**Constructors and destructors**

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become

operative and to avoid returning unexpected values during their execution. For example, what would happen if in

the previous example we called the member function area() before having called function set\_values()? Probably

we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called

whenever a new object of this class is created. This constructor function must have the same name as the class,

and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

// example: class constructor

#include <iostream>

using namespace std;

class CRectangle {

int width, height;

public:

CRectangle (int,int);

int area () {return (width\*height);}

};

CRectangle::CRectangle (int a, int b) {

width = a;

height = b;

}

int main () {

CRectangle rect (3,4);

CRectangle rectb (5,6);

cout << "rect area: " << rect.area() << endl;

cout << "rectb area: " << rectb.area() << endl;

return 0;

}

Output-

rect area: 12

rectb area: 30

As you can see, the result of this example is identical to the previous one. But now we have removed the member

function set\_values(), and have included instead a constructor that performs a similar action: it initializes the

values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are

created:

CRectangle rect (3,4);

CRectangle rectb (5,6);

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a

new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor

definition include a return value; not even void.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either

because its scope of existence has finished (for example, if it was defined as a local object within a function and the

function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no

value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the

moment of being destroyed we want to release the memory that the object was allocated.

// example on constructors and destructors

#include <iostream>

using namespace std;

class CRectangle {

int \*width, \*height;

public:

CRectangle (int,int);

~CRectangle ();

int area () {return (\*width \* \*height);}

};

CRectangle::CRectangle (int a, int b) {

width = new int;

height = new int;

\*width = a;

\*height = b;

}

CRectangle::~CRectangle () {

delete width;

delete height;

}

int main () {

CRectangle rect (3,4), rectb (5,6);

cout << "rect area: " << rect.area() << endl;

cout << "rectb area: " << rectb.area() << endl;

return 0;

}

Output:

rect area: 12

rectb area: 30

**Overloading Constructors**

Like any other function, a constructor can also be overloaded with more than one function that have the same

name but different types or number of parameters. Remember that for overloaded functions the compiler will call

the one whose parameters match the arguments used in the function call. In the case of constructors, which are

automatically called when an object is created, the one executed is the one that matches the arguments passed on

the object declaration:

// overloading class constructors

#include <iostream>

using namespace std;

class CRectangle {

int width, height;

public:

CRectangle ();

CRectangle (int,int);

int area (void) {return (width\*height);}

};

CRectangle::CRectangle () {

width = 5;

height = 5;

}

CRectangle::CRectangle (int a, int b) {

width = a;

height = b;

}

int main () {

CRectangle rect (3,4);

CRectangle rectb;

cout << "rect area: " << rect.area() << endl;

cout << "rectb area: " << rectb.area() << endl;

return 0;

}

Output-

rect area: 12

rectb area: 25

**Default constructor**

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default

constructor with no arguments. Therefore, after declaring a class like this one:

class CExample {

public:

int a,b,c;

void multiply (int n, int m) { a=n; b=m; c=a\*b; };

};

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply

declaring them without any arguments:

CExample ex;

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default

constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for

the class:

class CExample {

public:

int a,b,c;

CExample (int n, int m) { a=n; b=m; };

void multiply () { c=a\*b; };

};

**Parameterize constructor**

Here we have declared a constructor that takes two parameters of type int. Therefore the following object

declaration would be correct:

CExample ex (2,3);

But,

CExample ex;

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default

constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three

special member functions in total that are implicitly declared if you do not declare your own.

**Copy constructor**

These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data

members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be

something similar to:

CExample::CExample (const CExample& rv)

 {

a=rv.a; b=rv.b; c=rv.c;

}

Therefore, the two following object declarations would be correct:

CExample ex (2,3);

CExample ex2 (ex); // copy constructor (data copied from ex)