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. You may work in groups of up to two people.

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.cpp`, `Makefile`) available under Lab 2 on the course web page.

3. Compile the program using the following command:

   ```
   make
   ```

   `make` executes the compile commands in the `Makefile`. Note that in the `Makefile`, `insertion.cc` is compiled with the `-g` flag so the executable file 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.

   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 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 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. Once you've launched KDbg or `gdb`, **make sure to run it with the arguments given above.**
   - **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 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`. 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 `numY` is `print numY`). 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, 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.

`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. Fill in the blanks in the following 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.")

```
#include <iostream>
using namespace std;

int main () {
  int x = 5, y = 15;
  int * p1, *p2;

  p1 = &x;      // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____

  p2 = &y;      // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____

  *p1 = 6;      // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____

  *p1 = *p2;    // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____

  p2 = p1;      // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____

  *p1 = *p2+10; // x contains ____; y contains ____
                // p1 contains ____; p2 contains ____
  return 0;
}
```

6. Show your work to your TA either by the end of this lab or at the start of your next lab, or you will not receive credit for the lab!

7. (Optional) For added practice, experiment with the following code:

```
#include <iostream>
using namespace std;
int a = 7;
```

```
int b = 6;
int* c = &b;
void test( int& x, int y, int*& z ) {
  x++;
  y++;
  z= &a;
}
int main() {
  test(a,b,c);
  cout << a << " " << b << " " << *c << endl;
  return 0;
}
```

What happens when you modify the test arguments? Try changing the various arguments from pass-by-reference to pass-by-value and vice versa. What happens? What happens if you make `b` a pointer? What happens if you make `y` a pointer?