Getting Started with gdb

The first step in using gdb (or most any other debugging tool) is to compile the code with *debugging symbols* —extra information to help a debugging tool understand what the layout of the code and data in memory is—included in the binary. The - *g* option to gcc requests that it include this debugging information, but if you are using gdb in particular, you should use - *ggdb3* , which requests the maximum amount of debug information (*e.g.* , it will include information about preprocessor macro definitions) in a gdb-specific format. Note that if you compile your program in multiple steps (object files, then linking), you should include - *ggdb3* at all steps.

Once you have your program compiled with debugging symbols, you need to run *gdb* . You can run *gdb* directly from the command line, however, it is much better to run it from inside of emacs. To run *gdb* inside *emacs,* use the command *M-x gdb* (that is either *ESC x* or *ALT-x* depending on your keyboard setup, then type gdb, and hit enter). At this point, emacs should prompt you for how you want to run *gdb* ("Run gdb (like this):"), and provide a proposed command line. Typically the options that emacs proposes are what you want; however, you may want to change the name of the program to debug (specified as the last argument on the command line). Once you are happy with the command line, hit enter to start gdb.

At this point, you will end up with a buffer titled **gdb** , or **gdb-prog** (where prog is the name of the program). The stars in the buffer name indicate that the buffer corresponds to interaction with a process, not a file on disk. This buffer should contain some output from *gdb* which tells you its version, some information about where to find the manual, and a message about the ``help'' command. The last lines of output should be (replace "yourProgram" with the name of the program you are debugging):

```
Reading symbols from yourProgram...done.
(gdb)
```

Note that if you instead get the following, it indicates that you did not compile with debugging symbols (in which case, you should recompile with debugging symbols):

```
Reading symbols from yourProgram...(no debugging symbols found)...done.
(gdb)
```

Also, this message indicates that your requested program does not exist in the current directory:

```
yourProgram: No such file or directory.
(gdb)
```

Note that the (*gdb)* at the start of the last line of the output is gdb's command prompt. Whenever it displays this prompt, it is ready to accept a command from you. The first commands we are going to learn are:

start: Begin (or restart) the program's execution. Stop the program (to accept more commands) as soon as execution enters *main* .

run: This command runs the program (possibly restarting it). It will not stop unless it encounters some other condition that causes it to stop (we will learn about these later).

step: Advance the program one "step", in much the same way that we would advance the execution arrow when executing code by hand. More specifically, *gdb* will execute until the execution arrow moves to a different line of source code, whether that is by going to the next line, jumping in response to control flow, or some other reason. In particular, step will go into a function called by the current line. This command can be abbreviated *s* .

next: Advance the program one line of code. However, unlike *step,* if the current line of code is a function call, *gdb* will execute the entire called function without stopping. This command can be abbreviated *n* .

print: The print command takes an expression as an argument, evaluates that expression, and prints the results. Note that if the expression has side-effects, they will happen, and will affect the state of the program (*e.g.* , if you do print *x = 3* , it will set *x* to 3, then print 3). You can put */x* after *print* to get the result printed in hexadecimal format. This command can be abbreviated *p* (or *p/x* to print in *hex*). Every time you print the value of an expression, *gdb* will remember the value in its own internal variables which are named *1,1,2* , etc (you can tell which one it used, because it will say which one it assigned to when it prints the value— *e.g., $1 = 42*). You can use these $ variables in other expressions if you want to make use of these values later. *gdb* also has a feature to let you print multiple elements from an array—if you put *@number* after an lvalue, *gdb* will print number values starting at the location you named. This feature is most useful with arrays—for example, if a is an array, you can do *p a[0]@5* to print the first 5 elements of *a* .

display: The *display* command takes an expression as an argument, and displays its value every time *gdb* stops and displays a prompt. For example display *i* will evaluate and print *i* before each *(gdb*) prompt. You can abbreviate this command *disp* .

If you hit enter without entering any command, *gdb* will repeat the last command you entered. This feature is most useful when you want to use *step* or *next* multiple times in a row.

Note that if you need to pass command line arguments to your program, you can either write them after the *start* or *run* command (*e.g., run someArg anotherArg*), or you can use *set args* to set the command line arguments.