David's ~~Super~~ Marginally-Awesome GDB Guide for CSE 12

Setting Breakpoints

break *location*	Sets a breakpoint at a specific location inside your program. A valid location could be a function name (e.g. break main, break someFunc), a line number (e.g. 30, 50), a filename:linenumber (e.g. hw0.c:30), or an offset from the current line (e.g. break +2, break -4)
break *location* if *cond*	This sets a breakpoint a specific location but it will break only if the condition is true. For example, you have a loop and you only want to break when the loop counter is greater than a specific value (say 9000 in this case): for (int i = 0; i < 10000; i++) { … } You can use the command break *location* if i > 9000 to accomplish this. If you want to modify an existing breakpoint to have a condition, use the condition command inside gdb.
tbreak *location*	Sets a **t**emporary **break**point in your code. This will automatically remove itself after stopping at the location.
info break	Show all the breakpoints currently set inside your code as well as some additional information (such as how many times a breakpoint was hit).
delete *break#*	Delete a breakpoint that you had previously set using *break*. Get the break# from the info break command. Also if you don't want to permanently delete breakpoints you can use the enable and disable command (read the GDB docs).
rbreak *substr*	A good to know command. This will break on all functions matching a substring *substr*. So if there were functions doMath, doSleep, doSomething then typing in rbreak do will break on all three of those functions.
ignore *break# n*	Don't stop at a particular breakpoint specified by *break #* until it has hit *n* additional times.
continue *n*	The "oops, I didn't mean to break here yet" command. Continue executing your program. If you provide an *n*, it will instruct gdb to ignore the current breakpoint until it has hit *n* additional times.

Now that I've hit a breakpoint….

next *n*	Executes the next line of the program that is displayed on the screen currently. If the next line is a function call, this will **skip over** the function. If you provide an integer value for *n* it will execute the next command *n* times.
step *n*	Executes the next line of the program that is displayed on the screen currently. If the next line is a function call, this will **jump into** the function. If you provide an integer value for *n* it will execute the step command *n* times.
finish	The "oops, I didn't mean to step into the function" command. Continue executing your program until the end of the function (at which point it will display return values and other goodies).
print */f variable*	This command **p**rints out a variable to the screen (the /f is optional). This is different from e**x**amine because it takes into account what type the variable is. Examine provides lower level examination options. Also you can provide format flags to control how it will be displayed out to screen. These are valid values for *f*. x – print as hex d – print as decimal number u – print as unsigned decimal number o – print as octal t – print as binary c – print as a single character f – print as float s – print as string SPECIAL AWESOMENESS: You can change variables. So say if you're working on bank software,

	you can do something like this: print account_balance = 100000000 to change variables on-the-fly.
x */nfu addr*	This command has nearly the same functionality as **p**rint. However you can specify the number *n* (default = 1) of *u* (default = 4) sized chunks of memory and specifying the *f* format to display it in (default = x [hex]). These are some valid values for u: b – byte (1 byte duh on x86) h – halfword (2 bytes on x86) w – word (4 bytes on x86) So typing in x/10cb *a_string_var* would print out 10 characters of the string *a_string_var*.
display */f variable*	If you're looking a specific variable frequently, you may want to "pin" the value to the screen. This works similarly to print. Use info display to show all displayed variables and use undisplay display# (or delete display display#) to remove it. Similarly to enabling and disabling breakpoints, you can use enable and disable commands to temporarily show/hide the value.
list	Display 10 lines of code around the current line about to be executed. This is helpful in determining where you are inside your code.
where	Display the current traceback (aka the stack) for your program. This shows you which functions you've called to get to the line that you are executing now.
up/down	Goes to the previous/next stack frame. So lets say I have a function parent() that calls child(). If I am currently stopped in child() and I want to examine variables inside the parent, I need to use up to go to the parent's stack frame and examine the parent's variables. To go back, use down to go back to child's stack frame.