CSE 303: Concepts and Tools for Software Development

Hal Perkins
Winter 2009
Lecture 15—Breakpoint debugging & gdb
Where are We

We’re jumping ahead in the slide deck a little because debugging tools can save you a bunch of time while working with C code.

Today’s agenda:

- Debuggers, particular gdb
- Why?
  - To learn general features of breakpoint-debugging
  - To learn specifics of gdb
  - To learn general debugging “survival skills” (don’t panic)
An execution monitor?

What would you like to “see from” and “do to” a running program?

Why might all that be helpful?

What are reasonable ways to debug a program?

A “debugger” is a tool that lets you stop running programs, inspect (sometimes set) values, etc.

- A “CAT scan” for observing executing code
Issues

- Source information for compiled code. (Get compiler help.)
- Stopping your program too late to find the problem. (Art.)
- Trying to “debug” the wrong algorithm.
- Trying to “run the debugger” instead of understanding the program.

It’s an important tool.

Debugging C vs. Java

- Eliminating crashes does not make your C program correct.
- Debugging Java is “easier” because (some) crashes and memory errors do not exist.
- But programming Java is “easier” for the same reason!
gdb

(gdb) is on attu and other lab machines. It supports several languages, including C compiled by gcc.

Modern IDEs have fancy GUI interfaces, which help, but concepts are the same.

Compiling with debugging information: gcc -g

- Otherwise, gdb can tell you little more than the stack of function calls.

Running gdb: gdb executable

- Source files should be in same directory (or use the -d flag).

At prompt: run args

Note: You can also inspect core files, which is why they get saved. (Mostly useful for analyzing crashed programs after-the-fact, not for systematic debugging.)
Basic functionality

- backtrace
- frame, up, down
- print expression, info args, info locals

Often enough for “crash debugging”
Also often enough for learning how “the compiler does things” (e.g., stack direction, malloc policy, ...)
Breakpoints

• break function (or line-number or ...)

• conditional breakpoints (break XXX if expr)
  1. to skip a bunch of iterations
  2. to do assertion checking

• going forward: continue, next, step, finish
  – Some debuggers let you “go backwards” (typically an illusion)

Often enough for “binary search debugging”

Also useful for learning program structure (e.g., when is some function called)

*Why not skim the manual for other features.*
A few tricks

Everyone develops their own “debugging tricks”; here are a few:

• Printing pointer values to see how big objects were.
• Always checking why a seg-fault happened (infinite stack and array-overflow very different)
• “Staring at code” even if it does not crash
• Printing array contents (especially last elements)
• ...
Advice

Understand what the tool provides you.

Use it to accomplish a task, for example “I want to know the call-stack when I get the NULL-pointer dereference”

Optimize your time developing software.

• Think of debugging as a systematic experiment to discover what’s wrong — not a way to randomly poke around.

Use development environments that have debuggers?

See also: jdb for Java (on attu)

Like any tool, takes extra time at first but designed to save you time in the long run

• Education is an investment.