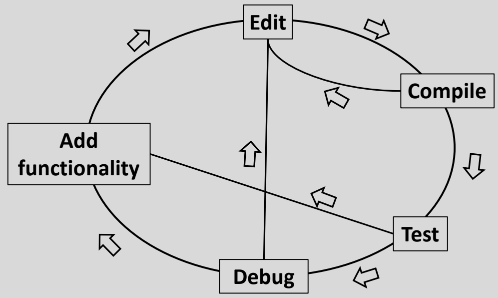
Debugging and debuggers

- **You have probably already had the experience of making a mistake in a program**
- **Speaking roughly, "debugging" is the process:**
  - After you know *that* your code is wrong
  - But before you know *how* it is wrong
- **Some kinds of debugging that don't need much tool support:**
  - Code review
  - Rubber duck debugging
  - Printf debugging

1

Debugging in the development cycle



2

What is a debugger for?

- **Not to fix your bugs for you, alas**
  - Computers aren't that smart yet
- **Instead, helps you examine your program's execution in more detail**
  - See what is happening if something is obviously wrong
  - Walk through normal execution, to compare with your expectations
- **Standard practice is source-level debugging**
  - I.e., the debugger shows your program in terms of its source code
  - For binaries, made possible by debugging information (enabled with compiler option `-g`)

3

The GNU debugger GDB

- **Standard command-line, source and binary-level debugger on Linux**
- **Start up with `gdb ./my_program`**
- **Supply program arguments to the GDB `run` command**
  - Abbreviated just `r`
- **Or, use `gdb --args ./my_program arg1 arg2`**
  - This mode doesn't work for redirection (shell <, >)
- **Today: using GDB as a source-level debugger**

4

`break`, `step`, `next`, `continue`

- **Normally, GDB will execute your program normally**
- **To get it to stop to let you look around, turn on a breakpoint with the command `break` (`b`)**
  - Argument can be function name, file and line number, others
- **When the breakpoint is reached, your program will stop and you can give GDB commands**
- **Run the program for one line with `step` (`s`)**
  - Variant `next` (`n`) does not go into other functions
- **To go back to full-speed execution, use `continue` (`c`)**

5

`print`

- **The most important command for examining program state is `print` (`p`)**
  - The argument is a source-level (i.e., C) expression
- **Some features to know about**
  - Can do arithmetic
  - Can refer to any variable in scope
  - Can call functions
  - Can do assignments
  - `p/x` prints in hexadecimal (other formats also available)

6

Crashes, interrupts, and `backtrace`

- **GDB will automatically stop if the program runs into a crash like a segfault (technically: a Unix signal)**
- **To stop in the middle of execution, type Ctrl-C**
  - Good for debugging infinite loops
- **The command `backtrace` (`bt`) summarizes all the currently executing functions**
  - Similar to what Java and Python print for an unhandled exception

7

Watchpoints

- **A watchpoint is sort of like a breakpoint, but based on data**
- **The command `watch` takes an argument like print**
- **A watchpoint stops execution when that value changes**
- **Useful for tracking down problems caused to pointers**
- **If you use a source-level expression, you'll usually get a software watchpoint, which is slow**
  - Later, we'll see hardware watchpoints

8

Overview: GDB without source code

- **GDB can also be used just at the instruction level**

Source-level GDB	Binary-level GDB
`step`/`next`	`stepi`/`nexti`
`break` <line number>	`break *`<address>
`list`	`disas`
`print` <variable>	`print` with registers & casts
`print` <data structure>	`examine`
`info local`	`info reg`
software `watch`	hardware `watch`

9

Disassembly and stepping

- **The `disas` command prints the disassembly of instructions**
  - Give a function name, or defaults to current function, if available
  - Or, supply range of addresses <start>`,`<end> or <start>`,+`<length>
  - If you like TUI mode, "`layout asm`"
  - Shortcut for a single instruction: `x/i` <addr>, `x/i $rip`
  - `disasm/r` shows raw bytes too
- **`stepi` and `nexti` are like `step` and `next`, but for instructions**
  - Can be abbreviated `si` and `ni`
  - `stepi` goes into called functions, `nexti` stays in current one
  - `continue`, `return`, and `finish` work as normal

10

Binary-level breakpoints

- **All breakpoints are actually implemented at the instruction level**
  - `info br` will show addresses of all breakpoints
  - Sometimes multiple instructions correspond to one source location
- **To break at an instruction, use `break *`<address>**
  - Address usually starts with 0x for hex
- **The `until` command is like a temporary breakpoint and a continue**
  - Works the same on either source or binary

11

Binary-level printing

- **The `print` command still mostly uses C syntax, even when you don't have source**
  - Registers available with `$` names, like `$rax`, `$rip`
  - Often want `p/x`, for hex
- **Use casts to indicate types**
  - `p (char)$r10`
  - `p (char *)$rbx`
- **Use casts and dereferences to access memory**
  - `p *(int *)$rcx`
  - `p *(char **)$r8`
  - `p *((int*)$rbx + 1)`
  - `p *(int*)($rbx + 4)`

12

Examining memory

- **The `examine` (`x`) command is a low-level tool for printing memory contents**
  - No need to use cast notation
- **`x`/<format> <address>**
  - Format can include repeat count (e.g., for array)
  - Many format letters, most common are `x` for hex or `d` for decimal
  - Size letter `b`/`h`/`w`/`g` means 1/2/4/8 bytes
- **Example: `x/20xg 0x404100`**
  - Prints first 20 elements of an array of 64-bit pointers, in hex

More useful printing commands

- **`info reg` prints contents of all integer registers, flags**
  - In TUI: `layout reg`, will highlight updates
  - Float and vector registers separate, or use `info all-reg`
- **`info frame` prints details about the current stack frame**
  - For instance, "saved rip" means the return address
- **`backtrace` still useful, but shows less information**
  - Just return addresses, maybe function names

Hardware watchpoints

- **To watch memory contents, use print-like syntax with addresses**
  - watch *(int *)0x404170
- **GDB's "Hardware watchpoint" indicates a different implementation**
  - Much faster than software
  - But limited in number
  - Limited to watching memory locations only
- **Watching memory is good for finding memory corruption**