GDB

`gdb` is the GNU debugger. Before trying to use it, make sure you compile your code with the `-g` option, which tells the compiler to include debugging information in the files it creates. If you forget to do this, you will notice that `gdb` does not give you line numbers or other useful things. Here is a quick reminder of the basic commands for navigating gdb (commands can be abbreviated with the underlined characters):

file This specifies the program that you are going to debug. It is generally the first command you give to gdb. Example: `file a.out`

set args If your program needs command line arguments, use this command to specify them all, after you do `file`, but before you start executing. Example: `set args -somearg 7 --somelongarg  whatever`

run Begins executing the program from the start until stopped by a breakpoint, watchpoint, signal etc.

continue Continues executing the current program until stopped by a breakpoint, watchpoint, signal, etc.

start Start execution, but break at the start of `main`.

next Go to the next line of code. This will step over a function call, evaluating the entire function before stopping (unless a breakpoint, watchpoint, or signal stops execution).

step Go to the next line of code, but step into a function.

finish Run until the current function returns.

backtrace Show the current call stack

up Examine the stack frame which called the current one.

down Examine the stack frame called by the current one.

break Set a breakpoint. When done with no arguments, sets a breakpoint on the current line of code. An argument of the form `file:lineNumber` can be specified to break at the given location, or an argument which

names a function can be given to break at entry to the specified function. Note that in emacs, the command `C x (space)` when done in your C code will tell gdb to break wherever the emacs point (cursor) is. Examples:

```
break
break myFile.c:37
break someFunction
```

When you do this command, it will give you output like this:

```
Breakpoint 3 at 0x8048475: file myFile.c, line 37.
```

The number that it gives you is important, because that is what you will use to refer to that breakpoint in the future.

cond Make a breakpoint (un)conditional. If you give cond only the breakpoint number, that breakpoint becomes unconditional. If you specifiy the breakpoint number and a condition (a C expression), then that condition gets evaluated to determine whether or not to stop the program when the breakpoint is reached. Examples:

```
cond 5
cond 3 (x == 7 || y > 0)
```

watch Create a hardware watchpoint of some expression. A watchpoint makes the program stop when the value of that expression changes. Note that since this has to use some special hardware, you can only have a few at once, and they can't be too complicated. Best is to watch some variable, or one memory location. Example: `watch x`

info break Tell the status of all current breakpoints and watchpoints.

del Delete a breakpoint or watchpoint (by number). You can specify more than one number. Examples:

```
del 3
del 1 4 7
```

dis Disable a breakpoint or watchpoint (by number). A disabled breakpoint still exists and can be re-enabled, but will not stop execution of the program. `dis` has the same sytax as `del`

en Enable a breakpoint or watchpoint (by number). Re-enable a disabled breakpoint. `en` has the same syntax as `dis`

print Print takes an expression as an argument, evaluates, and prints it out. When gdb prints out a value, it assigns it to a special gdb variable which starts with a $. You can use these just like any other variable, an they never go out of scope. Note that you can print any C expression, which includes function calls, and assignments. If you print a function call, it will be executed, with any side effects that it might have. Additionally, breakpoints will stop execution if they come up. If you print an assignemnt, it will actually change the value of the variable in your program. Examples:

```
p x
p x + y - 4
p fact(7)
p z = 7    /* changes the value of z to 7, and then prints 7 */
```

As another example, suppose you want to know when `*ptr` changes, but `ptr` may go out of scope first. Then you might do this:

```
(gdb) p ptr
$1 = (int *) 0xb7d96ff8
(gdb) watch *($1)
```

p/x Just liek print, except print in hex.

help Ask gdb to give you help on its commands

You can put commands in a file called `.gdbinit` (including the dot), which gdb will read and execute when it loads. For example, if you are constantly debugging one program with the same arguments, and always want to break on entry into an error function you wrote, you might have a `.gdbinit` file that looks like this:

```
file my_program
set args -foo 3 -bar 17

break myErrorFun
```

Other tools

If you can't find your problem with gdb, you might want to try `valgrind` or linking with `-lefence`. I recommend trying to gdb your program with libefence, and then resorting to valgrind if that fails. You can read more about these two tools in their respective man-pages (man valgrind, and man efence respectively).