OBJECT ORIENTED PROGRAMMING IN C++

<u>Module 3</u>

Classes and Objects

Object-oriented programming (OOP) is a conceptual approach to design programs. This session will continue our discussion on OOP and C++. Some of the features of OOP are Data encapsulation and data hiding, and these possible because of data abstraction. All these features have one thing in common – the vehicle to implement them. This vehicle is *class*. It is a new data type similar to structures. Since, the structure data type is a stepping-stone to this singularly important concept of C++, let us take another look at it.

Structures

A structure is a user-defined data type, which may contain different data types as its members. Creating a structure is a two-part process. First, a structure template is defined. This template gives a lot of information to the compiler. For instance, How it is stored in the memory? How many member variables are there in this composite data element? What are the types of its member variables? For data types like int, float, double, char this information is built into the compiler. But for the user-defined data types like structures, it has to be provided to the compiler. This is given in the definition of the structure template. This template creates a new data type. Variables of this new data type can then be declared and used like basic data types.

e.g.

```
struct Student
 {
        int Rollno;
        char Name[15];
        int Marks[6];
        float percent;
};
```

The above example depicts structure called students with enclosed data variables. The keyword *struct* is used to define a structure template. Student is a name or tag. The variables of this type can be declared as follows :

```
struct Student s1,s2;
        [or ]
```

In C++ you can even omit the struct tag and declare the variables as,

```
Student s1 = ( 100, “Sanket” , 20,10,30,40,50,60, 35.0};
Student s2,jack;
Student *sptr = &s1;
Student s[100];
```

In C++ , functions also can be declared in a structure template. These functions are called as member functions.

e.g.

```
struct Student
 {
        int Rollno;
        char Name[15];
        int Marks[6];
        float percent;
        void Getdetails();
        void Putdetails();
        int  ClacPercent();
};
```

There are three functions in the template given above. These functions have been incorporated in the structure definition to make it a fundamental unit. The building together of data and the functions that operate on that data is called as *data encapsulation.* But the problem is that the member variables of this structure can be accessed by functions other than the ones declared in the structure template. This direct access of member variables of a structure by functions outside the structure is not what OOP has to offer. The actual OOP methodology in this regard is implemented in C++ not by the structure data type, but by the data type *class.* The data type *class* is just like structures, but with a difference. It also has some access specifier, which controls the access to member variables of a class .

Introduction to Classes

Object-oriented programming (OOP) is a conceptual approach to design programs. It can be implemented in many languages, whether they directly support OOP concepts or not. The C language also can be used to implement many of the object-oriented principles. However, C++ supports the object-oriented features directly. All these features like Data abstraction, Data encapsulation, Information hiding etc have one thing in common – the vehicle that is used to implement them. The vehicle is " *class.*"

Class is a user defined data type just like *structures***,** but with a difference. It also has three sections namely *private***,** *public* and *protected.* Using these, access to member variables of a class can be strictly controlled.

Class Definition

The following is the general format of defining a class template:

```
class tag_name
 {
   public  :          // Must  type member_variable_name;
             :
                      type member_function_name();
            :
   private:           // Optional
                      type member_variable_name;
                      :
                      type member_function_name();
                      :
 };
```

The keyword *class* is used to define a class template. The private and public sections of a class are given by the keywords *'private' and 'public'* respectively. They determine the accessibility of the members. All the variables declared in the class, whether in the private or the public section, are the members of the class. Since the class scope is private by default, you can also omit the keyword private. In such cases you must declare the variables before public, as writing public overrides the private scope of the class.

e.g.

```
class tag_name
 {
                      type member_variable_name;    // private
                      :
                      type member_function_name();  // private
                      :


   public  :          // Must  type member_variable_name;
                      :
                      type member_function_name();
                      :

 };
```

The variables and functions from the public section are accessible to any function of the program. However, a program can access the *private* members of a class only by using the *public* member functions of the class. This insulation of data members from direct access in a program is called *information hiding.*

e.g.

```
class player
 {
  public :
                void getstats(void);
                void showstats(void);
                int no_player;

  private :
                char  name[40];
                int   age;
                int   runs;
                int   tests;
                float average;
                float calcaverage(void);
};
```

The above example models a cricket player. The variables in the *private* section – name, age, runs, highest, tests, and average – can be accessed only by member functions of the class calcaverage(), getstats() and showstats(). The functions in the public section - getstats() and showstats() can be called from the program directly , but function calcaverage() can be called only from the member functions of the class – getstats() and showstats().

With information hiding one need not know how actually the data is represented or functions implemented. The program need not know about the changes in the private data and functions. The interface(public) functions take care of this. The OOP methodology is to hide the implementation specific details, thus reducing the complexities involved.

Classes and Objects

As seen earlier, a class is a vehicle to implement the OOP features in the C++ language. Once a class is declared, an *object* of that type can be defined. An *object* is said to be a specific instance of a class just like Maruti car is an instance of a vehicle or pigeon is the instance of a bird. Once a class has been defined several objects of that type can be declared. For instance, an object of the class defined above can be declared with the following statement:

```
player Sachin, Dravid, Mohan ;
        [Or]
class player Sachin , Dravid, Mohan ;
```

where Sachin and Dravid are two objects of the class player. Both the objects have their own set of member variables. Once the object is declared, its public members can be accessed using the dot operator with the name of the object. We can also use the variable no_player in the public section with a dot operator in functions other than the functions declared in the public section of the class.

e.g.

```
Sachin.getstats();
Dravid.showstats();
Mohan.no_player = 10;
```

Public, Private and Protected members:

Class members can either be declared in public','protected' or in the 'private' sections of the class. But as one of the features of OOP is to prevent data from unrestricted access, the data members of the class are normally declared in the private section. The member functions that form the interface between the object and the program are declared in public section (otherwise these functions can not be called from the program). The member functions which may have been broken down further or those, which do not form a part of the interface, are declared in the private section of the class. *By default all the members of the class are private.* The third access specifier '*protected'* that is not used in the above example, pertains to the member functions of some new class that will be inherited from the base class. As far as non-member functions are concerned, *private* and *protected* are one and the same.

Memory Allocation for Objects

The memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed 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 function, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated seperately for each object. Separate memory locations for the objects are essential beacuase the member variables will hold different data values for different objects.

Member Functions of a Class

A member function of the class is same as an ordinary function. Its declaration in a class template must define its return value as well as the list of its arguments. You can declare or define the function in the class specifier itself, in which case it is just like a normal function. But since the functions within the class specifier is considered *inline* by the compiler we should not define large functions and functions with control structures, iterative statements etc should not be written inside the class specifier. However, the definition of a member function differs from that of an ordinary function if written outside the class specifier. The header of a member function uses the scope operator (::) to specify the class to which it belongs. The syntax is:

```
return_type  class_name :: function_name (parameter list)
{
        :
}
```

e.g.

```
void player :: getstats (void)
 {

        :

 }

void player :: showstats (void)
 {
        :
        :
 }
```

This notation indicates that the functions getstats () and showstats() belong to the class player.

Passing and Returning Objects

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

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

The first method is called **call – by – value**. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called **call – 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.

The pass – by – reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Objects can be passed to a function and returned back just like normal variables. When an object is passed by content , the compiler creates another object as a formal variable in the called function and copies all the data members from the actual variable to it. Objects can also be passed by address.

e.g.

```
class check
{
        public :
                check add(check);
                void get()
                {
                        cin >> a;
                }
                void put()
                {
                        cout << a;
                }
        private :
                int a;
};

void main()
{
        check c1,c2,c3;
        c1.get();
        c2.get();
        c3 = c1.add(c2);
        c3.put();
}

check check :: add ( check c2)
{
        check temp;
        temp.a = a + c2.a;
        return ( temp);
}
```

The above example creates three objects of class check. It adds the member variable of two classes, the invoking class c1 and the object that is passed to the function , c2 and returns the result to another object c3.

You can also notice that in the class add() the variable of the object c1 is just referred as ‘a’ where as the member of the object passed .i.e. c2 is referred as ‘c2.a’ . This is because the member function will be pointed by the pointer named this in the compiler where as what we pass should be accessed by the extraction operator ‘.’. we may pass more than one object and also normal variable. we can return an object or a normal variable from the function. We have made use of a temporary object in the function add() in order to facilitate return of the object.

Example for Objects as Function Arguments using Call – By – Value & Call – by – Reference

```
# include <iostream.h>

class item
{
        private:
                int data;
        public:
```

```
                    void getdata(int a)
                    {
                              data=a;
                    }

                    void putdata(item x, item y)
                    {
                              cout<<x.data;
                              cout<<y.data;
                    }


                    void showdata(item *x, item *y)
                    {
                              cout<<x->data;
                              cout<<y->data;
                    }
};

void main()
{
          item n1, n2,n3;
          n1.getdata(20);
          n2.getdata(30);
          n3.putdata(n1,n2);         //pass by value
          n3.showdata(&n1,&n2); // pass by reference
}
```

Example for Function Returning Objects

A function can not only receive objects as arguments but also can return them. The following program illustrates how an object can be created within a function and returned to another function.

```
# include <iostream.h>

class number
{
          private:
                    int no;
          public:
                    void getdata(int a);
                    {
                              no=a;
                    }

                    number sum(number x, number y)
                    {
                              number z;
                              z.no=x.no+y.no;
                              return(z)
                    }

                    void putdata()
                    {
                              cout<<no;
                    }
};
```

```
void main()
{
        number a,b,c;
        a.getdata(100);
        b.getdata(200);
        c=c.sum(a,b);
        c.putdata();
}
```

How to write an inline class member function

In addition to global functions, you may request that non-static member functions of a class be inlined. The normal method of doing this is to simply define the function within the scope of the class definition. In this case the inline request is made automatically, so the keyword *inline* does not need to be written.

Here the member function integer :: store ( ) is an inline member function because it is completely defined within the context of the class definition.

e.g.

```
class integer
{
        public :
                 //  This is an inline function
                void store (int n )
                {
                        number =n ;
                }

        private:
                int number ;
};
```

Functions inlined within the class definition are not evaluated by the compiler until the entire class scope has been processed. That's why, if they are defined first, they can refer to members defined later, or even compute the size of the class. The only problem with this class is that now it has been " cluttered up" with the actual definition of the integer::store() function.

A better approach is to move the function definition outside the class and use the scope resolution operator to give it class scope.

In order to make the inline request to the compiler, you must write the keyword inline in front of both the function declaration and its definition.

e.g.

```
class integer
{
        public:
                inline void store (int) ;

        private:
                int number ;
} ;

// This is an inline function

inline void integer :: store (int n )
{
        number = n;
}
```

The above example shows the correct way to define an inline function, the usage of the keyword inline in front of the declaration (within the class definition) is not always necessary. This is the case when the definition (still preceded by the keyword inline) is encountered by the compiler before it is ever called. Now there is no problem because the inlining will still occur.

Static Data Members

A data member of a class can be qualified as **static.** The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurences of all the objects. Program given below illustrates the use of a static data member.

```
# include <iostream.h>

class term
{
        static int count;
        int data;
public:
        void getdata(int a);
        void getcount();
};

void term::getdata(int a)
{
        data=a;
        count++;
}
void term::getcount()
{
        cout<<"Count :" ;
        cout<<count;
}

int term :: count; //count defined

main()
{
        term a,b;
        a.getcount();
        b.getcount();
        a.getdata(10);
        a.getcount();
        b.getdata(30);
        b.getcount();
        a.getcount();
}
```

The output would be :

```
Count : 0
Count : 0
Count : 1
Count : 2
Count : 2
```

Notice the following statement in the program

int term :: count;

The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored seperately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are known as *class variables.*

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 two times, the variable count is incremented three times. Because there is only one copy of "count" shared by all objects.

While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives "count" initial value 1.

```
int term::count = 1;
```

Static Member Functions

Like static member variable, we can also have **static member functions**. A member function that is declared static has the following properties:

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name, instead of its objects, as follows:

```
Class – name :: function – name;
```

```
# include <iostream.h>

class term
{
        static int count;
        int data;
public:
        void getdata(int a);
        static void showcount();
};

void term::getdata(int a)
{
        data=a;
        count++;
}
static void term::showcount()
{
        cout<<"Count :" ;
        cout<<count;
}
```

```
int term :: count;

main()
{
        term a,b;
        a.getdata(10);
        b.getdata(30);
        term::showcount();
}
```

Remember, the following function definition will not work:

```
static void term :: showcount()
{
        cout<<data;
}
```

Because “data” is not a static data member.

Constant Member Functions

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototype(in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

Pointers to Objects

Passing and returning of objects is, however, not very efficient since it involves passing and returning a copy of the data members. This problem can be eliminated using pointers. Like other variables, objects of class can also have pointers. Declaring a pointer to an object of a particular class is same as declaring a pointer to a variable of any other data type. A pointer variable containing the address of an object is said to be pointing to that object. Pointers to objects can be used to make a call by address or for dynamic memory allocation. Just like structure pointer, a pointer to an object also uses the arrow operator to access its members. Like pointers to other data types, a pointer to an object also has only one word of memory. It has to be made to point to an already existing object or allocated memory using the keyword *‘new’.*

e.g.

```
string str;        // Object
string *sp;        // Pointer to an object
sp = &str;         // Assigns address of an existing object
sp = new string    // Allocates memory with new.
```

A simple example of using pointers to objects is given below.

```
class player
{
        public :
                void getstats(void);
                void showstats(void);
        private :
                char  name[40];
                int   age;
                int   runs;
```

```
                    int   tests;
                    float average;
                    float calcaverage(void);
    };

    void main()
    {
            player Sachin;
            Sachin.getstats();
            Sachin.showstats();

            player  *Dravid;

            Dravid = new player;

            Dravid->getstats();
            Dravid->showstats();
     }
```

Array of Objects

As seen earlier, a class is a template, which can contain data items as well as member functions to operate on the data items. Several objects of the class can also be declared and used. Also, an array of objects can be declared and used just like an array of any other data type. An example will demonstrate the use of array of objects.

e.g.

```
    class student
    {
            public :
                    void getdetails();
                    void printdetails();
            private :
                    int rollno;
                    char name[25];
                    int marks[6];
                    float percent;
     };

    void student :: getdetails()
    {
            int ctr,total;
            cout << ”enter rollno”;
            cin >> rollno ;
            cout << ”enter name”;
            cin >> name;
            cout << ” enter 6 marks “ ;
            for( ctr = 1 ;ctr <= 6 ; ctr++ )
            {
                    cin >> marks[ctr];
                    total = total + marks[ctr];
            }
            percent = total / 6;
    }

    void student :: printdetails ()
    {
            cout << rollno << name << percent ;
    }
```

```
	void main()
	{
		student records[50];
		int x=0;
		cout << " How many students ";
		cin >> x;
		for ( int i =1; i<= x; i++)
		{
			records[i].getdeatils();
		}
		for ( int i =1; i<= x; i++)
		{
			records[i].printdeatils();
		}
	}
```

As can be seen above, an array of objects is declared just like any other array. Members of the class are accessed, using the array name qualified by a subscript.

The statement,

```
	records[y].printdetails();
```

invokes the member funs printdetails() for the object given by the subscript y. For different values of subscript, it invokes the same member function, but for different objects.

The Special Pointer '*this*'

When several instances of a class come into existence, it naturally follows that each instance has its own copy of member variables. If this were not the case, then for obvious reasons it would be impossible to create more than one instance of the class. On the other hand, even though the class member functions are encapsulated with the data members inside the class definition, it would be very inefficient in terms of memory usage to replicate all these member functions and store the code for them within each instance. Consequently, only one copy of each member function per class is stored in memory, and must be shared by *all* of the instances of the class.

But this poses a big problem for the compiler: How can any given member function of a class knows which instance it is supposed to be working on ? In other words, up to now in a class member function you have simply been referring to the members directly without regard to the fact that when the instantiations occur each data member will have a different memory address. In other words, all the compiler knows is the offset of each data member from the start of the class.

The solution to this dilemma is that, in point of fact, each member function does have access to a pointer variable that points to the instance being manipulated. Fortunately this pointer is supplied to each member function automatically when the function is called, so that this burden is not placed upon the programmer.

This pointer variable has a special name '*this' (reserved word)*. Even though the *this* pointer is implicitly declared, you always have access to it and may use the variable name anywhere you seem appropriate.

e.g.

```
	class try_this
	{
		public :
			void print();
			try_this add(int);

		private :
```

```
            int ivar;
};

void print()
{
        cout << ivar;
        cout << this -> ivar ;
}
```

The function print refers to the member variable ivar directly. Also, an explicit reference is made using the *this* pointer. This special pointer is generally used to return the object, which invoked the member function. For example,

```
void main()
{
        try_this t1,t2;

        t2 = t1.add(3);
        t2.print();
}

try_this  try_this :: add(int v)
{
        ivar = ivar + v;
        return ( *this);
}
```

In the above example if ivar for t1 is 10 and value in v is 2, then the function add() adds them and ivar for t1 becomes 12 . We want to store this in another object t2, which can be done by returning the object t1 using *this to t2. The result of t2.print() now will be 12.

Dereferencing the Pointer *this*

Sometimes a member function needs to make a copy of the invoking instance so that it can modify the copy without affecting the original instance. This can be done as follows :

```
try_this temp(*this);
try_this temp = *this ;
```

In OOP emphasis is on how the program represents data. It is a design concept with less emphasis on operational aspects of the program. The primary concepts of OOP is implemented using class and objects. A class contains data members as well as function members. The access specifiers control the access of data members. Only the public members of the class can access the data members declared in private section. Once class has been defined, many objects of that class can be declared. Data members of different objects of the same class occupy different memory area but function members of different objects of the same class share the same set of functions. This is possible because of the internal pointer '*this' which keeps track of which function is invoked by which object.

Constructor and Destructor Functions

Since C++ supports the concept of user-defined classes and the subsequent initiations of these classes, it is important that initialization of these instantiations be performed so that the state of any object does not reflect " garbage". One of the principles of C++ is that objects know how to initialize and cleanup after themselves. This automatic initialization and clean up is accomplished by two member functions – the constructor and the destructor.

Constructors

By definition, a constructor function of some class is a member function that automatically gets executed whenever an instance of the class to which the constructor belongs comes into existence. The execution of such a function guarantees that the instance variables of the class will be initialized properly.

A constructor function is unique from all other functions in a class because it is not called using some instance of the class, but is invoked whenever we create an object of that class.

A constructor may be overloaded to accommodate many different forms of initialization for instances of the class. i.e. for a single class many constructors can be written with different argument lists .

Syntax rules for writing constructor functions

- Its name must be same as that of the class to which it belongs.
- It is declared with no return type (not even void). However, it will implicitly return a temporary copy of the instance itself that is being created.
- It cannot be declared static (a function which does not belong to a particular instance), const(in which you can not make changes).
- It should have public or protected access within the class. Only in very rare circumstances the programmers declare it in private section.

e.g.

```
class  boxclass
{
        public :
                        boxclass ( int x1, int y1, int x2, int y2);
                        void disp(void);
        private :
                        int x1, y1;
                        int x2, y2 ;
};

boxclass::boxclass(int ax1,int ay1, int ax2, int ay2)
{
        x1 = ax1 ;
        y1 = ay1 ;
        x2 = ax2 ;
        y2 = ay2 ;
}
```

Using the Constructor

There are basically three ways of creating and initializing the object. The first way to call the constructor is *explicitly* as :

```
boxclass bigbox = boxclass ( 1,1,25,79);
```

This statement creates an object with the name bigbox and initializes the data members with the parameters passed to the constructor function. The above object can also be created with an *implicit* call to the constructor :

```
boxclass bigbox(1,1,25,79);
```

Both the statements given above are equivalent. Yet, another way of creating and initializing an object is by direct assignment of the data item to the object name. *But, this approach works if there is only one data item in the class*. This is obvious because we cannot assign more than one value at a time to a variable.

e.g.

```
class counter
{
        public :
                        counter ( int c)   // constructor.
                         {
                                count = c;
```

```
                    };
        private :
                    int  count;

    };
```

we can now create an object as,

```
    counter cnt = 0;
```

In the above example , object cnt is initialized by a value zero at the time of declaration. This value is actually assigned to its data member count. This is the third way to initialize an object's data member. Thus, all the following statements to initialize the objects of the class counter are equivalent:

```
        counter c1(20);
        counter c1 =  counter(30);
        counter c1= 10;
```

Once the constructor for a class has been declared and defined, the program must use it. In other words, objects cannot be declaring without initializing them. For instance, in the above example, an object of class counter has to be declared and initialized in one of the three ways given above. Uninitialized objects like :

```
        counter cx,cy;
```

are not allowed.

However, sometimes the data members of an object may not require initialization. May be they have to be accepted from the user later. C++ is not so rigid in its approach. It has defined means of circumventing such a problem. The key to defining an uninitialized object is to have a constructor with default arguments.

e.g.

```
    class counter
    {
        public :
                    counter ( int c = 0)   // Constructor.
                    {
                        count = c;
                    };

        private :
                    int  count;
    };
```

In the example above, if the actual argument is not given, then the value of the formal variable c in the constructor defaults to zero. Thus, declaration of objects of this class can be uninitialized as well.

```
        counter c1(20);
        counter c2 =  counter(30);
        counter c3 = 10;
        counter c4,c5;
```

A constructor can be used to generate temporary instances as well , as given below :

```
        counter create( int cc)
        {
            return counter(c);
        }
```

An array of objects can be initialized at the time of declaration. A constructor has to be provided foe the same. The example given below uses the class employee to illustrate this.

```
class employee
{
        public :
                employee(char *n, int a, double s);

        private:
                char name[40];
                int age;
                double salary;
};

employee ::  employee (char *n , int a, double s)
{
        strcpy( name, n );
        age = a;
        salary = s;
}

void main()
{
        employee E[3] = { employee("sanat",25,9600),
                          employee("sanket",35,12600),
                          employee("priya",28,21600)
                        };

}
```

Destructors

A destructor function gets executed whenever an instance of the class to which it belongs goes out of existence. The primary usage of a destructor function is to release memory space that the instance currently has reserved.

Syntax rules for writing a destructor function

- Its name is the same as that of the class to which it belongs, except that the first character of the name is the symbol tilde (~).
- It is declared with no return type (not even void) since it cannot ever return a value.
- It cannot be static, const or volatile.
- It takes no input arguments , and therefore cannot be overloaded.
- It should have public access in class declaration.

Generally the destructor cannot be called explicitly (directly) from the program. The compiler generates a class to destructor when the object expires. Class destructor is normally used to clean up the mess from an object. Class destructors become extremely necessary when class constructor use the *new* operator, otherwise it can be given as an empty function. However, the destructor function may be called explicitly allowing you to release the memory not required and allocate this memory to new resources, in Borland C++ version 3.1.

The default Constructor and Destructor

If you fail to write a constructor and destructor function, the compiler automatically supplies them for you. These functions have public access and essentially do nothing useful. If you were to write these functions yourselves, this would look like:

```
class employee
```

```
        {
                public :
                                employee()
                                {
                                }

                                ~employee();
                                {
                                }
        };
```

A Constructive Example

Consider an example , to model a user-defined data type for strings. The object simulates a character array (string) using a character pointer and an integer variable for its actual size, which can be determined at its run-time. The object doesn’t use a character array , since it may impose a limit on the number of characters that can be stored.

e.g.

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

        class string
        {
                public :
                        string ( char *s);
                        ~string();
                        void putstr();

                private :
                        char *cptr;
                        int size;

        };

        void main(void)
        {
                string s1(“ Hello student    \n”);
                string s2 = “ Welcome to C++   \n”;
                string s3 = string ( “ its fun \n”);

                s1.putstr();
                s2.putstr();
                s3.putstr();
        }

        string::string(char *s)
        {
                size = strlen(s);
                cptr = new char[size + 1];
                strcpy(cptr,s);
        }

        string::~string()
        {
                delete cptr;
        }
```

```
void string::putstr()
{
        cout << cptr;
}
```

The class defined in the above example contains a character pointer, which allocates memory at run–time, after determining the actual size required. This programme demonstrates the use of class along with the constructor and destructor to create a user defined data type String. The constructor function contains a default argument of null character, which will be assigned to the variable cptr in the absence of an actual parameter. The destructor uses the *delete* operator to release the memory allocated by the constructor .

A class can contain data members as well as function members. A member function is the only way to access the private section members of a class. For initializing the data members of the class, we may write a function just like any other function. But then, we have to invoke this function explicitly. C++ provides a special function called constructor. This function is called automatically when we create an object. It contains code to initialize the member variables using the formal parameters if they are passed via objects. If we write such a function then we must pass parameters each time. To evade this problem we can write many constructors to give some flexibility to create objects with different parameter lists. We can also use default arguments in the constructor.

C++ also provides a special member function called destructor, which is called automatically when an object expires as per the scope rules. Both constructor and destructors do not have return types, not even void. The destructor cannot have any parameters as well.

Exercises

1. Create a class called Time that has a separate data members for day, month and year. A constructor should be used to initialize these members. Then write a function to add these dates and store the result in a third object and display it.

2. WAP to add co-ordinates of the plane. The class contains x and y co-ordinates. Create three objects. Use a constructor to pass one pair of co-ordinates and a function to accept the second pair. Add these variables of two objects and store the result in the third.