GDB Tutorial

A Walkthrough with Examples

CMSC 212 - Spring 2009

Last modified March 22, 2009

What is gdb?

- "GNU Debugger"
- A debugger for several languages, including `C` and `C++`
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like *segmentation faults* may be easier to find with the help of gdb.
- `http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html` - online manual

Additional step when compiling program

- Normally, you would compile a program like:

```
gcc [flags] <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors prog1.c -o prog1.x
```

- Now you add a -g option to enable built-in debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors -g prog1.c -o prog1.x
```

Starting up gdb

Just try "gdb" or "gdb `prog1.x`." You'll get a prompt that looks like this:

```
(gdb)
```

If you didn't specify a program to debug, you'll have to load it in now:

```
(gdb) file prog1.x
```

Here, `prog1.x` is the program you want to load, and "file" is the command to load it.

Before we go any further

gdb has an interactive shell, much like the one you use as soon as you log into the linux grace machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

Tip

If you're ever confused about a command or just want more information, use the "`help`" command, with or without an argument:

```
(gdb) help [command]
```

You should get a nice description and maybe some more useful tidbits...

Running the program

To run the program, just use:

```
(gdb) run
```

This runs the program.

- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
- If the program *did* have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

So what if I have bugs?

Okay, so you've run it successfully. But you don't need `gdb` for that. What if the program *isn't* working?

Basic idea

Chances are if this is the case, you don't want to run the program without any stopping, breaking, etc. Otherwise, you'll just rush past the error and never find the root of the issue. So, you'll want to *step through* your code a bit at a time, until you arrive upon the error.

This brings us to the next set of commands...

Setting breakpoints

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "`break`." This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

This sets a breakpoint at line 6, of `file1.c`. Now, **if** the program ever reaches that location when running, the program will pause and prompt you for another command.

Tip

You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

More fun with breakpoints

You can also tell gdb to break at a particular function. Suppose you have a function `my_func`:

```
int my_func(int a, char *b);
```

You can break anytime this function is called:

```
(gdb) break my_func
```

Now what?

- Once you've set a breakpoint, you can try using the `run` command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "`continue`" (Typing `run` again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

- You can single-step (execute *just* the next line of code) by typing "`step`." This gives you really fine-grained control over how the program proceeds. You can do this a *lot*...

```
(gdb) step
```

Now what? (even more!)

- Similar to "`step`," the "`next`" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.

```
(gdb) next
```

Tip

Typing "`step`" or "`next`" a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

Querying other aspects of the program

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like *the values of variables*, etc. This *might* be useful in debugging. :)
- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:

```
(gdb) print my_var
(gdb) print/x my_var
```

Setting watchpoints

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a *watched* variable's value is modified. For example, the following `watch` command:

```
(gdb) watch my_var
```

Now, whenever `my_var`'s value is modified, the program will interrupt and print out the old and new values.

Tip

You may wonder how gdb determines which variable named `my_var` to `watch` if there is more than one declared in your program. The answer (perhaps unfortunately) is that it relies upon the variable's **scope**, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent :(.

Example programs

- Some example files are found in `~/212public/gdb-examples/broken.c` on the linux grace machines.
- Contains several functions that each should cause a segmentation fault. (Try commenting out calls to all but one in `main()`)
- The errors may be easy, but try using `gdb` to inspect the code.

Other useful commands

- `backtrace` - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)
- `where` - same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

Look at sections 5 and 9 of the manual mentioned at the beginning of this tutorial to find other useful commands, or just try `help`.

gdb with Emacs

Emacs also has built-in support for gdb. To learn about it, go here:
`http://tedlab.mit.edu/~dr/gdbintro.html`

More about breakpoints

Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping...

Basic idea

Once we develop an idea for what the error could be (like dereferencing a `NULL` pointer, or going past the bounds of an array), we probably only care if such an event happens; we don't want to break at each iteration regardless.

So ideally, we'd like to *condition* on a particular requirement (or set of requirements). Using **conditional** breakpoints allow us to accomplish this goal...

Conditional breakpoints

Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same `break` command as before:

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

This command sets a breakpoint at line 6 of file `file1.c`, which triggers only if the variable `i` is greater than or equal to the size of the array (which probably is bad if line 6 does something like `arr[i]`). Conditional breakpoints can most likely avoid all the unnecessary stepping, etc.

Fun with pointers

Who *doesn't* have fun with pointers? First, let's assume we have the following structure defined:

```
struct entry {
  int key;
  char *name;
  float price;
  long serial_number;
};
```

Maybe this struct is used in some sort of hash table as part of a catalog for products, or something related.

Using pointers with gdb I

Now, let's assume we're in gdb, and are at some point in the execution after a line that looks like:

```
struct entry * e1 = <something>;
```

We can do a lot of stuff with pointer operations, just like we could in C.

- See the value (memory address) of the pointer:

```
(gdb) print e1
```

- See a particular field of the struct the pointer is referencing:

```
(gdb) print e1->key
(gdb) print e1->name
(gdb) print e1->price
(gdb) print e1->serial_number
```

Using pointers with gdb II

- You can also use the dereference (*) and dot (.) operators in place of the arrow operator (->):

```
(gdb) print (*e1).key
(gdb) print (*e1).name
(gdb) print (*e1).price
(gdb) print (*e1).serial_number
```

- See the entire contents of the struct the pointer references (you can't do this as easily in C!):

```
(gdb) print *e1
```

- You can also follow pointers iteratively, like in a linked list:

```
(gdb) print list_prt->next->next->next->data
```