The Art of Debugging

Michael Chastain

The following documentation is excerpted from Merc 2.0's `hacker.txt` file. It was written by Furey of MERC Industries and is included here with his permission. I've packaged it with Circle (very slightly changed in a couple of places, such as specific filenames) because it offers good advice and insight into the art and science of software engineering. More information about CircleMUD, including up-to-date versions of this documentation in ASCII and Postscript, can be found at the *CircleMUD Home Page* `<http://www.circlemud.org/>` or *FTP site* `<ftp://ftp.circlemud.org/pub/CircleMUD/>`.

1 "I'm running a Mud so I can learn C programming!"

Yeah, right.

The purpose of this document is to record some of our knowledge, experience and philosophy. No matter what your level, we hope that this document will help you become a better software engineer.

Remember that engineering is work, and *no* document will substitute for your own thinking, learning and experimentation.

2 How to Learn in the First Place

- Play with something.
- Read the documentation on it.
- Play with it some more.
- Read documentation again.
- Play with it some more.
- Read documentation again.
- Play with it some more.
- Read documentation again.
- Get the idea?

The idea is that your mind can accept only so much "new data" in a single session. Playing with something doesn't introduce very much new data, but it does transform data in your head from the "new" category to the "familiar" category. Reading documentation doesn't make anything "familiar," but it refills your "new" hopper.

Most people, if they even read documentation in the first place, never return to it. They come to a certain minimum level of proficiency and then never learn any more. But modern operating systems, languages, networks, and even applications simply cannot be learned in a single session. You have to work through the two-step learning cycle *many* times to master it.

3 Basic Unix Tools

man

gives you online manual pages

grep

stands for "global regular expression print;" searches for strings in text files

vi, emacs, jove

use whatever editor floats your boat, but learn the hell out of it; you should know *every* command in your editor

ctags

makes "tags" for your editor allows you to goto functions by name in any source file

`>, >>, <, |`

input and output redirection at the command line; get someone to show you, or dig it out of "man csh"

These are the basic day-in day-out development tools. Developing without knowing how to use *all* of these well is like driving a car without knowing how to change gears.

4 Debugging: Theory

Debugging is a science. You formulate a hypothesis, make predictions based on the hypothesis, run the program and provide it experimental input, observe its behavior, and confirm or refute the hypothesis.

A good hypothesis is one which makes surprising predictions which then come true; predictions that other hypotheses don't make.

The first step in debugging is not to write bugs in the first place. This sounds obvious, but sadly, is all too often ignored.

If you build a program, and you get *any* errors or *any* warnings, you should fix them before continuing. C was designed so that many buggy ways of writing code are legal, but will draw warnings from a suitably smart compiler (such as "gcc" with the `-Wall` flag enabled). It takes only minutes to check your warnings and to fix the code that generates them, but it takes hours to find bugs otherwise.

"Desk checking" (proof reading) is almost a lost art these days. Too bad. You should desk check your code before even compiling it, and desk-check it again periodically to keep it fresh in mind and find new errors. If you have someone in your group whose *only* job it is to desk-check other people's code, that person will find and fix more bugs than everyone else combined.

One can desk-check several hundred lines of code per hour. A top-flight software engineer will write, roughly, 99% accurate code on the first pass, which still means one bug per hundred lines. And you are not top flight. So ... you will find several bugs per hour by desk checking. This is a very rapid bug fixing technique. Compare that to all the hours you spend screwing around with broken programs trying to find *one* bug at a time.

The next technique beyond desk-checking is the time-honored technique of inserting "print" statements into the code, and then watching the logged values. Within Circle code, you can call `printf()`, `fprintf()`, or `log()` to dump interesting values at interesting times. Where and when to dump these values is an art, which you will learn only with practice.

If you don't already know how to redirect output in your operating system, now is the time to learn. On Unix, type the command "`man csh`", and read the part about the ">" operator. You should also learn the difference between "standard output" (for example, output from "`printf`") and "standard error" (for example, output from "`fprintf(stderr, ...)`").

Ultimately, you cannot fix a program unless you understand how it's operating in the first place. Powerful debugging tools will help you collect data, but they can't interpret it, and they can't fix the underlying problems. Only you can do that.

When you find a bug ... your first impulse will be to change the code, kill the manifestation of the bug, and declare it fixed. Not so fast! The bug you observe is often just the symptom of a deeper bug. You should keep pursuing the bug, all the way down. You should grok the bug and cherish it in fullness before causing its discorporation.

Also, when finding a bug, ask yourself two questions: "What design and programming habits led to the introduction of the bug in the first place?" And: "What habits would systematically prevent the introduction of bugs like this?"

5 Debugging: Tools

When a Unix process accesses an invalid memory location, or (more rarely) executes an illegal instruction, or (even more rarely) something else goes wrong, the Unix operating system takes control. The process is incapable of further execution and must be killed. Before killing the process, however, the operating system does something for you: it opens a file named "`core`" and writes the entire data space of the process into it.

Thus, "dumping core" is not a cause of problems, or even an effect of problems. It's something the operating system does to help you find fatal problems which have rendered your process unable to continue.

One reads a "core" file with a debugger. The two most popular debuggers on Unix are `adb` and `gdb`, although occasionally one finds `dbx`. Typically one starts a debugger like this: "`adb bin/circle`" or "`gdb bin/circle lib/core`".

The first thing, and often the only thing, you need to do inside the debugger is take a stack trace. In `adb`, the command for this is "`$c`". In gdb, the command is "`backtrace`". In dbx, the command is "`where`". The stack trace will tell you what function your program was in when it crashed, and what functions were calling it. The debugger will also list the arguments to these functions. Interpreting these arguments, and using more advanced debugger features, requires a fair amount of knowledge about assembly language programming.

If you have access to a program named "Purify" ... learn how to use it.

6 Profiling

Another useful technique is "profiling," to find out where your program is spending most of its time. This can help you to make a program more efficient.

Here is how to profile a program:

1. Remove all the .o files and the "circle" executable:

   ```
   rm src/*.o bin/circle
   ```

2. Edit your `Makefile`, and change the `PROFILE=` line:

   ```
   PROFILE  = -p
   ```

3. Remake circle:

   ```
   make
   ```

4. Run circle as usual. Shutdown the game with the `shutdown` command when you have run long enough to get a good profiling base under normal usage conditions. If you crash the game, or kill the process externally, you won't get profiling information.

5. Run the `prof` command:

   ```
   prof bin/circle > prof.out
   ```

6. Read prof.out. Run "`man prof`" to understand the format of the output.

For advanced profiling, you can use "`PROFILE = -pg`" in step 2, and use the "gprof" command in step 5. The "gprof" form of profiling gives you a report which lists exactly how many times any function calls any other function. This information is valuable for debugging as well as performance analysis.

Availability of "prof" and "gprof" varies from system to system. Almost every Unix system has "prof". Only some systems have "gprof".

7 Books for Serious Programmers

Out of all the thousands of books out there, three stand out:

- Kernighan and Plaugher, *The Elements of Programming Style*.
- Kernighan and Ritchie, *The C Programming Language*.
- Brooks, *The Mythical Man Month*