CSE 374
Programming Concepts & Tools

Hal Perkins
Fall 2017
Lecture 11 – gdb and Debugging
Administrivia

• HW4 out now, due next Thursday, Oct. 26, 11 pm:
  – C code and libraries.
  – Some tools: gdb (debugger) and clint.py (style checker). gdb demo today.
  – Due next Thursday, 11 pm

• Midterm the following Wednesday, Nov. 1, in class
  – Everything up to hw4/basic C
  – Closed book – will provide reference summary info on test as needed
  – Old exams on web now for studying
  – Review Q&A Tuesday, Oct. 31, 4:30, location TBA
Agenda

• Debuggers, particularly gdb
• Why?
  – To learn general features of breakpoint-debugging
  – To learn specifics of gdb
  – To learn general debugging “survival skills”
    • Skill #1: don’t panic!
    • Skill #2: be systematic – have a plan
How to avoid debugging

• Don’t put bugs in the program!!
• Think before typing – design before coding
• Write down design (comments) as you go
  – Functions: declaration+comments should be complete spec
  – Significant data: declaration + comments should be complete spec
  – If someone has to read the code to figure out how to use something or understand data structures, comments are bad
  – Review/check comments and compare to code as you work
    • Will catch errors before you run the program
• Turn on compiler warnings (-Wall); use assert; get the computer to find problems for you.
• But things can still go wrong…
An execution monitor?

- What would you like to “see inside” 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, run a statement or two at a time, etc.
  - A “MRI” 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 (Gnu debugger) is part of the standard Linux toolchain.
- gdb supports several languages, including C compiled by gcc.
- Modern IDEs have fancy GUI interfaces, which help, but concepts are the same.
- Compile 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 got saved on older systems after every crash
  - (Mostly useful for analyzing crashed programs after-the-fact, not for systematic debugging. The original use of db.)
Basic functions

• 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)
• Skim the manual for other features.
A few tricks

• Everyone develops their own “debugging tricks”; here are a few:
  – Always checking why a seg-fault happened (infinite stack and array-overflow very different)
  – Printing pointer values to see how big objects were.
  – “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. Observation: the problem; hypothesis: I think the cause is …; experiment: use debugger to verify
- Use development environments that have debuggers?
- See also: jdb for Java
- Like any tool, takes extra time at first but designed to save you time in the long run
  - Education is an investment
gdb summary – running programs

- Be sure to compile with gcc -g
- Open the program with: gdb executable_file
- Start or restart the program: run program_args
- Quit the program: kill
- Quit gdb: quit
- Reference information: help

- Most commands have short abbreviations
- <return> often repeats the last command
  – Particularly useful when stepping through code
gdb summary – looking around

- `bt` – stack backtrace
- `up`, `down` – change current stack frame
- `list` – display source code (list n, list `function_name`)
- `print expression` – evaluate and print expression
- `display expression` – (re-)evaluate and print expression every time execution pauses.
  - `undisplay` – remove an expression from this recurring list.
- `info locals` – print all locals (but not parameters)
- `x` (examine) – look at blocks of memory in various formats
gdb summary – breakpoints, stepping

• break – set breakpoint. (break function_name, break linenumber, break file:linenumber)
• info break – print table of currently set breakpoints
• clear – remove breakpoints
• disable/enable – temporarily turn breakpoints off/on without removing them from the breakpoint table

• continue – resume execution to next breakpoint or end of program
• step – execute next source line
• next – execute next source line, but treat function calls as a single statement and don't step into them
• finish – execute to the conclusion of the current function
  – How to recover if you meant “next” instead of “step”