Eric Roberts
CS 106B

Handout #8
April 6, 2009

Strings and Recursive Functions

Strings in C++ Recursive Functions

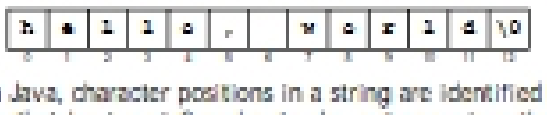
Eric Roberts
CS 106B
April 6, 2009

Data Types: A Pedagogical Experiment

- Chapter 2 of the reader introduces the primitive types in C++ along with the mechanisms that languages have traditionally offered to create compound types from primitive ones.
- In keeping with the overall strategy of showing you how to use data structures before you see their implementation, I'm going to see whether it is possible to postpone the discussion of the Chapter 2 material until **after** I need it to implement the various classes introduced in the later chapters. Once you have the class-based tools from Chapter 4, you won't have much occasion to use the low-level structures anyway.
- The only problem on Assignment #1 that requires data structures at all is the histogram problem, which needs an array to keep track of the scores in each range. If you're reading ahead, feel free to use the `Vector` class introduced in Chapter 4 instead. Otherwise, read enough of section 2.4 to see how to declare such an array.

C Strings

- Almost any program that you write in any modern language is likely to use string data at some point, even if it is only to display instructions to the user or to label results.
- Conceptually, a string is just an array of characters, which is precisely how strings are implemented in the C subset of C++. If you put double quotation marks around a sequence of characters, you get what is called a **C string**, in which the characters are stored in an array of bytes, terminated by a **null byte** whose ASCII value is 0. For example, the characters in the C string `"hello, world"` are arranged like this:



- As in Java, character positions in a string are identified by an **index** that begins at 0 and extends up to one less than the length of the string.

Strings as an Abstract Data Type

- Because C++ includes everything in its predecessor language, C strings are a part of C++, and you will occasionally have to recognize that this style of string exists.
- For almost every program you write, it will be far easier to use the C++'s `string` class, which implements strings as an **abstract data type**, which is a type defined by its behavior rather than its representation.
- The methods C++ provides for working with strings are often subtly different from those in Java's `String` class. As I emphasized on Friday, however, most of these differences fall into the **accidental** category.
- The only **essential** difference between strings in C++ and Java is that C++ allows clients to change the individual characters contained in a string; by contrast, Java strings are **immutable**, which means that they never change once they are allocated.

Calling String Methods

- Because `string` is a class, it is best to think of its methods in terms of sending a message to a particular object. The object to which a message is sent is called the **receiver**, and the general syntax for sending a message looks like this:

```
receiver.name(arguments);
```

- For example, if you want to determine the length of a string `s`, you might use the assignment statement

```
int len = s.length();
```

just as you would in Java.

- Unlike Java, C++ allows classes to redefine the meanings of the standard operators. As a result, several string operations, such as `+` for concatenation, are implemented as operators.

The Concatenation Operator

- As many of you already know from Java, the `+` operator is a wonderfully convenient shorthand for **concatenation**, which consists of combining two strings end to end with no intervening characters. In Java, this extension to `+` is part of the language. In C++, it is an extension to the `string` class.
- In Java, the `+` operator is often used to combine items as part of a `println` call, as in

```
println("The total is " + total + ".");
```

In C++, you would achieve the same result by writing

```
cout << "The total is " << total << "." << endl;
```

- Although you might imagine otherwise, you can't use the `++` operator in this statement, because the quoted strings after C strings and not `string` objects.

Operators on the `string` Class

`str[i]` Returns the *i*th character of `str`. Assigning to `str[i]` changes that character.
`s1 + s2` Returns a new string consisting of `s1` concatenated with `s2`.
`s1 = s2;` Copies the character string `s2` into `s1`.
`s1 += s2;` Appends `s2` to the end of `s1`.
`s1 == s2` (and similarly for `<`, `<=`, `>`, `>=`, and `!=`) Compares to strings lexicographically.

Common Methods in the `string` Class

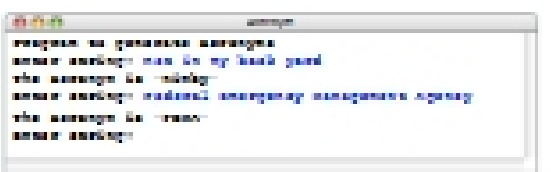
`str.length()` Returns the number of characters in the string `str`.
`str.at(index)` Returns the character at position `index`; most clients use `str[index]` instead.
`str.substr(pos, len)` Returns the substring of `str` starting at `pos` and continuing for `len` characters.
`str.find(ch, pos)` Returns the first index ≥ `pos` containing `ch`, or `string::npos` if not found.
`str.find(text, pos)` Similar to the previous method, but with a string instead of a character.
`str.insert(pos, text)` ⟸ Destructively changes `str` Inserts the characters from `text` before index position `pos`.
`str.replace(pos, count, text)` ⟸ Destructively changes `str` Replaces `count` characters from `text` starting at position `pos`.

Exercise: String Processing

- An **acronym** is a word formed by taking the first letter of each word in a sequence, as in

  `"self contained underwater breathing apparatus"` → `"scuba"`

- Write a C++ program that generates acronyms, as illustrated by the following sample run:



Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function introduced in Chapter 5, which can be defined in either of the following ways:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = \begin{cases} 1 & \text{if } n \le 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

```
int Fact(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * Fact(n - 1);
   }
}
```

The Recursive Paradigm

- Most recursive methods you encounter in an introductory course have bodies that fit the following general pattern:

```
if (test for a simple case) {
   Compute and return the simple solution without using recursion.
} else {
   Divide the problem into one or more subproblems that have the same form.
   Solve each of the problems by calling this method recursively.
   Return the solution from the results of the various subproblems.
}
```

- Finding a recursive solution is mostly a matter of figuring out how to break it down so that it fits the paradigm. When you do so, you must do two things:
  1. Identify **simple cases** that can be solved without recursion.
  2. Find a **recursive decomposition** that breaks each instance of the problem into simpler subproblems of the same type, which you can then solve by applying the method recursively.

Exercise: A Recursive GCD Function

One of the oldest known algorithms that is worthy of the title is Euclid's algorithm for computing the greatest common divisor (GCD) of two integers, *x* and *y*. Euclid's algorithm is usually implemented iteratively using code that looks like this:

```
public int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that the greatest common divisor of *x* and *y* is also the greatest common divisor of *y* and the remainder of *x* divided by *y*.