---

**Getting Started:** **YOU MUST USE THE LINUX NETWORK — NOT YOUR OWN LAPTOP OR DESKTOP — TO COMPLETE THIS LAB.** Remotely log in to the Linux network, create a new `lab3` directory in your `cmsc240` directory, and then copy all the files and directories from the appropriate directory in my home directory: `~dszajda/outbox/cs240/lab3/`

Answers to all underlined questions must go in the `cmsc240_lab3_NETID.txt` (**which you must rename (so that it contains your netID!**). This file, in addition to being your submission file, also contains parts of questions.

---

**Lab Exercise #1:**

- Use the appropriate commands to inspect the man pages for `atoi()` and `atof()`.
  1. What does the `atoi()` function do?
  2. What does the `atof()` function do?
- Now open `ProgramOne.cpp` in an editor and inspect the source code to determine what the program is doing.
  3. Precisely what is the `atoi()` function used for in this program?
- Compile the program, with the debugging flag (`-g`) included, so that you have an executable called `ProgramOne`.
- Start the debugger by issuing the command

  ```
  gdb ProgramOne
  ```

  You should see a few lines of information about gdb scrolling on your screen followed by a (gdb) prompt. This lab will focus on command line interaction with gdb.

  > *(Note that on the Linux boxes you are free to use DDD, a graphical interface for gdb. All gdb commands will work in ddd; however, there are also menus, buttons, and windows in ddd that perform many of the actions you can initiate with gdb text-based commands. If you're sitting at one of Linux terminals or you have been fortunate enough to get remote windowing to work, type* `ddd&` *at the terminal prompt — don't forget the '&' — it allows you to continue typing commands in your terminal window after ddd starts running. Then in ddd, just open the executable you are interested in inspecting.*

  As you work, it will be helpful to keep your source code open in an editor so that you may view it alongside the gdb output.
- To run the debugger on your executable, just type `run` or simply `r` (which is an alias for `run`) followed by any necessary command line arguments. For example, because this program expects a single integer command-line argument, you can use something like

  ```
  r 3
  ```

  You'll see the output of your program displayed at the prompt. Type your responses into that window.
- Debuggers are useful because they allow you to stop execution of the program at any breakpoint you set. Once at that breakpoint, you can examine the values of variables and walk through the code step by step. To set a breakpoint, you can enter a command that follows the following format:

`break` *source-code-filename*:*line-num-in-file*

(you can use `b` as an alias for `break`). Specifically, you could set a break on line 27 of `ProgramOne.cpp` by giving the command

```
b ProgramOne.cpp:27
```

Another way to set a breakpoint is to give the name of a particular function you want the break to occur on, e.g.,

```
b Sum
```

Now, put a breakpoint at the line declaring the variable `array` in the `main` function. (If you get errors from gdb stating there are no debugging symbols found at this point, you might want to scan down the g++ man page for the information about the –g flag. It is buried pretty deep, so using '/' to search for it may be helpful.)

When you run the program again, you'll notice the program will stop and the (gdb) prompt will reappear. gdb will also specify which breakpoint was reached and will display the source code for the specified location. (**NOTE: The line that gdb displays has not yet been executed!** The breakpoint is set between the previous line and the line that is displayed.)

4. Copy and paste both lines of the breakpoint information printed by gdb for the array declaration.

You can then move on to the next line of code by typing `n` or `next`. The `next` command moves you to the next line of the source code *as the program is executing*.

You should note that `next` does not allow you to step inside function calls. To step inside a function call, enter the command `s` (or `step`) when you're at the line of code that calls the function. Place a breakpoint at the line in `main` where the function call to `Sum` occurs and use the `c` command (or `cont` or `continue`) — this causes execution to resume as normal (not stepping) until the next breakpoint is encountered. You should see gdb pauses just before executing the line of code containing the `Sum` function call. Now use `s` to step into that function.

5. Copy and paste both lines of the breakpoint information printed by gdb immediately after stepping into `Sum`.

- You can also display the current value of any variable within the current scope by using the `print` (or `p`) command. For example, you can type

```
p array
```

at the (gdb) prompt to print the contents of the `array` variable.

To answer the following questions, you should **restart debugging** from the beginning of the program by issuing the `run` command again in gdb. Note that the line of source code that `gdb` displays is the one that will be executed next, not the one that has just been executed.

6. Copy & paste the gdb output when you issue the command `p array` immediately **after** the array is declared.
7. Copy & paste the gdb output when you issue the command `p *array` at that same place.
8. Copy & paste the gdb output when you issue the command `p array[0]` at that same place.
9. Copy & paste the gdb output when you issue the command `p &(array[0])` at that same line.
10. Select all correct responses with respect to the four values above.

If you want to continually display the value of a variable, you can use the `display` command (or more simply `disp`). For example, enter

```
display array
```

after the line that declares `array`, and then repeatedly use the `next` command until you reach the end of the program. You should notice that the value for `array` will be repeated after each `next` command. To stop displaying a variable at any point, enter `undisplay` (or `undisp`) followed by the number (to the left of each display of the variable) associated with that variable. Try this until you're comfortable with its functionality.

- Now place a new breakpoint on the line inside the for-loop's body within the `Sum` function (line 17). Rerun the program (use a small number, say 3, of numbers to enter) and step through to completion of the program. Within that for-loop inside `Sum`, use `display array[i]` and `display sum` to indicate the values for `array[i]` and `sum` as the loop progresses. (Note that displaying the variable named `array` in `main` differs from displaying the parameter named `array` in `Sum` because the scopes are different.)

  11. Copy and paste display information for `array[i]` and `sum` from any one iteration through the loop.

- Type `q` (or `quit`) to exit gdb and return to the command prompt.

**Lab Exercise #2:**

- Open `ProgramTwo.cpp` in an editor and inspect the source code to determine what the program is doing. Note that the program uses two different C-style strings: one an "automatic" which is allocated and deallocated for you automatically; the other uses explicitly dynamic allocation, and must be deleted before the scope ends. The program also uses the C-style string function `strncpy` to copy into those strings — inspect the C++ API to understand what this function does.

  Compile (with `-g`) and run the program at the terminal.

  12. Copy & paste the output from the program.
  13. Based on code inspection, what did you expect the last two `cout` statements to print?

- Now run the program in gdb, setting a breakpoint at `main`. After the allocation of the two char arrays but before execution of the `strncpy` calls, use gdb's display capability to show what is contained in the two arrays.

  14. Copy & paste the display output for the two arrays before the `strncpy` calls are executed.

  Use `n` to step through your program line by line.

  15. Copy & paste the display output for the two arrays before the *first* pair of `cout` statements are executed.
  16. Copy & paste the display output for the two arrays before the *second* pair of `cout` statements are executed.

- Rerun the program in gdb, but just before execution of the second pair of `cout` statements, issue the following commands in gdb:

```
set autoString[3] = '\0'
set dynamicString[4] = '\0'
```

  The `\0` character is used as the end-of-string character to indicate to C++ where the end of a C-style string of characters occurs.

- Then continue to step through the program, executing the second pair of `cout` statements.

17. What is now printed for `autoString`?
18. What is now printed for `dynamicString`?
19. Based on the previous observations, the API description of `strncpy`, and your knowledge of C-style strings, if our goal was to produce just `"289"` as the output of each of the second set of `cout` statements, what is missing from the C-style strings as a result of using `strncpy` as given?

- Modify `ProgramTwo.cpp` to include two assignment statements of the end-of-string character to ensure that C++ will print only `"289"` for both `autoString` and `dynamicString` in the second pair of `cout` statements. Compile, execute, and test for correctness.

**Lab Exercise #3:**

- Open `ProgramThree.cpp` and `INPUT.txt` in an editor and inspect the source code and input file to determine what the program is doing. Note that the program reads the same input file twice. The first loop uses >> to read characters from the file, skipping any whitespace. The second loop uses the `.get()` function to read one character at a time from the file, including any whitespace. When printing, both loops print each character read, immediately followed by its ASCII integer value.
- Compile (with `-g`) the program and execute.

20. Copy and paste the output of the program.
21. Based on inspection of `INPUT.txt`, what behavior does the first loop exhibit that is incorrect or unexpected?

- You will now use gdb to determine why the first loop is incorrect. Set a breakpoint for line 12 (the first while statement) of the program. Then run the program in gdb, and when the program pauses at line 12, use the following display commands to allow gdb to display the state of your `infile` stream and the contents of the variable `c` as you step through the program.

```
display c
display infile.good()
display infile.eof()
```

- Now step through the loop as the five non-whitespace characters are read from the file, until the loop fails.

22. When do the values of `infile.good()` and `infile.eof()` change?
23. Why is the value 9 printed twice by this loop?

- Modify `ProgramThree.cpp` to include appropriate logic (using `infile.eof()`) to recognize when the end-of-file character has been reached, so that the value 9 will not be printed twice. Compile, execute, and test for correctness.

**Lab Exercise #4:**

- Open and carefully inspect the source files `ClassOne.h`, `ClassOne.cpp`, and `ProgramFour.cpp` so that you understand their functionality and purpose.

24. What three types of data does a `ClassOne` object contain?
25. What functionality (methods) would a `ClassOne`-type object have?

Note that `ProgramFour` dynamically allocates two objects of type `ClassOne` (similar to what you saw above for dynamic allocation of a C-style string), and deletes those objects at the end of their scope.

- There are two errors with these programs — one that causes unexpected behavior and one that causes the program to crash. Compile and run the program at the terminal (remember to include both .cpp files, but no .h files, in the compile command).

  26. What is the *unexpected result in the output* (that occurs before the crash at the end)?

- Now execute gdb on this program and begin stepping through the program from the beginning.

  27. Copy and paste the result when you have gdb print or display `obj` immediately after its construction.
  28. Copy and paste the result when you have gdb print or display `*obj`.
  29. Copy and paste the result when you have gdb print or display `obj2` immediately after its construction.
  30. Copy and paste the result when you have gdb print or display `*obj2`.
  31. Copy and paste the result when you have gdb print or display `obj->charstar_data`.
  32. Copy and paste the result when you have gdb print or display `obj2->charstar_data`.
  33. Based on the information above, why do you get the unexpected result in the output? (You may need to inspect the copy constructor in `ClassOne.cpp` again, too.)

- You need to make changes to the copy constructor to fix the unexpected result you saw above. (What changes?) Fix the issue in that copy constructor, and then compile and run `ProgramFour` again. If you appropriately fixed the problem above, the crash should also be eliminated.

  34. Briefly describe your fix to the copy constructor.
  35. Why did your fix above eliminate the crash as well?

**Naming:**

Same as usual. The name of your submission file for this lab **MUST** be `cmsc240_lab3_netID.txt`, where the `netID` is, of course, your netID. Note this is a `.txt` file not a `.tar` file.

**Submission:**

The high level picture is that to submit any labs/project in this course, you send an email to a special email address with your **single** submission file attached. This has the effect of placing your submission in the appropriate Box folder. If your submission requires more than one file, you should `tar` or `zip` your files together to create your single submission file.

The the email address for this lab is

`lab3.9m7bbjoo274zn8kj@u.box.com`.