Review of Constructors/Destructors

© Charles Lin. 2001. All rights reserved.

Implicit Methods

> *"When should you write a constructor and destructor for a class?"*

What's your answer to this question?

Some people have been taught to write constructors and destructors all the time. That's not always the best thing to do. There's a simple reason for this. People make mistakes writing constructors and destructors.

C++ is very sneaky. It "wants" each class to have at least one constructor, one destructor, and an assignment operator. These methods are so important that even when you don't write one, the compiler generates an *implicit* version for you. You will discover that, in many cases, it's better to let C++ create these methods for you instead of defining it yourself.

- If no constructors are written, C++ creates an implicit default constructor for you. This constructor calls the default constructors for each of its instance variables (for basic types such as int, float, etc., there's no special initialization).
- If at least one non-default constructor is written, C++ does NOT create a default constructor.
- If there is no user-defined copy constructor, C++ creates an implicit copy constructor, which calls the copy constructor on each of its instance variables. The copy constructor for each of instance variables may be implicit or user-defined.
- If there is no user-defined assignment operator, C++ creates an implicit assignment operator, which calls the assignment operator on each of its instance variables. The assignment operator for each of the instance variables may be implicit or user-defined.
- If there is no user-defined destructor, C++ create an implicit destructor, which calls the destructor on each of its instance variables. The destructor for each of the instance variables may be implicit or user-defined.

As you can see, the copy constructor, destructor, and the assignment operator are very important methods in all classes. It's so important that C++ creates implicit versions of these methods for all classes, even when you does not explicitly define one. (Note: C++ doesn't always create a default constructor. It only does so when you don't explicitly define any constructors.)

When to Write Constuctors, Destructors, and Assignment Operators

Because the copy constructor, destructor, and overloaded assignment operator are so important, you need to know when to write these three methods and when *not* to write them.

Lesson 1 *Either explicitly define the copy constructor, destructor, and assignment operator as a group, or don't define any of these three methods.*

It's a simple lesson and easy to follow. Yet, quite often, some define, say, a destructor, without defining a copy constructor or assignment operator.

The first lesson is to write all three methods as a group, or avoid writing them at all. What's the second lesson?

Lesson 2 *Define the copy constructor, destructor, and assignment operator as a group only when the object manages dynamic memory allocation and deallocation directly.*

This lesson is a little harder to understand. When does an object manage dynamic memory allocation?

Let's consider the following example. Suppose you want to create a class that has a dynamic array. Let's call this class, `ArrayList`.

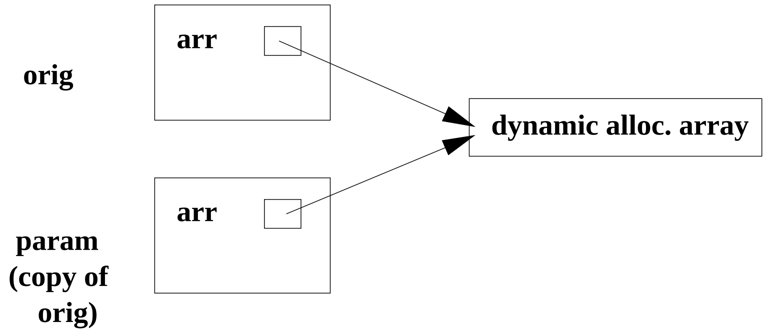
```
class ArrayList
{
   int *arr;       // a dynamic array
   int capacity;   // total size of array
   int size;       // number of elements used in array

public:
   ArrayList( int capacityIn = 5 );
};

ArrayList::ArrayList( int capacityIn ) : capacity( capacityIn ),
  size( 0 )
{
   arr = new int[ capacityIn ];
}
```

In an `ArrayList` object, the object must manage a dynamic array. It creates a dynamic array in the constructor shown above. Once you know the object uses dynamic memory allocation, in particular, once you know it uses "new", then you should write all three methods (copy constructor, assignment operator, and destructor).

Why Write These Methods with Dynamic Allocation?

Why *must* you write these methods when with objects that use `new` and `delete`? Suppose you write a destructor for `ArrayList`, but not a copy constructor or an overloaded assignment operator. In this case, an implicit version of these two methods is created by the compiler.

Imagine you are passing an `ArrayList` object by value to a method. This will call the implicit copy constructor. The implicit copy constructor does a memberwise copy. For pointers, it merely copies the address. Thus, you will have the following situation:

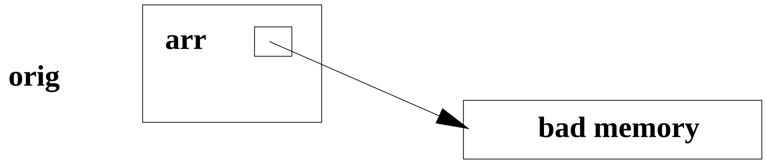


When the flow of control enters the method, the parameter will get a copy of the original argument, as shown above. Unfortunately, this creates a shallow copy. Both the original and copy both point to the same allocated memory.

When the flow of control leaves the method, the destructor (which you wrote) is called. Assuming that you wrote a destructor that looks like:

```
ArrayList::~ArrayList
{
   delete [] arr;
}
```

Then, the dynamic array will be reclaimed. The original object now contains a dangling pointer that points to memory that's been given back to the runtime system. If the object attempts to



manipulate the array, one of two bad things happen. Either that memory gets reallocated to some other object, and therefore contains "garbage" as far as the object is concerned, or worse, it causes errors in the runtime system, and a potential core dump may mysteriously occur sometime in the future.

All of this occurs because the person writing the code (you) didn't follow Lesson 2. If you write only one of the three methods (or none), you may run into this kind of problem.

What About This?

Suppose you create the following class:

```
class Container
{
   ArrayList quiz;
   ArrayList exam;
}
```

This class contains two `ArrayList`'s. Should you write a copy constructor, destructor, and assignment operator for this class.

Now think carefully, before you answer.

The answer is no. Ask yourself "will this class create dynamic memory or not"? You might say "of course, it does". Ah, `ArrayList` does, but how did you know that? You might say "I knew it because we just implemented it". But what if I gave you `ArrayList` and didn't tell you anything more. Then, how would you decide?

In fact, you are rarely told whether the class you use has dynamic memory allocation or not. The reason? It's not important. If `ArrayList` has been written correctly, then it's up to the class to make sure that it does dynamic allocation correctly. As far as `Container` is concerned, it doesn't have to deal with the dynamic memory allocation of `ArrayList`.

In this case, you wouldn't write of the three methods for `Container`.

Factoring Code Out

A common problem with beginning programmers is writing repetitive code. Have you ever written code and thought "I'm writing this for the tenth time. Surely there must be a better way". In fact, you know the better way—write functions!

The concept of "factoring code" is similar to factoring in algebra. For example, when you see an expression like:

$$x(y+z)+(w-1)(y+z)$$

you can factor out $y+z$ to get:

$$[x+(w-1)]\cdot(y+z)$$

Similarly, once you write code, you may discover that you should factor out common code.

In particular, I suggest writing the following helper methods (all of which should be private), for objects which use "new" and "delete" (i.e., which manage dynamic memory).

- `void copy( const Object &other )`

  Makes a "deep" copy of the object.

- `void init()`

  Initializes an "default" object.

- `void freeMem()`

  Frees any dynamic memory, and sets pointers to NULL.

How would you use these methods in the three methods described earlier?

(If there are any errors, please let Charles know, so he can fix them).