type T whenever appropriate
};
```

Advantages of C++ Class Templates:

- One C++ Class Template can handle different types of parameters.
- Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Function template

To perform identical operations for each type of data compactly and conveniently, we use function templates. we can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

The general form of a function template is

```
Template<class T>
returnType function_name (argument of type T)
{
//body of function with type T whenever appropriate
};
```

The End