Tip 72: Export one-off variables

Ben Klemens

23 February 2012

level: intermediate POSIX
purpose: faster control of your programs

First, I hope you are by now familiar with the difference between a shell variable and an exported shell variable.

The shell variable is just some text that gets substituted in later use on the shell. Like any old variable, it saves typing and redundancy.

```
workingdir=/home/b/tech/code_snippets/enviro
ls $workingdir
cd $workingdir
```

[Users of shells in the C shell family need to use `set VAR value` and `setenv VAR value`. That said, I'm going to use the POSIX-standard forms for the rest of this.]

When a variable is *exported*, then it rises in importance, and is now a means of communicating to child programs. When a program spawns a child (in C, when you call `fork()`; on the shell, when you type in a command and hit <enter>), then the environment variables as they stand are sent to the child program. Appendix A of *Modeling with Data* covers how obnoxiously easy it is to read environment variables from C.

Great—so we can call `export VAR=value` and our child programs will have `VAR` on hand. But we can do it even faster.

Try this:

```
PANTS=kakhi env
```

You may know `env` as a program that does nothing but list the environment variables. When you run this, you should see `PANTS` there at the end of the list. Or run it through grep (`env|grep PANTS`) to make sure it's there.

So by setting a variable on the same line as the command, the command receives the variable as an export. You can do this with several variables if need be:

```
PANTS=kakhi PLANTS="ficus fern" env
```

Notice how we handled the space in the `PLANTS` variable.

Now that you've tried this trick on your command line, try running `env` again without preface. You will find that `PANTS` isn't there. The variable really did get set and exported for the duration of a single command.

What can we do with this?

We can write our C programs around this. As per Appendix A, which gives a complete example on p 384, we can call `int varvalue = atoi(genenv("VAR"));` in our C programs to get the value of an environment variable and convert from string to integer. So you could do something like

```
RUNS=32 ./do_simulation
```

for the variables that you would be tweaking all the time. People are more used to flags like `-r 32`, and they're about as easy to parse, but if you like this form, run with it.

`make` cares deeply about environment variables, and expects to be making use of them. Also, the `configure` script that you use to set up programs packaged with Autotools (think `./configure; make;sudo make install`) also cares about environment variables.

Now things are getting useful. Make has a default rule for compiling C code, and (among others) it depends on two key variables: `CFLAGS` and `LDLIBS`. You might as well set up your `CFLAGS` once and for all via

```
export CFLAGS="-g -Wall -O3 -std=gnu99"
```

(or whatever works for you) in your `.bashrc`, `.zshrc` or whatever is appropriate.

If your program depends on any libraries, then they are specified via `LDLIBS`. Now for the magic: if you have those two variables set, then for a simple program, you don't need a makefile at all. We set `CFLAGS` for good above, so let's say we have a program that depends on glib-2.0 and is named `pantspress.c`. Then you may be able to compile it via

```
LDLIBS=-lglib-2.0 make pantspress
```

You will be running Make many times on a program, so you might want to at some point generate a makefile. It may consist of only a single line defining `LDLIBS=-lglib-2.0` and still work.

But you only run `configure` once or twice on a package, and it may require odd flags to get working. In this case, the form here is great:

```
#turn off optimization, generate all warnings, link to glib:
CFLAGS="-g -Wall -O0" LDADD="-lglib-2.0" ./configure
```

Did you notice that I used `LDADD` this time? GNU Configure likes that better than GNU Make's `LDLIBS`. Neither is POSIX-standard.