COMP 161 - Lecture Notes - 08 - Strings

Spring 2014

In these notes we take a look at the world of string data in C++. In doing so we're exposed to some basic ideas in working with object classes.

Objects and Classes

We've started to use the term OBJECT to refer to some of the data in or programs, so let's be sure we have a clear sense of what it means. In the most general sense, an object is a value in memory. We often refer to variables as objects because a variable is a named and typed piece of memory. For example, if we initialize the following variables,

```
int x{0};
double pi{3.14};
char let{'x'};
bool isOK{false};
```

Then we've introduced four objects to our program: the integer object *x*, the double *pi*, the character *let*, and the boolean *isOk*. We didn't refer to these as objects at first because their types are fundamental and it's just as easy to think of them strictly in terms of their types.

In the last set of notes we learned about some non-primitive types: *istringstream*, *ostream*, and *istream*. In particular we worked with a few pre-defined objects– *cout*, *cin*, and *cerr*– and initialized some *istringstream* objects of our own. Using the blanket term *object* for this data lets us ignore the underlying complexity of the value. It's not the least be clear what an *istringstream* looks like. More importantly, *we don't need to know what, exactly, it looks like*. We simply need to know how to work with it though operators and procedures.

When we're talking about types that are not built-in, types that are defined in libraries, types like *ostream*, *istream*, and *istringstream*, then we refer to the type as a CLASS. Put another way, a CLASS is synonym for TYPE where calling a type a class implies that the type is user-defined. The term object then gets used to refer to instances of that class, or values of a class in memory.

From the users perspective[1], working with classes is typically about working with procedures for a few general tasks:

- CONSTRUCTORS allow you to build objects from that class
- SELECTORS allow you to access some or all of the data within an object
- MUTATORS allow you to modify some or all of the data within an object

[1] program with the class as opposed to program the class

- QUERIES allow you to learn something about the object and its state

Many of the procedures we'll work with are designed as a part of the class definition itself and are called differently than the usual procedure. We'll often refer to these procedures as CLASS METHODS, or just methods. As we explore the C++ string class, you'll see all of these at work.

Strings

A STRING is a sequence of characters. In every language I know of, string literals are written as characters in double quotes. Like this:

```
"I am a string"
"  so am I "
"12345"
```

In C++, we must learn to navigate two types for string data. One is primitive, built-in and is the type associated with string literals. The other is a string class that has way more batteries included. Thus, or usual problem is we almost always start with some kind of built-in string value when what we want is a string class object.

*C-Strings: char**

The C-String type is unavoidable because any time you type a string literal[2] the compiler sees it as a C-String value– you can't express a string value in your program without stepping on the C-String type. The type annotation for C-Strings is *char**. Anytime you see this type, you're dealing with C-Strings. The name C-String comes from the fact that these values are handled in exactly the same manner as string data in the C programming language. In fact, it's also the way you tend to manage strings in assembly language as well.

[2] characters in double quotes

Until we have a reason to do otherwise, we'll avoid directly working with C-Strings like the plague. It's not that there's anything wrong with them really, it's just that we have other libraries that make doing interesting things with strings much easier than if we stayed with raw C-Strings. We saw this with our CLI based program in the previous lecture notes. The strings coming from the CLI are C-Strings. We immediately fed them to *istringstream* objects as initial values. That library then took care of reading the character sequence as if it were a double.

The C++ string Class

Our preferred string type is the *std::string* class as defined in the string library[3]. We can initialize string variables from string literals:

[3] http://www.cplusplus.com/reference/string/string/

```
string str{"hello"};
string ing{" world!"};
```

Occasionally we want to have a string object independent of an actual variable. This is very useful in testing for example. To do this we use a standard constructor form where the class name is followed by a set of parenthesis with the value inside them. Here we see this in the context of some tests that examine the values of our previously declared string variables:

```
EXPECT_EQ(string("hello"),str);
EXPECT_EQ(string(" world!"),ing);
```

We can, in fact, use gTest's *EXPECT_EQ* test on C++ string data because the class definition includes a definition for string equality. What we cannot do, is compare C++ strings to C-Strings. These tests will give you nothing but compiler errors:

```
EXPECT_EQ("hello",str);
EXPECT_EQ(" world!",ing);
```

As far as the machine is concerned, a C-String, the expected value, and a C++ string object's value, the actual value, are not the same thing. This is subtle, annoying, and will cause you lots of headaches if you don't internalize this ASAP. Logically, we know they're the same. This stupid box we're programming doesn't know the difference a priori.

It is sometimes useful to get the C-String version of a string object. To do this we can using a class method. Let's see this at work and then talk about what we're seeing. Here's the same tests as before but now we're comparing C-Strings using gTest's *EXPECT_STREQ*.

```
EXPECT_STREQ("hello",str.c_str());
EXPECT_STREQ(" world!",ing.c_str());
```

Class methods are invoked using the DOT OPERATOR, .. It's a binary operator whose left-hand operand is an object and right-hand operand is a method defined for that object's class. The object acts as an *implicit* argument to the method. We say that the methods acts on the object with any parameters passed to the method. In the case of *c_str*, there are no arguments so the only real "input" is the object upon which it's called. The *c_str* method could also be written as a single argument procedure that takes a string and returns a C string.

As a class method we say it takes no arguments but acts on a string object. It's a subtle but important difference.

You don't necessarily need variables to invoke class methods[4]. Here we see *c_str* at work on an unnamed string object.

[4] it's the most typical use case, but the dot operator doesn't care if the left operand has a name or not

```
EXPECT_STREQ("wierd",string("wierd").c_str());
```

The string library contains a host of useful methods and operators and you're encouraged to spend some time with the reference to see what's out there[5]. For now, I'll leave you with a few gTests that demonstrate a few of the more useful functional methods and operators and let you connect the dots on those:

[5] `http://www.cplusplus.com/reference/string/string/`

```
string str{"hello"};
string ing{" world!"};

// Size/Length
EXPECT_EQ(5,str.length());
EXPECT_EQ(5,str.size());

// Character Selection
EXPECT_EQ(' ',ing[0]);
EXPECT_EQ('w',ing[1]);
EXPECT_EQ(' ',ing.at(0));
EXPECT_EQ('w',ing.at(1));

// Substring Selection
EXPECT_EQ(string("ell"), str.substr(1,3));
EXPECT_EQ(string("orld"), ing.substr(2,4));
EXPECT_EQ(string(" wor"), ing.substr(0,4));
EXPECT_EQ(string(" ld!"), ing.substr(4));

// Concatenate
EXPECT_EQ(string("hello world!"), str + ing);
```

String class mutators expose us to the world of variable mutation. In order to demonstrate their effect on a string object with tests we must do a pre-post test. One test establishes the state of the variable prior to mutation, the second test is then run after the mutator is called to demonstrate that the desired change has occurred. This technique is import to understand– you'll be writing a lot of tests like this as we start designing procedures for state mutation. The first test you'll see demonstrates the effect of assignment[6] with strings. I'll then make use of assignment to reset variables post-mutation as needed.

[6] =

```
// The assignment operator with C++ and C strings
```

```
string s{""};
EXPECT_EQ(string(""),s); //before
s = string("hello!"); //mutation
EXPECT_EQ(string("hello!") , s); //after .. and before
s = "world";
EXPECT_EQ(string("world") , s);


string str{"hello"};
string ing{" world!"};

// string append
EXPECT_EQ(string("hello") , str);
str.append(ing);
EXPECT_EQ(string("hello world!") , str);
str = "hello"; // reset
str.append("world!"); //works with C strings too
EXPECT_EQ(string("helloworld!") , str);
str = "hello"; // reset

// erase
EXPECT_EQ(string("hello") , str);
str.erase(0,2);
EXPECT_EQ(string("llo") , str);
str = "hello";
str.erase();
EXPECT_EQ(string("") , str);
str = "hello";
str.erase(2,3);
EXPECT_EQ(string("he") , str);
str = "hello";

// Single character assignment and append
EXPECT_EQ(string("hello") , str);
str[0] = 'H';
EXPECT_EQ(string("Hello") , str);
str[2] = 'L';
EXPECT_EQ(string("HeLlo") , str);
str.at(4) = 'O';
EXPECT_EQ(string("HeLlO") , str);
str.push_back('!');
EXPECT_EQ(string("HeLlO!") , str);
```

The last thing we'll look at from the string library is the *getline* procedure. This is a classic example of a Input effect based proce-

dure. The typical use is to read a whole line from an input stream object like *cin*. Typically, input is broken up by white space. Sometimes you want to read several tokens in and stop when you encounter the newline character. For example, if you want to get my name as one string, "Logan Mayfield", you'd either need to read each piece in to a string, then concatenate or append, or use getline. Let's demonstrate[7].

[7] assume my name is typed in the obvious places

```
string first{""};
string last{""};
string name{""};

cin >> first >> last;

EXPECT_EQ(string("Logan",first);
EXPECT_EQ(string("Mayfield",last);

name = first + " " + last;

EXPECT_EQ(string("Logan Mayfield"), name);

name = "";

getline(cin,name);
EXPECT_EQ(string("Logan Mayfield"), name);
```

As an input procedure, *getline* is also a mutator as it must assigned the input value to the second argument, namely a variable. This implies a key issue when working with input and mutation procedures: *you must pass variables*. Something like *getline(cin,string())* might work but is useless. Even if the string object passed as the second argument were modified, we'd have no way of accessing the results.

A Note on Testing Effects

The previous section merits careful study not just because it introduces key string functionality, but because it introduces how to establish and test expectation in our new effect-based world. Our testing regime is critical both for evaluating the correctness of our code and for helping us to establish expected behavior of our code before we even get down to writing it. Writing these tests is a vital tool for software development as well as general problem solving.

When we test functions, we simply compare the actual function return value to the expected value. Effects are not about things returned but about change to the system. We need to know if a variable's stored value did or did not change. We need to know if input

from a stream was read correctly. We need to know if the characters written to an output stream were written properly.

String Streams

The string stream library[8]. Provides us with classes for using C++ strings as streaming I/O devices. At first glance this might seem strange because string objects and string data are obvious I/O devices like stdout and stdin. However, all data printed as output ends its journey as character data and data read as input begins as character data; the whole I/O system is built around working with sequences of characters, i.e. strings.

[8] #include<sstream>

Using strings as streams is, in a way, like ignoring the output destination or input source. Streaming data to string let's you get the character sequence that you'd like to send to *cout* prior to actually sending it. Conversely, streaming input from a string is a bit like having the characters from standard in first saved to a string for you to then read. The important thing here is that we can use these streams to mock up what happens at the standard output and standard input in such a way that we can test our I/O with frameworks like gTest.

The other use for string streams, in particular input string streams, that we've already seen is using the library to manage the conversion from character sequences to other types and vice versa. When you have a string that looks like a double value and you really want that double value, the string streaming lets you use the I/O library for reading the string to a double. The new C++11 additions to the string library actually provide functions for this kind of thing[9], but we'll focus on the streams as we'll need to get comfortable with them for I/O testing purposes.

[9] see *stoi* and *stod* and *to_string*

The *istringstream* type lets us read data from strings in the same fashion that we do from the standard input stream *cin*. We can initialize istringstream objects from C and C++ strings. Here are few demonstrations:

```
istringstream istr{"Hello 1345 3.4 a"};
string s{""};
int x{0};
double d{0.0};
char c{'\0'};

istr >> s >> x >> d >> c;
EXPECT_EQ(string("Hello") , s);
EXPECT_EQ(1345 , x);
```

```
EXPECT_DOUBLE_EQ(3.4,d);
EXPECT_EQ('a', c);
```

The *ostringstream* types lets us write data to a string in the same fashion we do from the standard output stream *cout*. You can initialize the stream with some data, but it's probably more likely that we'll start with an empty string stream and fill it with data. To extract the string we use the class method *str*.

```
ostringstream sout{""};

sout << "Hello" << 1345 << 3.4 << 'a';

EXPECT_EQ(string("Hello13453.4a"), sout.str() );
```

Let's quickly return to our getline example to see how we can have testable I/O. To test getline working with *cin* you physically have to type input. With a string stream you can fill the stream and then test the I/O. All of this can be done by the computer and requires zero human intervention at the time of testing.

```
string first{""};
string last{""};
string name{""};

istringstream sin{"Logan Mayfield"};
sin >> first >> last;

EXPECT_EQ(string("Logan"),first);
EXPECT_EQ(string("Mayfield"),last);

name = first + " " + last;

EXPECT_EQ(string("Logan Mayfield"), name);

name = "";
istringstream sin2{"Logan Mayfield"};

getline(sin2,name);
EXPECT_EQ(string("Logan Mayfield"), name);
```