GDB Tutorial

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.

This is a brief description of some of the most commonly used features of gdb.

Compiling

To prepare your program for debugging with gdb, you must compile it with the `-g` flag. So, if your program is in a source file called `memsim.c` and you want to put the executable in the file `memsim`, then you would compile with the following command:

```
gcc -g -o memsim memsim.c
```

Invoking and Quitting GDB

To start gdb, just type `gdb` at the unix prompt. Gdb will give you a prompt that looks like this: `(gdb)`. From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed). Or, you can start gdb and give it the name of the program executable you want to debug by saying

`gdb` *`executable`*

To exit the program just type `quit` at the `(gdb)` prompt (actually just typing `q` is good enough).

Commands

help

Gdb provides online documentation. Just typing `help` will give you a list of topics. Then you can type `help` *`topic`* to get information about that topic (or it will give you more specific terms that you can ask for help about). Or you can just type `help` *`command`* and get information about any other command.

file

`file` *`executable`* specifies which program you want to debug.

run

`run` will start the program running under gdb. (The program that starts will be the one that you have previously selected with the `file` command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying `run` instead of the program name:

```
run 2048 24 4
```

You can even do input/output redirection: `run > outfile.txt`.

break

A ``breakpoint'' is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the `break` command.

`break` *`function`* sets the breakpoint at the beginning of *`function`*. If your code is in multiple files, you might need to specify *`filename:function`*.

`break` *`linenumber`* or `break` *`filename:linenumber`* sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

delete

`delete` will delete all breakpoints that you have set.

`delete` *`number`* will delete breakpoint numbered *`number`*. You can find out what number each breakpoint is by doing `info breakpoints`. (The command `info` can also be used to find out a lot of other stuff. Do `help info` for more information.)

clear

`clear` *`function`* will delete the breakpoint set at that function. Similarly for *`linenumber`*, *`filename:function`*, and *`filename:linenumber`*.

continue

`continue` will set the program running again, after you have stopped it at a breakpoint.

step

`step` will go ahead and execute the current source line, and then stop execution again before the next source line.

next

`next` will continue until the next source line in the current function (actually, the current innermost stack frame, to be precise). This is similar to `step`, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with `step` execution will stop at the first line of the function that is called.

until

`until` is like `next`, except that if you are at the end of a loop, `until` will continue execution until the loop is exited, whereas `next` will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.

list

`list` *`linenumber`* will print out some lines from the source code around *`linenumber`*. If you give it the argument *`function`* it will print out lines from the beginning of that function. Just `list` without any arguments will print out the lines just after the lines that you printed out with the previous `list` command.

print

`print` *`expression`* will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called `list`, do

```
print list[0]@25
```