## **Destructors in C++**

**What is a destructor?**

Destructor is a member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed. All objects created in a function are automatically destroyed when that function returns, because the objects go out of scope.

**Properties of Destructor:**

* Destructor function is automatically invoked when the objects are destroyed.
* The destructor does not have arguments.
* It has no return type - not even void.
* A destructor should be declared in the public section of the class.

**Example:**

#include <iostream>

using namespace std;

class test{

int dummy;

public:

test() {

dummy=0;

cout<<"This is the constructor."<<endl;

}

//Destructor must be named as ~name\_of\_class

~test() {

cout<<"This is the destructor."<<endl;

}

};

int main()

{

test testobj;

return 0;

}

**Output:**

This is a constructor.

This is a destructor.

## **Copy Constructors in C++**

**What is a copy constructor?**

A copy constructor is a member function that initializes (or constructs) an object using another object of the same type. The general signature of a copy constructor is

name\_of\_class (cons name\_of\_class &obj)

**Example:**

#include <iostream>

using namespace std;

struct someobj {

int \*p;

};

class test{

int dummy;

someobj sobj;

public:

test() {

dummy=0;

sobj.p = &dummy;

cout<<"This is the default constructor."<<endl;

}

test(const test &testobj) {

dummy=testobj.dummy;

sobj.p = &dummy;

cout<<"This is the copy constructor."<<endl;

}

};

int main()

{

test testobj1; // this calls the default constructor

test testobj2(testobj1); //this calls the copy constructor

return 0;

}

**Output:**

This is the default constructor.

This is the copy constructor.

**Why is a user defined copy constructor needed?**

C++ will create a default copy constructor if the programmer does not provide one. This will produce a byte-by-byte shallow copy of the other object.

When the user wants additional/different behaviour in the copy constructor, the user has to define one themself.

As an example, consider the program given above - a default copy constructor for the class test will copy testobj.sobj.p to sobj.p , while the expected/desired behaviour may be what is given in the user defined copy constructor given above.

## **Overloading the Assignment Operator in C++**

The reasons for overloading an assignment operator are very similar to why a copy constructor needs to be created by an user.

**Example:**

#include <iostream>

using namespace std;

struct someobj {

int \*p;

};

class test{

int dummy;

someobj sobj;

public:

test() {

dummy=0;

sobj.p = &dummy;

cout<<"This is the default constructor."<<endl;

}

test& operator=(const test &testobj) {

dummy=testobj.dummy;

sobj.p = &dummy;

cout<<"This is the overloaded assignment operator."<<endl;

return \*this;

}

};

int main()

{

test testobj1; //this calls the default constructor

test testobj2; //this calls the default constructor

testobj1 = testobj2; //this calls the overloaded assignment operator

return 0;

}

**Output:**

This is the default constructor.

This is the default constructor.

This is the overloaded assignment operator.

**Why is a user defined overloaded assignment operator needed?**

C++ will create a default overloaded assignment operator if the programmer does not provide one. This will produce a byte-by-byte shallow copy of the other object.

When the user wants additional/different behaviour in the overloaded assignment operator, the user has to define one themself.

As an example, consider the program given above - a default overloaded assignment operator for the class test will copy testobj.sobj.p to sobj.p , while the expected/desired behaviour may be what is given in the user defined copy constructor given above.