UnixWorld Online: "Wizard's Grabbag": Column No. 001

Perl By Example

This month we feature several practical Perl programs contributed by Patrick Ryan.

Dear Editor:

I'm contributing several Perl scripts written for our system. I have abandoned the Bourne shell, awk, and sed, converting completely to Perl for all script programming. At its little heart, Perl is a fantastic tool for manipulating text. As a bonus, it has access to most of the C library and system calls. Perl is about as close as you can get to interpreted C.

I wrote most of these scripts as exercises for myself when I was learning Perl. As such, they are not the best designed programs I've ever written. Each one, however, demonstrates some useful functionality of Perl in a reasonably concise manner.

Note that the `pnews` network program makes use of the Perl 5 "use" keyword. This code is safely nestled inside some version test logic and an eval statement. These programs should work under either Perl4 or Perl5.

Some time ago, I wrote the ''Introduction to Perl'' document, which was kindly converted to HTML and made available on the Web by Neil Bowers.

Editor's Note: Here are brief descriptions of each Perl tool. Even if you aren't interested in the tool itself, the programs provide examples of well-crafted Perl.

edcron: Edit a user's `cron`-configuration file.

> The `cron` daemon executes specified commands at specified dates and times. The crontab command is supplied so a user can list or install a configuration file used by the `cron` daemon. Newer `crontab` versions recognize the `-e` option to invoke a visual editor, generally `vi`, to let the user update the configuration file. However, pre-SVR4 versions of `crontab` did not recognize this option so users had to perform several error-prone steps to update the configuration file.
>
> The `edcron` Perl program lets the user update their `cron`-configuration file on any system with the `crontab` command, whether or not it recognizes the `-e` option. Briefly, here's how it works: first `edcron` runs `crontab -l` to list the configuration file, storing the output in a temporary file. An editor is invoked to update the contents of the temporary file. The `vi` editor is employed unless the `VISUAL` or `EDITOR` environment variables specify an alternate editor. Finally, `crontab` is invoked to install the edited temporary file as the `cron`-configuration file for the user who invoked `edcron`.

kp: Kill selected processes.

> Occasionally, it is necessary to kill all the processes owned by a certain user or all processes identified by a common pattern. My "process killer" (`kp`) program traverses the output from `ps`. As it goes along, `kp` looks for certain patterns in the output and kills processes that it recognizes. The

invocation syntax is:
`kp [-n] [-`*signal-number*`] `*pattern1* `[`*pattern2*`...]`

Here, the arguments are:

`-n`
: The "not really" mode, that is, don't actually kill processes, just report processes that would have been killed.

-signal-number
: Here, *signal-number* is a integer representing a signal. The default is 1 (SIGHUP). To send a SIGKILL, for instance, you would specify `kp -9`.

pattern
: After the command-line options, one or more regular expressions can be specified to identify the desired processes. Remember to quote any expressions that contain characters special to the shell, such as `()*|$!^&`

Here are some examples:

Kill all processes owned by user `bob`:

```
# kp bob
```

Send a SIGKILL to all daemon processes:

```
# kp -9 '*d$'
```

View, but don't really kill ryan's processes:

```
# kp -n ryan
```

*Caveat:*Because the `ps` output format often varies from system to system--that is, the number of output columns and their ordering is not standardized--you may need to "tweak" `kp` for your system.

pgrep: A Perl "grep" program.

Perl has some sweet extensions to the regular expression syntax used by `grep`. My `pgrep` program is a quick-and-dirty implementation with Perl of pattern matching that gives you access to Perl's complete regular expression capabilities. I find Perl's word boundary matching feature (`\b`) especially useful.

The syntax is simply:

`pgrep` *pattern* `[`*file1* `[`*file2*`...]]`

Some usage examples:

Display all lines in `gh.c` containing "printf" pattern:

```
% pgrep printf gh.c
gh.c:       fprintf(stderr,
gh.c:                   fprintf(stderr,"bad address: %s\n",h);
gh.c:       printf("%s %s ", ip, host->h_name);
gh.c:            printf("%s ", *s);
gh.c:       printf("\n");
gh.c:       fprintf(stderr,"%s:",msg);
```

```
gh.c:       fprintf(stderr,"HOST_NOT_FOUND\n");
% []
```

Only display calls to "printf," not "fprintf":

```
% pgrep '/\bprintf\b/' gh.c
gh.c:       printf("%s %s ", ip, host->h_name);
gh.c:            printf("%s ", *s);
gh.c:       printf("\n");
```

Perform a case-insensitive search on the same file (analogous to "`grep -i`"):

```
% pgrep /host/i gh.c
gh.c:  *   get host info
gh.c:       struct hostent
gh.c:      *host;
gh.c:       for (s = host->h_aliases; *s; ++s)
gh.c:        case HOST_NOT_FOUND :
gh.c:       fprintf(stderr,"HOST_NOT_FOUND\n");
```

Note: The surrounding slashes in the regular expression argument are optional.

pnews: Simple news posting client that uses the NNTP protocol.

I wrote this utility as an exercise in Perl networking.

The invocation arguments are:

`-h` *host*
: Specify NNTP posting host. Otherwise, the default host specified by the `$NNTPSERVER` environment variable is used.

`-s` *subject*
: Set message subject. If not specified, the program will prompt for a subject.

`-d`
: Enable debugging output

`-t` *command*
: For testing purposes, send an NNTP command to your server, and display the result.

A few examples:

To post a message:

```
% pnews
```

The program will prompt you for newsgroups and a subject, invoke your favorite editor, and post the message.

To test your server:

```
% pnews -t help
```

Send a "help" message to your NNTP daemon and print the result.

tarls: List contents of an archive created by the UNIX `tar` command.

I developed `tarls` to demonstrate the use of the Perl `unpack` function. The `tar`-archive format is

one of the few widely accepted file formats in the UNIX system world. The record structure can usually be found in `/usr/include/tar.h`. The `tarls` program simply lists the contents of a tar file.

The command-line syntax is:

`tarls` *tar-format-file*

Here's an example displaying the contents of the `/tmp/cfiles.tar` file:

```
% tarls /tmp/cfiles.tar
    ryan/ims         786 Mon Aug 29 15:26:19 EDT 1994 touchport.c
    ryan/ims        4982 Thu Dec 15 14:28:10 EST 1994 utmpdmp.c
    ryan/ims         989 Thu Dec  8 15:37:57 EST 1994 daemon_start.c
% []
```

which: Perl version of the UNIX system `which` program.

For some reason unknown to me, many UNIX systems vendors do not include the helpful `which` program with their operating systems. The UNIX `which` command looks for a specified program along the command-search path specified in the C shell's `.cshrc` file, and displays its path name. (It also expands command alias definitions). When I came across a UNIX-based platform that didn't have this useful utility--UNICOS, to be exact--I cobbled up this version with Perl.

Unlike the distributed version, my version searches the list of directories in the `PATH` environment variable. In particular, the value of the Perl `$ENV{PATH}` is split into fields and each resulting directory examined for the specified command-line argument. Like the distributed version, my version also displays the alias expansion for the specified argument.

Some usage examples:

```
% which cc
/usr/ucb/cc
/opt/SUNWspro/SC2.0.1/cc
% []

% which pd
pd aliased to 'pushd'
% []
```

Copyright © 1995 The McGraw-Hill Companies, Inc. All Rights Reserved.
Edited by Becca Thomas / Online Editor / UnixWorld Online / beccat@wcmh.com