CS 213 -- Lecture #8

"Late Night Guide to C++"
Chapter 6 pg 145 - 157
DESTRUCTORS

Administrative...

- Assignment #3 graded
  - Average grade: NN
  - YY Assignments Turned In
- Remember, Prelim #1 10/14
- Before we talk about destructors...

A Lecture Within a Lecture

Why reading in Strings is such a pain

Why reading in Strings is such a pain

- As many of you know, the C++ string class and `cin` don't always get along.
- Actually, it's more a problem/feature with `cin` than the C++ string class. C strings have the same exact problem.
- Why? Let's start with a discussion of C strings.
- A C string is not a class, it is simply an array of characters.
- Any given C string of length `n` will correspond to a character array of *at least* `n+1` characters. The extra character is a NULL byte which terminates the string.
- Most of us old time C programmers (well, OK, I'm not really *that* old) tended to stick with C strings instead of "converting".
- Or we wrote our own String classes which suited our needs better than the C++ string class.
- Either way, it's good to know about C strings.

Why reading in Strings is such a pain (cont)

- Even though C strings are based on the concept of a simple array of characters, `cin` and `cout` try to deal with them as best as they can.
- Consider the following code:

```
int main()
{
  char cStr[80];  // Allocate a C string
  cout << "Enter a string> ";
  cin >> cStr;
  cout << "You entered… " << cStr;
}
```

- Let's make sure this works the way we think it does...

Demonstration #1

C Strings

Why reading in Strings is such a pain (cont)

- Why did input stop in the middle of our sentence?
- Because `cin` is designed to stop reading when *whitespace* is encountered.
- And since were directing `cin` at an array of characters (C string), it knows to terminate the target array with a NULL byte when the space is encountered.
- Is there a way to tell it *not* to stop on spaces?
- Well, you're suppose to be able to:

```
int main()
{
  char cStr[80];  // Allocate a C string
  cout << "Enter a string> ";
  cin >> noskipws >> cStr;  // This is suppose to work
  cout << "You entered… " << cStr;
}
```

Why reading in Strings is such a pain (cont)

- But it doesn't (at least not on my Mac :-))
- Then there is the getline() "trick".
- cin is just a class and it has member functions.
  - One such member function is cin.getline() which is suppose to get all characters from the input stream until '\n' is encountered.
- Great. Doesn't this solve our problem?
- Not really. Consider the following code:

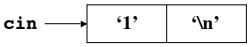
```
int main()
{
  int k;
  char cStr[80];
  cout << "Enter a number> "; cin >> k;
  cout << "Enter a string> "; cin >> cStr;
  cout << "Number is " << k << ", string is: " << cStr <<
endl;
}
```

Why reading in Strings is such a pain (cont)

- The output of this simple program might look like this:

```
Enter a number> 1
Enter a string>
Number is 1, string is:
```

- So what is going on?
- The input stream (that which cin reads from) is thought of as an array of characters.
- So when we enter "1" above, we put the following two characters into the input stream buffer:

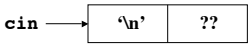


Why reading in Strings is such a pain (cont)

- When the following line of code is executed...

```
cout << "Enter a number> "; cin >> k;
```

- cin manages to grab the "1" out of the input stream buffer but leaves the newline there. This leaves us with something like this:



Why reading in Strings is such a pain (cont)

- Now, when the next line of code is executed...

```
cout << "Enter a string> "; cin >> cStr;
```

- cin sees the newline as the first character in the input stream buffer and assumes we haven't "seen" it before.
- It processes the newline viewing it as the terminating newline for our string read.
- This leaves us with an empty string in our string variable.
- So what can we do?
  - Try to extract the newline before actually calling "cin >> cStr"
  - Find a different way to read in strings (well, and other types as well)
  - How about our very own String class?

Demonstration #2

`MyString`:
The String Class for the Rest of Us

OK, back to the subject at hand

Destructors

Destructors

- Last lecture we talked about 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

- The best way to determine if you need a destructor in a given class is to look at any constructors which might be present.
- Consider the following class definition:

```
class MyString
{
public:
  MyString();
  MyString(string initialValue)
  {
    setValue(initialValue);
  }

  // Rest of class definition here
};
```

- Let's get a reminder of what `setValue(string)` does...

Destructors (cont)

```
bool MyString::setValue(string cppString)
{
  int spaceNeeded = cppString.length();

  // make sure we have enough space to hold the new string
  if (spaceNeeded > allocatedSpace)
  {
     // If we get here we'll be dynamically allocating
     // memory!
     if (!growStorage(spaceNeeded+1))
        return false;
  }

  strcpy(storagePtr,cppString.c_str());
  return true;
}
```

Destructors (cont)

- Notice that we we might be performing memory allocation in the constructor.
- Since this memory needs to be freed somewhere (presumably when we're done with the object) a destructor seems likes a logical choice.
- How do we delcare a destructor?

```
class MyString
{
public:
  // Constructors
  MyString();
  MyString(string initialValue);
  // Destructor
  ~MyString();  // Notice the '~' in front of the member name
  // rest of class definition here...
};
```

Destructors (cont)

- 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.
- Our destructor definition would be as follows:

```
MyString::~MyString()
{
  // Don't try to free space if it wasn't allocated!
  if (allocatedSpace > 20)
    delete [] storagePtr;  // Free dynamic memory
}
```

Demonstration #3

`MyString`'s Destructor

Destructors and Inheritance

- What happens if I want to derive a class from MyString?
- Take a very simple URL class...

```
class URL : public MyString
{
public:
  URL():MyString(){}
  URL(string initialValue):MyString(initialValue){}

  string getProtocol();
};
```

- Let's implement `getProtocol()` first...

Destructors and Inheritance (cont)

```
string URL::getProtocol()
{
  string aCopy = MakeString();  // Calls MyString::MakeString()
  return aCopy.substr(0,aCopy.find(':'));
}
```

- Now consider a very simple `main()` function...

```
int main()
{
  URL myURL;
  cout << "Enter a URL> ";
  myURL.readString(80);
  cout << "You entered… " << myURL.MakeString() << endl;
  cout << "And the protocol is: " << myURL.getProtocol() <<
          endl;
}
```

Demonstration #4

Inherited Destructor

Overriding Destructors?

- If `URL` had it's own destructor, would it override `MyString`'s?
- Suppose we arbitrarily define a destructor for `URL:`

```
class URL public MyString
{
public:
  URL():MyString(){}
  URL(string initialValue):MyString(initialValue){}
  ~URL()
  {
    cout >> "We're in the destructor for URL!" << endl;
  }
  string getProtocol();
};
```

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

Demonstration #5

Destructors in Base and Derived Classes

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:

```
int main()
{
  URL *myURL = new URL("http://salsa.cit.cornell.edu");
  MyString *aString;
  aString = myURL;
  delete aString;
}
```

Demonstration #6

Virtual Destructors?

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 `MyString`'s destructor to be virtual, we'll get the desired behavior...

Demonstration #7

Virtual Destructors!

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:

```
MyString MakeAString(string cppString)
{
  MyString returnStr(cppString);
  return returnStr;
}
```

Automatically Generated Functions (cont)

```
MyString MakeAString(string cppString)
{
  MyString returnStr(cppString);
  return returnStr;
}
```

- When we return `returnStr` the compiler actually makes a copy of `returnStr` on the stack and then `returnStr`'s destructor is called.
- When returnStr's destructor is called all dynamically allocated memory is freed.
- That leaves the copy of returnStr 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 week we'll work on fixing that problem...

Final Thoughts

- Assignment #4 is due on Thursday
- Come to office hours with problems