CSCI-1200 Data Structures — Spring 2023
Lab 3 — Memory Diagrams, Testing, and Debugging

Checkpoint 1

estimate: 20-40 minutes

Write a function `compute_squares` that takes 3 arguments: two C-style arrays (*not* STL vectors), a and b, of unsigned integers, and an unsigned integer, n, representing the size of each of the arrays. The function should square each element in the first array, a, and write each result into the corresponding slot in the second array, b. You may not use the subscripting operator (`a[i]`) in writing this function; instead, practice using pointer arithmetic. Also, write a main function and a couple of test cases with output to the screen to verify that your function is working correctly.

To complete this checkpoint: Show a TA your function, the test cases, and the corresponding output.

Checkpoints 2 & 3

Checkpoints 2 and 3 are an introduction to using a command line debugger: `gdb` or `lldb`.

Testing and debugging are important steps in programming. Loosely, you can think of testing as verifying that your program works and debugging as finding and fixing errors once you've discovered it does not. Writing test code is an important (and sometimes tedious) step. Many software libraries have "regression tests" that run automatically to verify that code is behaving the way it should.

Here are four strategies for testing and debugging:

1. When you write a class, write a separate "driver" main function that calls each member function, providing input that produces a known, correct result. Output of the actual result or, better yet, automatic comparison between actual and correct result allows for verifying the correctness of a class and its member functions.
2. Carefully reading the code. In doing so, you must strive to read what the code actually says and does rather than what you think and hope it will do. Although developing this skill isn't necessarily easy, it is important.
3. Using the debugger to (a) step through your program, (b) check the contents of various variables, and (c) locate floating point exceptions and segmentation violations that cause your program to crash.
4. Judicious use of `std::cout` statements to see what the program is actually doing. This is useful for printing the contents of a large data structure or class, especially when it is hard to visualize large objects using the debugger alone.

Points, Lines, and Slopes

The programming context for this lab is calculating and sorting lines by their slope. We could use information in a map program that is plotting bicycle routes. Some users might want to pick routes that avoid the steepest roads. Other users looking for a challenge might specifically seek out the biggest uphill sections!

Our program juggles a number of source code files that define two helper classes, a 3D `Point` and a `Line` connecting two `Point`s.

Please download the following source code files needed for this lab:
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/point.h
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/point.cpp
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/line.h
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/line.cpp
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/roads.cpp

As well as the following data files:
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/input_a.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/input_b.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/input_c.txt
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/labs/03_debugging/input_d.txt

Checkpoint 2

estimate: 20-30 minutes

1. Examine the provided files briefly. How are the files related or dependent upon each other? Hint: Look at the `#include` statements. Read through the comments from the developer about the purpose of each class and function.

   > We have intentionally placed a number of bugs in the program that will cause problems when you attempt to compile and then run these programs.
   >
   > Even if you spot the problems, ***don't fix them yet!***

2. When you're confident you understand what the original programmer was aiming to do, let's compile and run the program. Take careful step-by-step notes about every command you run, and every line of the program you add or edit (and why you make that edit). Create a new README.txt file to organize your notes. You'll need to show these to your TA/mentor to get checked off for today's lab.

3. What command line(s) are necessary to compile these files into an executable? Be sure to enable *all recommended warnings for the Data Structures course* by using the `-Wall -Wextra` flags. Also use the `-g` flag which will include source code line numbers and other useful information for the debugger.

4. The program as provided may not compile without error. Perform the *minimal edits* to the source code files necessary to remove the *compilation ERROR* and produce an executable. **IMPORTANT: Do not fix any of the *compilation WARNINGs* yet.**

5. Run the program with each of the provided input data files. Take notes on what appears to be working correctly and if anything looks buggy.

6. Now let's examine those *compilation warnings* a little more closely. Oftentimes programmers can get lazy and ignore compilation warnings because these warnings aren't necessarily showstoppers that prevent us from testing and running our program on initial datasets. But some compilation warnings are actually very dangerous and can prevent the program from successfully handling all possible input datasets or even from running at all!

   Here's a list of the categories for some of the more common compilation warnings we see in the Data Structures course. You should see all or most of these when you compile the provided code.

   - `warning: expression result unused / expression has no effect`
   - `warning: control reaches / may reach end of non-void function`
   - `warning: variable is uninitialized when used here / in this function`
   - `warning: comparison of integers of different signs: 'int' and 'unsigned int'`
   - `warning: returning reference to local temporary object /`
     `reference to stack memory associated with a local variable returned`
   - `warning: unused variable`

7. Study the source code referenced by each specific compilation warning. Do you see the cause of the warning? Some or all of these warnings might be actual logic or math bugs that will cause the problem to crash or return bad data for some or all inputs. Do you see the problem? Do you know how to fix

it? **IMPORTANT: Don't fix the problems yet.** And don't worry if you don't see the error – just staring at the code is not the only debugging technique nor is it the most effective debugging technique for large programs!

To complete this checkpoint, show a TA your detailed notes on compilation, the minimal edits necessary to produce an executable, and the results of your preliminary testing. Be prepared to discuss your observations and any logic or math bugs that you belive you have found (but not yet fixed!) in the code.

Checkpoint 3

estimate: 30-40 minutes

Now, we will practice using the debugger to find and fix errors. Today we'll learn how to use the `gcc/g++` command line debugger, `gdb` from the GNU/Linux/Ubuntu or MacOSX terminal. *NOTE: On Mac OSX, you are probably actually using the* `llvm/clang++` *compiler, so you'll use* `lldb` *instead of* `gdb` *in the instructions below.* If you didn't already install `gdb/lldb`, review the installation instructions on the course webpage. Many introductory `gdb/lldb` debugging tutorials can be found on-line; just type e.g., `gdb tutorial` into your favorite search engine. Here are a couple:
http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html
http://www.cs.cmu.edu/~gilpin/tutorial/

And here's a handy table mapping `gdb` commands to `lldb` commands:
https://lldb.llvm.org/use/map.html

After you learn and demonstrate to your TA the basics of debugging with `gdb/lldb` you are encouraged to explore other debuggers, for example, the debugger built into your IDE, but we do not provide those instructions. You may also want to try a graphical front end for `gdb`: http://www.gnu.org/software/ddd/ After today's lab you can use your favorite search engine to find basic instructions for other debuggers and figure out how to do each of the steps below in your debugger of choice.

Note that in addition to a standard step-by-step program debugger like `gdb/lldb`, we also recommend the use of a *memory debugger* (`drmemory` or `valgrind`) for programs with *dynamically-allocated memory* (we'll talk about this in Lecture 5 on Friday!), or anytime you have a segmentation fault or other confusing program behavior. We'll work the memory debugger in lab next week! Information about memory debuggers is available here:
http://www.cs.rpi.edu/academics/courses/spring23/csci1200/memory_debugging.php

After today's lab, you should be comfortable with the basics of command line debugging within your preferred development environment. Keep practicing with the debugger on your future homeworks, and be prepared to demonstrate debugger skills when you ask for help in office hours.

1. Identify a program and (as appropriate) a test dataset that has buggy behavior or output.

2. **Getting started:** When you plan to use the `gdb` (or `lldb`) debugger to investigate your program you need to make sure you have *linked* the code with the `-g` option to add *debug information* (including, source code line numbers!) to the executable.

   For example (replacing `file1.cpp file2.cpp` with your source code):

   ```
   g++ -g -Wall -Wextra file1.cpp file2.cpp -o executable.exe
   ```

   In order to start `gdb`, type (replacing `executable.exe` with your program):

   ```
   gdb executable.exe
   ```

   NOTE: You can specify command line arguments to your executable on the `gdb/lldb` command line *OR* you can specify those command line arguments a little bit later, when you run your executable inside the debugger. Refer to documentation, e.g.: https://lldb.llvm.org/use/map.html

3. Now we're inside the command-line debugger. Type `help` in order to see the list of commands.

   There are several commands for setting breakpoints. You can set a breakpoint by specifying a function name or a line number within a file. For example:

   ```
   break main
   ```

   sets a breakpoint at the start of the main function. You can also set a breakpoint at the start of a member function, as in:

   ```
   break Point::get_x
   ```

   Finally, to set a breakpoint at a specific line number in a file, you may type:

   ```
   break line.cpp:31
   ```

   Set a breakpoint at some point in your code just before (in order of execution!) you think the first error might occur. Finally, in order to actually start running the program under control of the debugger, you will need to type `run` at the `gdb` command line.

4. **Stepping through the program:**

   You can step through the code using the commands `next` (move to the next line of code), `step` (enter the function), and `finish` (leave the function). The command `continue` allows you to move to the next breakpoint.

5. **Examining the content of variables:**

   You can use `print` to see the values of variables and expressions.

   You can use the command `display` to see the contents of a particular variable or expression when the program is stopped.

6. **Program flow:**

   The command `backtrace` can be used to show the contents of the call stack. This is particularly important if your program crashes. Unfortunately, the crash often occurs inside C++ library code. Therefore, when you look at the call stack the first few functions listed may not be your code.

   This does not mean that the C++ library has an error! Instead it likely means that your code has called the library incorrectly or passed the library bad data. Find the function on the stack that is the nearest to the top of the stack that is actually *your code*. By typing `frame N`, where `N` is the index of the function on the stack, you can examine your code and variables and you can see the line number in your code that caused the crash. Type `info locals` and `info args` to examine the data in the function. Type `list` to see the source code.

7. **Breakpoint on Variable Change:** The last powerful debugger feature we will try today is variable monitoring.

   Add a new modifier member function to the `Point` class named `set_elevation` that changes the y coordinate of a `Point` instance. Create an instance of a `Point` object in the `main` function called `pt`. Use the `set_elevation` function on that instance.

   You can use the command `watch` to halt the program when a variable or expression changes.

   For example, type `watch pt.y`. You can try the the `rwatch` command which allows you to break on a read of the value (not just a change).

   NOTE: Students have recently (in the past term or two) reported difficulty getting watch points to work inside their command line debuggers. So don't worry if you also cannot get this feature to work today.

Continue to experiment with these different debugger operations. You can now start to fix the bugs in the program that were hinted at by the compilation warnings. Remember to recompile the program and re-launch the debugger after each change to the source code. Continue to take detailed notes on what edits you make to the provided source code.

When when you feel comfortable showing off all these debugger features, put your name in the queue for checkoff for this checkpoint. The TA/mentor will be asking you to demonstrate these features and discuss the compilation warnings and bugs in the provided code. You will also be asked questions about the other steps in debugging.