Code Troubleshooting Guide

Computer Science I

Department of Computer Science & Engineering
University of Nebraska–Lincoln

Troubleshooting Buggy Code

Often you'll encounter a bug or problem with your code that leads to unexpected results, segmentation faults or other issues. Learning how to approach these problems with a rigorous and systematic approach is an essential skill in software development. Though there are many approaches, we'll list a few basic steps that you should try before throwing in the towel.

1. Test! Write test cases and test suites or use a formal unit testing framework. Writing and using unit tests and developing test cases Test cases helps to provide assurance that your software is correct. The more formal the testing, the higher assurance you have and the more maintainable and repeatable your tests are. It also has additional benefits: by writing tests it forces you to think/rethink your software design, understand the problem better, and more rigorously outlines your expectations on how the software should work.
2. Use a linter. Always use the `-Wall` flag when compiling. This makes the compiler reports *all* warnings. These are not necessarily problems with your program, but may (and often do) indicate code that can lead to bugs or are bad practices.
3. Use a static analyzer. Similar to a linter, there are tools that you can use that perform *static analysis* of code. This goes beyond merely checking the syntax or code for common errors and performs sophisticated analysis of code to find potential bugs and issues. Two static analysis tools are available on CSE:
    - Clang is another compiler (actually a front-end for the LLVM compiler) and has a similar linter/analyzer capability. It often catches the same issues as GCC's linter, but often catches more or different issues. You can use it as follows:

`clang --analyze foo.c`

- Clang also has a Static Analyzer tool (see http://clang-analyzer.llvm.org/, http://clang-analyzer.llvm.org/available_checks.html) that you can use in your compilation or build process. Examples:

  Using gcc:
  `scan-build gcc foo.c`

  Using clang:
  `scan-build clang foo.c`

  Using make:
  `scan-build make`

4. Use a dynamic analyzer. Valgrind is a dynamic analysis tool that can monitor a program and analyze the resources it uses and operations it performs. Dynamic analysis is different in that it performs its analysis by running the program instead of just looking at the source code. To use it, you first need to use the `-g` flag to compile in order to preserve function names and line numbers in the analysis. Once compiled, you can start Valgrind using

   `valgrind --leak-check=full --show-leak-kinds=all ./a.out`

   It will run your program and produce a report of any issues including potential (and "definite") memory leaks.

5. Use a debugger. GNU's Debugger (GDB) is a powerful tool that allows you to run, test, and debug a program by simulating its execution. You can pause and resume the program, examine variable values, and step through the program line-by-line. A full introduction to GDB is available in the course resources:

   - Video Introduction to GDB: https://www.youtube.com/watch?v=bWH-nL7v5F4&index=43
   - Hack 9.0: https://github.com/cbourke/CSCE155-Hack9.0

6. "Perf" Test. Performance testing may be necessary if you have code that compiles, runs, and may even work, but seems to be inefficient. You can use a *profiler* to monitor the execution of a program and determine which function(s) or sections of code take the most time or resources. GNU's Gprof performance analysis tool can be used to profile a program. First, compile using the `-pg` flag:

   `gcc -pg foo.c`

   to preserve identifiers and make it so that gprof can profile the program. Then run the program normally. After it has concluded executing, it will have created a `gmon.out` file which contains a gprof report. To view it you can use the following:

   `gprof a.out`