1
C++ FUNDAMENTALS

1.1 OBJECT ORIENTED PROGRAMMING PARADIGM

Object oriented programming (oops) treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protect it form modification from outside functions. oops allows decomposition of a problem into a number of entities called objects and then builds data and functions around those objects. The organization of data and functions in object oriented program is shown as

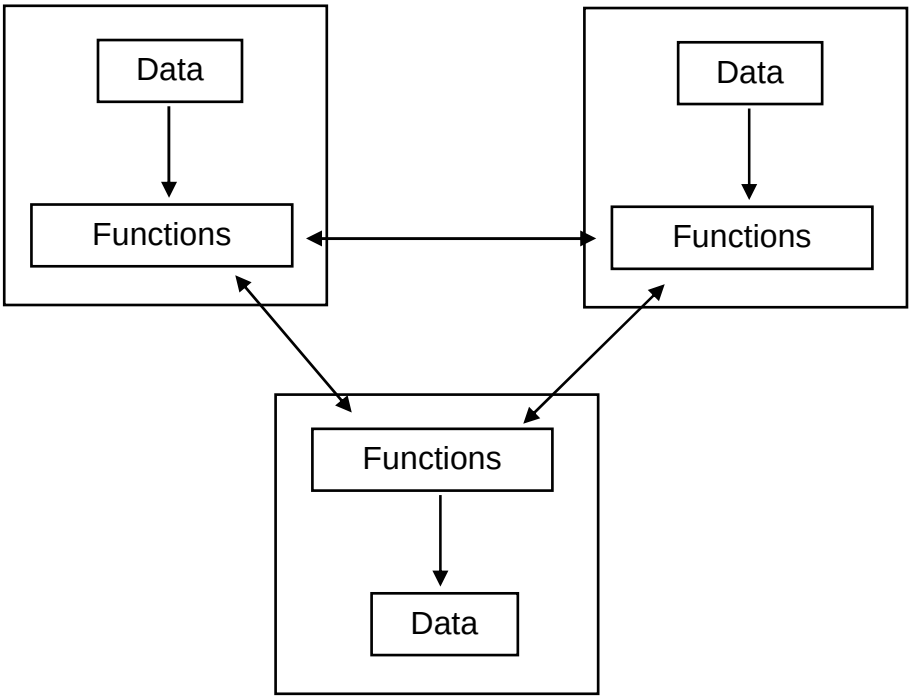


Fig. 1.1 Organization of data & functions in oops

Some of the features of oops are.

1) Emphasis is on data rather than procedure.
2) Programs are divided into what are known as objects.
3) Data structures are designed such that they characterize the objects.
4) Functions that operate on the data of an object are tied together in the data structure.
5) Data is hidden and cannot be accessed by external functions.
6) Objects may communicate with each other through functions.
7) New data and functions can be easily added wherever necessary.

8) Follows bottom up approach in program design.

1.5 BASIC CONCEPT IN OOPs –

Some important concept in oops are
1) Objects
2) Classes
3) Data abstraction & Encapsulation.
4) Inheritance
5) Dynamic binding.
6) Message passing.

1) Object :-

i) Object are the basic run-time entities in an object-oriented system.
ii) They may represent a person, a place a bank account, a table of data or any item that the program has to handle.
iii) Programming problem is analyzed in terms of objects and the nature of communication between them.
iv) Objects take up space in the memory & have an associated address like structure in c.
v) When a program executes, the object interacts by sending messages to one another. Ex. If there are two objects "customer" and "account" then the customer object may send a message to account object requesting for the bank balance. Thus each object contains data, and code to manipulate the data.

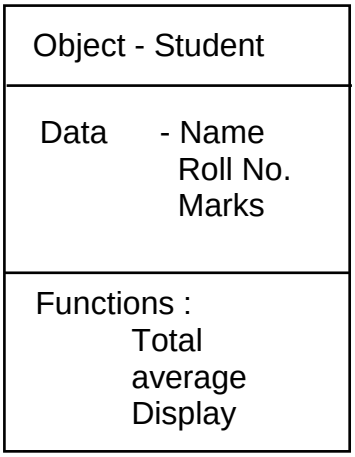


Fig. 1.2 Student Object.

2) Classes

i) The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are actually variable of the type class.
ii) Once a class has been defined, we can create any number of objects belonging to that class. Thus a class is collection of objects of similar type.
iii) Classes are user defined data types and behaves like the built in type of a programming language.
iv) The syntax for defining class is

```
class class-name
{
----------
----------
}
```

3) Data abstraction and Encapsulation

i) The wrapping up of data and functions into a single unit called class is known as encapsulation.
ii) The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
iii) These functions provide the interface between the objects data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.
iv) Abstraction refers to the act of representing essential features without including the background details or explanations.
v) Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and coast, and functions to operate on these attributes.

4) Inheritance

i) Inheritance is the process by which object of one class acquire the properties of objects of another class.
ii) In OOPs, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.
iii) This is possible by deriving a new class from the existing one. The new class will have combined features of both the classes.

5. Polymorphism

i) Polymorphism is important oops concept. It means ability to take more than one form.
ii) In polymorphism an operations may shows different behavior in different instances. The behavior depends upon the type of data used in the operation. For Ex- Operation of addition for two numbers, will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.
iii) The process of making an operator to show different behavior in different instance is called as operator overloading. C++ support operator overloading.

iv)

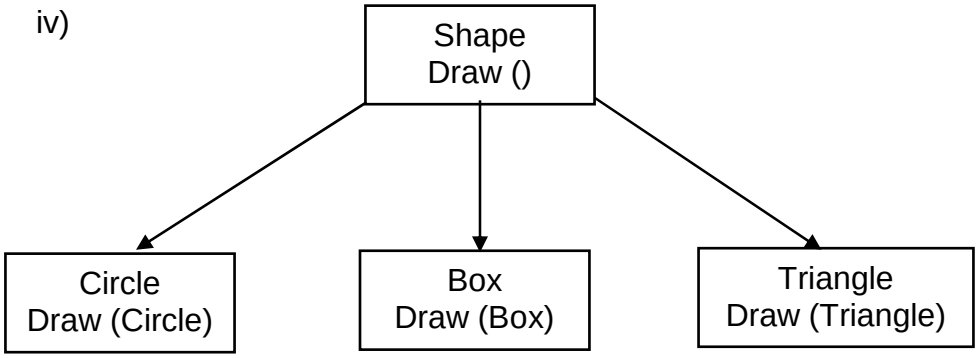


Fig 1.3 Polymorphism

The above figure shows concept of function overloading. Function overloading means using a single function name to perform different types of tasks.

5) Dynamic Binding

i) Binding referes to the linking of a procedure call to the code to be executed in response to the call.
ii) Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time.

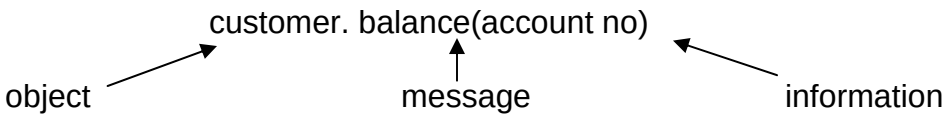
6) Message Passing

i) OOPs consist of a set of objects that communicate with each other.
ii) Message passing involves following steps-
    i) Creating classes that define objects and their behavior
    ii) Creating objects from class definitions and
    iii) Establishing communication among objects

iii) A message for an object is a request for execution of a procedure & therefore will invoke a function in the receiving object that generates the desired result.

Message passing involves specifying the name of the object, the name of the function i.e. message and the information to be sent.

Ex-



1.6 BENEFITS OF OOPs

OOP offers several benefits to both the program designer and the user. The principal advantages are.

i) Through inheritance, we can eliminate redundant code and extend the use of existing classes
ii) We can build program from the standard working module that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
iii) The principal of data hiding helps the programmer to build secure programs that cannot be invaded by code in other part of the program.
iv) It is possible to have multiple instance of an object to co-exist without any interference
v) It is easy to partition the work in a project, based on objects.
vi) Object oriented systems can be easily upgraded from small to large systems.
vii) Message passing techniques for communication between objects makes the interface description with external systems much simpler.
viii) Software complexity can be easily managed.

1.7 WHAT IS C++

- C++ is an object oriented programming language developed by Biarne stroustrup at AT & T Bell laboratories.

- C++ is an extension of c with a major addition of the class construct feature.

1.8 A SIMPLE C++ PROGRAM

```
# include<iostream.h>
int main()
{
        cout<<“Hello Wold”;
        return 0;
}
```

- C++ program is a collection of functions. Every C++ program must have a main() function.

- The iostream file:-
  The header file iostream should be included at the beginning of all programs that uses one output statement.

- Input / Output operator

  1. Input Operator cin :- The identifier cin is a predefined object in c++ that corresponds to the standard input stream. Here this stream represents keyboard.

     Syntax:- cin>>variable;

     The operator >> is known as extraction or get from operator & assigns it to the variable on its right.

  2. Output operator cout :-The identifier cout is predefined object that represents the standard output stream in c++. Here standard output stream represents the screen.

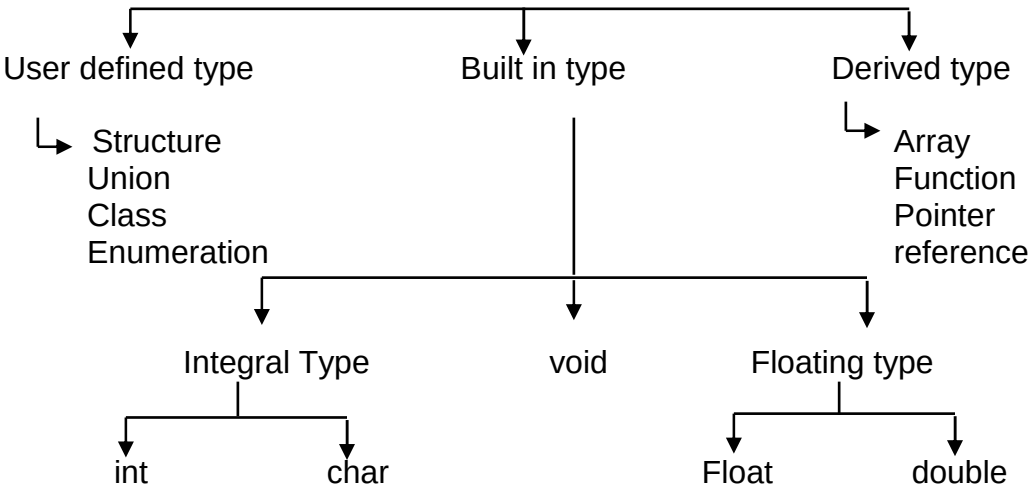
     Syntax:- cout<<string;

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

  3. Return type of main():- In C++, main returns an integer type value to the operating system. So return type for main() is explicitly specified as int. Therefore every main() in c++ should end with a return 0 statement.

1.9 BASIC DATA TYPES IN C++

C++ Data Types



1) Built-in-type

i) Integral type – The data types in this type are int and char. The modifiers signed, unsigned, long & short may be applied to character & integer basic data type. The size of int is 2 bytes and char is 1 byte.

2) void – void is used for.

i) To specify the return type of a function when it is not returning any value.
ii) To indicate an empty argument list to a function.
Ex- Void function1(void)
iii) In the declaration of generic pointers.
EX- void *gp
A generic pointer can be assigned a pointer value of any basic data type.

```
Ex –  int *ip    // this is int pointer
      gp = ip    //assign int pointer to void.
```

A void pointer cannot be directly assigned to their type pointers in c++ we need to use cast operator.

```
Ex -  Void * ptr1;
      Char * ptr2;
       ptr2 = ptr1;
```

is allowed in c but not in c++

ptr2 = (char *)ptr1

is the correct statement

3) **Floating type:** The data types in this are float & double. The size of the float is 4 byte and double is 8 byte. The modifier long can be applied to double & the size of long double is 10 byte.

2) User-defined type

i) The user-defined data type structure and union are same as that of C.

ii) Classes – Class is a user defined data type which can be used just like any other basic data type once declared. The class variables are known as objects.

3) Enumeration

i) An enumerated data type is another user defined type which provides a way of attaching names to numbers to increase simplicity of the code.

ii) It uses enum keyword which automatically enumerates a list of words by assigning them values 0, 1, 2,….etc.

iii) Syntax:-

```
enum shape {
        circle,
        square,
        triangle
        }
```

Now shape become a new type name & we can declare new variables of this type.

EX – shape oval;

iv) In C++, enumerated data type has its own separate type. Therefore c++ does not permit an int value to be automatically converted to an enum value.

Ex.
```
shape shapes1 = triangle, // is allowed
shape shape1 = 2;  // Error in c++
shape shape1 = (shape)2; //ok
```

v) By default, enumerators are assigned integer values starting with 0, but we can over-ride the default value by assigning some other value.

EX:- enum colour {red, blue, pink = 3};

it will assign red to o, blue to 1, & pink to 3 or

enum colour {red = 5, blue, green}; it will assign red to 5, blue to 6 & green to 7.

4) Derived Data types

1) Arrays

An array in c++ is similar to that in c, the only difference is the way character arrays are initialized. In c++, the size should be one larger than the number of character in the string where in c, it is exact same as the length of string constant.

Ex - char string1[3] = “ab”; // in c++
char string1[2] = “ab”; // in c.

2) Functions

Functions in c++ are different than in c there is lots of modification in functions in c++ due to object orientated concepts in c++.

3) Pointers

Pointers are declared & initialized as in c.

Ex- int * ip; // int pointer
ip = &x; //address of x through indirection

c++ adds the concept of constant pointer & pointer to a constant pointer.

char const *p2 = “HELLO”; // constant pointer

1.10 VARIABLES IN C++

- **Declaration of variables.**
  C requires all the variables to be defined at the beginning of a scope. But c++ allows the declaration of variable anywhere in the scope. That means a variable can be declared right at the place of its first use. It makes the program easier to understand.

- **Dynamic initialization of variables.**
  In c++, a variable can be initialized at run time using expressions at the place of declaration. This is refereed to as dynamic initialization.
  Ex.- int m = 10;
  Here variable m is declared and initialized at the same time.

- **Reference variables.**
  i) A reference variable provides an alias for a previously defined variable.

ii) Example:- If we make the variable sum a reference to the variable total, them sum & total can be used interchangeably to represent that variable.

iii) Reference variable is created as:-
data type & reference name = variable name.
Ex:- int sum = 200;
int &total = sum;
Here total is the alternative name declared to represent the variable sum. Both variable refer to the same data object in memory.

iv) A reference variable must be initialized at the time of declaration.

v) C++ assigns additional meaning to the symbol '&'.Here & is not an address operation. The notation int & means referenceto int. A major application of reference variable is in passing arguments to functions.

Ex.
```
void fun (int  &x) // uses reference
{
        x = x + 10;             // x increment, so x also incremented.
}
int main( )
{
        int n = 10;
        fun(n);                 // function call
}
```

When the function call fun(n) is executed, it will assign x to n
i.e. int &x = n;
Therefore x and n are aliases & when function increments x. n is also incremented. This type of function call is called call by reference.

1.11 OPERATORS IN C++

C++ had rich set of operators. Some of the new operators in c++ are-

1. :: - Scope resolution operators.
2. ::* - pointer to member decelerator.
3. → *-pointer to member operator.
4. .* - pointer to member operator.
5. delete – memory release operator.
6. endl – Line feed operator
7. new – Memory allocation operator.
8. stew – Field width operator.

- **Scope resolution Operator.**

1) C++ is a block - structured language. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.

2) Consider following program.

```
……
…….
{       int  x = 1;
        ===
}
=====
{       int x = 2;
}       =====
```

The two declaration of x refer to two different memory locations containing different values. Blocks in c++ are often nested. Declaration in a inner block hides a declaration of the same variable in an outer block.

3) In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by using scope resolution operator (::), because this operator allows access to the global version of a variable.

4) Consider following program.

```
# include<iostream.h >
int m = 10;
int main ( )
{
        int m = 20;
        {
                int k = m;
                int m = 30;
                cout << “K = “ <<k;
                cout << “m = “<<m;
                cout << : : m = “ << : : m;
        }
        cout << “m” = “<< m;
        cout << “::m = <<::m;
        }
        return 0;
 }
```

The output of program is

```
k = 20
m = 30
:: m = 10
m = 20
:: m =10
```

In the above program m is declared at three places. And ::m will always refer to global m.

5) A major application of the scope resolution operator is in the classes to identify the class to which a member functions belongs.

- **Member dereferencing operators –**

C++ permits us to define a class containing various types of data & functions as members. C++ also permits us to access the class members through pointers. C++ provides a set of three pointer. C++ provides a set of three pointers to member operators.

1) ::* - To access a pointer to a member of a class.
2) .* - To access a member using object name & a pointer to that member.
3) →* - To access a member using a pointer in the object & a pointer to the member.

- **Memory management operators.**

1) We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. C++ supports two unary operators **new** and **delete** that perform the task of allocating & freeing the memory.
2) An object can be created by using new and destroyed by using delete. A data object created inside a block with new, will remain existence until it is explicitly destroyed by using delete.
3) It takes following form.
   variable = new data type
   The new operator allocated sufficient memory to hold a data object of type data-type & returns the address of the object.
   EX - p = new int.
   Where p is a pointer of type int. Here p must have already been declared as pointer of appropriate types.
4) New can be used to create a memory space for any data type including user defined type such all array, classes etc.
   Ex- int * p = new int [10]
   Creates a memory space for an array of 10 integers.
5) When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form is
   delete variable.
   If we want to free a dynamically allocated array, we must use following form.
   delete [size] variable.
   The size specifies the no of elements in the array to be freed.
6) The new operator has following advantages over the function malloc() in c -.
   i) It automatically computes the size of the data object. No need to use sizeOf()

ii) If automatically returns the correct pointer type, so that there is no need to use a type cast.
iii) new and delete operators can be overloaded.
iv) It is possible to initialize the object while creating the memory space.

- **Manipulators**

Manipulators are operators that are used to format the data display. There are two important manipulators.

1) endl
2) stew

1) endl : - This manipulator is used to insert a linefeed into an output. It has same effect as using "\n" for newline.
Ex- cout<< "a" << a << endl <<"n=" <<n;
<< endl<<"p="<<p<<endl;
The output is
a = 2568
n = 34
p = 275

2) Setw : With the stew, we can specify a common field width for all the numbers and force them to print with right alignment.
EX – cout<<stew (5) <<sum<<endl;
The manipulator setw(5) specifies a field width of 5 for printing the value of variable sum the value is right justified.

		3	5	6

- **Type cast operator.**

C++ permits explicit type conversion of variables or expressions using the type cast operator.
Syntax – type name (expression)
Ex – avg = sum/float(i)

Here a type name behaves as if it is a function for converting values to a designated type.

❖❖❖

2

CLASSES AND OBJECTS

2.1 SPECIFYING A CLASS

Class: A class is a user defined data type which binds data and its associated functions together. It allows the data and functions to be hidden, if necessary from external use. Generally, a class specification has two parts.

i) **Class declaration:** it describes the type & scope of its members.
ii) **Class function definitions:** It describes how the class functions are implemented.

- **Class declaration.**

The general form of a class is

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

1) The class keyword specifies that what follows is an abstract data of type class name. The body of a class is enclosed within braces & terminated by semicolon.
2) The class body consists of declaration of variables & functions which are called as members & they are grouped under two sections i.e. private & public.
3) Private and public are known as visibility labels, where private can be accessed only from within the class where public members can be accessed from outside the class also. By default, members of a class are private.
4) The variable declared inside the class are known as data members & functions are known as member functions. Only the member function can have access to the private data members & private functions. However the public members can be accessed from outside the class.

- Simple class example

```
class item
{
```

```
        int number;          }      variable
        float cost;          }      declaration
public :
        void getdata (int a, float b);   }    function
        void putdata (void);             }    declaration
}
```

In above class class-name is item. These class data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variable number & cost, and putdata() for displaying their values. These functions provide the only access to data members of the class.

- **Creating objects**

Object is an instance or variable of the class. Once a class has been declared. We can create variables of that type by using the class name.

Ex – item x;

Here class variables are known as object therefore, x is called an object of type item and necessary memory space is allocated to an object.

- **Accessing class Members**

1) Private data of a class can be accessed by the member function. The following is the format for calling member function.

   Object-name.function-name(actual argument)

   Ex – x.getdata (10, 20.3);

This statement assign value 10 to number & 20.3 to cost of the object x.
By implementing the getdata() function similarly, the statement.

x.putdata();

would display the value of data member.

2) If the member variable is declared as private then it cannot be accessible by object. It can only be accessed by a member function. Whereas, if a variable is declared as public it can be accessed by the object directly.

Ex – class abc

```
{
        int x, y;
    public :
        int z;
};
main()
{
abc m ;
m.x = 100;  // error x is private
m.z=50;     // ok ,z is public
```

The above example shows concept of data hiding.

2.2 DEFINING MEMBER FUNCTIONS.

Member functions can be defined in two ways.

1) Outside the class definition
2) Inside the class definition.

1) Outside the class definition.

1) Member function that is declared inside a class has to be defined separately outside the class. These member functions associate a membership identify label in the header. This 'label' tells the compiler which class the function belongs to. The general format of a member function definition is.

```
Return type class name:: function-name(argument declaration)
{
        Function body
}
```

2) The membership label class-name:: tells the compiler that the function function-name belongs to the class-name. The symbol :: is called as scope resolution operator.
3) Ex- the function getdata is coded as

```
void item:: getdata(int a, float b)
{
        number = a;
        cost = b;
}
```

4) The member function have some special characteristics :-
   i) Several different classes can use the same function name the 'membership label' will resolve their scope.
   ii) Member function can access the private data of the class. A non member function cannot do so.
   iii) A member function can call another member function directly, without using the dot operator.

2) Inside the class definition.

In this method the function declaration inside the class is replaced by actual function definition.

For Ex-

```
class item
{
        int number;
        float cost;
pubic :
```

```
        void getdata (int a float b);   //declaration

        void putdata (void)
        {
        cout << number << endl;
        cout << cost << endl;
        }
}
```

Remember only small functions can be defined inside the class.

2.3 A C++ PROGRAM WITH CLASS

```
# include<iostream.h>
class item              // class declaration
{
      int number;           // private by default
      float cost;
public :
      void getdata(int a float b);
      void putdata(void)          // function defined here
      {
      cout <<"number "<< number << "\n";
      cout << cost: << cost << "\n"
      }
};
// member functions definition
void item :: getdata(int a, float b)
{
        number = a;
        cost = b;
}
// main program
main()
{
        item x; // create object x
        x.getdata(100, 20.3);
        x.putdata();
}
```

In the above program we have shown that one member functions is inline and other is external member function. Here is the output of program.

```
number : 100
cost : 20.3
```

2.4 MAKING AN OUTSIDE FUNCTION INLINE.

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

Ex –

```
class item
{ ----
  ----

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

2.5 NESTING OF MEMBER FUNCTION

When a member function can be called by using its name inside another member function of the same class, it is known as nesting of member function.

Ex –

```
# include<iostream.h>
class set
{
int m, n;
public:
        void input (void);
        void display (void);
        int largest(void);
};
int set:: largest (void)
{
        if (m >= n)
        return (m);
        else
        return (n);
}
void set : : input (void)
{
cout << "input values of m & n:";
```

```
cin >> m >> n;
}
void set :: display(void)
{
cout << “largest value = “ <<largest();
}
main()
{
        set a
        a.input();
        a.display();
}
```

2.6 PRIVATE MEMBER FUNCTION

i) Although we place all data items in a private section and all the functions in public, some situation may require certain function to be hidden from the outside calls.

ii) We can place these functions in the private section. A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

iii) Consider following class:

```
class sample
{
        int m;
        void read(void);
public:
        void update(void);
        void write(void);
};
```

If s1 is an object of sample then

```
    s1.read(); // will not work , object cannot access private data.
```

read() can be called by the function update() to update the value of m.

```
void sample :: update (void)
{
        read();    // a simple call
}
```

2.7 MEMORY ALLOCATION FOR OBJECTS

The member functions are created & place in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions no separate space is allocated for member functions when the objects are

created only space for member variables is allocated separately for each object.

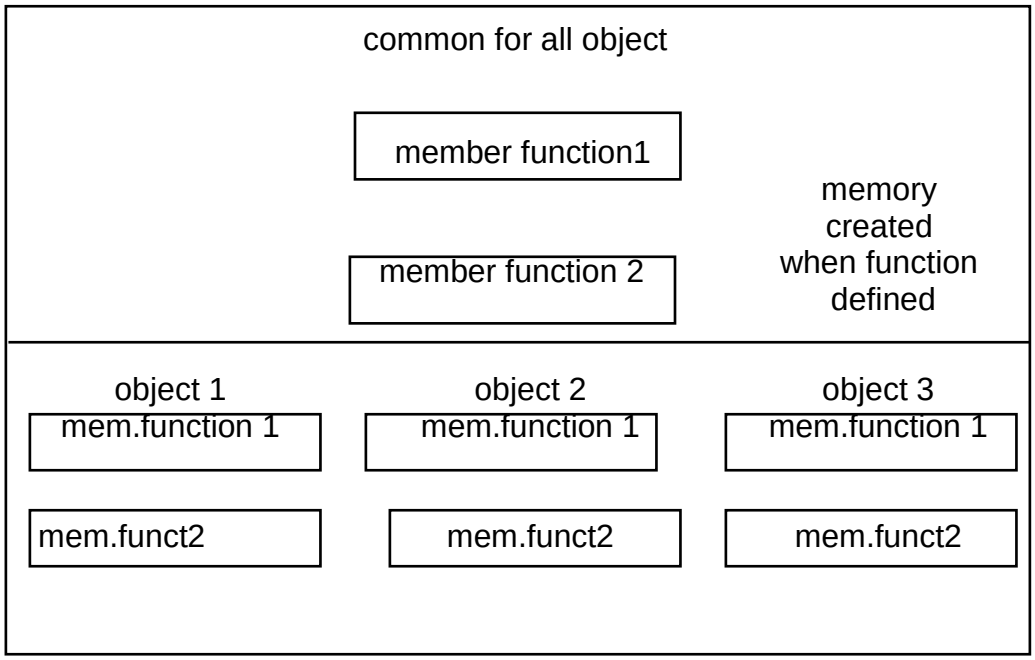


Fig. Object in memory.

2.8 STATIC DATA MEMBERS

1) A data member of a class can be qualified as static. A static member variable has certain special characteristics these are.
   i) It is initialized to zero when the first object of its class is crated. No other initialization is permitted.
   ii) Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
   iii) It is visible only within the class, but its lifetime is the entire program.

2) Static variables are normally used to maintain values common to the entire class for Ex - A static data member can be used as a counter that records the occurrences of all the objects.

```
# include<iostream.h>
class item
{
        static int count;
        int number;
```

```
public :
        void getdata(int a)
        {
                number = a;
                count ++;
        }
        void get count (void)
        {
        count << “count :”;
        count << count << “\n”;
        }
};
int item : : count;
int main ( )
{
        item a, b, c;
        a.getcount() ;
        a.getcount() ;
        a.getcount() ;
        a.getdata(100);
        a.getdata(200);
        a.getdata(300);
        cout <<“After reading data”;
        a.getcount();
        b.getcount();
        c.getcount();
        return 0;
}
the output would be
        count : 0
        count : 0
        count : 0
After reading data
        count : 3
        count : 3
        count : 3
```

The static variable count is initialized to zero when the objects are created.

The count is incremented whenever the data is read into an object. Since the data is read into objects three times, variable count is incremented three times.

3) Static variables are like non inline member functions in that they are declared in a class declaration & defined in the source file.

2.9 STATIC MEMBER FUNCITON

1) A static member function has following properties.
   i) A static function can have access to only other static members declared in the same class.
   ii) A static member function can be called using the class name as follows:
   class–name:: function–name;

2) Following program illustrates the implementation of these characteristics.

```
#include<iostream.h>
class test
{
        int code;
        static int count;
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout <<"object No " << code;
        }
        static void showcount(void)
        {
        cout<< "count : "<<count;
        }
}
int test:: count;
main()
{
        test t1, t2;
        t1.setcode();
        t2.setcode();
        test:::showcount();
        test t3;
        t3.setcode();
        test:::showcount();
        t1.showcode() ;
        t2.showcode();
        t3.showcode();
```

```
}
```

Output of the program

```
count : 2
count : 3
object No : 1
object No : 2
object No : 3
```

The static function showcount displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable count.

2.10 OBJECTS AS FUNTIONS ARGUMENTS

1) An object may be used as function arguments in two ways :-
   i) A copy of the entire object is passed to the function.
   ii) Only the address of the object is transferred to the function.

2) I) **Pass-by-value –** Since a copy of the object is passed to the function, any change made to the object inside the function do not effect the object used to call the function.
   iii) **Pass-by-reference –** when an address of the object is passed the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

3) Example -Following program illustrates the use of object as function arguments. It performs the addition of time in the hour & minute format.

```
# include<iostream.h>
class time
{
        int hours;
        int minutes;
public:
        void gettime(int h, int m)
        {
            hours = h;
            minutes = m;
        }
        void puttime(void)
        {
        cout << hours << “hours and:”;
        cout << minutes << “ minutes “ << “\n”;
        }
        void sum (time, time);
```

```
};

void time :: sum (time t1, time t2)
{
minutes =t1.minutes + t2.minutes ;
hours = minutes / 60;
minutes = minutes%60;
hours = hours = t1.hours + t2.hours;
}
main()
time T1, T2, T3;
T1.gettime(2, 30);
T2.gettime(3, 45);
T3.sum(T1, T2);
cout << " T1 = ";
T1.puttime();
cout << " T2 = ";
T2.puttime();
cout << " T3= ";
T3.puttime();
}
```

4) An object can also be passed as argument to a non-member function but, such functions can have access to the public member function only through the object passed as arguments to it.

2.11 FRIENDLY FUNCTIONS

1) A non member function cannot have an access to the private data of a class. However there could be situation where we would like two classes to share a particular function.
2) In such situation, c++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes such a function need not be a member of any of these classes.
3) The syntax for friend function is

```
class ABC
{
=======
public:
=======
    friend void function1(void);
}
```

The function is declared with friend keyword. But while defining friend function. It does not use either keyword friend or :: operator.

A friend function, although not a member function, has full access right to the private member of the class.

4) A friend, function has following characteristics.
   i) It is not in the scope of the class to which it has been declared as friend.
   ii) A friend function cannot be called using the object of that class. If can be invoked like a normal function without help of any object.
   iii) It cannot access the member variables directly & has to use an object name dot membership operator with member name.
   iv) It can be declared either in the public or the private part of a class without affecting its meaning.
   v) Usually, it has the object as arguments.

5) Ex-

```
# include<iostream.h>
class sample
{
        int a, b;
        public:
        void setvalue()
        {
             a = 25; b = 40;
        }
        friend float mean (sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/ 2.0;
}
main ( )
{
        sample x ;
        x.setvalue();
        cout << " Mean Value = ";
        mean(x);
}
```

output –

Mean value = 32.5

6) Member function of one class can be friend functions of another class. In such cases they are defined using the scope resolution operator.

EX – class x
{ ------

```
int fun(); // member function of x
};
class y
{---------
----------
friend int x : : fun()
}; ---------
```

7) We can declare all the member functions of one class as the friend functions of another class by declaring that class as a friend class.

Ex –
```
class z
{
        --------
        --------
friend class x ;
{;
```

8) A friend function work as a bridge between the classes. Ex

```
#include<iosream.h>
class ABC ; //forward declaration
class xyz
{
        int x;
public:
        void setvalue(int i) { x = i; }
        friend void max (xyz, ABC);
};
class ABC
{
        int a;
public a:
        void setvalue(int i) { a =i;}
        friend void max(xyz, ABC);
};
void max(xyz m, ABC n) // definition of friend.
{
        if(m. x > = n. a)
        count << m .x;
        else
        cout << n.a;
}
main ( )
{
        ABC K;
        K. setvalue (20);
        XYZ l;
        l.setvalue (40);
```

```
        max (l , k);
}
output is – 40
```

9) A friend function can be called by reference. In this local copies of the object are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.
Ex-following program shows how to use a common friend function to exchange private value of two classes.

```
# include <iostream.h>
class c2;
class c1
{
        int value1;
        public:
        void indata (int a) {value1 = a;}
        void display (void) {cout << value1;}
        friend void exchange (c1 &, c2 &);
};
class c2
        {
        int value2;
        public:
        void indata (int a) {value 2 = a;}
        void display (void) {cout << value2;}
        friend void exchange (C1 &, C2 &)
};
void exchange (cl &x, c2, &y)

        int temp = x.value1;
        x.value 1= y.value2;
        y.value 2 = temp
}
main()
{
        c1 co1;
        c2 co2;
        co1. indata(10);
        co2. indata(20)
        cout << “Values before exchange;
        co1. display()
        co2.display();
        exchange(co1, co2);
        cout << “values after exchange”;
```

```
        co2.display();
}
output of the program
        values before exchange
        10
        20
        values after exchange
        20
        10
```

2.12 RETURNING OBJECTS

A function can not only receive objects as arguments but also can return them. Ex.

```
#include <iostream.h>
class complex
{
    float  x, y;
public:
    void input (float real, float image)
    {
        x = real;
        y = image;
    }
    friend complex sum (complex, complex);
    void show(complex);
};
complex sum (complex c1, complex (C2)
{
    complex c2;
    c3. x = c1.x  + c2.x;
    c3.y = cl.y + c2.y;
    return(c3);
}
void complex ::show(complex c)
{
    cout << c.x << "+j" << c.y << "\n";
}
main ( )
{
    complex A, B, C;
    A.input (3.1, 5.65);
    B.Input (2.75, 1.2);
    C= cum (A, B);
    cout << "A"; A.show(A);
    cout << "B";  B.show(B);
```

```
        cout << "C";  C.show(C);
    }
```

output of the program
A = 3.1 + j5 65
B = 2.75 + ji.2
C = 5.85 + j6. 85

❖❖❖

3

CONSTRUCTORS AND DESTRUCTORS

3.1 INTRODUCTION

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as putdata() and setvalue() to provide initial values to the private member variables. For example, the following statement.

```
a.input();
```

invoke the member function input() which assigns the initial values to the data items of object A. Similarly, the statement.

```
x.getdata (100, 299.95);
```

passes the initial values as arguments to the function getdata() where these values are assigned to the private variables of object x. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not confirm with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as class, that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example.

```
int m = 20;

Float x = 5.75;
```

Are valid initialization statements for basic data types

Similarly, when a variable of built-in-type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the object when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. It also provides another member function called the destructor that destroys the objects when they are no longer required.

3.2 CONSTRUCTORS

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the value data members of the class.

```
//class with a constructor
class Integer
   {
    int m, n;
    public:
         Integer(void); / constructor declared
              ….
              ….
   };
    Integer :: Integer(void) // constructor defined
    {
     m = 0; n = 0;
    }
   };
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration.

```
Integer int1; // object int1 created
```

not only creates the object **int1** of type I**nteger** but also initializes its data members **m** and **n** to 0.There is no need to write any statement to invoke the constructor function.

A constructor that accepts no parameters is called the default constructor. The default constructor for class A is A::A(). If no such constructor is defined the compiler suppliers default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object a.
The constructor functions have some special characteristics.

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make implicit calls to the operations new and delete when memory allocation is required. Remember, when a constructor is declared for a class initialization of the class objects become mandatory.

3.3 PARAMETERIZED CONSTRUCTORS

The constructor Integer(), defined above, initialized the data members of all the objects to zero. However in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

The constructor integer() may be modified to take arguments as shown below :

```
class integer
{
        int m, n;
 public:
        integer (int x, int y); //parameterized constructor
        …..
```

```
 };
integer : : integer int x, int y)
{
        m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

integer int1;

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways;

- By calling the constructor explicitly.
- By calling the constructor implicitly.

Integer Int1 = Integer (0, 100); //explicit call

This statement creates in integer object int1 and passes the values 0 and 100 to it.

The second is implemented as follows :

Integer int1 (0, 100); //implicit call.

This method sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide arguments for the constructor. Program 3.1 demonstrates the passing of arguments to the constructor functions.

```
////////////////////////////// [CLASS WITH CONSTRUCTORS] //////////////////////////////
include <iostream.h>

class integer
{
        int m, n;
    public:
        integer (int, int) ;    // constructor declared
        void display (void)
        {
         count << " m = "<< m << "\n";
         count << " n  = "<< n <<  "\n";
         }
```

```
};

integer : : integer (int x, int y) // constructor defined
{
        m = x;  n = y ;
}
main()
{
         integer int1 (0, 100)  // IMPLICIT call
        integer int2 = integer (25, 75); // EXPLICIT  call
        cout << "\nOBJECT1" <<. "\n;
        int1.display();
        cout << "\nOBJECT2" << "\n";
        int2.display();
}
```

// < PROGRAM 3.1 > //

Program 3.1 displays the following output :

```
OBJECT1
m = 0
n = 100
OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as inline functions. Example:

```
class integer
{
        int m, n;
public
integer (int x, int y) //inline constructor
{
        m = x; y = n;
}
        …..
        …..
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
        …..
```

```
        public:
                A (A);
        };
```

is illegal

However a constructor can accept a reference to its own class as a parameter. That is,

```
Class A
    {
        …..
        …..
    public:
        A(A&);
    };
```

is valid. In such cases, the constructor is called the copy constructor.

3.4 MULTIPLE CONSTRUCTORS IN A CLASS

So far we have used two kinds of constructors. They are;

```
integer();          // No arguments
integer(int, int); // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from main ( ). C++ permits us to use both these constructors in the same class. For example, we could define a class as follows :

```
class integer
{
    int m, n;
public:
    integer ( ) {m = 0; n = 0;)        // constructor 1
    integer (int a, int b)
    {m = a; n = b;)                    // constructor 2
    integer (integer & i)
    {m = i.m; n = i.n;)                // constructor 3
};
```

This declared three constructors for an integer object. The first constructor receives no arguments, the second receives two integer

arguments and the third receives one integer object as an argument. For example, the declaration.

Integer I1;

would automatically invoke the first constructor and set both m and n of I1 to zero. The statement

integer I2 (20, 40);

would call the second constructor which will initialize the data members m and n I2 to 20 and 40 respectively. Finally, the statement.

Integer I3(I2);

would invoke the third constructor which copies the values of I2 into I3. That is, it sets the value of every data element of I2 to the value of the corresponding data element of I3. As mentioned earlier, such a constructor is called the copy constructor. The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor is overloaded, it is called constructor overloading.Program 3.2 shows the use of overloaded constructors.

```
//////////////////////////////// [ OVERLOAED CONSTRUCTORS ] ////////////////////////////
    #include <iosrtream.h>

    class complex
    {
            float x, y;
        public:
            complex ( ) {  }                    // constructor no arg
            complex (float a ) (x = y = a;) // constructor one arg
            complex (float real, float imag)//constructor –two args
            {x = real; y = imag;}
            friend complex sum (complex, complex);
            friend void show (complex);
    };
    complex sum (complex c1, complex c2) // friend
    {
        complex c3;
        c3.x = cl. x + c2. x;
        c3.y = cl. y + c2. y;
        return (c3);
    };
    void show (complex c) // friend
    {
        cout << c.x << "+j" << c.y << "\n";
```

```
        }

        main ( )
        {
                complex A (2, 7, 3.5);          //define & initialize
                complex B (1.6) ;               //define & initialize
                complex C;                      //define
                C = sum (A, B);                 //sum ( ) is a friend
                cout << "A = ";show (A);        show ( ) is also friend
                cout << "B = "; show (B);
                cout << "C = "; show (C) ;

//      Another way to give initial values (second method)
        complex P, Q, R;                        // define P, Q and R
        P = complex  (2.5, 3.9);                // initialize P
        Q = complex  (1.6, 2.5);                // initialize Q
        R = sum (P, Q);
        cout << "/n";
        cout << "P = " ; show (P);
        cout << "Q = ";   show (Q)
        cout << "R = "; show (R) ;
]
```

// < PROGRAM 3.2 > ////////////////////////////////////

The output of Program 3.2 would be :

```
A = 2.7 + j3.5
B = 1.6 + J1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

There are three constructors in the class complex. The first, which takes no arguments, is used to create objects which are not initialized. The second , which takes one argument, is used to create objects and initialize them. The third constructor, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex ( ) { }
```

It contains the empty body and does not do anything. We just stated that, this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? As pointed out earlier

C++ compiler has an implicit constructor which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we defined a constructor, we must also define the “do-nothing” implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

3.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor complex ( ) can be declared as follows.

complex (float real, float image = 0)

The default value of the argument image is zero.Then the statement

complex C (5, 0):

assigns the value 5.0 to the real variable and 0.0 to image (by default). However, the statement

complex C(2.0, 3.0);

assigns 2.0 to real and 3.0 to image. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A::A() and the default argument constructor A::A(int = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class it causes ambiguity.

3.6 DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as

well as for different periods of investment. Program .3.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
////////////// [ DYNAMIC INITIALIZATION OF CONSTRUCTORS ] ////////////////////
        // Long-term fixed deposit system
        # include <iostream.h>

        class Fixed_deposit
        {
        long int P_amount;                  //Principal amount
        int     years;          //Period of investment
        float   Rate;           //Interest rate
        float   R_value;        //Return value of amont
        public:
        Fixed_deposit ( )  {  }
        Fixed_deposit (long int p, int y, float r = 0.12);
        Fixed_deposit (long int p, int y, int r);
        void display (void);
        };
        Fixed_deposit :: Fixed_deposit (long int p, int y, float r)
        {
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for (int i = 1; i <= y; i++)
        R-vale = R_value * (1.0 + r);
        }
        Fixed_deposit :: Fixed_deposit (long int p, int y, int r)
        {
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for (int I = 1; I <= y; i++)
        R-vale = R_value * (1.0 +  float  ( r ) /100);
        }
        void fixed_deposit :: display (void)
        {
                cout< < "\n";
                << "Principal Amount = " << P_amount << "\n"
                << "Return Vale         = " << R_value    << "\n";
        }

        main()
        {
```

```
        Fixed_deposit FD1, FD2, FD3 ;       //deposits created
        long int p,                         //principal amount
        int y; //investment period, years
        float r;                            //interest rate, percent form

        cout << "Enter amount period, interest rate (in percent)" <<\n";
        cin >> p >> y > R;
        FD1 = Fixed_deposit (p, y, R);

        cout <<"Enter amount, period, interest rate (decimal form)";
        cin >> p >> y >> r;
        Fd2 = Fixed_deposit (p, y, r);

        cout << "Enter amount and period" << "\n";
        cin >> p >> y;
        FD3 = Fixed_deposit (p, y);

        cout << "/n Deposit 1";
        FD1. display ( );

        cout << "/n Deposit 2"
        FD2. display() ;

        cout << "\nDeposit 3:;
        FD3. display();

}
```

////////////////////////////////////// < PROGRAM 3.3 > //////////////////////////////////////

The output of Program 3.3 would be;

```
        Enter amount period. interest rate (in per cent)
        50005  5    15
        Enter amount, period, interest rate (in decimal form)
        5000   5    0.15
        Enter amount and period
        50005

        Deposit 1
        Principal Amount     = 5000
        Return Value         = 10056.786133

        Deposit 1
        Principal Amount     = 5000
        Return Value         = 10056.78613

        Deposit 3
```

Principal Amount = 5000
Return Value = 8811.708984

The program uses three overloaded constructions. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms.

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3) and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, is used the default value for r.

3.7 COPY CONSTRUCTOR

We used the copy constructor.

Integer (integer & I)

in section 3.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object For example, the statement.

integer I2 (I1);

would define the object I2 and at the same time initialize to the values of I1. Another form of this statement is

integer I2 = I1;

The process of initializing through a copy constructor is known as copy initialization. Remember the statement.

I2 = I1;

will not invoke the copy constructor. However, if I1 and I2 are objects, this statement is legal simply assigns the values of I1 to I2, member-by-member.

A copy constructor takes a reference to an object of the same class s itself as an argument. Let us consider a simple example of constructing and using a copy constructor as show in Program 3.4

```
////////////////////////////////////// [    COPY CONSTRUCTOR   ] ////////////////////////////////
    #include <iostream.h>

    class code
     {
            int id;
    public:
            code ( ) {    }                    // constructor
            code(int a ) {id = a;}             //  constructor again
            code (code & x)                    // copy constructor
            {
                  id = x.id;                   // copy in the value
            }
            void display (void)
            {
                  cout << id;
            }
    };

    main ( )
    {
            code A =(100);       // object A is created an initialized
            code B (A);             // copy constructor called
            code C = A;          // copy constructor called again
            code D;                 // D is created, not initialized.
            D = A;                          // copy constructor not called

            cout << "\n id of A: "; A. display ( );
            cout << "\n id of B: "; B. display ( );
            cout << "\n id of C: "; C. display ( );
            cout << "\n id of D: "; D. display ( );
    }
////////////////////////////////////////// < PROGRAM 3.4  > ////////////////////////////////////////
```

The output of Program 3.4 is shown below :

```
id of A : 100
id of B : 100
id of C : 100
id of D : 100
```

Note that a reference variable has been used in the argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined the compiler supplies its own copy constructor.

3.8 DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each objects when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 3.5 shows the use of new in constructors that are used to construct strings in objects.

```
//////////////////////////////// [  CONSTRUCTORS WITH new ] ////////////////////////////////
        #include <iostream.h>
        #include <string.h>

        class string
        {
            char name;
            int length;
        public:
            string()                            //constructor – 1
            {
                  length = 0;
                   name = new char[length + 1];      //one extrafor \0
             }
            string (char * s)                   //constructor –2
            {
                   length = strlen (s);
                   name = new char[length + 1];//one extra for \0
                   strcpy (name , s);
             }
             void display (void)
            {
                    cout << name <<   “”\n” ;
            }
        void join (string & a, string & b);
};

void string : : join (string & a, string & b)
{
        length = a. length + b. length;
        delete name ;
        name = new char [length + 1]; //dynamic allocation.
```

```
        strcpy (name, a name);
        strcpy (name, b. name);
};

main ( )
{
        char * first = “Joseph”;
        sring name1 (first), name2 (“Louis”), name3 (“Lagrance”), s2, s2;
        s1. join (name 1, name 2);
        s2. join (s1, name 3);
        name1. display ( ) ;
        name2. display ( );
        name3. display ( );
        s1. display ( );
        s2 .display ( );
}
```

// < PROGRAM 3.5 > //

The output of Program 3.5 would be.

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrance
```

Program 2.5 uses two constructors. The first is an empty constructor that allows us to declare an array of string. The second constructor initialized the length of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character ‘\0’

The member function join () concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions strcpy ( ) strcat ( ). Note that in the function join ( ), length and name are members of the object that calls the function, while a. length and a. name are members of the argument object a. The main ( ) function program concatenates three strings into one string. The output is as shown below.

Joseph Louis Lagrange

3.9 CONSTRUCTING TWO-DIMENSIONAL ARRAYS

We can construct matrix variables using the class type objects. The example in Program 3.6 illustrates how to construct a matrix of size m x n.

```
//////////////////////////// [ CONSTRUCTING MATRIX OBJECTS]////////////////////////
        #include <iostream.h>
class matrix
{
        int  **p;                       //pointer to matrix
        int d1, d2;                     //diamensions
public:
        matrix (int x, int y);
        void get_element (int i, int j, int value)
        {p[i][j] = value;}
        int & put_element (int i, int j)
        {return p[i][j]; }
};

matrix : : matrix (int x, int y)
{
        d1 = x;
        d2 =y;
        p = new int *[d1];              //creates an array pointer
        for  (int i  = 0, i < d1; i++)
                p[i] = new int [d2];    //creates space for each row
}
main ()
{
        int m, n;
        cout << "Enter size of matrix : ";
        cin >> m >> n;
        matrix  A(m, n);                // matrix object A constructed
        cout << "Enter matrix elements row by row \n";
        int i j, value;
        for (i = 0; i < m; i++)
                for (j = 0; j< n; j++)
                {
                cin >> value;
                A.get_element (i, j, value);
                }
                 cout << "\n";
                 cout << A. put_element (1, 2);
};
/////////////////////////////////////// < PROGRAM 3.6 > ///////////////////////////////////////
```

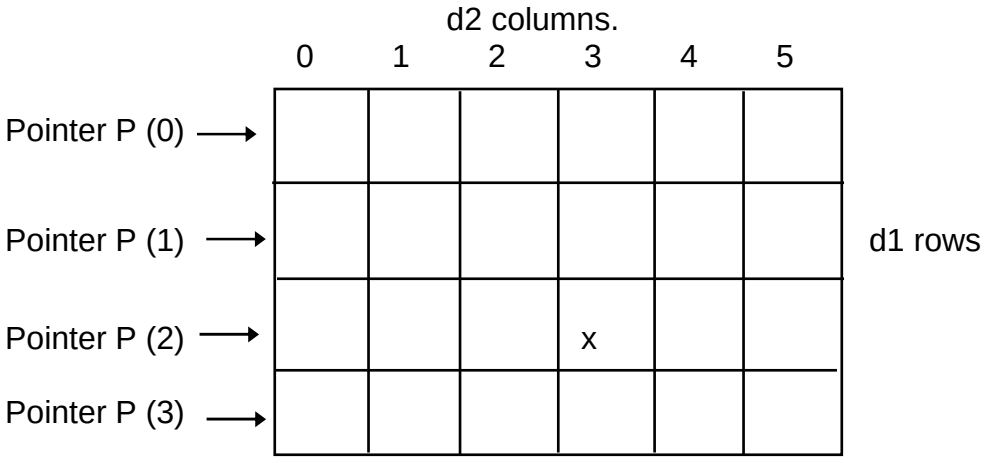
The out of a sample run of Program 3.6 is shown below :

```
                Input

                Enter size of matrix 3.4
                Enter matrix elements row by row
```

```
11  12  13  14
15  16  17  18
19  20  21  22

Output

17
```

17 is the value of the element (1.2)

The constructor first creates a vector pointer to an int of size d1. Then is allocates iteratively. ar type vector of size d2 pointed at each element p[I]. Thus for the element of a d2 x d2 matrix is allocated from free store as shown below :



x represents the element P [2] [3]

3.10 DESTRUCTORS

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, it is a member function whose name is the same as the class name but is preceded by a tilde, For example, the destructor for the class integer can be defined as shown below:

```
~ integer ( ) { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is good practice to declare destructors in a program since it release memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. For example the destructor for the matrix class discussed above may be defined as follows.

```
matrix :: ~ matrix ( )
{
        for  (int i = 0: j < d1;i++)
          delete p[i];
        delete p;
}
```

This is required because when the pointers to object go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been implicitly by the compiler.

////////////////////// [IMPLEMENTATION OF DESTRUCTORS] ////////////////////////

```
#include <iostream.h>
int count = 0;

class, alpha
{
        public :
             alpha ( )
             {
                    count ++;
                    cout << "\n No.of object created "<< count;
             }
             ~alpha()
             {
                    count << "\nNo. of object destroyed " << count;
                    count --;
              }
};

main ( )
{
        cout << "\n\n Enter MAIN \n";
        alpha A1, A2, A3, A4;
        {
               cout << "\n\nEnter Block/n";
               alpha A5;
        }
               count << "\n\nEnter Block2 \n";
```

```
            alpha A6;
        }
        cout<< "\n\n RE-ENTER MAIN \n";
    }
```

//////////////////////////////////// < PROGRAM 3.7 > ////////////////////////////////////

The output of a sample of run of Program 6.7 is shown below.

```
ENTER MAIN

No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4

ENTER BLOCK 1

No. of object created 5
No. of object destroyed 5

ENTER BLOCK 2

No. of object created 5
No. of object destroyed 5

RE-ENTER MAIN

No. of object destroyed 4
No. of object destroyed 3
No. of object destroyed 3
No. of object destroyed 1
```

As the objects are created and destroyed they increase and decrease the count. Notice that after the first group of objects is created. A5 is created and then destroyed. A6 is created, and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of scope is encountered, the destructors for each object in the scope are called. Note that the objects are destroyed in the reverse order of creation.

❖❖❖

4

FUNCTION AND OPERATOR OVERLOADING

4.1 FUNCITON OVERLOADING

I) Overloading refers to the use of the same thing for different purposes. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOPs.

II) The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number & type of the arguments but not on the function type.

III) Example-

```
//  declarations –
    int add(int a, int b);                  //declaration1
    int add(int a, int b, int c);           //declaration2
    double add (double x, double y);        //declaration3
    double add (int p, double q);           //declaration4
//  function calls –
    cout << add (5, 10);        // uses declaration1
    cout << add (5, 10.3);      // uses declaration4
    cout << add (2, 3, 4);      // uses declaration2
    cout << add (2.3, 5.4);     // uses declaration3
```

iv) A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. The function selection involves following steps.

1. The compiler first tries to find an exact match in which the types of actual arguments are the same & use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual argument, such as.
   char to int
   float to double to find a match.
3. When either of them fails, the compiler tries to use built in conversions to the actual arguments & then uses the functions whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error.
   Ex – long add(long n)
   double add(double m)
   A function call such as.
   add(20)

will cause an error because int argument can be converted either long or double.

4) For example –following program Illustrates function overloading.

```
#include<iostream.h>
int volume(int);
double volume(double, int);
long volume(long, int, int);
main()
{
        cout <<volume(10) << "\n";
        cout << volume (2.5, 8) << "\n";
```

```
        cout << volume (100 L, 75, 15);
}
int volume(int s)
{
   return (s * s * s);
}
double volume (double r, int h)
{
        return (3.14 * r * r * h);
}
long volume (long l, int b, int h)
{
        return (l * b * h);
}
```

The output of program

1000
157.2595
112500

4.2. FUNCTION WITH DEFAULT ARGUMENTS

i) The function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses & alerts the program for possible default values.

ii) For example-
float amount (float principal, int period. float rate = 0.15);
declares a default value to argument rate
A function call like
Value = amount (4000, 8);
Passes the value of 4000 to principal and 8 to period and then lets the function use default value of 0.15 for rate.
The call
value = amount (3000, 5, 0, 12);
passes an explicit value of 0.12 to rate.

iii) A default argument is checked for type at the time of call. Only the trailing arguments can have default values.

iv) We must add defaults from right to left we cannot provide a default value to a particular argument in the middle of an argument list.

v) For example-

```
# include<iostream.h>
main()
{
      float amount;
      float value(float p, int , float r = 0.15);
      void printline (char ch = '*',int len = 40);
      printline('=');
}
float value(float p, int n, float r)
{
      int year = 1;
      float sum = p;
      while(year <= n)
     {
        sum= sum * (1 + r);
```

```
            year = year + 1;
        }
         return(sum);
}
void printline(char ch, int len)
{
    for (int i =1 ; i < len;  i++)
    printf ("%c", ch);
}
```

The output is –

```
************************************************************
```

Final Value = 10056.786133

4.3 INLINE FUNCITON

i) Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack & returning to the calling function. When a function is small, lot of execution time may be spent in such overhead.

ii) Solution to this problem is to use macro definition, but because macros are not function, usual error checking does not occur during compilation.

iii) To eliminate the cost of calls to small functions, c++ introduces inline function. When inline function is called, the compiler replaces the function call with corresponding function code.

iv) Syntax is –

```
inline function header
{
        function body
}
Ex – inline int cube (int a)
{
        return (a * a * a);
}
```

this function can be called as –

```
c = cube(5);
```

v) All inline functions must be defined before they are called.

vi) The speed benefits of inline functions decreases as function grows in size. Therefore functions are made inline when they are small enough to be defined in one or two lines.

vi) Following are situations where inline expansion may not works are:

1) For functions returning values, if a loop, a switch, or a goto exists.
2) For functions not returning values, if a return statement exists.
3) If function contains static variables.
4) If inline functions are recursive.

vii) Example of inline function

```
# include <iostream.h>
inline int add (int x, int y)
{
    return (x + y);
}
inline int subtract (int a, int b)
{
    return (a – b);
```

```
}
    main ( )
{
    int p = 15;
    int q = 10;
    cout <<add (p, q) << "/n";
    cout <<subtract (p, q) << "/n';
}
```

output of program.

25
5

OPERATOR OVERLOADING

4.4 DEFINITION:

Overloading means assigning different meaning to an operation depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to the operators.

Example:-

The input/output operators << and >> are good examples of operator overloading although the built in definition of the << operator is for shifting of bits. It is also used for displaying the values of various data types.

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

1. Class member access operators (., . *)
2. Scope resolution operator (: :)
3. Size operator (sizeOof)
4. Conditional operator (?:)

Defining operator overloading –

i) To define an additional task to an operator, operator function is used.

The general form is –

Return type classname :: operator op (anglist)

```
{
        Function body //task defined
}
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. Operator op is the function name.

ii) Operator functions must be either member functions or friend function. The difference is a friend function will have only one argument for unary operators and two for binary operators. While a member function has no arguments for unary operators and only one for binary operators. This is because the object

used to invoke the member function is passed implicitly and therefore is available for the member function. This is not case with friend function, because arguments may be passed either by value or by reference.

iii) Example –

```
Vector operator – ( ) ; // unary minus
Vector operator + (vector); // vector addition
Friend vector operator + (vector, vector) // vector addition
Friend vector operator – (vector); // unary minus
Vector operator – (vector & a); // subtraction
Int operator == (vector); // comparision
Friend int operator == (vector,vector) //comparision
```

iv) The process of overloading involves –

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op () in the public part of the class. It may be either a member or friend function.
- Define the operator function to implement the required operation.

v) Overloaded operator functions can be invoked by expressions such as.

Op x or x op

For unary operators and

x op y

for binary operators.
And op x would be interpreted as
operator op (x)
for friend functions and x op y would be interpreted as
x.operator op (y)
in member function and
operator op (x,y) in friend function.

4.5 UNARY OPERATOR OVER LOADING -

i) In overloading unary operator, a friend function will have only one argument, while a member function will have no arguments.

ii) Let us consider the unary minus operator. A minus operator, when used as a unary takes just one operand.

iii) This operator changes the sign of an operand when applied to a basic data item. Following example shows how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variables.

```
# include <iostream.h>
class unary
{
        int x, y, z;
```

```
public:
        void getdata (int a, int , intc);
        void display (void);
        void operator – (); // overload unary minus.

};

void unary :: getdata (int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void unary : : display (void)
{
        cout << x << " " << y << " " << z << "\n";
}
void unary ::operator –()
{
        x = -x ;
        y = -y ;
        z = -z ;
}
main ( )
{
        unary u;
        u.getdata(20, -30, 40);
        cout << " u : " ;
        u. display ( ) ;
        -u;
        cout << " u : " ;
        u. display ( ) ;
}
```

output: -

```
        u :     20      -30     40
        u :     -20     30      -40
```

iv) The function operator – ( ) takes no argument because this function is a member function of the same class, it can directly access the member of the object which activated it.

v) It is possible to overload a unary minus operator using a friend function as follows :

```
Friend void operator – (unary & u);
Void operator – (unary & u)
{
        u.x = -u.x;
        u.y = -u.y;
        u.z = -u.z;
}
```

The argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator–(). Therefore changes made inside the operator function will not reflect in the called object.

4.6 BINARY OPERATOR OVERLOADING:-

i) In overloading binary operator, a friend function will have two arguments, while a member function will have one argument.

ii) Following example shows how to overload + operator to add 2 complex number

```
# include<iostream.h>
class complex
{
        float x, y;
  public :
        complex ( );
        complex (float real, float imag)
        {
                x = real;
                y = imag;
        }
       complex operator +(complex);
       void display(void);
};
complex complex:: operator+(complex c)
  {
        complex temp;
        temp.x = x + c.z;
        temp.y = y + c.y;
        return (temp);
  }
  void complex : : display (void)
  {
        cout << z << " + j "<< y << " \n ";
  }
  main ( )
  {
        complex c1,c2,c3;
        c1 = complex (2.5, 3.5);
        c2 = complex (1.6, 2.7);
        c3 = c1 + c2;
        cout << "c1 = "; c1.display( ) ;
        cout << "c2 = "; c2.display( ) ;
        cout << "c3 = "; c3.display( ) ;
 }
  output: -

                c1 = 2.5 + j 3.5
                c2 = 1.6 + j 2.7
                c3 = 4.1 + j 6.2
```

iii) In the above program, the function operator +, is expected to add two complex values and return a complex value as a result but receives only one value as argument.

iv) The function is executed by the statement

```
c3 = c1 + c2
```

Here, the object c1 is used to invoke the function and c2 plays the role of an argument that is passed to the function. The above statement is equivalent to

```
c3 = c1.operator +(c2)
```

Here, in the operator +() function, the data members of c1 are accessed directly and the data member of c2 are accessed using the dot operator. Thus in overloading binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

4.7 OVERLOADING BINARY OPERATORS USING FRIENDS

i) Friend functions may be used in the place of member functions for overloading a binary operator. The only difference being that a friend function requires two arguments to be explicitly passed to it while a member function requires only one.

ii) The same complex number program with friend function can be developed as –
friend complex operator +(complex, complex);
and we will define this function as –

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

in this case, the statement

```
c3 = c1 + c2;
```

is equal to c3 = operator + (c1, c2)

iii) In certain situation it is require to use a friend function rather than member function.

Example-
If we need to use two different types of operands for a binary operator, i.e. one is an object and another is a built in type data.

Example:- A = B + 2;

Where A & B are objects of same class.
This will work for member function, but

```
A = 2 + B;
```

Will not work. Because, here left-hand operand which is responsible for invoking the member function should be an object of the same class. However a friend

iv) The following program performs scalar multiplication of a vector, which shows how to overload operator >> and <<.

```
# include < iostream.h >
const size = 3;
class vector
{
        int v[size];
public:
        vector ( );
        vector (int * x);
        friend vector operator  * (int a, vector b);
        friend vector operator * (vector b, int a);
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
};
vector :: vector()
{
        for (int i = 0; i<size; i++)
                v[i] = 0;
}
vector :: vector(int * x)
{
        for(int i = 0; i<size; i++)
                v[i] = x[i];
}
vector operator * (int a, vector b)
{
        vector c;
        for (int  i= 0; i < size; i++)
                c.v[i] = a *  b.v[i];
       return c;
}
vector operator * (vector b, int a)
{
        vector c;
        for (int  i= 0; i < size; i++)
                c.v[i] = b.v[i] * a;
       return c;
}
istream & operator >> (istream & din, vector & b)
{
        for (int i = 0; i < size; i++)
                din >> b.v [I];
        return (din);
}


ostream & operator << (ostream & dout, vector & b)
{
        dout << “( “ << b.v[0];
        for (int i = ; i < size; i++)
         dout << “ , ” << b.v[i];
       dout << “ ) ” << b.v [I];
       dout << “ ) “;
```

```
        return (dout);
}

int x [size] = {2, 4, 6};
main ( )
{
        vector m; // invokes constructor 1
        vector n =x; // invokes vonstructor 2
        cout << "enter elements of vector m " << "\n";
        cin>> m;
        cout<<"\n";
        cout<<"m="<<m<<"\n";

        vector p, q;
        p = 2 * m; // invokes friend 1
        q = n * 2 ; // invokes friend 2
        cout << "p = " << p << "\n";
        cout << " q = " << q << "\n ";

}
```

Output: -

enter elements of vector m

5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)

4.8 RULES FOR OVERLOADING OPERATORS –

1] Only existing operators can be overloaded. New operators cannot created.

2] The overloaded operator must have at least one operand that is of user-defined type.

3] We cannot change the basic meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from the other.

4] Overloaded operators follow the syntax rules of the original operators.

5] There are some operators that cannot be overloaded.

6] We cannot use friend functions to overload certain operator.

Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values. But, those overloaded by means of a friend return (din);

7] function take one reference argument.

8] Binary operators overloaded through a member function take one explicit argument and those which one overloaded through a friend function take two

9] When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.

10] Binary arithmetic operators such as +, - *, and / must explicitly return a value. They must not attempt to change their own arguments.

4.9 OPERATOR RETURNING VALUE –

Operator function can return value.

Following program shows how the operator returns value.

```
# include <iostream.h>
class counter
{
        private :
                unsigned int count;
        public :
                counter( )
                {
                        count = 0;
                }

                int get-count ( )
                {
                        return count++;
                }
                counter operator ++ ( )
                {
                        count ++;
                        counter temp;
                        temp.count = count;
                        return temp;
                }

        };
        main ( )
        {
                counter C1, C2;
                cout << "\n C1 =" << C1.get_count ( );
                cout << "\n C2 =" << C2.get_count ( );
                C1++;
                C2 = C1++;
                cout << "\n C1 = " << C1. get_count ( );
                cout << "\n C2 = " << C2. get_count ( );
        }
```

in this program the operator ++ ( ) function creates a new object of type counter, called temp, to use as a return value. It increments count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object. Finally it returns the temp object. Expression like

C1 ++;

Now return a value, so they can be used in other expressions, such as:

C2 = C1++;

C2++. get_count();

In the first of these statements the value returned from C1++ will be assigned to C2, and in the second it will be used to display itself using the get_count ( ) member function.

4.10 OVERLOADING ARITHMETIC AND RELATIONAL OPERATOR

I. In C++, we can overload arithmetic operator +, - *, / and also relational operators <, >, <=, >=.

II. We can overload + and <= and >= operator to work with strings. There is no operators for manipulating the string in C. But the drawback of string manipulations in C is that whenever a string is to be copied, the programmer must first determines its length and allocate the required amount of memory.

III. But in C++, it permits us to create our own definition of operators that can be used to manipulate strings very much similar to the decimal numbers.

Example

String3 = string1 + string2;

If (string 1 >= string2) string = string ;

IV. Strings can be defined as class objects, which can be then manipulated like the built-in types. Because strings vary in size, we must use new operator to allocate memory for each string and a pointer variable to point to the string array. Therefore, string object should hold length and location information for string manipulation see the following program.

```
# include <iostream.h>
class string
{
        char *p;
        int len;
public:
        string ( ) { len = 0; P = 0; } // create null string.
        string (const chan * s); // create string from away S.
        string (const string &s); // copy constructor.
        ~string ( ) {delete p; } // destructor.
        friend string operator + (const string & s, const string & t);
        friend void show (const string S);

}

string : : string (const char * s)
{
        len = strlen (s);
        p = new char [lent+1];
        strcpy (p,s);
}
string operator + (const strings & s, const string & t)
```

```
{
        string t1;
        t1.len = s.len + t. len;
        t1.p = new char [t1.len + 1];
        strcpy (t1.p,s.p);
        strcat(t1.p,t.p);
        return(t1);
}
string :: string(const string  &s)
{
        len = s.len;
        p = new char [len+1];
        strcpy (p,s.p);
}
int operator <= (const string & s, const string & t)
{
        int m = strlen (s.p);
        int n = strlen (t.p);
        if (m < = n)
                return (1);
        else
                return (0);
}
void show (const string s)
{
        cout << s. p;
}
int main ( )
{
        string str1 = "ABC";
        string str2 = "DEFG";
        string str3 = "HIJKLM";
        string s1, s2, s3;
        s1 = str1;
        s2 = str2;
        s3 = strl + str 3;
        cout <<in s1 = "; show (s1);
        cout <<in s2 = "; show (s2);
        cout <<"in s3 = "; show (s3);
        if (s1 <= s3)
        {
                show (s1);
                cout <<"smaller than";
                show (s3);
        }
        else
        {
                show (s3);
                cout <<"smaller than";
                show (s1);
         }
       return 0;

}
```

output of the program: -

s1 = ABC
s2 = DEFG
s3 = ABCHIJKLM

ABC is smaller than ABCHIJKLM.

❖❖❖

5

INHERITANCE

INTRODUCTION

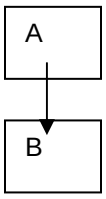
C++ supports the concept of reusability once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance*.

The old class is referred to as base class or super class.

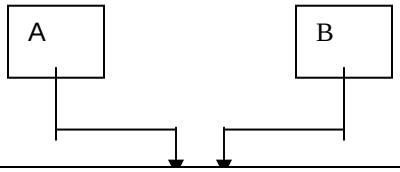
And the new one is called as derived class or subclass.

Types of Inheritance: -

1. **Single Inheritance** – A derived class with only one base class, is called Single Inheritance.



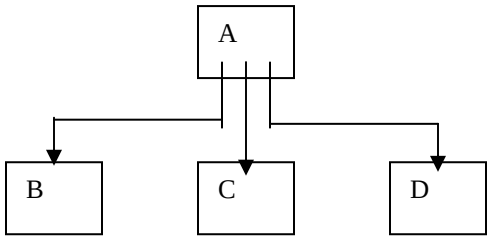
2. **Multiple Inheritance** – A derived class with several base classes, is called multiple Inheritance.



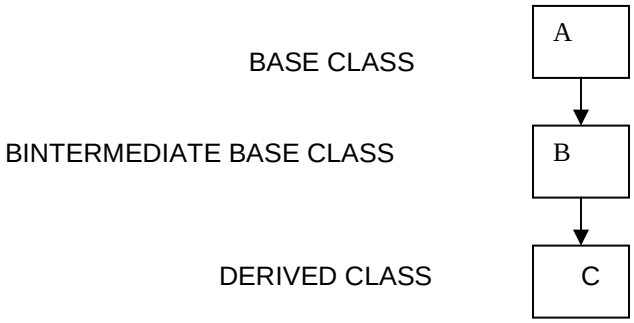
3. **Hierarchical Inheritance** – The properties of one class may be inherited by more than one class is called hierarchical Inheritance.

A

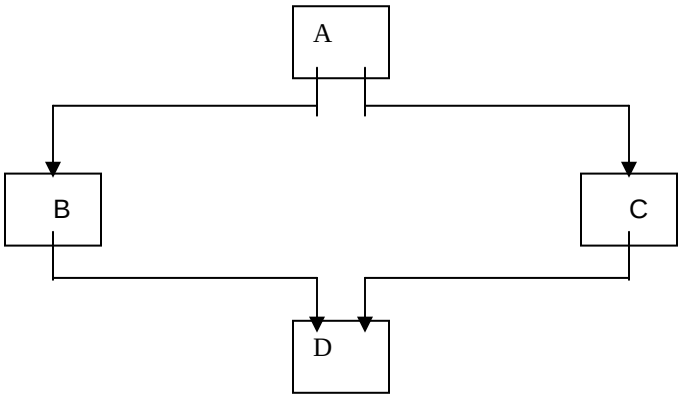
BASE CLASS



4. **Multilevel Inheritance** – The mechanism of deriving a class from another derived class is known as multilevel Inheritance.



5. **Hybrid Inheritance** – The hybrid Inheritance is a combination of all types of Inheritance.



DERIVED AND BASE CLASS

1. **Base class: -** A base class can be defined as a normal C++ class, which may consist of some data and functions.

Ex –

```
Class Base
        {
                : : : : :
        }
```

2. **Derived Class: -**
   i). A class which acquires properties from base class is called derived class.

   ii). A derived class can be defined by class in addition to its own detail.

The general form is –
Class derived-class-name: visibility-mode base-class-name

```
{
        : : : : :
}
```

The colon indicates that the derived class name is derived from the base-class-name. The visibility-mode is optional and if present it is either private or public.

PUBLIC INHERITANCE

i) When the visibility-mode is public the base class is publicly inherited.

ii) In public inheritance, the public members of the base class become public members of the derived class and therefore they are accessible to the objects of the derived class.

iii) Syntax is –

```
class ABC : public PQR
        {
                members of ABC
        };
```

Where PQR is a base class illustrate public inheritance.

iv) Let us consider a simple example to illustrate public inheritance.

```
# include<iostream.h>
class base
{
    int no1;
      public:
        int no2;
        void getdata();
```

```
                int getno1();
                void showno1();

};

Class derived: public Base // public derivation.
{
      int no3;
         public:
                    void add();
                    void display();
};

void Base :: getdata()
{
      no1 = 10;
      no2 = 20;

}

int Base :: getno1()
{
      return no1;

}

void Base :: showno1()
{
     cout <<"Number1 ="<<no1 <<"\n";

}

void Derived :: add()
{
    no3 = no2 + getno1(); // no1 is private
}

void Derived :: display()
{
     cout << "Number1 = "<<getno1() <<"\n";
                              cout << "Number2 = "<<no2<<\n";
                              cout << "Sum " <<no3 << "\n";
}
main ( )
{
      derived d;
      d.getdata ( );
      d.add ( );
      d.showno1 ( );
      d.display ( ) ;
      d.b = 100;
      d.add ( ) ;
      d.display ( ) ;
      return 0;
}
```

The output of the program is

```
Number1 = 10
Number1 = 10
Number2 = 20
Sum = 30

Number1 = 10
Number2 = 100
Sum = 110
```

v) In the above program class Derived is public derivation of the base class Base. Thus a public member of the base class Base is also public member of the derived class Derived.

PRIVATE INHERITANCE

i) When the visibility mode is private, the base class is privately inherited.

ii) When a base class is privately inherited by a derived class, public members of the base class becomes private members of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

iii) Public member of a class can be accessed by its own objects using the dot operator. So in private Inheritance, no member of the base class is accessible to the objects of the derived class.

iv) Syntax is –

```
class ABC: private PQR
{
  member of ABC
 }
```

v) Let us consider a simple example to illustrate private inheritance.

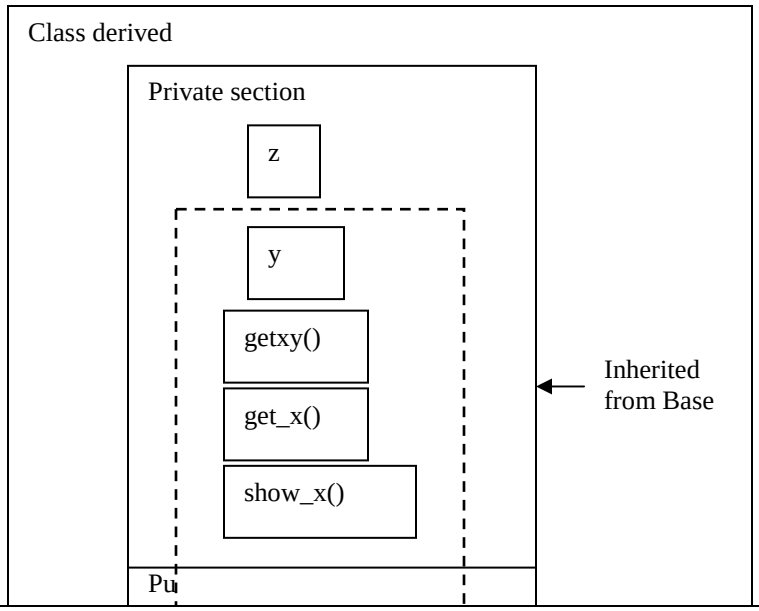
```
#include <iostream.h>
class base
{
      int x;
          public :
                    int y;
                    void getxy() ;
                    int get_x(void) ;
                    void show_x(void) ;
};
void base ::getxy(void)
{
          cout<<"Enter Values for x and y : " ;
          cin >> x >> y ;
}
int base : : get_x()
{
          return x;
}
void base :: show_x()
```

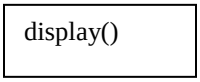
```
        cout <<"x = " << x << "\n";
}
void Derived : : mul( )
{
        getxy();
       z = y * get_x ( ) ;
}
void Derived ::display()
{
        show_x ( ) ;
        cout <<"y = " <<y<<"\n";
        cout<<"z = " <<z<<"\n";

}
main ( )
{
        Derived d;
        d. mul();
        d.display() ;
        //d.y = 4a; won't work y has become private.
        d.mul();
        d.display();
}
```

Output: -

```
        Enter values for x and y: 510
        X = 5
        Y = 10
        Z = 50
        Enter values for x and y:1220
        X = 12
        Y = 20
        Z = 240
```

The membership of the derived class derived is shown in following figure,





PROTECTED MEMBERS

1. If we want to inherit private data by a class, the only option is to change the visibility limit from private to public, but this will eliminate the advantage of data hiding.

2. Therefore to achieve data hiding, C++ provides a third visibility modifier, protected which has limited purpose in inheritance.

3. A member declared protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes.

4. When a protected member is inherited in public mode, it becomes protected in the derived class too, and therefore is accessible by the member.