Constructors and Destructors

9

CHAPTER

CHAPTER OUTLINE

- 9.1 Introduction
- 9.2 Constructors and Destructors
- 9.3 Characteristics of Constructors and Destructors
- 9.4 Applications with Constructors
- 9.5 Constructors with Arguments (Parameterized Constructor)
- 9.6 Overloading Constructors (Multiple Constructors)
- 9.7 Array of Objects Using Constructors
- 9.8 Constructors with Default Arguments
- 9.9 Copy Constructors
- 9.10 The `const` Objects
- 9.11 Destructors
- 9.12 Calling Constructors and Destructors
- 9.13 Qualifier and Nested Classes
- 9.14 Anonymous Objects
- 9.15 Private Constructors and Destructors
- 9.16 Dynamic Initialization Using Constructors
- 9.17 Dynamic Operators and Constructors
- 9.18 `main()` as a Constructor and Destructor
- 9.19 Recursive Constructors
- 9.20 Program Execution before `main()`
- 9.21 Constructor and Destructor with Static Members
- 9.22 Local Versus Global Object
- 9.23 More Programs

9.1 INTRODUCTION

When a variable is declared and if not initialized, it contains garbage value. The programmer needs to assign appropriate value to the variable. The compiler itself cannot carry out the process of initialization of a variable. The programmer needs to explicitly assign a value to the variable. Initialization prevents the variable from containing garbage value.

Consider an example

```
float height;  // variable declaration
height=5.5;    // assigning value to variable
```

In the above example, `height` is a variable of `float` type. It holds any garbage value before initialization. In the next statement, variable `height` is initialized with 5.5.

C++ handles abstract data type, which is the combination of one or more basic data types. An object holds copies of one or more individual data member variables. When an object is created, its data member contains garbage value.

We learnt in the previous chapter that declaring static member variables facilitates the programmer to initialize member variables with desired values. The drawback of static members is that only one copy of the static member is created for the entire class. All objects share the same copy, which does not provide security.

Another approach is to define an object as static. When an object is declared as static, all its member variables are initialized to zero. This is a useful approach. Declaring static object does not create common copies of member variables. Every static object has its own set of member variables. The drawback of static object is that the contents of static object remain throughout the program occupying more memory space. The following program illustrates this point.

9.1 Write a program to declare static objects and display their contents.

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

class data
{
   int x;
   float y;
   public:
   void show()
   {
   x++;
   y++;
   cout<<"\n x="<<x <<"\n y="<<y;
   }
};
void main()
{
   clrscr();
   static data A,B;
   cout<<"\n Object A";
   A.show();
   cout<<"\n Object B";
   B.show();
}
```

OUTPUT

```
Object A
x=1
y=1
Object B
x=1
y=1
```

Explanation: In the above program, objects `A` and `B` are declared as static. Their data member variables are initialized to zero. The member function `show()` increments value of member variables by one and displays them. Both the objects `A` and `B` invoke the function `show()`. The contents displayed are same for both the objects. Therefore, we can say that individual copies are created for each static object.

The main drawback of static object is that its value remains in the memory throughout the program. The following program illustrates that the static object remains in the memory even if it goes out of scope.

9.2 Write a program to demonstrate that static object persists its values.

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

class data
{
    int x;
    float y;
    public:
    void show()
    {
    x++;
    y++;
    cout<<"\n x="<<x <<"\n y="<<y;
    }
};
void main()
{
    clrscr();
    void display (void);
    display();
    display();
}
void display()
{
    static data K;
    K.show();
}
```

OUTPUT

```
x=1
y=1
x=2
y=2
```

Explanation: In the above program, function `display()` is a normal function. The member function `show()` performs the same task as described in the previous program. The static object K is declared in the function `display()`. The object K is local object of function `display()`. The function `main()` invokes function `display()` two times. In the first call, the contents displayed are one and one. In the second call, the contents displayed are two and two. It means in second call, the previous values are not cleared. Hence, we can say that static objects are intact or persist their values. The static object K remains throughout the program and is not destroyed even if execution of `display()` terminates.

9.2 CONSTRUCTORS AND DESTRUCTORS

In the previous chapter, we defined a separate member function for reading input values for data members. Using object, member function is invoked and data members are initialized. The programmer needs to call the function. C++ provides a pair of in-built special member functions called `constructor` and `destructor`. The constructor constructs the objects and destructor destroys the objects. In operation, they are opposite to each other. The compiler automatically executes these functions. The programmer does not need to make any effort for invoking these functions.

The C++ run-time arrangement takes care of execution of constructors and destructors. When an object is created, constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. The destructor is executed at the end of the function when objects are of no use or go out of scope. It is optional to declare constructor and destructor. If the programmer does not define them, the compiler executes implicit constructor and destructor.

Constructors and destructors are special member functions. They decide how the objects of a class are created, initialized, copied, and destroyed. Their names are distinguished from all other member functions because their names are same as the class they belong to. The only difference is that destructor is preceded by a ~ (tilde) operator.

Constructors and destructors have many attributes as that of normal member functions. We can declare and define them within the class, or declare them within the class and define them outside; however, they have few unique characteristics.

9.2.1 Constructors

If a `class B` has one or more constructors, one of them is invoked each time when we define an `object b` of `class B`. The constructor creates `object b` and initializes it. Constructors are also called when local or temporary objects of a class are created.

Example

```
B() { }
```

9.2.2 Destructors

Destructors are opposite to the constructor. The process of destroying the class objects created by constructors is done in destructor. The destructors have the same name as their class, preceded by a tilde (~). A destructor is automatically executed when object goes out of scope. It is also invoked when delete operator is used to free the memory allocated with class pointer. Similar to the constructor, it is not possible to define overloaded destructors and pass arguments to them. The class can have only one destructor. Destructors are called when these objects go out of scope.

Example

```
~B() { }
```

9.3 CHARACTERISTICS OF CONSTRUCTORS AND DESTRUCTORS

9.3.1 Constructors

(1) Constructors have the same name as that of the class they belong to.
(2) Constructors are executed when an object is declared.
(3) Constructors have neither `return` value nor `void`.
(4) The main function of constructor is to initialize objects and allocate appropriate memory to objects.
(5) Though constructors are executed implicitly, they can be invoked explicitly.
(6) Constructors can have default values and can be overloaded.
(7) The constructor without arguments is called as default constructor.

9.3.2 Destructors

(1) Destructors have the same name as that of the class they belong to preceded by ~ (tilde).
(2) Similar to constructors, the destructors do not have `return type` and not even `void`.
(3) Constructors and destructors cannot be inherited, though a derived class can call the constructors and destructors of the base class.
(4) Destructors can be virtual, but constructors cannot.

(5) Only one destructor can be defined in the destructor. The destructor does not have any arguments.
(6) Destructors neither have default values nor can be overloaded.
(7) Programmer cannot access addresses of constructors and destructors.
(8) TURBO C++ compiler can define constructors and destructors if they have not been explicitly defined. They are also called on many cases without explicit calls in programs. Any constructor or destructor created by the compiler will be public.
(9) Constructors and destructors can make implicit calls to operators `new` and `delete` if memory allocation/de-allocation is needed for an object.
(10) An object with a constructor or destructor cannot be used as a member of a union.

9.4 APPLICATIONS WITH CONSTRUCTORS

The initialization of member variables of class is carried out using constructors. The constructor also allocates required memory to the object. An example of a constructor is as follows:

```
class num
{
   private:
   int a, b,c;
   public:

   num (void); // declaration of constructor
   - - - - -
   - - ---
};
num:: num (void) // definition of constructor
{
   a=0;b=0;c=0; // value assignment
}
main()
{class num x;
}
```

In the above example, class `num` has three member integer variables `a,` `b,` and `c`. The declaration of constructor can be done inside the class and definition outside the class. In definition, the member variables of a class num are initialized to zero.

In the function `main()`, `x` is an object of type class `num`. When an object is created, its member variables (private and public) are automatically initialized to the given value. The programmer need not write any statement to call the constructor. The compiler automatically calls the constructors. If programmer writes a statement for calling a constructor, a constructor is called again. In case there is no constructor in the program, the compiler calls a dummy constructor. The constructor without argument is known as default constructor.

9.3 Write a program to define a constructor and initialize the class data member variables with constants.

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

class num
{
   private:
   int a,b,c;
   public:
   int x;
   num(void); // declaration of constructor
   void show()
   { cout<<"\n x="<<x <<"a="<<a<< "b="<<b<< "c="<<c; }
};
num:: num (void) // definition of constructor
{
   cout<<"\n Constructor called";
   x=5; a=0;b=1;c=2;
}
main()
{
   clrscr();
   num x;
   x.show();
   return 0;
}
```

OUTPUT

```
Constructor called
x = 5 a= 0 b= 1 c= 2
```

Explanation: In the above program, the class `num` is declared with four integers `a, b, c,` and `x`. The variables `a, b,` and `c` are `private` and `x` is a `public` variable. The class also has `show()` function and constructor prototype declaration. The function `show()` displays the contents of the member variables on the screen. The definition of a constructor is done outside the class. In the function `main()`, `x` is an object of class `num`. When an object is created, constructor is automatically invoked and member variables are initialized to given values as per the constructor definition. The values of variables `x, a, b,` and `c` are as per the output shown. The compiler calls the constructor for every object created. For each object the constructor is executed once; that is, the number of times the execution of constructor is equal to the number of objects created. The below-given program explains this point.

9.4 Write a program to show that for each object constructor is called separately.

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

class num
{
   private:
   int a;
   public:
   num (void) // definition of constructor
   {
   cout<<"\n Constructor called.";
   a=1;
   cout<<"a="<<a;
   }
};
main()
{
   clrscr();
   num x,y,z;
   num a[2];
   return 0;
}
```

OUTPUT

```
Constructor called. a = 1
Constructor called. a = 1
Constructor called. a = 1
Constructor called. a = 1
Constructor called. a = 1
```

Explanation: In the above program, `x, y,` and `z` are objects of class `num`. A[2] is an array of objects. For each individual object constructor is called. The total number of objects declared is five; hence, the constructor is called five times.

9.5 Write a program to read values through the keyboard. Use constructor.

```
#include<iostream.h>
#include<conio.h>
class num
{
   private:
   int a,b,c;
   public:
   num(void); // declaration of constructor
   void show()
   { cout<<"\n"<<"a=" <<a <<"b="<<b <<"c="<<c; }
};
num:: num (void) // definition of constructor
{
   cout<<"\n Constructor called";
   cout<<"\n Enter Values for a,b and c:";
   cin>> a>>b>>c;
}
main()
{
   clrscr();
   class num x;
   x.show();
   return 0;
}
```

OUTPUT

```
Constructor called
Enter Values for a,b and c: 1 5 4
a= 1 b= 5 c= 4
```

Explanation: The above program is same as the previous one. In this program, whenever an object is created, the constructor is called and it reads the integer values through the keyboard. Thus, the entered constants are assigned to member variables of the class. Here, the constructor is used like other functions.

9.5 CONSTRUCTORS WITH ARGUMENTS (PARAMETERIZED CONSTRUCTOR)

In the previous example, constructors initialize the member variables with given values. It is also possible to create constructor with arguments, and such constructors are called parameterized constructors. For such constructors, it is necessary to pass values to the constructor when an object is created. Consider the following example:

```
class num
{
   private:
   int a, b, c;
   public:
```

```
num (int m, int j, int k); // declaration of constructor
with arguments
   - - - - -
   - - ---
};
num:: num (int m, int j, int k) // definition of construc-
tor with arguments
{
   a=m;
   b=j;
   c=k;
}
main()
{
   class num x=num (4,5,7); // Explicit call
   class num y (9,5,7); // Implicit call
}
```

In the above example, the declaration and definition of a constructor contains three integer arguments. In the definition, the three arguments `m`, `j`, and `k` are assigned to member variables `a`, `b`, and `c`.

In a situation like this when class has a constructor with arguments, care is to be taken while creating objects. So far, we created the object using the following statement:

```
num x
```

This statement will not work. We need to pass required arguments as per the definition of constructor. Hence, to create object, the statements must be as given below:

```
(a) class num x=num (4,5,7); // Explicit call
(b) class num y (9,5,7); // Implicit call
```

The statements (a) and (b) can be used to create objects, which not only create objects but also pass given values to the constructor. The method (a) is called explicit call and the method (b) implicit call.

9.6 Write a program to create a constructor with arguments and pass the arguments to the constructor.

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

class num
{
   private:
```

```
   int a,b,c;
   public:
   num(int m, int j, int k); // declaration of constructor with
arguments
   void show()
   {
   cout<<"\na="<<a <<"b="<<b <<"c="<<c;
   }
};
num:: num (int m, int j, int k) // definition of constructor with
arguments
{
   a=m;
   b=j;
   c=k;
}
main()
{
   clrscr();
   num x=num(4,5,7); // Explicit call
   num y(1,2,8); // Implicit call
   x.show();
   y.show();
   return 0;
}
```

OUTPUT

```
a= 4 b= 5 c= 7
a= 1 b= 2 c= 8
```

Explanation: In the above program, `x` and `y` are objects of class `num`. When objects are created, three values are passed to the constructor. These values are assigned to the member variables. The function `show()` displays the contents of member variables.

9.6 OVERLOADING CONSTRUCTORS (MULTIPLE CONSTRUCTORS)

Similar to functions, it is also possible to overload constructors. In the previous examples, we declared single constructors without arguments and with all arguments. A class can contain more than one constructor. This is known as constructor overloading. All constructors are defined with the same name as the class they belong to. All the constructors contain different number of arguments. Depending upon the number of arguments, the compiler executes appropriate constructor. Table 9.1 describes object declaration and appropriate constructor to it. Consider the following example:

```
class num
{
   private:
   int a;
   float b;
   char c;
   public:
   a) num (int m, float j , char k);
   b) num (int m, float j);
   c) num();
   d) class num x (4,5.5,'A');
   e) class num y (1,2.2);
   f) class num z;
   }
```

Table 9.1 Overloaded Constructors

Constructor Declaration	Object Declaration
(a) `num (int m, float j, char k);`	(d) `num x (4,5.5,'A');`
(b) `num (int m, float j);`	(e) `num y (1,2.2);`
(c) `num();`	(f) `num z;`

In the above example, the statements **(a)**, **(b)**, and **(c)** are constructor declarations and **(d)**, **(e)**, and **(f)** are the object declarations. The compiler decides which constructor to be called depending on the number of arguments present with the object.

When object x is created, the constructor with three arguments is called because the declaration of an object is followed by three arguments. For object y, constructor with two arguments is called and lastly the object z, which is without any argument, is candidate for constructor without argument.

9.7 Write a program with multiple constructors for the single class.

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

class num
{
   private:
   int a;
   float b;
   char c;
   public:
   num(int m, float j , char k);
   num (int m, float j);
   num();
   void show()
   {
   cout<<"\n\ta="<<a<<"b="<<b<<"c="<<c;
   }
};
num:: num (int m, float j , char k)
{
   cout<<"\n Constructor with three arguments";
   a=m;
   b=j;
   c=k;
}
num:: num (int m, float j)
{
   cout<<"\n Constructor with two arguments";
   a=m;
   b=j;
   c=' ';
}
num:: num()
{
   cout<<"\n Constructor without arguments";
   a=b=c=NULL;
}
main()
{
   clrscr();
   class num x(4,5.5,'A');
   x.show();
   class num y(1,2.2);
   y.show();
   class num z;
   z.show();
   return 0;
}
```

OUTPUT

```
Constructor with three arguments
a= 4 b= 5.5 c= A
Constructor with two arguments
a= 1 b= 2.2 c=
Constructor without arguments
a= 0 b= 0 c=
```

Explanation: In the above program, three constructors are declared. The first constructor is with three arguments, second with two, and third without any argument. While creating objects, arguments are passed. Depending on number of arguments, compiler decides on the constructor to be called. In this program, `x`, `y`, and `z` are three objects created. The `x` object passes three arguments, the `y` object passes two arguments, and the `z` object passes no arguments. The function `show()` is used to display the contents of the class members.

9.8 Write a program to overload constructor and display date and time.

{coderipe}

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

class date_time
{
   int d,m,y,hrs,min,sec;
   public:
   date_time(int i,int j,int k,int a,int b,int c);
   date_time();
   void print()
   {
   cout<<"\nDate="<<d<<"Month="<<m<<"Year="<<y<<endl;
   cout<<"\n Hour="<<hrs<<"Minutes=" <<min<<"seconds="<<sec;
   }
};
date_time::date_time(int i,int j,int k,int a,int b,int c)
{
   d=i;
   m=j;
   y=k;
   hrs=a;
   min=b;
   sec=c;
}
date_time::date_time()
{
   cout<<"\nEnter Date Month Year (dd/mm/yy):";
   cin>>d>>m>>y;
   cout<<"\nEnter Hours Minutes Seconds (hr:min:sec):";
   cin>>hrs>>min>>sec;
}
int main()
{
   clrscr();
   date_time n(1,1,2012,14,45,21);
   n.print();
   cout<<"\n-----------------------------------------";
   date_time m;
   cout<<"\n-----------------------------------------";
   m.print();
   return 0;
}
```

OUTPUT

```
Date = 1 Month = 1 Year =2012
Hour = 14 Minutes = 45seconds =21
---------------------------------------------
Enter Date Month Year (dd/mm/yy):4 4 2012
Enter Hours Minutes Seconds (hr:min:sec):23 45 12
---------------------------------------------
Date = 4 Month = 4 Year =2012
Hour = 23 Minutes = 45seconds =12
```

Explanation: The program has two constructors. The first constructor has six parameters. Six arguments are passed through the object to the first constructor.

The second constructor takes the date and time details entered through keyboard.

9.7 ARRAY OF OBJECTS USING CONSTRUCTORS

An array is a collection of similar data types. We can also create an array of objects. The array elements are stored in contagious memory y locations. An array of objects is one of the most important data structures when data is itself an object. Program on it is illustrated using constructor.

9.9 Write a program to declare array of objects by using constructor.

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

class capital
{
   public:
   char name[20],cpt[20];
   capital();
};
capital::capital()
{
   cout<<"\nState and its capital:";
   cin>>name>>cpt;
}
int main()
{
   clrscr();
   capital c[3];
   return 0;
}
```

OUTPUT

```
State and its capital: Maharashtra Mumbai
State and its capital: Tamilnadu Chennai
State and its capital: Andhra-Pradesh Hyderabad
```

Explanation: The above program is generalized and can be extended to large number of objects. We have created an array of objects c[3] in the above program. The object c[3] initializes three names of the states and their capitals through keyboard, and the object c[3] calls the constructor three times.

9.8 CONSTRUCTORS WITH DEFAULT ARGUMENTS

Similar to functions, it is also possible to declare constructors with default arguments. Consider the following example:

```
power(int 9,int 3);
```

In the above example, the default value for the first argument is nine and two for second.

```
power p1 (3);
```

In this statement, object p1 is created and 9 raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated. Consider the following example on the above discussion:

9.10 Write a program to declare default arguments in a constructor. Obtain the power of the number.

{coderipe}

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class power
{
   private:
   int num;
   int power;
   int ans;
   public:
   power(int n=9,int p=3); // declaration of constructor with
default arguments
   void show()
   {

   cout<<"\n"<<num <<"raise to"<<power <<"is" <<ans;
   }
};
power:: power (int n,int p )
{
   num=n;
   power=p;
   ans=pow(n,p);
}
main()
{
   clrscr();
   class power p1,p2(5);
   p1.show();
   p2.show();
   return 0;
}
```

OUTPUT

```
9 raise to 3 is 729
5 raise to 3 is 125
```

Explanation: In the above program, the class `power` is declared. It has three integer member variables and one member function `show()`. The `show()` function is used to display the values of member data variables. The constructor of class `power` is declared with two default arguments. In the function `main()`, p1 and p2 are two objects of class power. The p1 object is created without argument. Hence, the constructor uses default arguments in `pow()` function. The p2 object is created with one argument. In this call of constructor, the second argument is taken as default. Both the results are shown in output.

9.9 COPY CONSTRUCTORS

The constructor can accept arguments of any data type including user-defined data types, exclusive of the class to which it belongs. Consider the following examples shown in Table 9.2.

Table 9.2 Copy Constructors

Statement (a)	Statement (b)

`class num`
`{`
`        private:`
`        ----`
`        ----`
`        ----`
`        public:`
`        num(num);`
`}`
	`class num`
	`{`
	`        private:`
	`        ----`
	`        ----`
	`        ----`
	`        public:`
	`        num(num&);`
	`}`

In the `example (a)`, an argument of the constructor is same as that of its class. Hence, this declaration is wrong. It is possible to pass reference of object to the constructor. Such declaration is known as copy constructor. The `example (b)` is valid and can be used to copy constructor.

When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using refer-

9.11 Write a program to pass an object with reference to constructor. Declare and initialize other objects.

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

class num
{
   int n;
   public:
   num() { } // constructor without argument
   num (int k) { n=k;} // constructor with one argument
   num (num &j) // copy constructor
   {
   n=j.n;
   }
   void show (void) {cout<<n;}
};
main()
{
   clrscr();
   num J(50);
   num K(J);
   num L=J;
   num M;
   M=J;
```

```
    cout<<"\n Object J Value of n:";
    J.show();
    cout<<"\n Object K Value of n:";
    K.show();
    cout<<"\n Object L Value of n:";
    L.show();
    cout<<"\n Object M Value of n:";
    M.show();
    return 0;
}
```

OUTPUT

```
Object J Value of n: 50
Object K Value of n: 50
Object L Value of n: 50
Object M Value of n: 50
```

ence of another object. Thus, whenever a constructor is called, a copy of an object is created.

Explanation: In the above program, class num is declared with one integer member variable n and three constructors. In function main(), the object J is created and 50 is passed to constructor. It is passed by value; hence, constructor with one argument is invoked. When object K is created with one object, the copy constructor is invoked, object is passed, and data member is initialized. The object L is created with assignment with object J; this time also copy constructor is invoked. The compiler copies all the members of object J to destination object L in the assignment statement num L = J. When object is created without any value, such as M, default constructor is invoked. The copy constructor is not invoked even if object J is assigned to object M.

The data member variables that are dynamically allocated should be copied to the target object explicitly by using assignment statement or by copy constructor as per the following statement:

```
num L=J; // copy constructor is executed
```

Consider the statement

```
M=J;
```

Here, M and J are predefined objects. In this statement, copy constructor is not executed. The member variables of object J are copied to object M member by member. An assignment statement assigns value of one entity to another.

The statement num L = J; initializes object L with J during definition. The member variables of J are copied member by member into object L. This statement invokes constructor. This statement can be written as num L (J), which we frequently use to pass values to the constructor.

9.10 THE const OBJECTS

In the previous chapter, we have studied the constant functions. The const declared functions do not allow the operations that alter the values. In the same fashion, we can also make the object constant by the keyword const. Only constructor can initialize data member variables of constant object. The data member of constant objects can be read-only and any effort to alter values of variables will generate an error. The data members of constant object are also called read-only data members. The constant object can access only constant functions. If constant object tries to invoke a non-member function, an error message will be displayed.

9.12 Write a program to declare constant object. Also, declare constant member function and display the contents of member variables.

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

class ABC
{
   int a;
   public:
   ABC (int m)
   { a=m; }
   void show() const
    { cout<<"a="<<a; }
};
int main()
{
   clrscr();
   const ABC x(5);
   x.show();
   return 0;
}
```

OUTPUT

```
A=5
```

Explanation: In the above program, class `ABC` is declared with one member variable and one constant member function `show()`. The constructor `ABC` is defined to initialize the member variable. The `show()` function is used to display the contents of member variable. In `main()`, the object `x` is declared as constant with one integer value. When object is created, the constructor is executed and value is assigned to data member. The object `x` invokes the member function `show()`.

9.11 DESTRUCTORS

The destructor is also a special member function like constructor. Destructors destroy the class objects created by constructors. The destructors have the same name as their class, preceded by a tilde (~).

For local and non-static objects, the destructor is executed when the object goes out of scope. In case the program is terminated by using `return` or `exit()` statements, the destructor is executed for every object existing at that time. It is not possible to define more than one destructor. The destructor is only one way to destroy the object. Hence, they cannot be overloaded.

A destructor neither requires any arguments nor returns any value. It is automatically called when object goes out of scope. Destructor releases memory space occupied by the objects.

The program given below explains the use of destructor.

9.13 Write a program to demonstrate execution of constructor and destructor.

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

struct text
{
   text() // Constructor
   {
   cout<<"\n Constructor executed.";
   }
   ~text() // Destructor
   {
   cout<<"\n Destructor executed.";
   }
};
void main()
{
   clrscr();
   text t; // Object declaration
}
```

OUTPUT

```
Constructor executed.
Destructor executed.
```

Explanation: In the above program, the class `text` contains constructor and destructor. In function `main()`, object `t` is executed. When object `t` is declared, constructor is executed. When object goes out of scope, destructor is executed.

9.14 Write a program to create an object and release them using destructors.

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

int c=0; // counter for counting objects created and de-
stroyed.
class num
```

```
{
   public:
   num()
   {
   c++;
   cout<<"\n Object Created: Object("<<c <<")";
   }
   ~num()
   {
   cout<<"\n Object Released: Object("<<c <<")";
   c--;
   }
};
main()
{
   clrscr();
   cout<<"\n In main() \n";
   num a,b;
   cout<<"\n\n In Block A\n";
   {
   class num c;
   }
   cout<<"\n\n Again In main()\n";
   return 0;
}
```

OUTPUT

```
In main()
Object Created: Object(1)
Object Created: Object(2)
In Block A
Object Created: Object(3)
Object Released: Object(3)
Again In main()
Object Released: Object(2)
Object Released: Object(1)
```

Explanation: In the above program, the variable `c` is initialized with zero. The variable `c` is incremented in constructor and decremented in destructor. When objects are created, the compiler calls the constructor. Objects are destroyed or released when destructor is called. Thus, the value of a variable changes as the constructors and destructors are called. The value of a variable of `c` shows number of objects created and destroyed. In this program, the objects `a` and `b` are created in `main()`. The object `c` is created in another block, that is, in block A. The object `c` is created and destroyed in the same block. The objects are local to the block in which they are defined. The objects a and b are released as the control passes to main() block. The object created last is released first.

9.12 CALLING CONSTRUCTORS AND DESTRUCTORS

The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructors and destructors. In practice, it may not be useful; but for the sake of understanding, few examples are illustrated on this concept.

9.15 Write a program to invoke constructor and destructor.

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

class byte
{
   int bit;
   int bytes;
   public:
   byte()
   {
   cout<<"\n Constructor invoked";
   bit=64;
   bytes=bit/8;
   }
   ~byte()
   {
   cout<<"\n Destructor invoked";
   cout<<"\n Bit="<<bit;
   cout<<"\n Byte="<<bytes;
   }
};
int main()
{
   clrscr();
   byte x;
   byte(); // calling constructor
   // x.byte() ; // invalid statement
   // x.byte::byte() // valid statement
   // ~ byte(); // invalid statement
   // byte.~byte(); // invalid statement
   // x.~byte(); // Member identifier expected
   x.byte::~byte();
   return 0;
}
```

OUTPUT

```
Constructor invoked
Constructor invoked
Destructor invoked
Bit = 64
Byte = 8
Destructor invoked
Bit = 64
Byte = 8
Destructor invoked
Bit = 64
Byte = 8
```

Explanation: In the above program, the class `byte` is declared with two integer members. The class has constructors and destructors. In the function `main()`, `x` is an object of class `byte`. When an object is created, the compiler automatically invokes the constructor. The program also invokes constructor. Thus, constructor is executed two times, that is, implicitly and explicitly. The constructor is called without the help of object of that class. If we try to call the constructor with the help of object `x` as per the statement `x.byte()`, the compiler displays an error message.

The destructor cannot be invoked like constructor. Calling destructor requires the help of object and class name. Only object of the class alone is not able to invoke the destructor, the object needs the help of the class. The statement `x.~byte()` is invalid, the compiler displays the error message Member identifier expected. The destructor can be invoked using the statement `x.byte::~byte()`. Here, in addition, class name is also specified.

The constructor and the destructor can call each other. The statement `byte::byte()` is used for calling destructor `~byte()` within the body of constructor `byte()`. The destructor `~byte()` can call the constructor `byte()` using the statement `byte()`. The constructor and the destructor may contain condition statements; such destructors or constructors are called conditional destructor or conditional constructor, respectively. The conditional constructor can be created using `if-else` or `switch()` statements. The example given below illustrates this.

9.16 Write a program to define conditional constructor and destructor.

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

class byte
{
   int bit;
   int bytes;
   public:
   ~byte(); // Prototype declaration
   byte()
   { cout<<"\n Constructor invoked";
   bit=64;
   bytes=bit/8;
   byte:: ~byte(); // call to destructor
   }
};
byte::~byte()
{
   if (c==0)
   {
   // byte() // call to constructor
   cout<<"\n Destructor invoked";
   cout<<"\n Bit="<<bit;
   cout<<"\n Byte="<<bytes;
   }
   c++;
}
int main()
{ clrscr();
   byte x,y;
   return 0;
}
```

OUTPUT

```
Constructor invoked
Destructor invoked
Bit = 64
Byte = 8
Constructor invoked
```

Explanation: In the above program, `x` and `y` are objects of class `byte`. The compiler automatically invokes the constructor. The constructor invokes the destructor. The variable `c` is declared and initialized with zero. It is a global variable declared before `main()` and can be used anywhere in the program. The destructor contains conditional statement `if`. The `if` block is executed only when `c` is zero. When destructor is called the first time, the `if` block is executed and `c` is turned to a non-zero value. Next time when destructor is called, the `if` condition evaluates false and the `if` block is not executed. Though destructor is called, no result is displayed on the screen. Remember that the constructors and destructors are executed for equal number of times. It is possible for the programmer to use the constructor and the destructor like other user-defined functions.

9.13 QUALIFIER AND NESTED CLASSES

The class declaration can also be done inside the other class. While declaring object of such class, it is necessary to precede the name of the outer class. The name of outer class is called qualifier name, and the class defined inside is called nested class.

The use of nested classes enhances the power of abstract data type and helps to construct more powerful data structure. The qualifier (host class) and the nested class follow the same access conditions. The nested classes are useful to hide a particular class and its objects within a qualifier class. The following program illustrates this.

9.17 Write a program to declare classes within classes and access the variables of both the classes.

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

class A // Host class or qualifier class
{
   public:
   int x;
   A() { x=10; }
   class B // Nested class
   {
   public:
   int y;
   B() { y=20; }
   void show()
   {
   A a;
   cout<<"x="<<a.x<<endl;
   cout<<"y="<<y<<endl;
   }
   }; // End of nested (inner) class
}; // End of qualifier (outer) class
void main()
{
   clrscr();
   A::B b;
   b.show();
}
```

OUTPUT

```
x=10
y=20
```

Explanation: In the above program, class B is declared inside class A. All the members of both the classes are declared public so that they can be easily accessed. Both the classes have constructors to initialize the member variables. The class B also has one member function `show()` to display the contents on the screen.

In function `main()`, `A::B b` declares object of class B. The name of class A is preceded because the class B is inside the class A. If we declare an object using the statement `B b,` the program will be executed with a warning message `"Use qualified name to access nested type 'A::B'"`. Here, the class A acts as a qualified class.

The `show()` function is a member of class `B.` Due to this, variable `y` is its member variable and can be directly accessed. To access the variable of a qualified class A, object of class `A a` is declared and through this object variable `x` is accessed. The data variables are public. In case of private, we can access member variables using member functions. There is no limit for declaring classes within classes. The following program explains this.

9.18 Write a program to declare multiple qualifier classes and declare an object of every class.

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

class A
{
   public:
   int x;
   A() { x=5;
   cout<<"x="<<x;
}
class B
{
   public:
   int y;
   B()
   {
   y=10;
   cout<<"y="<<y;
   }
   class C
   {
   public:
   int z;
   C()
   {
   z=15;
   cout<<"z="<<z;
   }
   };
};
};
void main()
{
clrscr();
A a; // outer class object
A::B b; // middle class object
A::B::C c; // inner class object
}
```

OUTPUT

```
x= 5 y =10 z =15
```

Explanation: In the above program, classes `A, B,` and `C` are declared. The class `B` is declared inside the class `A`. The class `C` is declared inside the class `B`. The classes `A` and `B` are qualified classes for class `C`. The class `A` is qualified class for class `B`. Each class has a constructor that initializes and displays the contents of the object.

9.14 ANONYMOUS OBJECTS

Objects are created with names. It is possible to declare objects without name. Such objects are known as `anonymous objects`. We learnt how to invoke constructors and destructors. When constructors and destructors are invoked, the data members of the class are initialized and destroyed, respectively. Thus, without object we can initialize and destroy the contents of the class. All these operations are carried out without object or we can also assume that the operations are carried out using an anonymous object, which exists but is hidden. The second thought is correct because if we apply pointer `this` (explained in Chapter 13 "Pointers and arrays"), we get the address of the object. It is not possible to invoke member function without using object but we can invoke the special member functions `constructors` and `destructors` which compulsorily exist in every class. Thus, without the use of `constructor` and `destructor`, the theory of `anonymous object` cannot be implemented in practice. Consider the following example:

9.19 Write a program to create an anonymous object. Initialize and display the contents of member variables.

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

class noname
{
   private:
   int x;
   public:
   noname (int j)
   {
   cout<<"\n In constructor";
   x=j;
   show();
   }
   noname()
   {
   cout<<"\n In constructor";
   x=15;
   show();
   }
   ~noname() { cout<<"\n In destructor"; }
   noname *const show()
   {
   cout<<endl<<"x:"<<x;
   return this;
   }
};
void main()
{
   clrscr();
   noname();
   noname(12);
}
```

OUTPUT

```
In constructor
x: 15
In destructor
In constructor
x: 12
In destructor
```

Explanation: In the above program, class `noname` is declared with one integer data member. The class also has constructor, destructor, and member function. The member function `show()` is used to display the contents on the screen.

In function `main()`, no object is created. The constructor is called directly. Calling constructor directly implies creating anonymous objects. In the first call of the constructor, the member variable is initialized with 15. The constructor invokes the member function `show()` that displays the value of `x` on the screen.

It is not possible to create more than one anonymous object at a time. When constructor execution work is over destructor destroys the object. Again, the constructor with one argument is called and integer is passed to it. The member variable is initialized with 12 and it is displayed by `show()` function. Finally, the destructor is executed that marks the end of program.

☞ **TIP**

Calling constructor directly means creating anonymous objects.

9.15 PRIVATE CONSTRUCTORS AND DESTRUCTORS

We learnt that when a function is declared as private, it could be invoked by the public function of the same class. So far, we declared constructors and destructors as public. The constructor and destructor can be declared as private. The constructor and destructor are automatically executed when an object is executed and when an object goes out of scope. Nevertheless, when the constructor and destructor are private, they cannot be executed implicitly, and hence, it is a must to execute them explicitly. The following program is illustrated concerning this.

9.20 Write a program to declare constructor and destructor as private and call them explicitly.

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

class A
{
   private:
   int x;
   ~A() { cout<<"\n In destructor ~A()"; }
   A() {x=7; cout<<"\n In constructor A()"; }
   public:
   void show()
   {
   this->A::A(); // invokes constructor
   cout<<endl<<"x="<<x;
   this->A::~A(); // invoke destructor
   }
};
void main()
{
   clrscr();
   A *a;
   a->show();
}
```

OUTPUT

```
In constructor A()
x = 7
In destructor ~A()
```

Explanation: In the above program, class `A` is declared. It has one data member, constructor, and destructor. The function `show()` is declared in public section. In function `main()`, `a` is a pointer to class `A`. When pointer object is created, no constructor is executed. When `show()` function is invoked, the constructor is invoked and its return value is assigned to pointer `*this`. (The pointer `*this` is explained in Chapter 13 "Pointers and arrays.") Every member function holds `this` pointer that points to the calling object. Here, the pointer points to object `a`. When the pointer invokes the constructor, constructor is invoked. Consider the statement `this = this->A::A()`. The statement invokes the zero-argument constructors. The contents are displayed by the `cout` statement and again the `this` pointer invokes the destructor.

9.16 DYNAMIC INITIALIZATION USING CONSTRUCTORS

After declaration of the class data member variables, they can be initialized at the time of program execution using pointers. Such initialization of data is called dynamic initialization. The benefit of dynamic initialization is that it allows different initialization modes using overloaded constructors. Pointer variables are used as argument for constructors. The following example explains dynamic initialization using overloaded constructor.

9.21 Write a program to initialize member variables using pointers and constructors.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class city
{
   char city[20];
   char state[20];
   char country[20];
   public:
   city() { city[0]=state[0]=country[0]=NULL; }
   void display( char *line);
   city(char *cityn)
   {
   strcpy(city, cityn);
   state[0]=NULL;
   }
   city(char *cityn,char *staten)
   {
   strcpy(city,cityn);
   strcpy(state,staten);
   country[0]=NULL;
   }
   city(char *cityn,char *staten, char *countryn)
   {
   _fstrcpy(city,cityn);
   _fstrcpy(state,staten);
   _fstrcpy(country,countryn);
   }
};
void city:: display (char *line)
{
   cout<<line<<endl;
   if (_fstrlen(city)) cout<<"City:"<<city<<endl;
   if (strlen(state)) cout<<"State:"<<state <<endl;
   if (strlen(country)) cout<<"Country:"<<country <<endl;
}
void main()
{
   clrscr();
   city c1("Mumbai"),
   c2("Nagpur","Maharashtra"),
   c3("Nanded","Maharashtra","India"),
   c4('\0','\0','\0');
   c1.display("=========*=============");
   c2.display("=========*=============");
   c3.display("=========*=============");
   c4.display("=========*=============");
}
```

OUTPUT

```
=========*=============
City: Mumbai
=========*=============
City: Nagpur
State: Maharashtra
=========*=============
City: Nanded
State: Maharashtra
Country: India
=========*=============
```

Explanation: In the above program, the class `city` is declared with three-character arrays: `city`, `state`, and `country`. The class `city` also has four constructors: zero-argument constructor, one-argument constructor, two-argument constructor, and three-argument constructor. The `display()` member function is used to display the contents on the screen.

In function `main(), c1, c2, c3,` and `c4` are declared and strings are passed. According to the number of arguments, respective constructor is executed. The function `display()` is called by all the four objects and information displayed is as shown in the output. While calling function `display()`, the line format "=====" is passed that is displayed before displaying the information. The `display()` function also contains if statements that check the length of the string and display the string only when the variable contains string. In case the data variable contains NULL, the string will not be displayed. The object `c4` is initialized with NULL character. The use of `c4` object is only to display a line at the end of a program.

9.17 DYNAMIC OPERATORS AND CONSTRUCTORS

When constructor and destructor are executed they internally use `new` and `delete` operators to allocate and de-allocate memory. Dynamic construction means allocation of memory by constructor for objects, and dynamic destruction means releasing memory using the destructor. Consider the following program that explains the use of `new` and `delete` operators with constructor and destructor.

9.22 Write a program to use `new` and `delete` operators with constructor and destructor. Allocate memory for the given number of integers. Also release the memory.

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

int number (void);
class num
{
   int *x;
   int s;
   public:
   num()
   {
   s=number();
   x= new int [s];
   }
   ~num() { delete x; }
   void input();
   void sum();
};
void num:: input()
{
   for ( int h=0;h<s;h++)
   {
   cout<<"Enter number ["<<h+1<<"]:";
   cin>>x[h];
   }
}
void num:: sum()
{
   int adi=0;
   for ( int h=0;h<s;h++)
   adi+=x[h];
   cout<<"sum of elements="<<adi;
}
number()
{
   clrscr();
   int n;
   cout<<"How many numbers:";
   cin>>n;
   return n;
}
void main()
{
   num n1;
   n1.input();
   n1.sum();
}
```

OUTPUT

```
How many numbers: 3
Enter number [1]: 1
Enter number [2]: 4
Enter number [3]: 5
sum of elements = 10
```

Explanation: In the above program, class `num` is declared with two variables and they are integer pointer `*x` and integer variable `s`. The class also contains constructor and destructor. The responsibility of constructor is to allocate memory using `new` operator for a given number of integers by the user. The non-member function `number()` is used to input number of elements through the keyboard and the entered numbers are stored in the variable `s`. The variable `s` is used with the `new` operator to allocate memory. The memory is allocated to the pointer `x`.

The member function `input()` reads elements and the `for` loop repeats the statement `cin>>x[h]` for `s` (number of total elements) times. The `x[h]` is used like array. The `x` is the starting address and `h` indicates the successive locations. The address of pointer `x` remains unchanged. The value of h is added to value of x. In this way, all these numbers are stored in continuous memory locations. If you are still confused how successive memory locations are accessed, go through the pointer arithmetic operations.

The sum() function is used to perform addition of all the entered numbers and displays the same on the screen. The destructor is finally executed which releases the memory using delete operator.

☞ **TIP**

In the above program, if we remove the operators new and delete, the program will work successfully. However, this is suitable for small programs, developed for demonstration purpose. In real application, it needs large amount of memory to be allocated or released. Hence, it is very essential to check the memory before doing the process. For example, when any program is loaded in the memory, the operating system checks for available system resources that are sufficient enough to load the requested application by the user. In such a case, the operating system checks memory resources and if memory is insufficient, the compiler displays the same message to the user.

9.18 main() AS A CONSTRUCTOR AND DESTRUCTOR

We learnt that the constructor and destructor have the same name as their class. To use main() as a constructor and destructor, we need to define class with the name main. So far, we have declared object of classes without using the keyword class or struct because C++ treats classes like built-in data types. The use of keyword class or struct is only compulsory when the class name is main. This is because execution of program starts with the function main(). The following program clears this point.

9.23 Write a program to declare class with name main.

```
#include<conio.h>
#include<iostream.h>
class main
{
   public:
   main() { cout<<"\n In constructor main()"; }
   ~main() { cout<<"\n In destructor main()"; }
};
void main()
{
   clrscr();
   class main a;
}
```

OUTPUT

```
In constructor main()
In destructor main()
```

Explanation: In the above program, the class is declared with the name main. Hence, it is compulsory to use keyword `class` or `struct` while declaring objects.

☞ **TIP**

When the class name is main, it is compulsory to use keyword class or struct to declare objects.

9.19 RECURSIVE CONSTRUCTORS

Similar to normal and member functions, constructors also support recursion. The following program explains this.

9.24 Write a program to invoke constructor recursively and calculate the triangular number of the entered number.

{coderipe}

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

class tri_num
{
   int f;
   public:
   tri_num() { f=0; }
   void sum( int j) { f=f+j; }
   tri_num(int m)
   {
   if (m==0)
   {
   cout<<"Triangular number:"<<f;
   exit(1);
   }
   sum(m);
   tri_num::tri_num(--m);
   }
};
void main()
{
   clrscr();
   tri_num a;
   a.tri_num::tri_num(5);
}
```

OUTPUT

Triangular number: 15

Explanation: In the above program, class tri_num is declared and with private integer f and member function sum(). The class also has zero-argument constructor and one-argument constructor.

In function main(), a is an object of tri_num class. When object a is declared, the zero-argument constructor is executed. The object a invokes the constructor explicitly and passes integer value to it. The passed value is received by the variable m of the constructor. The if statement checks the value of variable m and if it is zero, the triangular number is displayed and the program is terminated.

The sum() function is called by the constructor and the cumulative sum is calculated and stored in the member variable f. Followed by the call of function sum(), the constructor tri_num() is executed and the value of m is decreased by one. Thus, the recursion takes place and each time value of m decreases. When the value of m becomes zero, as mentioned above, the program terminates and we get the triangular number.

☞ **TIP**

Recursion is not difficult, though somewhat confusing. If you are still confused about recursion concept, execute the program in a single step to know the flow of the program. First, try the recursion with normal function in C style and after perfect understanding try with member function and constructors, but do not drop it.

9.20 PROGRAM EXECUTION BEFORE main()

It is impossible to execute a program without main(). The declaration of objects is allowed before main(). We know that when an object is created, constructor and destructor are executed. The constructor can call other member functions. When an object is declared before main(), it is called as global object. For global object, constructor is executed before main() function and destructor is executed after the completion of execution main() function. The following program illustrates this.

9.25 Write a program to declare global object and observe execution of constructor.

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

struct text
{
   text() // Constructor
   {
   cout<<"\n Constructor executed.";
   }
   ~text() // Destructor
   {
   cout<<"\n Destructor executed.";
   }
};
text t; // Global object
void main()
{ }
```

OUTPUT

Constructor executed.
Destructor executed.

Explanation: In the above program, object `t` is declared before `main()`. It is a global object. As soon as an object is declared, its constructor is invoked immediately. However, the execution of every program starts from function `main()`, but for the object declared before `main()`, constructor is executed before execution of `main()`. The destructor for such object is executed after the complete execution of function `main()`. The constructor can invoke other member functions. These member functions are also executed before `main()`. The following program explains this.

9.26 Write a program to declare object before `main()` and invoke member function.

```
#include<iostream.h>
#include<conio.h>
class A
{
   private:
   char *y;
   int x;
   public:
   A()
   {
   clrscr();
   cout<<"\n In constructor ";
   x=15;
   show(); // invokes member function
   }
   void show() { cout<<endl<<"In show() x="<<x; }
   ~A() { cout<<endl<<"In destructor"; }
};
A a;
void main() { }
```

OUTPUT

In constructor
In show() x=15
In destructor

Explanation: In the above program, class `A` is declared with member variable character pointer and integer `x`. The class also has constructor, destructor, and member function `show()`. Before `main()`, object `a` is declared and constructor is executed that initializes the member variables. The constructor invokes the member function `show()`. Finally, the destructor is executed.

9.21 CONSTRUCTOR AND DESTRUCTOR WITH STATIC MEMBERS

Every object has its own set of data members. When a member function is invoked, only copy of data member of calling object is available to the function. Sometimes, it is necessary for all the objects to share fields which are common for all the objects. If the member variable is declared as static, only one copy of such member is created for entire class. All objects access the same copy of static variable. The static member variable can be used to count the number of objects declared for a particular class. The following program helps you to count the number of objects declared for a class.

9.27 Write a program to declare a static member variable. Count the number of objects created and destroyed.

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

class man
{
   static int no;
   char name;
   int age;
   public:
   man()
   {
   no++;
   cout<<"\n Number of Objects exists: "<<no;
   }
   ~ man()
   {
   --no;
   cout<<"\n Number of objets exists:"<<no;
   }
};
int man:: no=0;
void main()
{
   clrscr();
   man A,B,C;
   cout<<"\n Press any key to destroy object";
   getch();
}
```

OUTPUT

```
Number of Objects exists: 1
Number of Objects exists: 2
Number of Objects exists: 3
Press any key to destroy object
Number of objects exists: 2
Number of objects exists: 1
Number of objects exists: 0
```

Explanation: In this program, the class man has one static data member no. The static data member is initialized to zero. Only one copy of static data member is created and all objects share the same copy of static data member.

In function `main()`, objects `A, B,` and `C` are declared. When objects are declared, constructor is executed and static data member `no` is increased with one. The constructor also displays the value of `no` on the screen. The value of static member shows us the number of objects present. When the user presses a key, destructor is executed, which destroys the object. The value of static variable is decreased in the destructor. The value of static member variable shows the number of existing objects.

9.22 LOCAL VERSUS GLOBAL OBJECT

The object declared outside all function bodies is known as global object. All functions can access the global object. The object declared inside a function body is known as local object. The scope of local object is limited to its current block. We learnt the behavior of constructor and destructor with local and global object.

When global and local variables are declared with the same name, the scope access operator is used to access the global variable in the current scope of local variable. We know that the local variable gets first precedence than the global variable. The same is applicable for objects. When a program contains global and local objects with the same name, the local object gets first preference in its own block. In such a case, scope access operator is used with global object. The following program describes this.

9.28 Write a program to show difference between local and global object.

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

class text
{
   public:
```

```
   void show (char *c)
   {
   cout<<"\n"<<c;
   }
};
text t; // global object declaration
void main()
{
   text t; // local object declaration
   ::t.show("Global"); // call using global object
   t.show("Local"); // call using local object
}
```

OUTPUT

```
Global
Local
```

Explanation: In the above program, the object `t` is declared in local as well as global scope. In function `main()`, using scope access operator and `t`, the function `show()` is invoked. The first time function `show()` is invoked using global object. The second call to function `show()` is made by local object. The local object does not require scope access operator. The scope access operator is used only when the same name for object is used in global and local scopes. If scope access operator is not used before object, for both times function `show()` is invoked by local object.

9.23 MORE PROGRAMS

9.29 Write a program to pass arguments to constructor. Define conditional constructor.

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

struct num
{
   num (int,char);
   ~num()
    { cout<<"\n Destructor invoked"; }
};
num::num (int s,char b)
{
   switch(b)
   {
   case 'R':
   cout<<"\n Square Root of"<<s <<"is" <<sqrt(s);
   break;
   case 'S':
   cout<<"\n Square of"<<s <<"is" <<pow(s,2);
   break;
   case 'C':
   cout<<"\n Cube of"<<s <<"is" <<pow(s,3);
   break;
   default:
   cout<<"\n Invalid choice";
   }
}
int main()
{
   clrscr();
   int u;
   char o;
   cout<<"Enter a number:";
   cin>>u;
   cout<<"Enter operation (S)quare,(C)ube,Square (R)oot";
   cin>>o;
   num x(u,o);
   return 0;
}
```

OUTPUT

```
Enter a number: 64
Enter operation (S)quare,(C)ube,Square (R)oot R
Square Root of 64 is 8
Destructor invoked
```

Explanation: In the above program, `struct num` is declared. The structure is without any data member. The `struct` contains `constructor` and `destructor`. In function `main()`, `x` is an object. Before creating object `x`, an integer and a character are entered through the keyboard in variables `u` and `o`, respectively. The variables `u` and `o` are passed to the `constructor`. The `switch` case statement within the constructor selects appropriate operation depending upon the value of variable `o`.

9.30 Write a program to copy constructor. Change the values of an object by calling the constructor.

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

class data
{
   int u;
   char c;
   float f;
   public:
   ~data() { };
   data() {u=c=f=NULL; };
   data (int a,char b, float h) { u=a,c=b,f=h;}
   data (data &k)
   {
   u=k.u;
   c=k.c;
   f=k.f;
   }
   void show()
    { cout<<"\n u="<<u <<"c="<<(char)c <<"f="<<f; }
};
void main()
{
   clrscr();
   data k(5,68,8.2);
   k.show();
   data l(k);
   l.show();
   data z;
   z.show();
   z=data(1,70,5.8);
   z.show();
}
```

OUTPUT

```
u = 5 c = D f= 8.2
u = 5 c = D f= 8.2
u = 0 c = f= 0
u = 1 c = F f= 5.8
```

Explanation: In the above program, the class `data` is declared with three data member variables and one member function `show()`. The class `data` also contains destructor and overloaded constructors. For explanation of this program, please refer Program 5.7. In this program, the object `z` is created and no arguments are passed. In such a case, the constructor with no arguments is executed and the entire data members are initialized to NULL. The statement `z = data (1,70,5.8)` explicitly calls the constructor and the object `z` is initialized. The result of the program is as given above.

9.31 Write a program to call the constructor without using object.

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

class A
{
   public:
   char c;
   int i;
   float f;
   A()
   {
   cout<<"\n Enter char, int and float value";
   cin>>c>>i>>f;
   out(c,i,f);
   }
   void out (char l,int k, float j )
   {
   cout<<"\n char value="<<l;
   cout<<"\n int value="<<k;
   cout<<"\n float value="<< j;
   cout<<"\n";
   }
};
int main()
{
   clrscr();
   A()=A()=A();
   return 0;
}
```

OUTPUT

```
Enter char, int and float value A 65 6.5
char value = A
```

```
int value = 65
float value = 6.5
Enter char, int and float value B 66 6.6
char value = B
int value = 66
float value = 6.6
Enter char, int and float value C 67 6.7
char value = C
int value = 67
float value = 6.7
```

Explanation: In the above program, class `A` has three member variable `c`, `i`, and `f` of `char`, `int`, and `float` type. The class `A` also one-member function `out()` and a constructor. The constructor invokes the `out()` function to output the contents on the screen. In function `main()`, no object is declared. The constructor is called directly.

9.32 Write a program to demonstrate the use of copy constructor.

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

class data
{
   private:
   int x;
   float y;
   public:
   data() { }
   data (int xx, float yy) { x=xx; y=yy; }
   data operator=(data & d)
   {
   cout<<endl<<"Assignment operator executed";
   x=d.x;
   y=d.y;
   return data (x,y);
   }
   data (data &d)
   {
   cout<<endl<<"copy constructor executed";
   x=d.x;
   y=d.y;
   }
   void show() { cout<<"X="<<x<<"Y="<<y; }
};
void main()
{
   clrscr();
   data d1(12,5.8);
   data d2,d4;
   d4=d2=d1;
   data d3=d1;
   cout<<"\n Object d1:";
   d1.show();
   cout<<"\n Object d2:";
   d2.show();
   cout<<"\n Object d3:";
   d3.show();
   cout<<"\n Object d4:";
   d4.show();
}
```

OUTPUT

```
Assignment operator executed
Assignment operator executed
copy constructor executed
Object d1: X=12 Y=5.8
Object d2: X=12 Y=5.8
Object d3: X=12 Y=5.8
Object d4: X=12 Y=5.8
```

Explanation: In the above program, `d1, d2, d3,` and `d4` are objects of the class data. The statement `d4 = d2 = d1` invokes the overloaded operator = and assignment is carried out. The statement data `d3 = d1` executes the copy constructor. The operator `= ()` returns an object of class data type and initializes it using three-parameter constructor. The above approach creates an extra copy of the object that occupies extra memory space. To overcome this problem, we can use copy constructor method in which no new object is created.

9.33 Write a program to invoke constructors of nested classes.

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

class A
{
   public:
   int x;
   A() { x=5;
   cout<<endl<<"In constructor A x="<<x;
}
class B
{
   public:
   int y;
   B()
   {
   y=10;
   cout<<endl<<"In constructor B y="<<y;
   }
   class C
   {
   public:
   int z;
   C()
   {
   z=15;
   cout<<endl<<"In constructor C z="<<z;
   }
   };
};
};
void main()
{
   clrscr();
   A();
   A::B();
   A::B::C();
}
```

OUTPUT

```
In constructor A x= 5
In constructor B y =10
In constructor C z =15
```

Explanation: In the above program, `classes A, B,` and `C` are declared. The `class B` is declared inside the `class A`. The `class C` is declared inside the `class B`. `A` and `B` are qualified classes for `class C`. The `class A` is qualified class for `class B`. Each class has a constructor that initializes and displays the contents of the object.

SUMMARY

(1) C++ provides a pair of in-built functions called `constructor` and `destructor`. The compiler automatically executes these functions. When an object is created, constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. It is executed at the end of program when objects are of no use.

(2) Constructors and destructors decide how the objects of a class are created, initialized, copied, and destroyed. They are member functions. Their names are distinguished from all other member functions because they have the same name as the class they belong to.

(3) It is also possible to create constructor with arguments as in normal functions.

(4) Similar to functions, it is also possible to overload constructors and assign default arguments.

(5) When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using reference of another object. Thus, whenever a constructor is called, a copy of an object is created.

(6) We can also make the object constant by the keyword `const`. Any effort to alter values of variables made by it will generate an error. The constant object can access only constant functions.

(7) The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructor and destructor.

(8) Objects are created with names. It is possible to declare objects without name. Such objects are known as `anonymous objects`.

(9) When the constructor and destructor are private, they cannot be executed implicitly, and hence, it is a must to execute them explicitly.

(10) We can also call the constructor and destructor in the same fashion as we call the normal user defined function.

(11) The class declaration can also be done inside the other class. While declaring object of such class, it is necessary to precede the name of the outer class. The name of outer class is called qualifier name, and the class defined inside is called as nested class.

(12) The dynamic construction means allocation of memory by constructor for objects and dynamic destruction means releasing memory using the destructor.

(13) To use `main()` as a constructor and destructor, we need to define class with the name `main`.

(14) Similar to normal and member functions, constructors also support recursion.

(15) For global object, constructor is executed before `main()` function and destructor is executed after the completion of execution `main()` function.

EXERCISES

(A) Answer the following questions

(1) What are constructors and destructors?

(2) Explain the characteristics of constructors and destructors.

(3) Explain constructors with arguments. How are arguments passed to the constructor?

(4) What do you mean by overloading of constructors? How it benefits the programmer?
(5) Explain constructor with default arguments.
(6) What is copy constructor?
(7) What are constant objects? How are they declared?
(8) How are constructors and destructors called explicitly?
(9) What is the difference between calling methods for constructors and destructors?
(10) Is it possible for a constructor and destructor to call each other?
(11) What are conditional constructors and destructors?
(12) What is anonymous object?
(13) What are nested and qualifier classes?
(14) What is the difference between local object and global object?
(15) What is static object? How it is different from normal object?
(16) How are private constructors and destructors executed?
(17) How will you declare constant object?
(18) What is default constructor?
(19) What is parameterized constructor?
(20) Explain the negative aspect of static objects.

(B) Answer the following by selecting the appropriate option

(1) Constructor and destructor are automatically invoked by
(a) compiler
(b) operating system
(c) `main()` function
(d) object

(2) Constructor is executed when
(a) object is declared
(b) object is destroyed
(c) both (a) and (b)
(d) none of the above

(3) The destructor is executed when
(a) object goes out of scope
(b) object is not used
(c) object contains nothing
(d) none of the above

(4) When memory allocation is essential, the constructor makes implicit call to
(a) new operator
(b) `malloc()`
(c) `memset()`
(d) random access memory

(5) Destructors can be
(a) overloaded
(b) of any data type
(c) able to return result
(d) explicitly called

(6) Constructors have the same name as
(a) the class they belong to
(b) the current program file name
(c) class name and preceded by ~
(d) both (a) and (c)

(7) The following program displays

```
#include<iostream.h>
class A
{
   int x;
   public:
   A() {x=10;}
   ~A() {}
};
void main()
{
   A a;
   cout<<(unsigned)
   a.A::A();
}
```

(a) address
(b) value
(c) both (a) and (b)
(d) none of the above

(8) The following program returns address of

```
#include<iostream.h>
class A
{public:
   A(){} ~A(){} A *dis-
   play()
   {return (&A());}
};
void main()
{  void *p;
   clrscr();
```

```
        A a;
        p=a.display();
        cout<<p;
    }
```

(a) constructor
(b) `display()`
(c) first element if any
(d) none of the above

(C) Attempt the following programs

(1) Write a program to declare a class with private data members. Accept data through constructor and display the data with destructor.

(2) Write a program to pass an object to constructor and carry out copy constructor. Display contents of all the objects.

(3) Write a program to declare a class with three data members. Declare overloaded constructors with no arguments, one argument, two arguments, and three arguments. Pass values in the object declaration statement. Create four objects and pass values in such a way that the entire four constructors are executed one by one. Write appropriate messages in constructor and destructor so that the execution of program can be understood.

(4) Write a program to declare a class with two data members. Also, declare and define member function to display the content of the class data members. Create object A. Display the contents of object A. Again initialize the object A using explicit call to constructor. This time pass appropriates values to constructor. Display the contents of object A using member function.

(5) Write a program to call constructor recursively. Calculate factorial of a given number.

(6) Write a program to create an array of strings. Read and display the strings using constructor and destructor. Do not use member functions.

(7) Write a program to create object without name.

(D) Find the bugs in the following programs

(1)
```
class text
{ text() { cout<<"Start"; }
~text() { cout<<"\n End"; }
};
void main() { text t; }
```

(2)
```
struct text
  { public:

  text (char *c)

  { cout<<"\n"<<c; }

  };
void main() { text t; }
```

(3)
```
class text
  { char *c;

  public:

  text ( *c) { cout<<"\n"<<c;
  }

  };
  void main() { text t("I
  WON"); }
```