COMP 161 - Lecture Notes - 09 - Strings, Objects, and Arrays

Spring 2014

In these notes we look at Object types by way of the C++ string class.

Strings

A string is a sequence of zero or more characters. Any arbitrarily long sequence can be viewed as recursive object[1]:

[1] think Racket lists

```
A string is:
---------------
 - the empty string ""
 - a single char value followed by another string
```

An alternative way of viewing a string is an INDEXED COLLECTION of characters. From this perspective we think of the string as a kind of RANDOM ACCESS collection where each individual character is just as easy to access as any other character[2] Access is made by way of the integer index values. For a string of length n, the first character in the collection is indexed with 0 and the last with $n-1$.

[2] Notice this is very different then our recursive structure where we only have easy access to the first character.

The C++ *string* library provides a string type that allows the program to work from either perspective.

Classes and Objects

Non-primitive data types are defined as CLASSES in C++. In general, a literal value from a class is called an OBJECT. Primitive types have clearly defined sets of operators where classes provide a set of METHODS and possibly operators. A *method* is a special type of procedure defined along side the class and only usable with an object or variable of that class type.

Objects and Variables

Let's look at the *string* class as an example[3]. String literals are an exception to rule in that they have a special literal syntax. The following are string essentially literals:

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

```
""
"a string"
"hello"
"a"
```

I say essentially because the compiler attributes the type *char ** to these values, and not *string*. In many[4] cases, the two types can be

[4] most?

used inter-changeably, but you should be aware of the difference. If you want a proper *string* type literal, then you'd do the following:

```
string("")e
string("a string")
string("hello")
string("a")
```

This syntax commonly used for object literals. This syntax actually invokes a special procedure called the CONSTRUCTOR. For C++ classes, constructors are named after the class. So the procedure named *string* is one of the string type CONSTRUCTORS[5]. These examples show the constructor that creates a *string* type value from a *char ** type value. Another important constructor is the COPY CONSTRUCTOR that constructs a string from a string.

[5] like *make-posn* from Racket

```
string(string("hello"))
```

More often then not, we'll need string variables to store our string values. Declaring class-based variables is the same as primitive types. Un-initialized declaration looks like this:

```
string s;
string t;
```

We have several options for initializing our variables. The preferred syntax is this:

```
string s("hello");
string t(s);
```

The first syntax uses the *char ** constructor to build string s. The second uses the COPY CONSTRUCTOR to build *t* as a copy of the string *s*. We could also initialize our variable with the following syntax:

```
string s = "hello";
string t = s;
string u = string("hello");
```

These forms are less preferred as they involve the assignment operator. They'll work just fine for *string* types, but when we write our own classes, we'll have to write our own assignment operator to ensure that these work as intended. For now, you should understand that *for strings*, all of these initialization forms have the same *effect* but they achieve this effect in slightly different ways. These differences can, for non-string types, mean different effects.

Class Methods

We know how to express string literals, declare string variables, and initialize string variables. So far, things look more or less like they did with primitive types. Now, we'll make a departure. Classes almost always provide a set of class methods. These methods are special procedures meant to be invoked with an object from that class. Rather than passing that object to the method, we use the DOT OPERATOR to invoke the method *with respect to the object*.

Let's look at the string *length* method that returns the length of a string. If it were a procedure, then we'd expect something like this[6]:

[6] as gTest statements

```
// string literal
EXPECT_EQ(3 , length(string("dog")));

// string variable
string s("dog");
EXPECT_EQ(3,length(s));
```

However, length[7] is a class method, so we do this:

[7] http://www.cplusplus.com/reference/string/string/at/

```
EXPECT_EQ(3 , string("dog").length());

string s("dog");
EXPECT_EQ(3,s.length());
```

The *dot operator* effectively calls the class method to the left of the dot on the object or variable to the right of the dot. In the first example we used a string literal[8]. It is helpful to think of the object value to the left of the dot as the first[9] parameter of the method to the right of the dot.

[8] note that "dog".length() will not work as "dog" is of type *char ** not *string*

[9] or implied

Now lets consider the methods that allow us to work with strings as a collection, namely the methods that let us select one or more characters from the string. These are interesting methods as they require parameters in addition to the object. First, *at*[10], which selects by index.

[10] http://www.cplusplus.com/reference/string/string/at/

```
EXPECT_EQ('a',string("at").at(0));
EXPECT_EQ('t',string("at").at(1));
EXPECT_EQ('t',string("at").at( string("at").length() - 1 ) );

string s("dog");
EXPECT_EQ('o',s.at(1));
EXPECT_EQ('g',s.at(2));
```

The *substr*[11] method is a general purpose selector that allows you to get part of a string as a string.

[11] http://www.cplusplus.com/reference/string/string/substr/

```
EXPECT_EQ("a",string("at").substr(0,1) );
EXPECT_EQ("at",string("at").substr(0,2) );
EXPECT_EQ("ello",string("hello").substr(1) );
EXPECT_EQ("ello",string("hello").substr(1, 4) );
EXPECT_EQ("ello",string("hello").substr(1, string::npos) );

string s("daydream");
EXPECT_EQ("dre",s.substr(3,3) );
EXPECT_EQ("daydrea",s.substr(0,s.length()-1) );
```

Overloaded Operators

In order to allow programmers to use objects like primitive values, C++ allows class designers to define operators[12] for their classes. One of the most important operators for the string type is *operator[]*[13] that allows for indexed based selection. This syntax is the most commonly used syntax in programming for element selection from index collections.

[12] called operator overloading

[13] http://www.cplusplus.com/reference/string/string/operator[]/

```
EXPECT_EQ('a',string("at")[0]);
EXPECT_EQ('t',string("at")[1]);

string s("dog");
EXPECT_EQ('o',s[1]);
EXPECT_EQ('g',s[2]);
```

We read the expression $s[1]$ as "s at 1". Also note that while C++ refers to the operator as *operator[]*, the actually operator only involves the []. For more examples, check out the relational operators for string comparison[14].

[14] http://www.cplusplus.com/reference/string/string/operators/

String Mutators

The string class provides mutators methods as well as accessor methods, so string variables are mutable. The *push_back* method adds a single character to the end of a string.

```
string s("do");
EXPECT_EQ(2,s.length());
EXPECT_EQ("do",s);

s.push_back('g');

EXPECT_EQ(3,s.length());
EXPECT_EQ("dog",s);
```

It is also possible to directly mutate individual characters through the use of single character assignment. Both the *at* method and *op-*

erator[] provide direct references to individual characters within the string.

```
string s("dog");

EXPECT_EQ("dog",s);
s[0] = 'h';
EXPECT_EQ("hog",s);

s = "bat";
EXPECT_EQ("bat",s);
s.at(2) = 'g';
EXPECT_EQ("bag",s);
```

This type of character level manipulation is indicative of treating a string as an *array of char* values. An array type is the low-level C++ implementation of indexed collections. When we treat a string as an array, we tend to ignore the higher-level string logic and focus on the fact that it's a collection of chars.