Strings with Class
"Oh such grace, such beauty"

CS 211

Adam Sweeney

November 2, 2017

Wichita State University, EECS

Introduction

- We have learned about C-Strings, which are inherited from C
- C++ has its own version of strings

Agenda

- Section 9.2
- An introduction to the concept of classes
- Declaring
- String objects are not so different

Classes?

Classes?

- One thing C++ does better than C is object-oriented programming
- This class will not be very concerned with the object-oriented programming (OOP) paradigm
- We are currently focusing strictly on the procedural paradigm
- But, because C++ has OOP as a pillar of its design, we can't escape using classes
- And we haven't, we've been using classes since our "Hello world" program

So, What is a Class?

- Very loosely speaking, a class is a data structure made up of variables and functions gathered together around a common goal
- The part that matters to us is that a class can contain data and functions
- When we declare a variable of a class type, we are creating an object
- It is through these objects that we can access the functions of a class

Accessing the Functions of a Class

- It has been stated that we need to declare a variable of a class type in order to access the functions of a class
- Because we need an object, any object that calls a class function is called a *calling object*
- Below is an example that calls the `length()` function of a string object

```
string foo = "Hello";
unsigned int fooLength = foo.length();
```

- Functions of a class are accessed through the **dot operator**, which is just a period
- Note that we treat these functions just like any other, they are simply invoked in a way we haven't seen before
- Also note, because a calling object is required, the function `length()` can only find the length of `foo` in the example above, since it was `foo` that called the function

That's it?

- This all that is necessary to know about classes and objects in order to understand how to appropriately invoke a class function
- Classes will be discussed in more detail in CS 311

Declaring `string` Objects

Declaring `string` Objects

- Unlike C-Strings, which are an array of characters terminated by the null character, string objects will be much simpler to use
- In order to declare them, you must include the <string> library

```
#include <string>

string word = "Hello";
```

- One thing to keep in mind is that the string class was defined to behave as closely to a primitive data type (`int`, `double`, etc.) as possible
- This means that if we are able to do a thing with an `int`, we should expect to do the same with a string object

“What once was broken is whole again” –Someone(hopefully)

- With a C-String, we were only allowed to assign a C-String as a value at initialization
- This works with strings

```
char word[10];
word = "Hello";     // ILLEGAL with C-Strings

string word;
word = "Hello";     // LEGAL with string objects
```

Other Things That Work Better

- Joining two string objects is much simpler

```
string word = "Hello", other = "Hi", more = "Good Day";

cout << (word + other)     // Prints HelloHi
word += more;              // word is now HelloGood Day
```

- Joining string objects is extremely easy, and safe
- We don't have to worry about array sizes at all
- The operator == behaves as we expect
- `cin` and `cout` work as well, again, without having to worry about going out of bounds

Another Function

- It should be noted that `cin` will still behave the same as ever
- This means that it will only read until it hits white-space
- We can get around this by using the function `getline()`

```
string name;

cout << "What's your name, man? ";
getline(cin, name);
```

- Note the two parameters. The first is the name of the stream object (more on this when we discuss I/O in a couple weeks) that we want to use to read in the line. The second is the name of the string variable where we are placing what was read

A Pitfall with `getline()`

- It is very common to mix `cin` and `getline()` in a program
- If you do, there is something extremely important to keep in mind
- `cin` and `getline()` treat the `\n` character quite differently
- When using `cin`, after the user presses the Enter key, that keystroke is not discarded
- When using `getline()`, it is
- This different behavior leads to `getline()` appearing to not work when they are used together in the same program
- This is because a call to `getline()` after `cin` will allow `getline()` to see the `\n` that `cin` left behind; `getline()` will immediately think it's done and exit, storing nothing into the string object

How to Avoid This Mess

- Every time we use `cin`, we should follow it with `cin.ignore()`
- The `ignore()` function of `cin` will grab the first character that was left behind, if it exists, and essentially throw it away
- In most cases, this is enough to throw out the `\n` that `cin` usually leaves behind
- This wil allow `getline()` to function as expected

```
int age;
string name;

cout << "What's your age again? ";
cin >> age;
cin.ignore();    // Without this line, getline would
                 // not work as expected
cout << "What's your name, man? ";
getline(cin, name);
```

`string` Objects Are Not So Different

`string` Objects Are Not So Different

- Remember, the `string` class was designed to behave as closely to the basic data types as possible
- So, if we can do a thing with a `double`, we should expect to be able to do the same thing with a `string`
- The `string` class was also designed so that those familiar with C-Strings would not have to learn everything over again
  - One example is the C-String function `strcmp()` has an analog in the `string` class function `compare()`
- Relational operators ($<$, $>$, $<=$, etc.) work with strings
- We are able to have arrays of strings the same as arrays of `int`s
  - An array of C-Strings is possible, but it is conceptually more difficult and more limited

Still an Array of Characters?

- We are able to use the [ ] operators the same as with a C-String

```
string word = "Hello";
cout << word[2] << endl;     // Prints the first 'l'
```

- We are still able to treat a `string` object just like an array of characters
- If we do so, it comes with the same liabilities
- The `string` class provides a much safer method

Array-like Access Without the Hazards

- We can use the `string` class function `at()`

```
string word = "Hello";
cout << word.at(2) << endl;    // Prints the first 'l'
```

- The `string` function `at()` provides the same array-like access, but it does not allow out of bounds access at all
- The [ ] still does, because that is how they would normally behave
- With the `at()` function, a special error called an exception is thrown
- Unless you as the programmer catch and handle the exception, your program will end
- Exceptions will not be covered in this class, so your program will just end

A Few More Functions from <string>

Function	Description
`.substr(`*strtIndex*`, `*numChars*`)`	Returns a sub-string of the calling object; parameters are a starting index and the number of characters to read
`.c_str()`	Converts a `string` object into a C-String; this can be necessary for compatibility reasons, particularly with certain other libraries
`.swap(`*target*`)`	Swaps values of the calling object and the target string