Overview: 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, including 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.

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.

Executing the Lab: On the Assignments page of the course web site, you will find the gzipped tarball lab2-examples.tgz — download and unpack the file into the directory of your choice. Here’s a quick primer on handling tar files:

```
tar xvf lab2-examples.tgz  # Unpack the gzipped tar archive in the current directory

tar cvf lab2-examples.tgz *.c  # Create the gzipped tar archive of all .c files in the current directory
```

To create uncompressed .tar files, leave out the z flag. Tar has a huge number of other capabilities, which you can learn about from the tar man page. (Historical note: the popular .zip format is a combination of tar and the Unix compress compression utility. Gzip is a more effective compression algorithm that contains no patented software technology. Note also that Java .jar archive are really tar archives, and the command line flags for the two tools are very similar.)

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.

Complete your electronic answer sheet (included in the tarball as lab2-Ans.txt) as you complete the assignments below.

Program #1:

1. Open prog1.c 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++ -Wall -g -o prog1 prog1.c
```
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 --tool=memcheck prog1
   ```
   
   (Ask a friend about aliases if you don’t feel like typing --tool=memcheck every time.) 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 question 1 on your answer sheet.

6. Now correct the program, recompile, and rerun Valgrind to make sure the error is no longer present.

**Program #2:**

1. Open prog2.c 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.

   It is possible that nothing disastrous happens — both arrays are initialized and print out exactly as expected. But this program will blow up just as it is on some systems. Mismanagement of memory (how’s that for alliteration?) is lurking beneath the hood, and can easily come back to bite you.

3. If the program ran without error on your system, change the value of intArraySize from 10 to 100. Recompile and execute again. This should produce a segmentation fault.

4. 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 references to prog2.c with associated line numbers.

5. Complete question 2 on your answer sheet — do not just write the error(s) fed back by Valgrind; I want a code-specific explanation. You may need to refer back to the original code.

6. Now correct all errors in the program, recompile, and rerun using Valgrind to make sure no errors are present.

7. Remove any existing core files present in your directory. There shouldn’t be any unless you have used the ulimit command to change the default behavior for your account, but check just in case — core files are big and can easily gobble up your disk quota if you aren’t careful about properly disposing of them.

**Program #3:**

1. Open prog3.c 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. Similar to the previous program, try changing intArraySize to 1000. Recompile and rerun. What happens?

4. Change intArraySize to 100000. Recompile and rerun. What happens?

5. Now run the program again using Valgrind.

6. Complete question 3 on your answer sheet—do not just write the error fed back by Valgrind; I want a code-specific explanation.

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.c. Make sure to complete question 4 on your answer sheet.

2. Repeat the process with prog5.c. Make sure to complete question 5 on your answer sheet.

3. Repeat the process with prog6.c. Make sure to complete question 6 on your answer sheet.

Program #7:

The final program, prog7.c, 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. Note that Valgrind does report that some memory was lost, it just doesn’t count it as an error.

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

   ```
   valgrind --tool=memcheck --leak-check=full --show-reachable=yes prog7
   ```

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

3. Complete question 7 on your answer sheet—do not just write the error fed back by Valgrind; I want a code-specific explanation.

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

Create a gzipped tarball of your modified files (following the instructions above), choosing a name that identifies you and the assignment, and drop the tarball in my netfiles inbox. Be sure to include your answer sheet as well.

Summary:

You have seen that Valgrind is capable of finding uninitialized variables, problems with allocating and deallocating 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: Thanks to Will Jones in the ECE Department at Clemson University for graciously permitting us to use and modify the example programs he created. If you found these to be helpful, send Will an email to that effect: wjones@parl.clemson.edu. Thanks also to Barry Lawson for the original formulation of this lab exercise.