# Constructors and Destructors

## 5.1 Constructors

A constructor is a special member function whose task is to initialize the objects of its class. It has the name same as that of class name. The constructor is automatically invoked whenever an object is created. It is used for automatic initialization of objects*. Automatic initialization is the process of initializing object’s data members when it is first created, without making a separate call to a member function.* It is called constructor because it constructs the values of data members of the class.

**For example,**

class rectangle

{

private:

int length;

int breadth;

public:

rectangle( ) ***//constructor with no argument***

{

length = breadth= 0;

}

rectangle(int x); ***// constructor declared, constructor with one argument.***

………

………

};

rectangle:: rectangle(int x) ***// constructor defined***

{

length = breadth= x;

}

**Characteristics of constructor:**

* They executed automatically when objects are created.
* Constructors have the same name as that of class name.
* They should be declared in the “public” section.
* They do not have return types, not even void and therefore, and they cannot return any values.
* Like other C++ functions, they can have default arguments.
* Constructors cannot be **virtual**.

## 5.2 Types of Constructor

There are, basically, three types of constructor:

* Default constructor
* Parameterized Constructor
* Copy Constructor

### 5.2.1 Default Constructor

A constructor that does not take any parameter is called default constructor.

**Calling default constructor:**

rectangle rect1; ***(syntax:*** *class\_name object\_name;****)***

There are three possible situations for this.

**1. If we do not provide any constructor with a class,** the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

class A

{

//no constructor.

};

**2. If we provide a constructor without any arguments then that is the default constructor.**

class A

{

A ()

{}

//or

A (void)

{}

};

**3. If we provide constructor with all default arguments,** then that can also be considered as the default constructor.

class A

{

A (int x=5)

{ }

};

### 5.2.2 Parameterized Constructor

A constructor that takes arguments is called a parameterized constructor. Arguments are passed when the objects are created.

**Example:**

class rectangle

{

private:

int length;

int breadth;

public:

rectangle(int l, int b) **//parameterized constructor**

{ .

length = l;

breadth = b;

}

………

*………*

*};*

Arguments are passed when the objects are created. This can be done in two ways.

-By calling the constructor explicitly

* + **Example:** rectangle rect1 = rectangle(4, 4); ***//explicit call***

**-** By calling the constructor implicitly

* + **Example:** rectangle rect1(4, 4); ***//implicit call***

### 5.2.3 Copy Constructor

In the C++ programming language, a copy constructor is a special constructor for creating a new object as a copy of an existing object.

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.A copy constructor is called when an object is created by copying an existing object. The process of initializing through a copy constructor is known as ***copy initialization***. A copy constructor takes a reference to an object of the same class as itself (as an argument). *We cannot pass the argument by value to a copy constructor.*

**Example:** The copy constructor may have the following signature (prototype):

rectangle (rectangle &); **// reference to object of same class.**

***Calling copy constructor***

rectangle obj2(obj1);

The above statement would define the object obj2 and initialize it to the values of obj1. The above statement can also be written as,

rectangle obj2 = obj1;

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

Remember the statement

obj2 = obj1;

will not invoke the copy constructor. However, if obj1 and obj2 are objects, this statement is legal and simply assigns the values of obj1 to obj2, member by member. This is the task of the overloaded assignment operator (=).

#### COMPLETE EXAMPLE:

class alpha

{

int x, y;

public:

alpha (){}; **//default constructor**

alpha(int a, int b) **//parameterized constructor**

{

x=a;

y=b;

}

alpha (alpha & s) **//copy constructor**

{

x =s.x;

y=s.y;

}

void display(void)

{

cout<<"x is: " <<x<<" y is: " <<y<<endl;

}

};

int main()

{

alpha a(4, 5); **//calling parameterized constructor; object a is created.**

alpha b(a); **//copy constructor called**

alpha c = b; **// copy constructor called again**

alpha d; **//d is created but not initialized.**

d = a; **//copy constructor not called, implicit overloading of assignment operator (=).**

a.display();

b.display();

c.display();

d.display();

return 0;

}

## 5.3 Constructor Overloading

We can define more than one constructor in a class which is called constructor overloading.

### Example:

#include <iostream>

using namespace std;

class Item

{

int code, price;

public:

Item(){ code= price =0; } **//Default Constructor**

Item(int c, int p) **//Parameterized Constructor**

{

code=c;

price=p;

}

Item(Item &x) **//Copy Constructor**

{

code= x.code;

price= x.price;

}

void display()

{

cout<<"Code: "<<code<<endl<<"Price: "<<price<<endl;

}

};

int main()

{

Item I1; **//calling default constructor**

Item I2(102,300); **//calling parameterized constructor**

Item I3(I2); **//calling copy constructor**

I1.display();

I2.display();

I3.display();

return 0;

}

In the above example of class **Item**, we have defined three constructors. The first one is invoked when we don’t pass any arguments. The second gets invoked when we supply two arguments, while the third one gets invoked when an object is passed as an argument.

## 5.4 Constructor with default argument

It is possible to define a constructor with default argument like in normal function. Constructor with default arguments are called ***default argument constructor***.

**For example:**

class complex

{

int x, y ;

public:

complex (int a=4, int b=0) // constructor with two default arguments.

{

x=a;

y=b;

}

} ;

In above example, the default value of b is zero i.e. (0 is assigned to y). We can create the following type of objects for above class:

1. complex C; //default value 4 is assigned to x and 0 is assigned to y.

2. complex c1(5) //default value 0 is assigned to y.

3. complex c2(5, 6) //Here, default values are overridden.

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

A a;

* + The ambiguity is whether to call A::A() or A::A(int x = 0).

## 5.5 Dynamic initialization of objects

Class objects can be initialized dynamically (i.e. at the run time). The user provides the values at the run time.

**Advantage:** various initialization formats can be provided using constructor overloading.

void display(void)

{

cout<<"x is: " <<x<<" y is: " <<y<<endl;

}

};

int main()

{

float s, t;

cout<<"Enter of values: "<<endl;

cin>>s>>t;

alpha A(s, t);

A.display();

return 0;

}

### Example:

#include<iostream>

using namespace std;

class alpha

{

float x, y;

public:

alpha (){}; //default constructor.

alpha(float a, float b) //constructor with two parameters.

{

x=a;

y=b;

}

## 5.6 Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release memory and to perform other “cleanup” activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

Destructor will automatically be called by a compiler to clean up storage taken by objects. The objects are destroyed in the reverse order from their creation in the constructor.

**Characteristics:**

* Destructor is invoked automatically, when an object goes out of scope (i.e. exit from the program, or block or function).
* Like constructors, destructors do not have a return value.
* Destructor never takes any argument. Hence, we can use only one destructor in a class (*i.e., they cannot be overloaded*).
* Like constructors destructors are also defined in the public section.

### Example:

class A

{

int a;

public:

A( ) **//constructor**

{

cout<< "\n object created\n";

a=0;

}

~A( ) **//destructor**

{

cout<< "Object destroyed \n";

}

};

int main( )

{

A s, z;

return 0;

}**//Remember destructor is invoked automatically.**

### Objects are destroyed in the reverse order by the destructor:

#include<iostream>

using namespace std;

class A

{

int a;

static int count;

public:

A( ) **//constructor**

{

count++;

cout<< count<<"object created\n";

a=0;

}

~A( ) **//destructor**

{

cout<<count<< "object destroyed \n";

count--;

}

};

int A:: count; **//automatically initialized to 0.**

int main( )

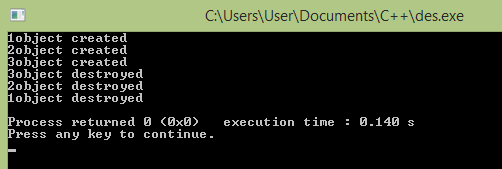
{

A p, q, r; **//3 objects are created and initialized to 0.**

return 0;

}

**Output:**

****

### Another Example:

#include<iostream>

int main( )

{

A p, q, r; **//3 objects are created and initialized to 0.**

{

cout<< "\n \n Enter Block1\n" ;

A s;

}

{ cout<< "\n \n Enter Block2\n" ;

A t;

}

cout<< "\n \n Re-enter main\n" ;

return 0;

}

using namespace std;

int count = 0;

class A

{

int a;

static int count;

public:

A( ) **//constructor**

{

count++;

cout<< count<<"object created\n";

a=0;

}

~A( ) **//destructor**

{

cout<<count<< "object destroyed is: "<<count<<endl;

count--;

}

};

### Run the following program, and analyze the output:

#include <iostream>

using namespace std;

class Item

{

int code, price;

public:

Item() //Default Constructor

{

code= price =0;

cout<<"Object created is: "<<this<<endl; //***this* is an automatic pointer to the calling**

**//object.**

}

Item(int c, int p) //Parameterized Constructor

{

code=c;

price=p;

cout<<"Object created is: "<<this<<endl; //***this* is an automatic pointer to the**

**//calling object.**

}

Item(Item &x) //Copy Constructor

{

code= x.code;

price= x.price;

cout<<"Object created is: "<<this<<endl; //***this* is an automatic pointer to the**

**//calling object.**

}

void display()

{

cout<<"Code: "<<code<<endl<<"Price: "<<price<<endl;

}

~Item()

{

cout<<"Object Destroyed is: "<<this<<endl;

}

}; **//class ends here.**

int main()

{

Item I1; **//calling default constructor**

Item I2(102,300); **//calling parameterized constructor**

Item I3(I2); **//calling copy constructor**

I1.display();

{

Item I5;

I5.display();

}**//local functional block ends here. Object I5 is destroyed.**

I2.display();

I3.display();

return 0;

} **//main() block ends here, so object destroyed in the order I3,I2,I1.**

## 5.7 const (Constant) Object and const Member Functions

Some objects need to be modified and some do not. We can use the keyword **const** to specify that an object is not modifiable and that any attempt to modify the object should result compiler error. The statement

const distance d1(5, 6.7);

declares a **const** object **d1** of class **distance** and initializes it to 5 feet and 6.7 inches. These objects must be initialized. A **const** object can only invoke a **const** member function. If a member function does not alter any data in the class, then we may declare and define it as a **const** member function as follows:

void display()const

{

cout<< "("<<feet<< ", " <<inches<< ")"<<endl;

}

The qualifier **const** is inserted after the function’s parameter list in both declarations and definitions. The compiler will generate an error message if such functions try to alter the data values. A **const** member function cannot call a non-**const** member function on the same class.