Quick GDB Information

Displaying stuff:

- `print` *stuff* displays the value in (*stuff*) or evaluates something (such as `print sizeof(foo)`
- `print/x` *stuff* or `p/x` *stuff* displays *stuff* in hexadecimal
- `print $eax` displays the value of register `%eax`
- `x` *stuff* displays the value pointed at by *stuff*
- `display` *stuff* displays *stuff* after each command
- `undisplay` *stuff* removes display number *stuff*
- `info registers` displays the contents of all registers, including some you've never heard of, in both hexadecimal and decimal.
- `layout` gives you a multiple-window view of code, registers, and commands. Try `layout split` and `layout regs`.

Any of these commands can have a format argument appended:

- `/d` decimal
- `/u` unsigned
- `/x` hex
- `/t` binary
- `/i` instruction
- `/s` string (displays ascii values until a NUL is encounte
- `/c` char

`display /i $eip` is a useful command. It displays the value pointed at by `$eip` after each command and interprets it as an instruction. Basically it shows the next instruction to be run. Also, if a size and number are given, it will print that many of those size items after the given thing. So, for example, `x/20w $esp` displays 20 words at and after `$esp`. The available sizes are:

- `/b` byte
- `/w /code> word`

Breakpoints:

`Setting and removing single breakpoints:`

- `break (`*`some function`*`)`
- `break (`*`line number`*`)`
- `break *(`*`some memory address`*`)`
- `delete (`*`breakpoint number`*`)`

`Removing all breakpoints:`

- `clear [clears current break point]`
- `clear (`*`some function`*`)`
- `clear (`*`some line number`*`)`
- `clear *(`*`some memory address`*`)`

Running:

- `run` *`arg1 arg2 ...`* `starts or restarts the program with the given arguments`
- `run starts or restarts the program at full speed. If restarting, uses the same arguments used last time.`

- s or step steps by one line of source code, going into function calls. This only works after the program is running, so you usually need to set a breakpoint somewhere so that you can get to where you want to start stepping.
- n or next steps by one line of source code, not going into function calls
- stepi steps by one instruction, going into function calls
- nexti steps by one instruction, not going into function calls
- c or continue goes at full speed after a breakpoint
- kill end the running program
- enter do the same command again.
- finish step out of the current function.

Stack:

- bt or backtrace shows the current stack.
- frame *N* goes to the *N*th stack frame.
- info locals prints all local variables.
- info args prints all of the arguments to the current function as they are now (as opposed to as they were at the top of the function).
- call *function* calls *function*. Arguments can be provided. **Note:** this works by pushing arguments on the stack, resetting %eip to point the the function, and letting the program run. In some circumstances, this can fail
- whatis *something* prints the type of *something*

Command Line:

- set args (*stuff*) passes *stuff* as command line arguments to the program the next time run is used.
- file *stuff* sets *stuff* as the program to be run and debugged.

Lazy Typing:

- *Enter* (the key) at an empty command prompt repeats the last command. This is especially handy for step and next commands
- Ambigious abreviations will resolve to the last command with that abbreviation

Lazy Math:

- print/d 0x*some hex number* will convert it to decimal
- print/x *some decimal number* will convert it to hex
- print *complicated expression* will evaluate the expression and print the result in decimal or hex. You can use standard C notation, variables of your program, and register names ($eax, etc.)

Useful Tricks:

- info b will tell you how many times a breakpoint has been hit.
- continue *n* (or c *n*) will continue past a breakpoint *n* times. For example, if the fourth call to a function is the one that fails, you can use "c 3" to skip the first three calls.
- You can combine the two above tricks to deal with a function that crashes after many calls. Set a breakpoint in the function, run the program, and type "c 9999999". When it crashes, use info b to find out how many times the function was called. Then rerun the program and use continue *n-1* to get to the invocation that crashes.

This page is maintained by Geoff Kuenning.