Lab 3
Strings

15 September 2015

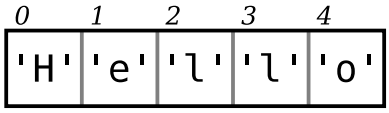
The drill for this lab is given below, after a short reading on how strings work and some of the operations we can do on strings. Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

String info

There's not really any single good place in our textbook that covers string internals (at least, not that would be accessible this early in the term), so I want to present some of them here:

- Strings are made up of characters; each element of a particular `string` value is itself a `char` value.
- The position of each character in the string is called its "index" within the string. Some people use "indexes" as the plural of "index" and others use "indices"; you'll probably hear me use both.
- An index represents how far past the first character you need to go, so the first character is actually at index 0 (because if you're at the first character you don't need to go any further). The character at index 1, one past the first, is the second element of the string, and so on.
- The length of the string is the actual number of characters in it; the last index is thus always one less than the length.

Here's an illustration of the five-character string `"Hello"`, with indices marked:



And, a selected list of things you can do to or with strings, a couple of which you've already seen and several that are new. All the expressions in the table below assume you have a `string` variable named `s` that's already got a value, but they'll work with *any* variable that's a `string` (hopefully, in actual code, with a more descriptive name than `s`).

`s = "foo"`	assigns the value `"foo"` to be the new contents of the string variable—works with any `string` after the equal sign
`s == "foo"`	determines whether the string is the same as the value `"foo"`—works with any `string` after the double-equal sign
`s + "foo"`	computes the string that would result from concatenating the string `s` with the string `"foo"`—works with any `string` after the plus sign
`s[2]`	retrieves the character at index 2 (that is, the third character)—works with any non-negative `int` expression in the square brackets, as long as it's not past the end of the string
`s.front()`	retrieves the first character in the string
`s.back()`	retrieves the last character in the string
`s.length()`	retrieves the number of characters in the string
`s.substr(2,3)`	computes the portion of the string that starts with the character at index 2 and continues for a total of 3 characters—works with any non-negative `int` expressions instead of 2 and 3
`s.find('x')`	computes the index of the first occurrence of the character `'x'` in the string—works with any `char` expression inside the parentheses. If the given character is not in the string, returns the special number `string::npos` (so you can ask if the `s.find('x') == string::npos` and if it does, then `'x'` was not found in the string)

String drill

There's no drill in the book on this topic, but this part of the lab is philosophically similar to an end-of-chapter drill: it's somewhat contrived but lets you write a short program to practice the basics.

Make your directory for this lab, and start its its `README.TXT` file. (From here on out, I'm not going to give you a checklist of what-all to include in your readme, because you have done a few and know what belongs in them, but I do always expect you to have one, and it should reflect the actual instructions and documentation for how to work with your code.)

The first program in this week's lab will be written in a file called `stringinfo.cpp` and it will read a single string—specifically, it will read one string that doesn't have spaces in it—and print out information about it.

1. First, write the program to read the string and print out the message

   ```
   The string "______" has _ characters.
   ```

   except with the blanks filled in correctly for that string: the string itself and its length (how many characters it has). Note that to include a double-quote inside a double-quoted string, you have to use `\"` for the quote inside the string (as with `\n`, the backslash indicates that something special is happening).

2. Add second line to the output that says

   ```
   Its first is '_' and its last is '_'.
   ```

   filling in the blanks with the first and last characters of the string (you can assume it will not be empty).

3. In cases where the string is at least three characters long, add a third line

   ```
   If you trim the first and last characters it leaves "____".
   ```

   The blank here should have the whole string *except* for its first and last characters.

4. Finally, in cases where there is a hyphen in the string, make it print a last line

   ```
   It has a hyphen at index _.
   ```

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about dotfiles). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, continue on to the next section.

Command line feature of the day: Dotfiles

From your home directory, type

```
ls -a
```

It should list a bunch of files you don't normally see that start with a period. Many command line programs have a number of configuration options that the user can set to control how they appear and how they behave. To make these config files easily accessible, they were put in the user's home directory; and to make them unintrusive, they were named by convention with names beginning with periods, which `ls` doesn't normally list. Such config files are popularly known as "dotfiles".

Let's start by doing some things to configure Vim, since we spend so much time using it. Edit (using vim, of course) the `.vimrc` file. (It probably doesn't exist yet unless I've already help you set it up.) Add the following lines to it:

```
set number
syntax on
```

You'll be typing configuration options into this file, and then seeing what happens when you edit your C++ code and other files; this will be easiest to see if you open a second terminal window, and in the left one edit the `.vimrc` (in your home directory) and in the right one edit your C++ files (in one of your lab directories).

Right away, when you edit a C++ file you should see a difference: the syntax highlighting and the line numbers that you've seen when I edit code.

Now, in the `.vimrc` file, add the line

```
set laststatus=2
```

and save it. Now, edit a C++ file. Do you see the difference?

(Go ahead, look for it.)

The change is that there is a status bar on the (second-to-) last line of the window which contains the name of the file (useful!) and, once it has been edited, the symbol `[+]`.

Here are a few other things to add to your `.vimrc`:

```
set cindent shiftwidth=2
set showmatch
```

Try adding them, and see if you can figure out how they change vim's behaviour when editing a C++ file. We'll talk about them in class tomorrow, and you can see if you guessed right.

One last one: PuTTY defaults to a terminal window with dark background and white letters; and you may notice that this makes Vim's syntax colouring a little hard to read. If so, you can add

```
set background=dark
```

to your .vimrc to improve the contrast there. (The default colour scheme is designed for light backgrounds; you can ask for it explicitly as "`set background=light`" if you prefer it.)

Another dotfile that you probably already have is a `.bashrc`, which controls how your command-line shell runs. (The name of the shell is "`bash`", which stands for "Bourne-again shell", a very geeky joke whose explanation you can google for yourself.) From your home directory, edit this file.

If you have one already, it probably has a lot of stuff in it. Some other time you might want to read through it all, but for now I'll just point out one small thing you can edit: your prompt. That's the text that is printed after every command you execute. For now, make it something simple; go to the bottom of the file and insert

```
PS1="Type something: "
```

Inelegant, but we'll fix it in a minute.

Save and quit the editor. Now, the changes haven't taken effect yet, because your current command-line shell was started before you changed the `.bashrc`. One way to check how the changes went is to open a new terminal window—this will make a new shell, using the new settings. Another is to force the `.bashrc` to be reread by typing

```
source .bashrc
```

Not happy with the prompt? Let's change it again. Edit the `.bashrc` file again, and this time change the prompt to

```
PS1="\h \w -\#-$ "
```

(That's approximately what I use.) This gives you, respectively, the name of the machine, the directory you're currently in, and the "command event" number, i.e. a running count of commands you've executed in this shell. You probably don't want to forget the space before the double quote.

Still not happy? There are lots of things you can put into your prompt. In the manpage for `bash` (way down around line 2700—type "2700" and hit enter to scroll down that far in the manpage) there's a list of them, including username, time, date, and other formats for the current directory. Feel free to tweak it (but don't spend too much time on it right now).

One last thing to add to the `.bashrc` file. Some of you have asked how to make the `ls` be highlighted like mine is. You can ask for it explicitly by typing `ls --color`, and you can say "always run that whenever I type `ls`" by adding the following to the end of your `.bashrc` file:

```
alias ls='ls --color'
```

Then all the executables should be in green and all the directories in blue.

Back to strings: testing

In this next part of the lab, we'll revisit the drill and think about how to effectively test it. We'll start planning our test cases on paper. Here, write down a string that's at least five characters long (no spaces or hyphens; it can be your name or anything else with at least five characters):

Now, write out both lines of the response the program would give to that input string:

Go ahead and create a file `stringinfo1.in` that contains the word, similar

to what we did last week (except this time, the entire file is just one line with a single string in it). This week, we'll go a little bit further in automating our testing, and write the expected output in a file as well: create a file `stringinfo1.expect` that contains the two expected lines of text output. This pair of files constitutes your first test case.

Last week, we saw that we could run a program this way:

```
./a.out <stringinfo1.in
```

But as I said then, redirection is more powerful than just that. We can also redirect output—instead of printing to the screen, the output can go someplace else. Such as a file:

```
./a.out  <stringinfo1.in  >stringinfo1.out
```

(the spacing is not very important, but notice the direction of the arrows). When you run that, it won't show any output at all, but it will create the file `stringinfo1.out`, which contains everything sent to `cout`. If you type

```
cat stringinfo1.out
```

you can see it. More importantly, you can use a program called `diff` to compare it to what was expected:

```
diff stringinfo1.expect stringinfo1.out
```

If they're the same, it says nothing, but if they are *at all* different, including spelling or *even just whitespace*, then `diff` will report a difference, and show you the lines from each one that differ. (Any line that matches is omitted from the output.)

Here's why that's super-nifty: as you add more test cases, if you run a bunch of tests at once, it will only show you if and when any of the tests failed.

More test cases

As we started to talk about last week, we want our test cases to have *coverage* with respect to the problem. Without going overboard, we want to have a test case for each meaningfully different input scenario. What's meaningfully

different? Well, changing a character in an input might affect the output, but only incrementally. On the other hand, if we have a short string of just two characters for the input, that makes this program act significantly differently (omitting a line!), so that's worth testing. Pick a two-letter string and write it in the first row of the Input column below.

Add more meaningfully different inputs to the table, along with comments explaining what case they test. (How many do you need to test all the cases?) Then, create the `.in` files with the words you wrote in the Input column, and corresponding `.expect` files with the expected output.

Test case	**Input**	**Comment**
`stringinfo2`		Less than three characters

In general, some version of a table like this is a nice feature for a readme: the Input column can be omitted there, but if you list the name of each case along with a comment about what it's testing, that will help you keep track of it and will also help me figure out what you're trying to do.

Filename utility

In general, Linux and Mac filenames can be broken down as follows:[1]

dirname. The part of the filename that indicates the directory it's in, or if the directory is not explicit in the filename, a single period `"."`.

[1] Windows is similar, but with slight differences.

extension. The part of the filename after the dot, if any. (The dot is part of the extension; but some filenames have no extension at all.)

basename. What's left over if you strip out the dirname and the extension.

For instance, a filename like

```
lab1/hello_world.cpp
```

has a dirname of `"lab1"`, a basename of `"hello_world"`, and an extension of `".cpp"`.

Write a program called `fileinfo` that reads in a single filename and prints out the parts of that filename. Note that while some filenames may have no extension, every filename has a dirname—if there are *no* slashes in the filename, the file must be in the current working directory, so the dirname is reported as a single period (`"."`).

As you program this, carefully reread the description of dirname, extension, and basename, and make sure you cover all the possible situations; also make sure you *test* all the possible situations. In fact, the points for having good test case coverage are available even if your program doesn't pass all of your own tests—it shows me that you understood the cases, even if you didn't get the program 100% working. If that happens, say so in the readme.

It's due as usual on Monday at 4pm. Hand in using the usual command, this time with assignment name `lab3`.

Rubric

RUBRIC (Tentative)

1 Attendance at lab with drill done or question written down
1 Readme present and has the usual stuff

Drill (stringinfo)

1 Program compiles and runs, reads string, prints it out
1 Uses `if`, `length` correctly
1 Uses `substr` correctly
1 Uses `find` correctly, including comparing result to `string::npos`
1 Test cases for stringinfo, good coverage *

fileinfo

1 Handles case where dirname and extension are both present
1 Handles all cases correctly
1 Test cases for fileinfo, good coverage *

* To get each full point for good test case coverage, you have to EITHER pass all test cases in the group OR indicate in the readme which ones aren't passing. This is your way of showing me that you've actually run your tests.

Those comments are related to, but separate from, your rubric check. Don't forget to claim your rubric check bonus!