Introduction

This lab explores programming problems related to Chapter 5 of Koenig and Moo and related to the lecture notes from Thursday, January 31 and Monday, February 4.

There are two major goals of the lab. The first is to help you develop your program testing and debugging skills. The second is to give you practice working with lists and iterators. Please note that the 3rd and 4th problems are typical of problems you might see on a test.

As usual, there are four checkpoints. Upon completing each checkpoint, raise your hand and one of the TAs will quickly check your work. The TAs will help to ensure that you successfully get through the first two checkpoints in the first half of lab so that you have enough time for the last two checkpoints.

Checkpoints

1. The first two checkpoints involve testing and debugging a slightly mistaken version of the Sieve of Eratosthenes program written during class on Monday.

Debugging programs that use the standard library presents some challenges because debuggers will take you right into the middle of the standard library source code. This is something we want to avoid until we understand the mechanics of the library better. Therefore, we will learn a bit more about traditional debugging techniques and use these in combination with the visual studio debugger.

   (a) Download the slightly buggy version of the primes program:

   http://www.cs.rpi.edu/~stewart/cs2/labs/week04/primes.cpp

   (b) Create a project and workspace for the program.

   (c) Add a function to primes.cpp called print_list to print the contents of the primes list (or any other list of ints). This should print the list all on one line. For example, my function prints

   \{ 2 3 4 5 6 7 8 9 10 11 12 \}
when \( n == 12 \) and the function is passed the initial list, just after the for-loop initialization. Be sure to pass the list by \textbf{constant reference}. For simplicity, you may just insert the function at the top of the file, after the includes. Note: when you pass the list by constant reference, you must use

\[
\text{list<int>::const\_iterator}
\]

instead of

\[
\text{list<int>::iterator}
\]

(d) Add code to the main function to call your new function to print the list in two places: just after the for loop and at the end of the outer while loop (just before the \( ++\ pi; \) statement).

(e) Compile and link your program, removing all compilation errors.

(f) Run your program using the debugger. To do this, instead of clicking on the “!” icon (or typing Ctl-F5), type F5 or click on the icon that looks like a page of text with a downwards-arrow on the right. This icon is to the right of the “!” icon on my screen. Type in the value 12 at the prompt. The program should generate some output that looks approximately correct and then halt. Minimize the command-prompt (execution) pane. You should see a small error window with a statement about an “Unhandled exception”.

(g) Stop at this point and show a TA both your execution pane and the small error window.

\[2\]

The next series of steps will involve some use of the debugger and will require adding code to print out more about your program’s behavior.

(a) Click on ‘ok’ in the error window. This will throw you into the debugger. Your visual studio screen should have changed significantly. The icons at the top should be different. In the middle of the screen there should be a large pane with the source code. At the bottom will be three small panes arranged side-by-side. The left of these panes is the “call stack”, the middle gives the “context” and the right gives the “watch” list. We will eventually learn to use all of these, but not today.

(b) What you see may be hard to interpret. The reason is that the “exception” occurred within the \texttt{list} code, even though the bug in our program caused it. To learn something from the error messages, look at the call stack on the lower left. Mine says:

\[2\]
The yellow arrow points at the top line. Yours will look similar, but may not be identical.

(c) Double-click on the second line and the screen should change. It will point you into your own code, with a yellow arrow in the middle pane pointing at the line

```cpp
while ( qi != primes.end() )
```

This is the place where your code called the list class function that in turn caused the exception. We have therefore used the call stack to find out where our mistake caused the program to behave so badly that it crashed. It is not the location of our mistake, however. We must do more work to find out.

**Note:** If visual studio does not behave in exactly the way described here, stop debugging (Shift-F5 or Debug -> Stop Debugging), and try again. Sometimes it seems a bit flaky.

(d) It seems like the problem has something to do with the inner loop. At this point we could use the debugger further, but that would require knowing a little more about the list class. Instead, stop the debugger (Shift-F5 or Debug -> Stop Debugging), and return to editing primes.cpp. (If your edit screen still shows the source code for the list class, click on the window to terminate or minimize it.) Add the following code just inside the inner while loop:

```cpp
cout << "Inner loop: *qi = " << *qi << endl;
```

This will print the value in the list associated with the iterator.

(e) Compile and run your program. You can do this under the debugger if you wish. (If you don’t you will get an error message that will throw you into the debugger in a separate (second) instantiation of visual studio. Technically, there’s nothing wrong with this, but having two versions of visual studio active can cause confusion.) Look at the output. You will see that the program appears to wrap-around and start over in the list! (This is why 2

std::list<int,std::allocator<int> >::iterator::operator++() line 105 + 46 bytes
main() line 35
mainCRTStartup() line 206 + 25 bytes
KERNEL32! 77e97d08()
was eliminated incorrectly.) The problem is that the qi iterator is not always being updated correctly. The wrap-around occurs because the list is indeed a circular structure. (We will see why later in the semester.)

(f) At this point you have enough information to find and fix the problem. Study the inner loop carefully, and think about what it is doing. If you still don’t see what’s wrong, ask a TA for help.

To complete this checkpoint, demonstrate a fixed version of the program.

The last two checkpoints focus on working with vectors, lists and iterators.

3. Write function that reverses the contents of a vector. For example, if the contents of the vector are in increasing order before the reverse statement, then they will be increasing order afterwards. The program

http://www.cs.rpi.edu/~stewart/cs2/labs/week04/reverse.cpp

is designed to test this function. Download this program and add your function. Note that you may use either indexing (subscripting) or iterators in your reverse function.

The trick is to step through the vector one location at a time, swapping values between the first half of the vector and the second half. As examples, the value at location 0 and the value at location size()-1 must be swapped, and the value at location 1 and the value at location size()-2 must be swapped.

4. Write a function to reverse the contents of a list. Add it to the same program. (You will need to remove the comments for the testing code.) You must use iterators here and you may NOT use a second list — in other words you MUST do the reversing in place. Remember that the list container class does have a size member function that gives the number of items stored in the list.

You will need to use a straightforward concept we haven’t yet discussed in class: a reverse iterator. A reverse iterator is designed to step through a list (or vector) from the back to the front. An example will make the main properties clear:
list<int> a;
unsigned int i;
for ( i=1; i<10; ++i ) a.push_back( i*i );

list<int>::reverse_iterator ri;
for( ri = a.rbegin(); ri != a.rend(); ++ri )
    cout << *ri << endl;

Observe the type for the reverse iterator, the use of the functions \texttt{rbegin} and \texttt{rend} to provide iterators that delimit the bounds on the reverse iterator, and the use of the ++ operator to take one step backwards through the list. It is very important to realize that \texttt{rbegin} and \texttt{end} are NOT the same thing.

When you have finished, show a TA both your source code and a working executable.

**Discussion:**

- Writing extra functions to print the contents of structures (and containers) and using strategically placed \texttt{cout} statements are both extremely useful debugging tools. In some cases they can be replaced by more direct use of a debugger. In writing the extra functions, remember to always pass structures by constant reference to your printing functions and remember to end all debugging outputs with an \texttt{endl} to ensure that the output buffer is flushed.

- The location where the primes program crashed, which we found using the debugger, is not necessarily the exact line to concentrate on for finding the error. Instead, you must realize that something in the sequence of program execution at or before this line caused the problem. Therefore, you must use a combination of output statements, the debugger, and careful thinking to find the mistake. Remember, to **ALWAYS look for the first place the program goes wrong** during the course of execution and work from there. A simple mistake early can cause a major mistake much later. Eliminating the early mistake can eliminate the major mistake.

- Writing code using iterators takes some practice. The examples provided here are good ones. Try to make up and test others.