by Jeffreys Copeland and Haemer

PAUL STODDARD

*Sir, I have found you an argument; but I am not obliged to find you an understanding.*
– Samuel Johnson

# We Argue with Our Code

There are several ways to get information into a UNIX program to affect the way it behaves. In this column we're going to talk about one of these: command-line arguments.

First, though, let's briefly list all the ways. There are four:

1. The environment – Each UNIX process inherits a suite of information from its parent process, which it can interrogate. All the programs you run from the shell have a collection of state information, including the current working directory and the current user's ID, which the operating system keeps track of during the life of the process.

These values are private to the process. Once the process has started running, it can manipulate most of them, but you can't change them from outside. The fact that you can't change an environment from the outside is very important. In all the programming newsgroups, you'll find a never-ending stream of posts asking, "How do I change the value of an environment variable in a parent process from one of its children?" The answer (always found in the FAQ) is, "You can't."

Nevertheless, if you tweak the environment before you start a program, it can examine that environment at start-up time and decide how you want it to run. Most of this state information is held in environment variables, which you can see with the command `env`.

We could say a lot more about the environment, but we won't.

2. Interprocess communication – Running processes can talk directly to one another in a host of interesting ways, collectively called "IPCs." The most common of these are pipes (named and unnamed), signals, sockets, semaphores, message-passing and shared memory. Not all vendors supply all of these, but we wouldn't call anything "UNIX-like" if it lacked unnamed pipes, signals or sockets.

3. Input – Duh.

4. Command-line arguments – Now we're at the subject of this column, so let's dig deeper.

## A Little Background

The only Standard-C function with two alternative prototypes is `main()`, which you can use as either `int main(void)` or `int main(int argc, char *argv[])`. Whenever you use the latter, the operating system passes `main()` two parameters. The first is the number of arguments that your program was invoked with, and the second is a `NULL`-terminated list of those arguments.

If you're not as old as we are, you may take command-line arguments for granted. Believe us, they're a big deal. This is from Brian Kernighan's *Why Pascal is Not My Favorite Programming Language* (Bell Labs' Computing Science Technical Report No. 100):

*There is no notion of access to command-line arguments, again probably reflecting Pascal's batch-processing origins. Local routines may allow it by adding non-standard procedures to the environment.*

We'll give you the obligatory reminder that all arguments are processed by the shell before calling the program. The shell does file name expansion (globbing) and splits the argument list on white space. If you say

```
$ rename foo* bar*
```

the shell expands the strings `foo*` and `bar*` before it starts `rename` (`rename` will never see the asterisks).

What do these arguments mean? Whatever you want: file names, flags, assignment statements, your children's names, your birthday and so on. The C Standard assigns them no built-in meaning.

Admittedly, unless you go to a lot of work, `argv[0]` contains the name the program was invoked by. The remainder of the arguments are the other strings from the command line that invoked the command. (Here "a lot of work" typically means invoking one of the `exec()` family by hand. If you want to know more about that, check out the man page for `execl()`.)

This isn't much restriction, though. In the following command,

```
$ foo './abc*' -x cat=fezmo nkids=3 riley gillian zoe 11/21/48
```

the argument `nkids=3` could be a file name, as far as the C Standard is concerned:

```
$ ls -l
total 3
-rw-rw-r--  1 jsh   rd    12 Mar 29 15:08 abc
-rw-rw-r--  1 jsh   rd    24 Mar 29 15:08 nkids=3
-rw-rw-r--  1 jsh   rd    18 Mar 29 15:08 riley
```

## Conventions

C isn't the only standards game in UNIX-town. Whenever we discuss command-line issues, we consult the POSIX standard for the Shell and Utilities, POSIX.2–IEEE 1003.2-1992.

In addition to standardizing commands and shell syntax, Dot 2 contains a variety of definitions and rules designed to make shell programs and programmers more portable. In particular, Section 2.10, "Utility Conventions," gives a detailed set of rules about command options. Following these guidelines, if we see this line in a shell script:

```
gut_morgn -s 'Vos makhstu?' -a -- -zoe
```

we can reasonably infer that `gut_morgn` is the name of a command, while the leading hyphens of `-a` and `-s` indicate they're flags (options) controlling the command's behavior.

`'Vos makhstu?'` is probably an argument to the `-s` option because of its position. But `-zoe` is not an option at all, despite the leading hyphen, because it follows the end-of-argument indicator `--`. Admittedly, we're talking about a standards document: precise, but not always penetrable.

***In addition to standardizing commands and shell syntax, Dot 2 contains a variety of definitions and rules designed to make shell programs and programmers more portable.***

Guideline 7 of Dot 2 says "option-arguments should not be optional." Fortunately, you don't have to understand it. Dot 2 also encourages conformance to its rules by providing a C-language function, `getopt()`, and a shell-level utility, `getopts`, that encapsulate the rules of correct behavior. Use these to parse arguments, and you'll end up doing the right thing.

Of course these, too, are specified in standards-speak. Here's a paragraph taken from the four-page description of `getopts`:

*If an option character not contained in the* optstring *operand is found where an option character is expected, the shell variable specified by* name *shall be set to the question-mark (*?*) character. In this case, if the first character in* optstring *is a colon (*:*), the shell variable* OPTARG *shall be set to the option character found, but no output shall be written to standard error; otherwise, the shell variable* OPTARG *shall be unset and a diagnostic message shall be written to standard error. This condition shall be considered to be an error detected in the way arguments were presented to the invoking application, but shall not be an error in* getopts *processing.*

And vice versa. Offer not good in Alaska, Albuquerque and southeastern counties of northern West Virginia between the hours of 12 p.m. and 12 a.m., Central Standard Time, inclusive. Void where prohibited by law.

## Picking Arguments for Fun and Profit

The Dot 2 rationale–which contains nonnormative, explanatory and historical footnotes to the normative first volume–also provides code illustrating how to use both `getopt()` and `getopts`. (*Nonnormative* is standard-ese for "This is interesting and helpful, but language lawyers can officially ignore it." *Normative* is standard-ese for "The semicolons in the remainder of this document shall be assigned and conform to all special meanings detailed to them by section 1.3.4.2.8.7.16 of this document, except as modified by ISO Standard 1769.8a-1987, q.v., if supported by the implementation.")

Unfortunately, you probably do not have a copy of Dot 2 handy. To help smooth your way, we'll give you examples of both `getopts(1)` and `getopt(3)`.

We'll make the two code examples parallel one another as closely as we can, so you can contrast them, and we'll follow these with a third parallel example in Perl.

Let's begin with the most verbose of the three, a C program that demonstrates how to use `getopt(3)`:

```
#include <stdio.h>
#include <limits.h>
#include <getopt.h>
#include <string.h>

int r;
```

```
char g[128];

die(char *s)
{
  fprintf(stderr, "%s\n", s);
  exit(1);
}

main(int argc, char *argv[]) {
  int i, name;
  char usage[128];
  char **s;

 sprintf(usage,
  "%s: [-g value] [-r] filename [...]",
  argv[0]);

 printf("The full command line is' ");
  for (s = argv; *s; s++) {
   printf(" %s", *s);
 }
 printf("'\n");

 printf("The command name is '%s'\n", argv[0]);
 printf("There are %d other arguments",
     argc - 1);
 if (argc > 1) {
   printf(" and the first of these is '%s'",
       argv[1]);
 }
 printf("\n");

 while ((name =
   getopt(argc, argv, "g:r")) != -1) {
  switch(name) {
   case 'g': strncpy(g, optarg, 128);
     break;
   case 'r': r = 1;
     break;
   case '?': die(usage);
   }
 }

 argc -= optind-1;
 memmove(argv+1, argv+optind,
    (argc+1) * sizeof(char *));
 if (argc < 2) { die(usage); }

 printf("The options are -g = '%s', -r = %d.\n",
  g, r);
 printf("The %d filename arguments are '",
  argc-1);
 for (s = argv+1; *s; s++) {
  printf(" %s", *s);
 }
 printf(" '\n");
}
```

Next, a shell script with a parallel `getopts(1)` example:

```
#!/bin/sh

die() {
 echo $* 1>&2; exit 1
}
usage="$0: [-g value] [-r] filename [...]"

echo "The full command line is ' $0 $* '."
echo "The command name is '$0'"
echo -n "There are $# other arguments"
test $# -gt 0 &&
   echo -n " and the first one is '$1'."
echo

while getopts g:r name
do
  case $name in
   g) g="$OPTARG" ;;
   r) r=1 ;;
   ?) die $usage ;;
  esac
done

shift $(($OPTIND - 1))
test $# -gt 0 || die $usage;

echo "The options are -g = '$g', -r = '$r'."
echo "The $# filename arguments are ' $* '."
```

Finally, here's the same thing in Perl:

```
#!/usr/local/bin/perl -w
use Getopt::Std;
local ($opt_r, $opt_g) = (0, "");

$usage = "$0: [-g value] [-r] filename [...]\n";

print "The full command line is
  ' $0 @ARGV '.\n";
print "The command name is '$0'\n";
print "There are ", scalar @ARGV,
  " other arguments";
print @ARGV ?
  " and the first one is '$ARGV[0]'.\n" :
  "\n";

getopts "g:r" and @ARGV or die $usage;

print "The options are -g =
   '$opt_g', -r = '$opt_r'.\n";
print "The ", scalar @ARGV, "
   filename arguments are ' @ARGV '.\n";
```

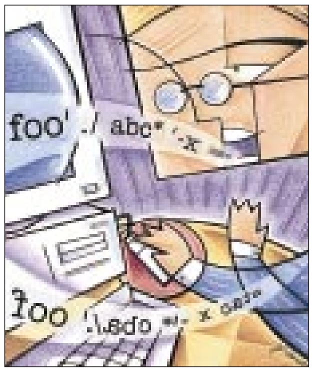
The size differences between the code are noteworthy. If we count nonblank lines, each example is about half the size of its predecessor:

```
$ for i in getopts.c getopts.sh getopts.pl
> do
> echo ==$i
> sed '/^$/d' $i | wc -l
> done

==getopts.c
 46
==getopts.sh
 22
==getopts.pl
 11
```

Even with blank lines, the entire Perl script fits comfortably inside a 24x80 terminal window.

At the other end of the spectrum, the C program fills an entire printed page, but if you're writing a C program, you won't care how much shorter the argument parsing would be if you were writing a shell script; you will care that a `getopt()` call is shorter, more robust and easier to debug than a hand-crafted `argv[]` parser.



But *why* are they different sizes? Comparing the C and shell sources is instructive. Almost every meaningful individual unit in the C program translates directly into something the shell can say in fewer words. For example, loops turn into single lines, variables and functions no longer need declarations and even case statements can be laid out more directly in the shell. Common actions have been made easy to express.

Perl is a completely different story. C's declarations and `#include` directives, which the shell script got by without, have returned. The stunning shrinkage comes from something else: much-simpler `getopts` handling. Parsing the options, storing their values, verifying that there are file name arguments and issuing a usage message for improper invocation takes 13 lines in C, 11 lines in the shell and only one in Perl.

If we stand back and remind ourselves that the bulk of these three programs are just scaffolding to report that we've parsed the arguments successfully, the author of `Getopt::Std` has clearly done something right.

## More Arguments

But what if you need to do really, *really*, ***really*** complicated argument processing?

In the shell, you're back to doing things by hand. In C, take a look at the man page for GNU's `getopt_long()`, which can do so many things that the Bugs section reads:

```
This man page is confusing.
```

Perl, too, has a comprehensive, confusing, `Getopt::Long` module, but if you don't like it, the Comprehensive Perl Archive Network (CPAN) at `http://www.perl.com/` lists seven other packages, which offer even more styles of argument handling. However, even if you choose to use `Getopt::Std`, get the latest version from the CPAN. Versions before 5.004 only let you save the value of the `-d` option into the variable `$opt_d`. Recent versions let you save arguments into the hash (associative array) of your choice. Instead of saying this:

```
getopts('abcdefg') or die $usage;
print "\$opt_a is $opt_a\n" if defined $opt_a;
print "\$opt_b is $opt_b\n" if defined $opt_b;
...
```

you can now say this:

```
getopts('abcdefg', \%opts) or die $usage;
foreach (sort keys %opts) {
  print "\opt{'$_'} is $opts{$_}\n"
}
```

Does it matter what hash we choose? There is one interesting choice: the hash `%ENV`, which contains the environment. By using `%ENV`, you can have your code set options from the command line, *or* take in option values from the environment. Observe:

```
# Set the default values
# override environment value
$ENV{'r'} = 0;
# set to "hello" only if $g
#  isn't already set in the environment
$ENV{'g'} ||= "hello";

# optionally override defaults
#   with command-line arguments
getopts($args, \%ENV)
or die "usage: $0 [-r] [-g string]
```

Of course, this takes us to the beginning of our column. We'll let you reread it while we go off and argue until next month.

Until then, happy trails.

***Jeffrey Copeland*** (`copeland@alumni.caltech.edu`) *lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats and being a thorn in the side of his local school board.*

***Jeffrey S. Haemer*** (`jsh@usenix.org`) *works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

*Note: The software from this and past Work columns is available at* `http://alumni.caltech.edu/~copeland/work.html`.