Lecture 12

Destructors

“Absolute C++”
Chapters 10.3, 15.2

Destructors

- Quick review of constructors:
  - They’re called when an object is created
  - You may perform one-time initializations in constructors
  - Memory allocation
  - member variable initializations
  - You may define multiple constructors
  - Each may take a varying number of parameters
  - If you do not define a constructor for a given class, C++ will define one for you (that basically does nothing).
- Destructors are called whenever an object is destroyed (or just before destruction)
- Destructors give you a place to:
  - Free memory allocated in a constructor
  - Release system resources (Windows, disk drives, etc.)

Destructors

A destructor is declared by declaring a member function which has the same name as the class (like a constructor) only prefixing it with a tilde (~) character.
- There is ever only one destructor per class.
- Destructors do not take arguments.
- It is not usually necessary to zero out member variables
  - The destructor is called right before the object goes away, so dangling values in member variables shouldn’t matter.
- Let’s look at a class definition that contains a destructor...

```cpp
class SomeClass
{
public:
    SomeClass()
    {
        cout << "We're in the constructor" << endl;
    }
    ~SomeClass()
    {
        cout << "We're in the destructor" << endl;
    }
};
```

Demonstration #1

A Simple Destructor

Destructors

- The following class simply shows a constructor and a destructor.
- We’ll print out messages from each so that we know when the runtime environment is executing them...

```cpp
class SomeClass
{
public:
    SomeClass()
    {
        cout << "We're in the constructor" << endl;
    }
    ~SomeClass()
    {
        cout << "We're in the destructor" << endl;
    }
};
```

Destructors

- The best way to determine if you need a destructor in a given class is to look at any constructors which might be present.
- Remember the following modified version of Course from last lecture:

```cpp
class Course
{
public:
    Course();
    Course(string theCourse, string theInstructor, int classSize);
private:
    string courseName;
    string instructor;
    int size;
    Student *studentList;
    int nextStudent;
};
```
When looking at the definition of the constructors, we see the following:

```cpp
Course::Course()
{
    courseName = "unknown";
    instructor = "unknown";
    size = nextStudent = 0;
    studentList = NULL;
}

Course::Course(string theCourseName, string theInstructor, int classSize): courseName(theCourseName),
    instructor(theInstructor), size(classSize),
    nextStudent(0)
{
    studentList = new Student[size];
}
```

In the overloaded constructor that takes three arguments, we are dynamically allocating memory.

Since this memory needs to be freed somewhere, a destructor seems like a logical choice.

```cpp
class Course
{
public:
    // Constructors
    Course();
    Course(string theCourse, string theInstructor, int classSize);
    ~Course(); // Notice the "~" in front of the member name
    // Rest of class definition here..
};
```

Just like constructors, destructors can be defined either inside the class definition or in the corresponding .cpp file.

Using the latter, we might see this:

```cpp
Course::~Course()
{
    // Check to see if studentList has allocated memory
    if (studentList)
    {
        // it does! Delete the dynamically allocated array
        delete [] studentList;
    }
}
```

What happens if I want to derive a class from Course?

Take a more specific computer science course:

```cpp
class CSCourse : public Course
{
    public:
        CSCourse();
        CSCourse(string theCourse, string theInstructor, int classSize, bool argCSUGaccount);
        CSCourse(Course theCourse, bool argCSUGaccount);
        bool getsCSUGaccount() { return csugAccount; }
    private:
        bool csugAccount;
};
```
Overriding Destructors?
- If CSCourse had its own destructor, would it override Course’s?
- Suppose we arbitrarily define a destructor for URL:

```cpp
class CSCourse : public Course
{
public:
    CSCourse():Course(),csugAccount(false){}
    CSCourse(string theCourse,string theInstructor,
           int classSize, bool argCSUGaccount):
        Course(theCourse,theInstructor,classSize),
        csugAccount(argCSUGaccount){}
    ~CSCourse() { cout << "In CSCourse destructor" << endl; }
    bool getsCSUGaccount() { return csugAccount; }
private:
    bool csugAccount;
};
```

- What will happen now when we run our simple program?

Overriding Destructors? (cont)
- OK, why did it call both destructors?
- Because it’s the “right thing to do” :-) 
- Destructors are not “overridden” by derived classes.
- When an object is destroyed, the destructor for that class is called followed by the destructors for any base classes.
- Does this mean we don’t need to worry about virtual destructors?
- No, we do. Consider the following code:

```cpp
int main()
{
    CSCourse *myCSCourse = new CSCourse();
    Course *aCourse;
    aCourse = myCSCourse;
    delete aCourse;
}
```

Overriding Destructors? (cont)
- OK, so how do we declare a virtual destructor?
- Remember, it is the destructor in the base class that needs to be declared as virtual.
- So, if we declare Course’s destructor to be virtual, we’ll get the desired behavior...
Automatically Generated Functions

- If you don’t define a constructor, destructor or copy constructor, C++ will define a “default” one for you.
- A default constructor does nothing
- A default destructor does nothing
- A default copy constructor will populate all the member variables of the new class with values found in all the member variables of the class being copied from.
- Why do we care?
- Consider the following code:

```cpp
class Course {
public:
    string name;
    string instructor;
    int size;
private:
    // Constructor, destructor, copy constructor
};
```

Course MakeACourse(string name, string instructor, int size) {
    Course returnCourse(name, instructor, size);
    return returnCourse;
}
```

Automatically Generated Functions (cont)

- When we return `returnCourse` the compiler actually makes a copy of `returnCourse` on the stack and then `returnCourse`’s destructor is called.
- When `returnCourse`’s destructor is called all dynamically allocated memory is freed.
- That leaves the copy of `returnCourse` on the stack with a pointer to deallocated memory waiting to be assigned to whatever variable is receiving it in the calling function… OUCH!
- Next time we’ll work on fixing that problem...

Lecture 12

Final Thoughts