If you did not setup the lecture repo on your container, please go view Sakai Announcement:
Today's Quiz, Videos, and Lab 01 - Part 1

/unc/comp211

Systems Fundamentals

Debugging !

Exercise: Stepping Through a Program

1. Pull today's materials and navigate into our directory:
   - **`cd lecture`**
   - **`git pull origin master`**
   - **`cd ls18-gdb`**

2. Compile debug.c with debugging symbols
   - **`gcc -std=c11 -g debug.c`**
   - **`-g`** adds debugging symbols suitable for GDB to the compiled program

An Introduction to Debugging Programs with gdb

- **gdb** is the **G**NU **De**b**u**gger

- Programs compiled with the **-g** flag include ***debugging symbols***
  - Normal machine code binaries *do not* include information about how source code file and variable names correspond to memory addresses, etc.
  - When an executable is compiled with *debugging symbols*, however, this information is included to make debugging possible.

- With a debugger you can
  - Trace a program line-by-line
    - **Step *into*** function calls
    - Continue to the ***next*** line (stepping *over* function calls)
  - Set **breakpoints** and let the program run until a breakpoint is reached
  - Inspect variable values, stack frames
  - And more!
    - Watches, conditional breakpoints, scripting, etc.

Stepping Into (`step`), Over (`next`), and Out (`finish`)

- **step** - **s** - **Step *Into*** - *step into any function calls on this line*
  - If you give the **where** command you will see the current function call stack

- **next** - **n** - **Step *Over* -** *continue to the next line in the current frame*
  - Function calls on current line *are evaluated,* but the debugger does not pause.
  - In the example above, the debugger would move to *char *clone* ...

- **finish** - **fin** - **Step *Out*** - complete the current function call and return
  - This command works in all function calls *except* for from within main.

Exercise: Stepping Through a Program

1. Pull today's materials and navigate into our directory:
   - **`cd lecture`**
   - **`git pull origin master`**
   - **`cd ls18-gdb`**

2. Compile debugging.c
   - **`gcc -std=c11 -g debugging.c`**

3. Begin a gdb Session
   - **`gdb ./a.out`**

4. Use the
   - **`start`**
   - **`Proceed`** `with a combination of:`
     - **`step`** (or just write s for short)
     - **`finish`**`(return from a function except main)`
   - Until you find the value returned by the 2nd call to max. Respond on pollev with what it was.
   - **`quit, y`** `– to end the gdb session`

Inspect the State of your Program

- The **`where`** (aliases: backtrace, bt) command prints stack frames
  - The function name and argument values are printed first.
  - Where you see filename.c:## the numbers are the line number execution is currently paused at.

- You can **`list`** 10 lines of code centered around the current line
  - Or list a specific range of lines: **`list start_line,end_line`**

- Information about the variables in a function call
  - **`info locals`** - local variables
  - **`info args`** - arguments passed to a function call

- Print a variable's value or an expression:
  - **`print [expression]`**

Be thoughtful about where to pause the debugger.

- Stepping through a non-trivial program line-by-line is painful
  - Better to instruct debugger *when* to pause and `continue` execution until then.
  - You can set many breakpoints and watchpoints in a program you're debugging.

- **Breakpoints** - pause at a specific line
  - `break [line#]`
  - Shorthand: `br [line#]`
  - Multifile programs: `br [file:line#]`

- **Conditional Breakpoints** - pause at a specific line *if some expression is true*
  - `break [file:line#] if [expression]`
  - Example: **break 21 if i == 3**
  - Useful for breaking in loops or frequently called functions when some expression is true.

- **Watchpoints** - pause when the value of an expression changes, print old and new values
  - `watch [expression]`
  - Example: **watch i > 10**
  - Any identifiers (variable names) used in expression must be in scope when watch is made.

- Try to set a breakpoint *just before* a bug in your code occurs.
  - If you hit it multiple times before the issue, then try to write a conditional breakpoint you'll hit just once.
  - Use watchpoints on variables or simple expressions to confirm your expectations on how variables are updating in an algo

Intentional Debugging

- Before fixing an issue, be **intentional** about quickly reproducing it.

- Rather than starting gdb up and robotically retyping each command...
  - ...make use of gdb's **-ex** *execute* arguments at the command line.

- You can "script" the commands you want gdb to run immediately:
  - **gdb -ex 'command one' ... -ex 'command N' [program]**
  - Ex: **gdb -ex 'break 26' -ex 'run' -ex 'info locals' ./a.out**
    - Sets a breakpoint at line 26 , runs the program, displays info about locals once paused.

- Think critically about **where** and **why** you want to pause execution.
  - Then encode this in a CLI command you can easily repeat (as above).
  - This can dramatically reduce the amount of time spent fixing a bug.

Fundamental gdb Commands

`start` - Begin execution of the program.

`where` - Display the current source code file, line number, and call stack
`list` - List the lines of source code with around where execution is

`break <file:line#>` - Set a breakpoint at line #
`break <file:line#> if <expression>` - Set a conditional breakpoint
`watch <expression>` - Set a watchpoint
`continue` or `c` - Continue execution until a breakpoint is reached

`step` or `s` - "Step Into" - Move to the next line of code, following function call jumps
`next` or `n` - "Step Over" - Move to the next line of code, without following into function calls
`finish` or `fin` - "Step Out" - Return from the current function

`info locals` or `info args` - Print the values of local variables or arguments
`print <expression>` or `p <expression>` - Print the current value of a variable

`quit` - End your gdb session (followed by yes)