These must be completed and shown to your lab TA either by the end of this lab, or at the start of your next lab.

1. Please look at the brief introduction to C++ slides available on the course web page under Lab 1, if you haven't already.

2. Download the Insertion Sort program (`insertion.h`, `insertion.cc`, `Makefile`) available under Lab 2 on the course web page.

3. Compile the program using `make` and run it using the following command:

   ```
   ./insertion 12 5 9 3 2 25 8 19 200 10
   ```

   The program will hang at this point. Quit the program (`Ctrl-c`). Look at the code. What's wrong?

4. Debug the program. Identify and correct errors until `insertion` works correctly.

   You may find a debugger to be helpful with this task. A debugger allows you to *pause* a program, *step* through it line-by-line, and *inspect* the values of its variables in vivo.

   There are many choices of debuggers, and which one you use highly depends on your OS and personal preferences. If you use an IDE, your best bet is to try their debugger. If you use a text editor + command line, then you can use either a command line debugger (like `gdb`) or a graphical debugger (like `ddd`).

   This is your chance to practice debugging, so use the debugger as much as possible in this lab and consult the TAs when you need help. Often bugs can be found more quickly by placing print statements in your program, but some bugs are faster to find using a debugger, and still other bugs are nearly impossible to defeat without the use of a debugger. It will be a valuable member of your toolbelt.

   All debuggers should have a certain set of common commands:

   - **Run** In many debuggers, loading the program and running it are separate operations. To load the program into debugger, run the debugger with the argument being the name of the program you want to debug (for instance, `ddd ./insertion` or `gdb ./insertion`). Once you've launched `ddd` or `gdb`, run it with the arguments for the program that you are debugging. For example, `run 5 4 3 2 1` will start execution of `./insertion` with these arguments.
   - **Pause** (or **Interrupt**) Your `insertion` program hangs, so if you don't have any breakpoints set, you'll have to interrupt the program (Ctrl-c in `gdb`). What's a breakpoint, you ask? Well...
   - **Breakpoint** A breakpoint is a place in the program where you tell the program to pause. Whenever execution reaches the breakpoint, the program pauses. A breakpoint can be put on a particular line, or on a whole method. You can even add a **condition** to a breakpoint so that it only pauses when the condition is `true`. What can you do when the program has paused, you ask? Well...
   - **Step** While paused, you can step through the program line-by-line. You can also step in fancier ways. Try them out.
   - **Print** Debuggers have many different ways of displaying the values of variables while the program is paused. In some cases, you must explicitly call for the value to be printed in `gdb`, the command to show the value of `y_size` is `print y_size` (show the value once) or `display y_size` (show the value after every `gdb` command). In graphical IDEs, the values are automatically displayed in a sidebar, or even shown in a tooltip when you hover over them with the mouse.
   - **Expression Evaluation** Many debuggers even give you a way to evaluate C-like expressions and output their result. For example, you could have something like

     ```
     print truthiness && go_gadget_go(num + 1)
     ```

where `go_gadget_go` is a method in your source code.

- **Watch** You can set up a "watch" on a variable or expression. This causes the program to pause and show the value of the variable or expression whenever it changes.
- **Continue** You can also resume the program execution.

`gdb` has a very comprehensive internal `help` command, but if that fails you, here is the full user manual: `https://sourceware.org/gdb/current/onlinedocs/gdb/`

5. Write the values of `x`, `y`, `p1`, and `p2` after each statement in the following program. You may compile the program and use a debugger or `cout` statements to determine the values of `x` and `y` to check your work, but be prepared to explain your answers to the TA. (Instead of writing out the full hexadecimal value of memory addresses, you can use some shorthand to indicate "address of x" and "address of y.")

```
int main () {
  int x = 5, y = 15;
  int * p1, *p2;
              // value of     x       y       p1      p2
              //              5       15      uninit  uninit
  p1 = &x;
              //
  p2 = &y;
              //
  *p1 = 6;
              //
  *p1 = *p2;
              //
  p2 = p1;
              //
  *p1 = *p2+10;
              //
  return 0;
}
```

6. Download and compile program `deque.cc`. This program implements Deque ADT with Doubly Linked List. It adds operation `removeDuplicates()` that does what the name says. Observe that the program works correctly. It first enqueues strings into Deque so that it contains the following strings `ba,ba,ab,ab,ab,ab,ba,ba`. It then runs the `removeDuplicates()` and pop/prints the remaining elements: `ba,ab,ba`.

   While the program runs correctly, it's causing memory leaks. There is a handy tool for checking and catching the memory leaks in your programs, called `valgrind`. Run `valgrind --leak-check=full ./deque` to check for memory leaks. See `http://valgrind.org/docs/manual/faq.html` for more info.

7. **Bonus:** Modify the program by "plugging" memory leaks. Verify that your modified code works correctly and that it does not produce any memory leaks.

8. Be sure to show your work to your TA before you leave or at the start of your next lab, or you will not receive credit for the lab!