These exercises must be completed and shown to your lab TA either by the end of this lab session, or at the start of the next week's lab session. You may work in groups of up to two people.

1. Please have a look at the brief introduction to C++ slides available on the course webpage under Lab 1, if you have not already. Additionally, you may find some resources explaining the Insertion Sort algorithm.

2. Download the Insertion Sort program (`insertion.h`, `insertion.cpp`, `Makefile`) available under Lab 2 on the course webpage.

3. Compile the program using the following command:

   ```
   make
   ```

   `make` executes the compile commands for the default target in the `Makefile`. Note that in the `Makefile`, `insertion.cpp` is compiled with the `-g` flag so that executable files can be run in the debugger. You can run the buggy Insertion Sort program using:

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

   The program outputs all zeros. What's wrong?

4. Debug the program. Identify and correct errors until `insertion` works correctly to produce the sorted output.

   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, chances are it will include a debugger. If you use a text editor + command line, then you can use either a graphical or a command line debugger.

   Our recommendation for a CLI debugger is `gdb`. Our recommendation for a graphical debugger is KDbg (available on the CS openSUSE environment).

   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 tool in your skill set.

   All debuggers should have a certain set of common commands:

   - **Run** – in many debuggers, loading the program and running it are separate operations. Once you have launched KDbg or `gdb`, make sure to run it with the arguments given above.
   - **Pause** (or **Interrupt**) – your insertion program hangs, so if you do not have any breakpoints set, you will have to interrupt the program (Ctrl-c in `gdb`).

- **Breakpoint** – a breakpoint pauses execution automatically whenever a breakpoint is encountered. A breakpoint can be placed 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.
- **Step** – while paused, you can step through the program line-by-line. There are different types of stepping. You can **step over** the current line in the current file. If the current line contains a function call, you can **step into** the function. If the current line is within a function called from somewhere else, you can **step out** of the current function, executing all remaining lines of the current function and pausing after the current function returns to its caller.
- **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 `numY` is `print numY`). In graphical IDEs, the values are usually automatically displayed in a sidebar, or sometimes shown in a tooltip when you hover your cursor over them.
- **Watch** – you can set up a "watch" on a variable or expression, which is sometimes an easier way to see how it changes as you step through the program.
- **Continue** – You can also resume the program as normal (from a paused state).

5. Fill in the blanks in the comments of the pointers.cpp program. You may compile the program using make pointers and use a debugger or print 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.

Do not forget to show the TA your work before you leave!