Practicum 4 – Function Pointers & More

Assignment Details

Assigned: February 11th, 2013. Due: February 13th, 2013 at midnight.

Background

This assignment will provide some direct experience with function pointers and some more information about the build tools available on Linux.

The Practicum

Ok, to get started, log in to your multilab account as normal.

Then we're going to go into the folder you checked out last week – if you called it "`main_copy`" like in the instructions from last week, "`cd main_copy`" will get you there.

Then type "`svn update .`", enter your password, and you should see something like this:

```
bloodroot:~> cd main_copy
bloodroot:~/main_copy> svn update .
Authentication realm: <svn://progit.netlab.uky.edu:3690> UKCS Source
Control
Password for '216sec001':

-----------------------------------------------------------------------
ATTENTION!  Your password for authentication realm:

   <svn://progit.netlab.uky.edu:3690> UKCS Source Control

can only be stored to disk unencrypted!  You are advised to configure
your system so that Subversion can store passwords encrypted, if
possible.  See the documentation for details.

You can avoid future appearances of this warning by setting the value
of the 'store-plaintext-passwords' option to either 'yes' or 'no' in
'/u/zon-d2/ugrad/dbbrow00/.subversion/servers'.
-----------------------------------------------------------------------
Store password unencrypted (yes/no)? no
A    CS216/216sec001/la3
A    CS216/216sec001/la3/quick.cpp
Updated to revision 1933.
bloodroot:~/main_copy>
```

What just happened was an `svn update` operation. Instead of checking out a new copy, we just updated our current copy to what's sitting on the server – most notably; it added the two folders for this week's assignments, and a .cpp file inside one of them.

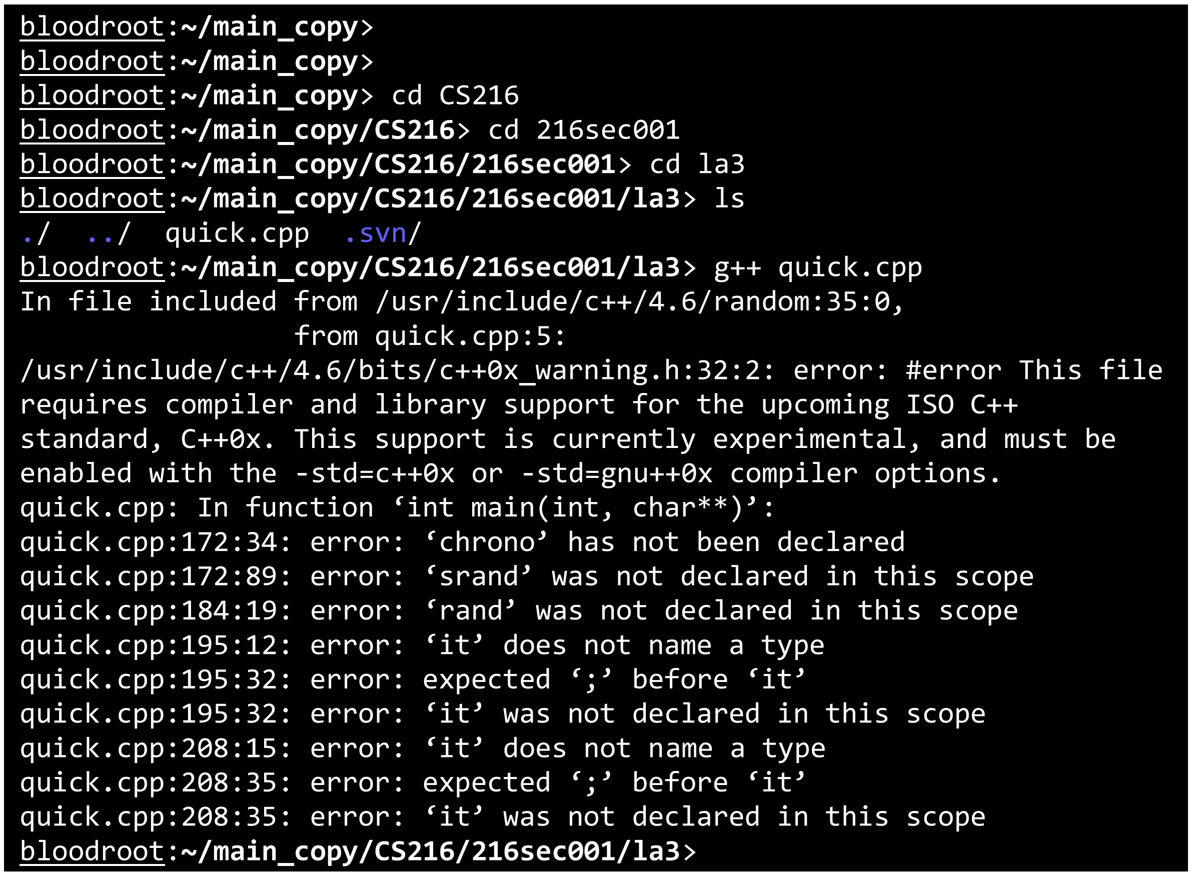
(If the `svn update` failed the easiest thing to do is just check out a new copy; unfortunately, `svn` on multilab can be a bit flakier than Tortoise, if you're used to that client…)

Now, navigate to the `la3` folder (via the `cd` command), this is where we're going to be working today.

The `quick.cpp` file given contains a working (and thoroughly – very thoroughly – commented) implementation of quick sort. Before we get started changing it, let's compile it, and give it a test run.

```
g++ quick.cpp
```

Is the command to compile (no `-c` option, as the file does contain a `main`, and we're trying to compile it to a program). Run that, and you should get:



```
bloodroot:~/main_copy>
bloodroot:~/main_copy>
bloodroot:~/main_copy> cd CS216
bloodroot:~/main_copy/CS216> cd 216sec001
bloodroot:~/main_copy/CS216/216sec001> cd la3
bloodroot:~/main_copy/CS216/216sec001/la3> ls
./  ../  quick.cpp  .svn/
bloodroot:~/main_copy/CS216/216sec001/la3> g++ quick.cpp
In file included from /usr/include/c++/4.6/random:35:0,
                 from quick.cpp:5:
/usr/include/c++/4.6/bits/c++0x_warning.h:32:2: error: #error This file
requires compiler and library support for the upcoming ISO C++
standard, C++0x. This support is currently experimental, and must be
enabled with the -std=c++0x or -std=gnu++0x compiler options.
quick.cpp: In function ‘int main(int, char**)’:
quick.cpp:172:34: error: ‘chrono’ has not been declared
quick.cpp:172:89: error: ‘srand’ was not declared in this scope
quick.cpp:184:19: error: ‘rand’ was not declared in this scope
quick.cpp:195:12: error: ‘it’ does not name a type
quick.cpp:195:32: error: expected ‘;’ before ‘it’
quick.cpp:195:32: error: ‘it’ was not declared in this scope
quick.cpp:208:15: error: ‘it’ does not name a type
quick.cpp:208:35: error: expected ‘;’ before ‘it’
quick.cpp:208:35: error: ‘it’ was not declared in this scope
bloodroot:~/main_copy/CS216/216sec001/la3>
```

Oops.

Well, close enough to working – if you open up the file, and go to the line it complains about – line 100 – you'll see that there's a use of the `auto` keyword. As `auto` is a relatively new addition to C++, `g++` (or, at least, the version on the multilab) does not, by default, support it. So we'll have to modify the command line a bit, and use the `--std=c++0x` option to enable some of the advanced features:

```
g++ --std=c++0x quick.cpp
```

Executing that, and then running the resulting executable, you should get something like this (noting that each run of the program randomizes the starting vector!):

```
bloodroot:~/main_copy/CS216/216sec001/la3> g++ --std=c++0x quick.cpp
bloodroot:~/main_copy/CS216/216sec001/la3> ./a.out
String vector before:
  ubpwk
  ceczq
  mhlon
  urbpa
  fjnir
  rxpqy
  plaeh
  mgnoh
  farqq
  gljha
  lnlzx
  eqwtj

String vector after:
  ceczq
  eqwtj
  farqq
  fjnir
  gljha
  lnlzx
  mgnoh
  mhlon
  plaeh
  rxpqy
  ubpwk
  urbpa
bloodroot:~/main_copy/CS216/216sec001/la3>
```

Now, as long as we're talking about new `g++` command line options, let's add the `-o` (for output) option. Calling this specifies what the output of the program should be called. So let's modify the command to:

```
g++ --std=c++0x quick.cpp -o quick
```

Which will instead of an executable named `a.out`, will give us an executable named `quick`:

```
bloodroot:~/main_copy/CS216/216sec001/la3> g++ --std=c++0x quick.cpp -o
quick
bloodroot:~/main_copy/CS216/216sec001/la3> ./quick
String vector before:
  gzsha
  vnrmy
  zwttk
  bcelp
  lfwml
  oaapp
  pvohf
  oesir
  qjplc
  bogfz
  wsgsf
  rihuy

String vector after:
  bcelp
  bogfz
  gzsha
  lfwml
  oaapp
  oesir
  pvohf
  qjplc
  rihuy
  vnrmy
  wsgsf
  zwttk
bloodroot:~/main_copy/CS216/216sec001/la3>
```

Ok, now for the actual programming part!

So, we've got a block of code that performs a quick sort on a vector of strings, and sorts based on their relative position in alphabetical order.

Now, let's make it a bit more useful – let's generalize it to sort based on a condition we pass it – a function pointer!

What we're going to do is modify both the `quickSort` and `quickSortInternal` functions to take another parameter – a function pointer. That function pointer will point to a function with a `bool` return type and two strings as parameters – the function should return true if the first string passed should occur earlier in the sorted vector than the second string.

Then, you should implement three functions (all of these will be very short) to handle the comparisons – one should use the default behavior already implemented in `quick.cpp` (i.e., sorted in alphabetical order), the second should sort the vector in reverse alphabetical order (which should have one character difference in the body of the function from the first…). The third should sort the vector alphabetically by the last character in the string[1]

After adding these functions, update the main function to output the initial random vector after sorted all three ways.

Once you're done with that, one more step – there's a neat feature in the command line that lets us redirect the output of a program to a file. If you execute a command on the command line, and follow it up with a `>` and then a file name, Linux will create the file and put the output of the program in it.

So, after you finish the program (and assuming you compile it to a file named `quick`), do:

```
./quick > output.txt
```

Which should generate a file in the folder named `output.txt`.

Requirements

- An updated `quick.cpp` with the changes detailed above
- An `output.txt` generated from running the compiled program
- Remember to call `svn add` on your output file and `svn commit` once you're done with everything!
- The answer to the following questions, saved in a file called `reflection.txt`

Reflection Questions

Take a look at the quick sort presented in the practicum as well as the merge sort we covered in class on Monday (and posted on the class web site).

Given that quick sort has the same average case performance but worse worst case performance, why might someone prefer quick sort to merge sort? What does the merge sort implementation do that the quick sort implementation not do?

Note: For this exercise, "it takes less lines of code to implement" is not a good answer.

[1] You are welcome to do it in alphabetical order based on a reversed string, but given time constraints in lab, just sorting by the last character is fine.

Notes

- Check the class web site for code with examples of function pointer syntax. As I've said before, the C-syntax for function pointers is ***horrible***. I mean, so bad bold and italics aren't enough, I feel the need to use Comic Sans to properly express the badness of their syntax. Instead, I suggest using the C++11 syntax…
- For reference, declaring a function pointer named pFunc:
  - C-style: `bool (*pFunc)(string, string);`
  - C++11-style: `function<bool(string,string)> pFunc;`
  - (note that the C++11 version requires `#include <functional>`)
- The syntax for declaring function pointer parameters is just like for declaring function pointer variables.
- You can use either C style or C++ style function pointers.
- Don't overthink this assignment. Each of the three functions you have to write should only take one line, and there's only one place where you have to actually *call* the function pointer – but remember to make sure you pass the function pointer from function to function.
- `vector`s have a method `back`, which returns the element on the end of the `vector`. `string`s, as they are *similar to* a `vector` of `char`s, share this method.

Example output

```
bloodroot:~/main_copy/CS216/216sec001/la3> g++ --std=c++0x quick.cpp -o
quick
bloodroot:~/main_copy/CS216/216sec001/la3> ./quick
String vector before:
  dgeeo
  lztgs
  qbeor
  ykbzs
  xstpu
  rsbbw
  rhcyl
  qjmls
  fbtlq
  kjcoj
  uldoc
  xfuzg

String vector after:
  dgeeo
  fbtlq
  kjcoj
  lztgs
  qbeor
  qjmls
  rhcyl
  rsbbw
  uldoc
  xfuzg
  xstpu
  ykbzs

String vector after reverse sort:
  ykbzs
  xstpu
  xfuzg
  uldoc
  rsbbw
  rhcyl
  qjmls
  qbeor
  lztgs
  kjcoj
  fbtlq
  dgeeo

String vector after last char sort:
  uldoc
  xfuzg
  kjcoj
  rhcyl
  dgeeo
  fbtlq
  qbeor
  ykbzs
  lztgs
  qjmls
  xstpu
  rsbbw
bloodroot:~/main_copy/CS216/216sec001/la3>
```