Unit 5 Constructors and Destructors

Structure:

5.1 Introduction

In the previous unit you studied classes, objects and procedure to define them. You also learnt the abstract classes and friend functions and their use in C++. In this unit you study constructors and destructors. There are several special C++ member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions – you declare and define them within the class, or declare them within the class and define them outside - but they have some unique features. In this unit we will discuss the role of constructor and overloading constructor. We will also focus on destructor with its role to de allocate the objects which were created by the constructor.

Objectives:

After studying this unit you should be able to:

- explain the use constructors.
- discuss overloading constructor.
- describe the role of destructors.

5.2 Constructors

Constructors are member functions of a class and have the same name as the class name. Constructors are called automatically whenever an object of

the class is created. This feature makes it very useful to initialize the class data members whenever a new object is created. It can also perform any other function that needs to be performed for all the objects of the class without explicitly specifying it.

A constructor knows only to build an object of its own class. Constructors aren't automatically inherited between base and derived classes. If the constructors are not provided in the derived class then C++ will provide a default constructor but it may not perform the functions as you like. And if you do not provide a constructor, the C++ will create a default constructor with no parameters in it. The default constructor will not be created if you provide a constructor in the derived class.

Characteristics of a constructor

- These are the functions having the name similar to its class. Their role is to initialize the class members when an object of the class is created.
- They should be declared in public section of class for availability to other functions.
- No return type, not even void is specified for the constructors.
- They cannot be inherited but the derived class can call the constructor of its baser class.
- They can call member functions of its class.
- They cannot be virtual.
- They can have default values and can be overloaded.
- It is possible to create multiple constructors of the same class but they should have different parameters so that they can be easily distinguished.

The syntax to define a constructor is as follows:

```
class classname
{
  public:
  classname();  //constructor
classname(argument list); //another constructor
…
};
```

5.3 Multiple Constructors

As you have already studied, you can declare multiple constructors of a class. And then you can use any of the constructors while creating an object of that class. Customization of the created object is facilitated by multiple constructors. As discussed above multiple constructors are differentiated by their argument list. When you declare multiple constructor of a class it is called constructor overloading. As you already know, constructors don't have return type. The example below demonstrates how to define multiple constructors in a class and how to call a desired one while instantiating the class.

```
#include<iostream.h>
class point
{
  public:
  int x, y;
public:
point()
{
 x=0;
 y=0;
cout<<''this is the constructor with no arguments\n";
};
point(int x1)
{
 x=x1;
 y=0;
cout<<"this is the constructor with one argument\n";
};
point(int x1, int y1)
{
  x=x1;
  y=y1;
cout<<"this is the constructor with two arguments\n";
};
void main()
```

```
{
 point p1;
 cout<<"point p1 (" << p1.x << "," << p1.y << ")" << endl;
point p2(5);
cout<<"point p2 (" << p2.x << "," << p2.y << ")" << endl;
point p3(10,10);
cout<<"point p3 (" << p3.x << "," << p3.y << ")" << endl;
}
```

In the above program you can see the point class having two data members x and y and three constructors. The first constructor is without any arguments and initializes both the data members to zero. The second constructor takes a parameter of integer type. In that one data member is initialized with the parameter x1 passed to the constructor and another is initialized with zero. The last constructor takes two arguments. The data members are initialized with the parameters x1 and y1 passed to the constructor. When a constructor is called, a message is printed on the user console.

Constructor pitfalls are:

As mentioned above, defining any constructor explicitly requires explicitly defining a default constructor if needed.

Defining constructors that take only one parameter allows the constructor to be implicitly used by the compiler to convert from the parameter type to the class type of the constructor. This is OK in many situations, such as converting a string literal to a string class instance. However in many other places it does not make sense. For example, a vector (single dimensional array) class that takes an integer type parameter to specify an initial number of elements in the vector object. In this case such an integer type can be implicitly converted to a vector type - which is probably not too useful and will probably hide errors (bugs!). In such cases you have to mark the constructor explicit:

```
class Vector
{
public:
explicit Vector( unsigned int size );
```

```
// ...
};

// Explicitly call the Vector::Vector(unsigned int) constructor
Vector v(10);
// Do stuff with v, fill it up with values etc.
v = 10; // ERROR as Vector::Vector(unsigned int) is explicit.
```

The above example assumes Vector has a meaningful assignment operator (operator=), which implies it also has a meaningful copy constructor as if one makes sense so does the other. I have not shown either in the parts of the Vector class shown to keep things simple. Without the 'explicit' qualification on the Vector::Vector (unsigned int) constructor declaration the line with the comment starting ERROR next to it would be legal. In this case 10 would be converted to a new (temporary) Vector by implicitly calling the Vector::Vector (unsigned int) constructor. Although this sounds as if it is neat, it would cause more errors and bugs. Assuming vector stored integer values, it is more likely that v = 10 should have been something like v[index] = 10 and the author forgot the subscript. By adding the 'explicit' qualification to a constructor it prevents the constructor calling it implicitly when we are not looking, i.e. an explicit constructor can only be called when we explicitly use it, as in the Vector v(10);

Self Assessment Questions

1. Special member function which has the name as class called _____.
2. The _____ constructor will not be created if you provide a constructor in the derived class.
3. When you declare multiple constructor of a class it is called _____.

5.4 Parameterized Constructors using Dynamic Objects

You can call a parameterized constructor while creating an object dynamically. It is as simple as calling a desired constructor while creating an object statically. The desired list of parameters is enclosed in parentheses following the point declaration. For example, the statement shown below calls the constructor of the class point and prints the values of data members in the user console.

```
point *p1 = new point();
```

The following statement calls a single argument constructor and initializes x to 5 and y to 0.

```
console.cout<<"point p1 (" << p1.x << "," << p1.y << ")" << endl;

point *p2= new point(5);
```

The program to demonstrate this concept is shown below:

```
#include<iostream.h>
class point
{
 public:
 int x, y;
public:
point()
{
 x=0;
 y=0;
cout<<''this is the constructor with no arguments\n";
};
point( int x1)
{
 x=x1;
 y=0;
cout<<"this is the constructor with one argument\n";
};
point(int x1, int y1)
{
  x=x1;
  y=y1;
cout<<"this is the constructor with two arguments\n";
};
void main()
```

```
{
 point *p1 = new point(); // it calls no argument constructor f the point class
                          and prints the values of data members in user
console.cout<<"point p1 (" << p1.x << "," << p1.y << ")" << endl;
point *p2= new point(5); // it calls a single argument constructor and
                                  initializes x to 5 and y to 0.
cout<<"point p2 (" << p2.x << "," << p2.y << ")" << endl;
point *p3= new point(10,10); //it calls the two argument constructor which
                                  initializes both x and y to 10.
cout<<"point p3 (" << p3.x << "," << p3.y << ")" << endl;
}
```

5.5 Copy Constructors

In some situations you may have to create a copy of an already existing object. You can do this by creating a constructor in the definition of your class. And you have to pass the object to be copied as an argument in that constructor. This type of constructor is known as copy constructor. Shown below is the definition of such a copy constructor:

```
point(point &p)
{
  x=p.x;
y=p.y;
}
```

In the example shown above the single argument is passed in the constructor that is the reference to the object of point class. The data members of the passed object are explicitly copied in the respective data members of the new object.

The example below demonstrates how to define and use a copy constructor.

```
#include<iostream.h>
class point
{
public:
```

```
  int x,y;
 public:
 point( int x1, int y1)
{
 x=x1;
 y=y1;
cout<<"constructor called\n";
};
point (point &p)
 {
  x=p.x;
  y=p.y;
 cout<<" copy constructor is called";
}
};
void main()
{
  point p1(5,5);
  cout<< "point p1 (" << p1.x << "," << p1.y << ")" << endl;
  point p2(p1)
  cout << "point p2 (" << p2.x << "," << p2.y << ")" << endl;
//testing for two objects
cout << "" setting data members of two objects\n";
p1.x =10; p1.y=10;
p2.x=20, p2.y=20;
cout<< "point p1 (" << p1.x << "," << p1.y << ")" << endl;
cout << "point p2 (" << p2.x << "," << p2.y << ")" << endl;
}
```

As you can see in the above example, two constructors are declared in the point class, i.e. the constructor with two arguments and a copy constructor. We create an object p1 that sets both data members of the point object to the value 5. The program prints these values on the user console for verification.

```
point p1(5,5);
cout<< “point p1 (“ << p1.x << “,” << p1.y << “)” << endl;
```

The statement point p2(p1) makes the use of copy constructor to construct object p2 of type point.

You will also want to verify that the two independent objects are created by the program. For this, as you can see in the program, individual data members of the two objects have to be set to different values and dumped the two objects on console. The following code member does this.

```
cout << “setting data members of two objects \n”;
p1.x =10; p1.y = 10;
p2.x =20, p2.y =20;
cout<< “point p1 (“ << p1.x << “,” << p1.y << “)” << endl;
cout << “point p2 (“ << p2.x << “,” << p2.y << “)” << endl;
```

Self Assessment Questions

4. In some situations it is required to create a copy of an already existing object this can be done by creating a _____.
5. In copy constructor we have to pass the object to be copied as a _____ in that constructor.

5.6 Destructors

Destructors are the member functions that are called automatically when an object of a class is destroyed or goes out of scope. It also performs cleanup work necessary before an object is destroyed. Likewise constructor the name of the destructor is also similar to its class and is prefixed by a ~ (tilde). For example:

```
class A
 {
public:
  // Constructor for class A
A();
  // Destructor for class A
  ~A();
};
```

Characteristics of a destructor:

- Destructors are called automatically when an object of a class is destroyed.
- They are declared in public section of a class.
- They don't have any return type (not even void) and do not take any argument as well.
- They cannot be declared const, volatile, const volatile or static. A destructor can be declared virtual or pure virtual.
- If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.
- They cannot be inherited. But the derived class can invoke the destructors of the base class.
- They cannot be overloaded.
- They can call other member functions of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for X:

```
X::~X() { }
```

The following program implements the constructor and destructors for a class

```
// constdest.cpp
# include<iostream.h>
class example
{ private:
int a;;
public:
example() {a=0; cout<<"Constructor invoked"<<endl;}
~example() {cout<<"Destructor invoked";}
void show()
{ cout<<"Data is="<<a<<endl;}
```

```
};
void main()
{  example e1;
e1.show();
}
```

If you run the above program you will get the output as follows:

```
Constructor invoked
Data=0
Destructor invoked
```

When an object e1 of example class is created, the constructor is invoked automatically and data value is initialized to zero. When the program is terminated the object is destroyed and the destructor is automatically invoked. When the program ends the object is destroyed which invokes the destructor. Please note that both the constructor and destructor are declared as public and they have no return value. The following program implements the overloaded constructors for the distance class.

```
//overloadconst.cpp
#include<iostream.h>
class distance
{ private:
   int feet;
int inches;
public:
distance()
{ feet=0;
inches=0;}
distance(int ft, int i)
{ feet=ft;
inches=i;}
void print()
{ cout<<feet<<”feet”<<inches<<”inches;}
};
void main()
```

```
{ distance d1, d2(10,2);
d1.print();
d2.print()
}
```

Of the two constructors are used in the above program, one is without parameters and the other is with parameters. These are automatically invoked. The first constructor is invoked when the object d1 is created and the second constructor i.e. distance (int ft, int i) is invoked when d2 is created as two arguments are passed. Please note that to invoke a constructor with arguments, argument values have to be passed along with the object name during declaration.

The compiler first implicitly defines the implicitly declared destructors of the base classes and non-static data members of class A before defining the implicitly declared destructor of A.

A destructor of class A is trivial if all the following are true:

- It is implicitly defined
- All the direct base classes of A have trivial destructors
- The classes of all the nonstatic data members of A have trivial destructors

The destructor is non-trivial if, any one of the above conditions is false. A union member cannot be of a class type that has a nontrivial destructor. Class members that are class types can have their own destructors. You already know that the destructors cannot be inherited, even though both base and derived classes can contain constructors. Suppose a base class X or its member contains destructor and the derived class of X does not declare a destructor, then the default destructor will be created. The destructor of the base class and the members of the derived class are called by the default destructor. The destructors of base classes and members are called in the reverse order of the completion of their constructor:

- The destructor for a class object is called before destructors for members and bases are called.
- Destructors for nonstatic members are called before destructors for base classes are called.

- Destructors for non-virtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block. When an automatic object (an object which is declared auto or register or not declared as static or extern) or a temporary object passes out of scope, then the destructors are implicitly called. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the delete operator for objects created with the new operator. For example

```
#include <string>
class A
 {
private:
  char * string;
  int n;
public:
  // Constructor
  A(const char*, int);
  // Destructor
  ~A() { delete[] string; }
};
// Define class A constructor
A::A(const char* p, int x)
 {
  string = strcpy(new char[strlen(p) + 1 ], p);
  nr = x;
}
int main ()
{
  // Create and initialize
  // object of class A
  A aobj = A("somestring", 10);
```

```
  // ...
  // Destructor ~A is called before
  // control returns from main()
}
```

The destructors can be used explicitly to destroy the objects, even though this method is not recommended. However to destroy an object created with the new operator, you can explicitly call the object's destructor. This concept is explained in the example below:

```
#include <iostream.h>
class X
{
  public:
    X() { cout << "X::X()" << endl; }
    ~X() { cout << "X::~X()" << endl; }
};
int main () {
  char* ptr = new char[sizeof(X)];
  X* xptr = new (ptr) X;
  xptr->X::~X();
  delete [] ptr;
}
```

The statement X* aptr =new(ptr) X creates a new object of type X dynamically. The object is created not in the free store but in the memory allocated by ptr. The storage allocated by ptr will be deallocated by the statement delete [] ptr. But the run time will still believe that the object pointed to by ap still exists until you explicitly call the destructor of X (with the statement aptr->X::~X()).

Self Assessment Questions

6. Destructor takes arguments and returns value. State (True/False)
7. The destructor for a _____ is called before destructors for members and bases are called.
8. Destructors are declared in _____ section of a class.
9. Destructors can be overloaded. (True/False)

10. Destructors are implicitly called at program termination for constructed external and static objects. (True/False)

5.7 Name Space

If you are using different modules/or libraries there are always chances of name clash. The reason is that these modules or libraries use the same identifier for many things. This issue is resolved by using namespaces in C++. A namespace is a certain scope for identifiers. Unlike a class, it is open for extensions that might occur at any source. Thus, it is possible for you to use a namespace to define components that are distributed over several physical modules. The C++ standard library is an example of such a component. In namespace known as std, all the identifiers of C++ standard libraries are defined.

Defining a Namespace

To define a namespace you should use 'namespace' keyword followed by the name of the namespace. The example to explain this concept is shown below:

```
namespace namespace_name {
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code;  // code could be variable or function.
```

For example:

```
namespace xyz //defining identifiers in namespace xyz
{
  class file;
 void myglobalfunc();
….
}
xyz::file obj;// using a namespace identifier
……
xyz::myglobalfunc();
```

Let us see how namespace scope the entities including variable and functions:

```
#include <iostream>
using namespace std;
// first name space
namespace f1_space{
  void f1()
  {
    cout << "this is the first name space" << endl;
  }
}
// second name space
namespace f2_space
{
  void f2(){
    cout << "this is the second namespace" << endl;
  }
}
int main ()
{
   // Calls function from first name space.
  f1_space::f1();

  // Calls function from second name space.
  f2_space::f2();
  return 0;
}
```

If the above code is executed, it will produce the result shown below:

this is the first namespace

this is the second namespace

The using directive

You can also avoid prepending of namespaces with the 'using' namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

```
#include <iostream>
using namespace std;
// first name space
namespace f1_sapce
{
   void f1()
  {
      cout << "this is the first space" << endl;
   }
}
// second name space
namespace f2_space{
   void f2(){
      cout << "this is the second space" << endl;
   }
}
using namespace f1_space;
int main ()
{
    // This calls function from first name space.
     f1();
      return 0;
}
```

If we compile and run above code, this would produce the following result:

this is the first space

You can also use the directive to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows: using std::cout;

Self Assessment Questions

11. A _____ is a certain scope for identifiers. Unlike a class, it is open for extensions that might occur at any source.
12. We can also avoid prepending of namespaces with the using namespace directive. (True/False).

5.8 Summary

- Constructors are member functions of a class, which have same name as the class name and are called automatically whenever an object of the class is created.
- We can declare multiple constructors of a class and it is called constructor overloading.
- Defining constructors that take only one parameter allows that constructor to be implicitly used by the compiler to convert from the parameter type to the class type of the constructor.
- We can call a parameterized constructor while creating an object dynamically. The required list of parameters is enclosed in the parentheses while we call the constructor with no arguments of the point class.
- In some situations, you may require to create a copy of an already existing object this can be done by using copy constructors.
- Destructors are the member functions that are called automatically when an object of a class is destroyed or goes out of scope.
- The destructors of base classes and members are called in the reverse order of the completion of their constructor.
- A namespace is a certain scope for identifiers. Unlike a class it is open for extensions that might occur at any source.
- We can also avoid prepending of namespaces with the ‘using’ namespace directive.

5.9 Terminal Questions

1. Implement a class stack, which simulates the operations of the stack allowing LIFO operations. Also, implement push and pop operations for the stack.
2. Write the output of the following program

```
# include<iostream.h>
class A
{ int a;
static int s;
public:
A()
{a=0; cout<<"Default Constructor Invoked"<<endl;}
A(int i)
{a=i; cout<<"Overloaded constructor Invoked"<<endl;s=s+a;}
void display()
{cout<<a<<endl<<s<<endl;}
}
int A::s=0;
void main()
{
A obj1(12);
obj1.display();
A obj2(10);
obj2.display();
}
```

3. Write a short note on constructors.
4. Describe multiple constructors with the help of an example.
5. Explain destructor in detail with the help of an example.
6. Describe the concept of namespace in detail.

5.10 Answers

Self Assessment Questions

1. Constructor
2. Default
3. Constructor overloading
4. Copy constructor
5. Argument
6. False
7. class object
8. Private
9. False
10. True
11. Namespace
12. True

Terminal Questions

1.
```
//stack.cpp
# include <iostream.h>
 # define size 100
 class stack
  { int stck[size];
  int top;
  public:
     stack() {top=0;
     cout <<"stack initialised"<<endl;}
     ~stack() {cout <<"stack destroyed"<<endl;}
    void push(int i);
    int pop();
    };
    void stack::push(int i)
    {
    if (top==size)
    { cout <<"stack is full";
      return;
```

```
    }
    stck[top]=i;
    top++;
    }
    int stack ::pop()
    { if (top==0) {
    cout << "stack underflow" ;
    return 0;
    }
    top--;
    return stck[top];
    }
  void main()
 { stack a,b;
 a.push(1);
 b.push(2);
 a.push(3);
 b.push(4);
 cout<<a.pop() << " ";
 cout<<a.pop()<<" ";
 cout<<b.pop()<< " ";
 cout<<b.pop()<<endl;
 }
```

2. Overloaded Constructor Invoked
   12
   12
   Overloaded constructor Invoked
   10
   22

3. Constructors are member functions of a class which have same name as the class name. Constructors are called automatically whenever an object of the class is created. This feature makes it very useful to

initialize the class data members whenever a new object is created. It also can perform any other function that needs to be performed for all the objects of the class without explicitly specifying it. For more details refer section 5.2.

4. Multiple constructors of a class. And then we can use any of the constructors while creating an object of that class. Customization of the created object is facilitated by multiple constructors. For more details refer section 5.3.
5. Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. For more details refer section 5.6
6. If you are using different modules/or libraries there are always chances of name clash. The reason being that these modules or libraries use the same identifier for many things. This issue is resolved by using namespaces in C++. A namespace is a certain scope for identifiers. Unlike a class it is open for extensions that might occur at any source. Thus it possible for you to use a namespace to define components that are distributed over several physical modules. For more details refer section 5.7

References:

- Object Oriented Programming with C++, by Poornachandra Sarang. Prentice Hall India Pvt., Limited.
- Object-Oriented Programming with ANSI and Turbo C++- Seventh Edition, by Kamthane. Pearson Education India.