Getting Started: Remotely log in to the Linux network, create a new `lab5` directory in your `cmsc240` directory, and then recursively copy all the files and directories from the appropriate directory in my home directory: `cp -r ~dszajda/outbox/cs240/lab5/* .`

Answers to all underlined questions must go in the `cmsc240_lab5_NETID.txt` (which you must rename!). And as usual, this "answer" file also has parts of questions. And of course the name should include your netID.

Valgrind: In this lab, you will learn to use Valgrind, a very handy memory debugging tool. Using Valgrind, you can find many memory management errors in your programs, include memory leaks (more on memory leaks in a moment). So why should you use Valgrind? An entire list of reasons is provided on the Valgrind web site (`http://valgrind.org/info/about.html`), but the following is, from your perspective, the most important:

> **"Valgrind will save you hours of debugging time."**

particularly when using dynamically allocated memory. Because many of you are new to, or at least relatively uncomfortable with, dynamic-memory management in C/C++, you should make frequent use of Valgrind.

Now let's walk through some intentionally buggy example programs, demonstrating the use of Valgrind to find the bugs and highlighting common memory mistakes along the way.

Program #1:

1. Open `prog1.cpp` in your favorite editor. Carefully read through the code, and locate (but do not correct) the program error. (Note: The error is not a compile-time error; it is a subtle run-time error that, in this program, won't even show itself when you execute the program.)
2. Now compile the program. Valgrind requires you to compile with a `-g` flag to turn debugging information on, e.g.,

   ```
   g++ -std=c++17 -g prog1.cpp -o prog1
   ```

3. Execute the program without using Valgrind. You will see that the error doesn't rear its ugly head (but that doesn't mean that in bigger, more complex programs it wouldn't!).
4. Now run the program using Valgrind:

   ```
   valgrind -v ./prog1
   ```

   (The `-v` flag provides an error summary in the output. And if you want to redirect the output of valgrind to a file (so you can use a text editor to search through it, you need to use the flag `--log-file=''filename''`, so to write output to the file `foo` it would be `valgrind --log-file=''foo'' -v ./prog1`.) Valgrind will find one error, listing the name of the program, the line on which the error occurred, and the type of error.
5. Complete questions 1 and 2 on your answer sheet.
6. As another approach here, recompile the program with the `-Wall` flag[1] (indicating that all compile warnings should be displayed), and note what happens.
7. Now correct the program, recompile, and rerun using Valgrind to make sure the error is no longer present.

[1] This should be read as "double-u all", not as "wall".

Program #2:

1. Open `prog2.cpp` and try to locate (but do not correct) the error. This error, although not a compile-time error, can result in a very serious run-time error.
2. Compile the program and execute without Valgrind. Ugh — a segmentation fault. Mismanagement of memory (how's that for alliteration?) is lurking beneath the hood.
3. Now run the program again using Valgrind — still a segmentation fault, but at least you have more meaningful feedback. In the Valgrind output, find all instances of `prog2.cpp` with associated line numbers.
4. Complete questions 3 and 4 on your answer sheet.
5. Now correct all errors in the program, recompile, and rerun using Valgrind to make sure no errors are present.
6. Remove any existing core files present in your directory. (There may be none.)

Program #3:

1. Open `prog3.cpp` and locate the error.
2. Compile the program and execute without Valgrind. You should not receive a runtime error; however, an error does exist in the program.
3. Try changing `intArraySize` to 10 000. Recompile and rerun. What happens?
4. Change `intArraySize` to 100 000. Recompile and rerun. What happens?
5. Now run the program again using Valgrind.
6. Complete questions 5 and 6 on your answer sheet.
7. Now correct the program, recompile, and rerun using Valgrind to make sure no errors are present.
8. Remove any existing core files present in your directory.

Programs #4, 5, 6:

1. Repeat the process with `prog4.cpp`. Make sure to complete questions 7 and 8 on your answer sheet.
2. Repeat the process with `prog5.cpp`. Make sure to complete question 9 and 10 on your answer sheet.
3. Repeat the process with `prog6.cpp`. Make sure to complete question 11 and 12 on your answer sheet.

Program #7: The final program, `prog7.cpp`, contains a subtle error that can come back to really bite you in programs that dynamically allocate a lot of memory.

1. Try to locate the error, then run using Valgrind.

   > Wait — Valgrind reports no errors. We're no longer dealing with an egregious memory error, but a more subtle one known as a *memory leak*. A memory leak occurs when you allocate space dynamically, and then subsequently lose any pointer to that memory (e.g., if you reassign the pointer) without having freed the memory that was associated with that pointer.

2. Run the program again using Valgrind, but now turn on Valgrind's full memory leak reporting:

   ```
   valgrind -v --leak-check=full ./prog7
   ```

   So you see that memory is definitely being leaked somewhere. Determine where.

3. Complete questions 13 and 14 on your answer sheet.

4. Now correct the program, recompile, and rerun using Valgrind to make sure that no memory leaks are present.

Summary: You have seen that Valgrind is capable of finding uninitialized variables, problems with allocated and deallocated memory, and memory leaks. Valgrind is capable of much more than we can learn in one lab — I encourage you to visit the Valgrind web site for more information.

Acknowledgments: I must thank Will Jones in the ECE Department at Clemson University for graciously permitting us to use and modify the example programs he created.

Naming:

Same as usual. The name of your submission file for this lab **MUST** be `cmsc240_lab5_netID.txt`, where `netID` is 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

`Lab5.ei432eo17uble55g@u.box.com`.