98-172 : Great Practical Ideas for Computer Scientists

GDB

Josh Zimmerman (jzimmerm@andrew)

What is GDB?

GDB, or the GNU debugger, is a command-line debugging tool that lets you examine the state of your program and its variables while the program is running.

It's an immensely powerful program, so for now we'll only go over a few of the more common things that you can do with it.

In this handout we'll focus on using it to debug C programs.

We'll be walking through a simple C program, available at `http://www.andrew.cmu.edu/course/98-172/labitations/demo.c`. You can easily download this directly to your AFS space using the command `wget`:

```
wget http://www.andrew.cmu.edu/course/98-172/labitations/demo.c
```

This will download the file into `demo.c` in your working directory.

Starting to use GDB

In order to use GDB effectively, you'll want to compile your C code with the `-g` flag to gcc. This flag tells gcc that it should include debugging symbols in the compiled binary, which GDB can use to give you more helpful information in debugging your program.

Then, to run your program (say, `a.out`), just type `gdb ./a.out`. (Note: don't give any of your program's arguments yet).

Next, gdb will open. Type `run [arguments]` and gdb will run your program with the given arguments.

Let's look at a specific example. Follow along with `demo.c`.

```
$ gcc -g demo.c -o demo
$ gdb ./demo
<omitted some text that gdb prints out on launch>
(gdb) run 2 3
Starting program: /afs/.../demo 2 3
f(2, 3) = 8

Program exited normally.
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.80.el6_3.6.x86_64
```

(Feel free to ignore the line that says "Missing separate debuginfos" — it occurs because the Andrew Linux machines don't have a certain package installed. I'll omit them from the rest of the handout.)

Breakpoints, watchpoints

Just running a program normally is not terribly useful and not the point of using GDB. Breakpoints and watchpoints are a very large part of what makes it useful.

A *breakpoint* lets you break (stop execution of a program) on a specific line (or even on a specific assembly instruction — you'll learn more about that in 15-213). A *watchpoint* allows you to stop execution of code when the value of a certain expression changes its value. (Note that watchpoints can sometimes make your code run *very* slowly.)

There are several commands that are useful in combination with breakpoints and watchpoints, which we'll now walk through.

If you type `break n` (where `n` is a line number) or `break foo` (where `foo` is a function), execution will stop either on that line or on entry to that function. If you're debugging multiple files, you can disambiguate line numbers and function names by prepending the filename and a colon. To break on line 42 of file `test.c`, you would do `break test.c:42`.

For example, if we're suspicious about the behavior of the function `f` in the file `demo.c`, we can do the following:

```
(gdb) break demo.c:f
Breakpoint 1 at 0x40055e: file demo.c, line 5.
(gdb) run 2 3
Starting program: /afs/.../demo 2 3

Breakpoint 1, f (a=2, b=3) at demo.c:5
5       int res = 1;
```

If we're not compiling with any other files, we can also omit the `demo.c` and just type `break f`.

Now that we have a breakpoint, we can interact with the stopped program in several ways, with the `step`, `next`, and `continue` commands.

`step` will execute one line of code (and `step 10` will execute 10 lines of code). `next` will execute one line of code, skipping any function calls (you can also give it a number of lines to execute). Finally, `continue` simply resumes execution of your code.

Let's pick up where we left off before. (Note: hitting enter in gdb without typing a command will run the last command you typed.) After we stop execution, we can print the values of variables to examine the state of our program.

```
(gdb) step
6       while (b != 0) {
(gdb)
7           res *= a;
(gdb) print a
$1 = 2
(gdb) print b
$2 = 3
(gdb) print res     # Note: res hasn't changed: we stopped before running line 7
$3 = 1
(gdb) step
8           b--;
(gdb) print res
$4 = 2
(gdb) step
```

```
6       while (b != 0) {
(gdb) print b
$5 = 2
(gdb) continue
Continuing.
f(2, 3) = 8

Program exited normally.
```

Sometimes, we only want to break at a specific place if a certain condition is true. We do this with `break X if cond`. Execution of code will only stop at `X` (which can be a function or a line number) if `cond`, which is a boolean expression, evaluates to true.

For example, if I'm suspicious of what the function `f` does when given a negative value for `b`, I can do the following:

```
$ gdb ./demo
<text omitted>
(gdb) break f if b < 0
Breakpoint 1 at 0x40055e: file demo.c, line 5.
(gdb) run 2 3
Starting program: /afs/.../demo 2 3
f(2, 3) = 8

Program exited normally.
(gdb) run 2 -1
Starting program: /afs/.../demo 2 -1

Breakpoint 1, f (a=2, b=-1) at demo.c:5
5       int res = 1;
```

`watch`. If you type `watch expr` (where expr is a variable, an arithmetic expression, a boolean expression, etc) then execution of your program will stop whenever the value of that expression changes. This can be useful if you have a variable changing to a value you don't expect — it lets you find out where exactly that change happens. Note that `watch` makes your code run very slowly since that condition has to be checked anywhere an update that affects the expression could possibly occur.

```
$ gdb ./demo
(gdb) break main
Breakpoint 1 at 0x400590: file demo.c, line 14.
(gdb) run 2 3
Starting program: /afs/.../demo 2 3

Breakpoint 1, main (argc=3, argv=0x7fffffffe0e8) at demo.c:14
14      if (argc < 3) {
(gdb) watch c    # We need to do this when c is in scope or gdb won't know what we mean.
Hardware watchpoint 2: c
(gdb) continue
```

```
Continuing.
Hardware watchpoint 2: c

Old value = 0
New value = 8
main (argc=3, argv=0x7fffffffe0e8) at demo.c:21
21      printf("f(%d, %d) = %d\n", a, b, c);
```

Listing and removing breakpoints

If we have accumulated a lot of breakpoints, it can be useful to list and remove some of them. We can do this with `info breakpoints` and `delete`. (Note: I added a few breakpoints so you might not have exactly what I have.)

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400590 in main at demo.c:14
    breakpoint already hit 1 time
2       hw watchpoint  keep y                      c
    breakpoint already hit 1 time
3       breakpoint     keep y   0x000000000040055e in f at demo.c:5
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      c
    breakpoint already hit 1 time
3       breakpoint     keep y   0x000000000040055e in f at demo.c:5
```

Using `backtrace` (also known as `where`) to see who called a function

The `backtrace` command (abbreviated `bt`) shows you a backtrace of function calls. Two useful commands are `backtrace` and `backtrace full`. `backtrace full` will also print out local variables:

```
(gdb) backtrace
#0  f (a=2, b=3) at demo.c:6
#1  0x00000000004005f7 in main (argc=3, argv=0x7fffffffe0e8) at demo.c:20
(gdb) backtrace full
#0  f (a=2, b=3) at demo.c:6
        res = 1
#1  0x00000000004005f7 in main (argc=3, argv=0x7fffffffe0e8) at demo.c:20
        a = 2
        b = 3
        c = 0
```

Using `list` to print out source code

If you want more context than gdb normally gives about where in your program's exeuction you are, you

can use `list` to print source code centered around the current place you're stopped, a specific function, or a specific line number. `list` can take no arguments (it defaults to printing code near where you currently are in the program), a line number, or a function name.

Getting help

Need help with `gdb`? There are a lot of great resources.

If you're not sure how to use a certain command, type `help command-name-here`, filling in the name of the command for `command-name-here`.

If that doesn't help, `http://www.yolinux.com/TUTORIALS/GDB-Commands.html` is a good resource, as is `http://www.chemie.fu-berlin.de/chemnet/use/info/gdb/gdb_toc.html`. The second is a far more comprehensive source, meaning it might be more difficult to find the information you need in it.

If all else fails, Google or ask someone on course staff and we'll be glad to help.