Tip 14: easier interrogations with GDB variables

Ben Klemens

29 October 2011

level: intermediate debugger
purpose: name suspects

OK, after Tip 13 I hope you're sold on the value of a debugger, and you are using it to see your data as it gets transformed by your program.

There are graphical front-ends for the debugger, which have built-in means of showing you certain common structures. But you also no doubt have your own favorite structures, and need tailored routines for viewing those structures the way you are used to seeing them.

This tip will cover some useful elements of GDB that will help you look at your data with as little cognitive effort as possible. All of the commands to follow go on the GDB command line. I assume you can already use GDB to do the exercise at the end of Tip 13.

Here's tip zero: the @ shows you a sequence of elements in an array. For example, here are a dozen of a `gsl_vector*`'s elements:

```
print *vector->data@12
```

Note the star at the head of the expression; without it we'd get a sequence of a dozen pointers. Also, herein I'll abbreviate `print` to `p`.

Next tip, which will only be new to those of you who didn't read the GDB manual, which is probably all of you. You can generate convenience variables, to save yourself some typing. For example, if you want to inspect an element deep within a hierarchy of structures, you can do something like

```
set $vd = my_model->dataset->vector->data
p *$vd@10
```

That first line generated the convenience variable to substitute for the obnoxious path. Following the lead of some shells, a dollar sign indicates a variable, and use `set` on first use. Unlike the shell, you need a dollar sign on the `set` line. The second line demonstrates a simple use. We don't save much typing here, but over the course of a long interrogation of a suspect variable, this can certainly pay off.

This isn't just a label; it's a real variable that you can modify:

```
p *$vd/14  #print the pointed to item divided by 14.
p *($vd++)  #print the pointee, and step forward one
```

That second line uses the only piece of pointer arithmetic worth knowing: that adding one to a pointer steps forward to tne next item in the list. [More on this in the next few tips.]

This is especially useful because hitting the enter key without any input repeats the last command. Since the pointer stepped forward, you'll get a new next value every time you hit enter, until you get the gist of the array. This is also useful should you find yourself dealing with a linked list. Pretend we have a function that displays an element of the linked list; then:

```
show_structure $list
show_structure $list->next
```

and leaning on the <enter> key will step through the list. [I'm not being creative here. This is still all from the manual.]

Tip 15 will be about making that imaginary function to display a data structure a reality.

But for now, here's one last trick about these $ variables. Let me cut an paste a few lines of interaction with a debugger in the other screen:

```
(gdb) p *out->parameters
$54 = {
  vector = 0x8056380,
  matrix = 0x0,
  names = 0x80561c0,
  text = 0x0,
  textsize =     {0,
    0},
  weights = 0x0,
  more = 0x0
}
```

You probably don't even look at it anymore, but notice how the output to the print statement starts with `$54`. Indeed, every output is assigned a variable name, which we can use like any other:

```
(gdb) p *$54->vector->data
$55 = 25.0001
(gdb) p $55*4
$56 = 100.0004
```

To be even more brief, `$` is the last output, so the above could have been:

```
(gdb) p out->parameters
(gdb) p *$
(gdb) p *$->vector->data
$55 = 25.0001
(gdb) p $*4
$56 = 100.0004
```

The `p *$` form is especially useful when you expected data but got the address of the data because you forgot that star.