Unix utilities, Part 3

The easiest component you'll ever write

Peter Seebach (unixcomponents@seebs.plethora.net)
Freelance writer
June 2001

Unix tools are much easier to develop than you might expect. In this article we'll explore how to build some simple tools.

Contents:

Related content:

Unix utilities, Part 1
Unix utilities, Part 2

Building a tool is, once again, surprisingly easy. Your interface is just reading lines of input and writing lines of output. Network communications are handled by wrapper tools. It's often useful to accept file names on the command line (or even on standard input), but it's not necessary; there are tools to do that for you.

So, in the end: read your input, process it, and write your output. If possible, write the output as soon as you can; don't wait until all the input is generated unless it's logically necessary. (`sort` needs to see all its input; `grep` and `uniq` don't.)

First decide what you want to do. Then just read your input and write your output.

Partial results are better than final results

A lot of users are initially shocked to discover that a PKZIP-style utility is not standard on Unix systems. There are compression programs, and there are archivers, but there are very few programs that compress *and* archive.

Why?

Because when the GNU project released a compression program, which was substantially better than the *compress* utility everyone used to use, the archiver didn't need to change. The compression utility writer doesn't need to think about archiving, or how to handle different file systems. The archive utility writer doesn't need to think about compression algorithms.

Now I'm often using yet another compression utility, called *bzip2*; it gives even better compression than *gzip*. I've changed compression algorithms twice; I'm still using the same archive programs. Unix has allowed me to upgrade an "application" (archive and compress files) by upgrading one component (the compression utility). Similarly, I can change my archiver and not have to worry about the compression characteristics; it uses the same compression I use now.

If you can divide a task into two parts, and there's any way one task could be useful without the other, develop them as two tools, not as a single tool.

I can use any archival program I want (even a specialized one I write for my own purposes) with any compression program I want. This is a feature no other OS has matched.

In this article, we'll see how to apply these principles in the creation of new tools.

Read your input

A well-behaved Unix utility will generally take input in either of two ways. It can read from the "stream"

of standard input; this is the most common way for a filter (a tool that simply manipulates a stream of data) to run. Or it can read from files, which are generally named on the command line.

A Unix utility only really "has" to read from standard input. Given a list of files, you can always run *cat* on them to turn them into a stream, suitable for connecting to the standard input of any tool you want.

There's not much else to this. In most languages, reading a line of input is totally trivial. In some component architectures, you might need a page of code to handle all the input "methods" you'd need.

Of course, there's a catch. In C, which is one of the more common Unix development languages, it's actually sort of complicated to read lines of arbitrary length. We'll look at a concrete example of this later in this article.

Writing output
A utility will generally write output as soon as it's ready, so the next program in the pipeline can start working as soon as possible: distributing workloads is everything.

Writing output is actually easier than reading input; you don't even have the allocation problem to worry about.

Examples
A couple of simple example tools should give you a feel for how simple an easy Unix tool can be. Then a couple of specialized examples will show how you can approach a specific problem.

Note that in many current Unix implementations, such as Linux or NetBSD, the system comes with complete source; even on other systems, you can use the source to these utilities (most of which are quite portable among Unix-like systems) to learn more about tool development. If there's a utility that *almost* does what you want, the chances are you can get the source and change it.

Example: sl
This is a quiet testimony to the awesome power of Perl. This script sorts lines by length.

```
#!/usr/bin/perl5
print sort { length $a <=> length $b } <>;
```

That's all you need to write a simple Unix tool. Input and output are so standardized that the logical assumptions can be made by default, allowing simple programs to be very expressive.

Example: unsort
Once upon a time, I was testing a sorting program. Of course, I needed large quantities of unsorted input, so I wrote a program to unsort the input for the tests.

The program is a little larger than I initially expected it to be, because I wanted it to be robust in the face of arbitrarily large input files. However, now that the code is written, I can always reuse it. The structure describes exactly what Unix utilities do: they read input, they modify it in some way, they write output. My utility is just like the other ones, and I can plug it into anything. (For instance, my editor can now unsort a block of text.)

unsort is a particularly simple utility. It doesn't need to handle any options -- randomizing the order of lines doesn't depend on a "key" in any way. The *unsort* source is attached (see Resources for a link).

This is a complete utility. It can process files or just an input stream, and it handles fairly large quantities of data quite well. The only real flaw is that the standard C random number generator is often atrocious, especially on Unix platforms. (Of course, you can easily substitute another.)

Example: shred
As I mentioned before, it's occasionally nice to be able to take a utility that operates in one way, and run it in another. For instance, most of the standard Unix filters don't run "in place" -- they take input and produce output, so when you run them on a file, they write the (modified) contents of the file on standard output.

The shell script *shred* takes a filter as an argument, and runs it, in place, on a number of files. It can take

files as a command-line argument, it can take a list of files on standard input, or it can generate a list of files matching a given criterion, such as "files with a .c suffix."

Note that this program doesn't use quite the same interface as most other Unix tools. This is because it's one of the tools that changes the model. Just as *rsh* needs to know about network connections so that other utilities don't need to, *shred* does a fair amount of work trying to locate the files you want to operate on, and trying to figure out how to manipulate them safely.

shred is a particularly interesting example of a tool, because it *can't do anything*. If you don't give it a command, all it can do is print a usage message and exit. And yet, once you do give it a command, it can do all sorts of things. The reason the Unix tool interface is still simple after all these years is that you can always *add* an interface to all the existing tools.

Pretty printing records
Imagine that we have some records in a format that looks something like this:

```
Account: NO_ACCT                          Name: foo
Sales Rep: n/a                            District: blah
City: Unknown                             Start Date: Unknown

Account: 2923912                          Name: Smith
Sales Rep: Anyone But Bob                 District: Unknown
City: Unknown                             Start Date: 4/1/01
```

How would we make these work with Unix utilities? We'd convert them! The utilities *prettyread* and *prettyprint* work with this kind of data. *prettyread* assumes that the second column starts exactly at 40 characters. What if the data is indented using tabs, instead of spaces? Luckily, there's a separate tool for that, *expand*. It replaces tabs with the corresponding number of spaces.

Thus, if we passed the above to our *prettyread* utility, we'd end up with:

```
NO_ACCT:foo:n/a:blah:Unknown:Unknown
2923912:Smith:Anyone But Bob:Unknown:Unknown:4/1/01
```

If we pass it to *prettyprint*, we get the original data back out.

This lets us use all the standard utilities on our data, without having to worry about accidentally affecting the labels instead of the data, or anything like that. Rather than writing a program to sort records in this new format, or modifying the standard sort to handle these weird records, we simply translate the records into a usable format, work on them that way, and then translate them back. Obviously, *prettyprint* will need a bit of work if we want to generalize it, but it won't need much work -- because it only has to do one thing.

Furthermore, this gives us an additional feature. The handling of defaults and the independence from the order of input records means that *prettyread* will do okay if you hand it only partial records, or records where the data is out of order. For instance, given the following input:

```
City: Saint Paul                          Name: Jane Doe
District: Midwest
Start Date: 2/1/00
```

We get the following output:

```
Account: NO_ACCT                          Name: Jane Doe
Sales Rep: Unassigned                     District: Midwest
City: Saint Paul                          Start Date: 2/1/00
```

When building your own tools, always remember: Do one thing, do it well, and if something else needs to be done, do it separately.

In the fourth and final installment in this series, I'll cover some Unix lessons for component architectures.

Resources

- This article is the third in a four-part series. The other parts are:
  - Unix utilities, Part 1: Yet another new component architecture
  - Unix utilities, Part 2: Simple tools, complex problems
- *prettyprint*, *prettyread*, and the associated `pretty.pl`, are available at http://www.plethora.net/~seebs/comp/. *unsort.c* is also available for download so that you may examine the source.
- *The Mythical Man Month*, Frederick P. Brooks, Addison-Wesley 1995.
- *A Quarter Century of Unix*, Peter H. Salus, Addison-Wesley 1994.
- In *Let's Make Unix Not Suck*, Miguel De Icaza does not agree that Unix utilities are components.
- Chris Browne on The Unix Philosophy.
- Thomas Scoville's well-known *UNIX as Literature* essay.
- IBM Tools and Toys for Unix page, where you can also download and study the code for these IBM open-source Unix utilities.
- An collection of links to excellent Unix resources from the IBM eServer group
- Application development tools for AIX and Unix. For all of those who think Unix is "old," we just had to link to this page, which states "Behind every cutting-edge software development, there's a Unix developer." How true that is ...

About the author

As a small child, Peter Seebach thought that "rm" was a perfectly intuitive name for the command to "ReMove" a file. He has been a Unix bigot ever since. You can reach him at unixcomponents@seebs.plethora.net.

e-mail it!

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback

About IBM | Privacy | Legal | Contact