The journal of Apple technology.

Volume Number: 18 (2002)
Issue Number: 8
Column Tag: Section 7

A Brief Look at Perl

a real gem of a scripting language

by Rich Morin

Last month's column closed with some short descriptions of BSD's scripting languages. It offered some opinions, but stopped short of recommending any particular language. This month, I'll get a bit braver, explaining why you might want to use Perl for most of your BSDish scripting needs.

First, however, I should caution that Perl is not always the appropriate choice. If you're modifying a system shell script, don't try to rewrite it in Perl. Just use the language the script is written in (typically the Bourne Shell). If you are just mechanizing a simple list of commands, Perl is probably overkill, but see below. Finally, if the script has to run early in the startup process, the Perl interpreter may not be available.

For new, substantial scripts, however, I would strongly recommend that you use Perl. Here are some reasons:

- efficiency - The Perl interpreter, unlike the shells, seldom has to start up new processes. This means that substantial Perl scripts will often run much faster than equivalent shell scripts. I found this out several years ago, when I transliterated several large shell scripts to Perl. The run times went down by a factor of five!
- syntax - Most shells have very weak notions of syntax. One result of this, in Mac OS X, is that white space in file names can be interpreted as splitting the names into multiple tokens. Perl handles strings in a much more sophisticated manner, so it doesn't get confused.
- integration - Unlike shell scripts, which may stitch together dozens of commands, Perl is an integrated language. This eliminates a great deal of hassle and possible confusion.
- facilities - Perl has powerful data structures, convenient control-flow operators, and access to almost any imaginable system call. The shells have none of these features. As a result, a Perl program can often do things that would be essentially impossible in any shell.
- support - Perl has numerous books, a vast library of modules, and a very active user community. Most shells have few to none of these resoueces.
- portability - Perl scripts can be run on essentially any modern operating system. With a little forethought, they can run unmodified on several different systems. The shells, in contrast, only work on BSD and other Unix-like systems.

Having said all of this, perhaps I should tell you some of the bad news about Perl:

- complexity - Perl is a very large language, with some really peculiar nooks and crannies. Even if you don't use all of these features, you may well encounter them in a module or some other bit of code you "inherit".

- informality - Perl's motto ("There's More Than One Way To Do It" gives fair warning that this isn't a nice tidy "orthogonal" language. In fact, Larry Wall (Perl's creator) says that Perl is a "diagonal" language; cutting across the middle often speed things up!
- mutability - Unlike the shells, Perl is still evolving. Perl 5 has (mostly) stabilized, but Perl 6 development is quite active. So, you might need to relearn some things in a few years.

Show me some code!

This being MacTech, you're probably wondering when you're going to see some actual Perl code. Well, here's a short Perl script that I hacked together to do some backups. It's not a full-featured backup utility, by any means, but it gets the job done (and shows off some Perl language features)...

```
#!/usr/bin/env perl
#
# macbac - Create backup files, using tar(1).
#
# Written by Rich Morin, CFCL, 2002.06
{
  $date = `date +%y%m%d.%H%M`;
  chomp($date);
  @dir = ('/Users/rdm',
          '/Volumes/Work');
  for $dir (@dir) {
    $bac = cvt($dir);
    $cmd = "nice -10 tar czf $bac $dir";
    printf(">>> %s\n", $cmd);
    system($cmd);
  }
}
sub cvt {  # convert the directory name
  my ($tmp) = @_;

  $tmp =~ s|/|.|g;
  $tmp =~ s|\s|_|g;
  return("/Backups/$date$tmp.tgz");
}
```

The first line of any BSD script, as discussed previously, tells the system which program should be invoked as the interpreter for the following lines. Because I may decide to install a later version of the Perl interpreter at some point (e.g., in /usr/local/bin), I don't want to specify a full path name for the Perl interpreter. So, I tell the system to run /usr/bin/env, letting it find and run the appropriate version of Perl.

The remainder of the script, in any case, is read by the Perl interpreter. Perl's syntax and feature set are borrowed from a variety of (mostly Unix) languages and tools, including awk, Basic-Plus, C, sed, sh, and tr. This makes Perl seem familiar to Unix aficionados, but can cause some culture shock to others. Stay calm; it's not really all that bad!

Unlike C, Perl has no "block comments". So, I use a column of sharp signs (#) for my header comments. I also like to wrap the "main" routine in braces. This causes its contents to be indented at the same level as the contents of any sub (routine). It also gives me a visual cue that this is a "block" of code.

I could have asked Perl to grab and format the date information (and should have, if I were trying for cross-OS portability), but the method above shows off Perl's ability to run BSD commands and retrieve their results. The backquotes tell Perl to run the enclosed command, returning the result as a text string. The result gets put into a scalar variable, $date. The chomp() function, by the way, removes the trailing newline from date's output.

The script then tells Perl to create an array variable named @dir and load it with a list containing two text strings. I use single quotes to wrap these strings, indicating that I don't want Perl to do any variable interpolation (see below) or other tricks.

The for loop sets $dir, successively, to each of the values in @dir. Note that the sigil (e.g., $, @) is part of a Perl variable's name, so @dir and $dir are two different variables. This seems a bit weird at first, but ends up being quite handy as you get used to it.

The cvt() routine shows off some of Perl's capabilities and peculiarities. First, it grabs @_ (the array of calling parameters) and copies the contents into a private list of variables. In this case, the list only has one element, but it might well have more.

The next two lines tell Perl to do global substitutions of periods for slashes and underscores for "white space". This gives me "flattened" names (no directory levels) without any annoying spaces, tabs, etc. This is a trivial example of Perl's powerful "regular expression" capability. Regular expressions can be used to perform all sorts of magic on text strings. In fact, there is a substantial book on regular expressions alone!

The last line tells Perl to "interpolate" the variables $bac and $date into a text string. The use of double quotes tells Perl to look for dollar signs and other "magic" characters. Note that, although $tmp is a private variable, $date is shared with the main routine. Finally, the return is not strictly needed (the value of the last expression evaluated in a sub is automatically returned), but I think it adds to the clarity of the code.

Returning to the main routine, we build up a command string, print it (ala C), and hand it off to the operating system to be run. Look up the man pages for nice and tar to see what their roles are in this script.

In a script of this size, there isn't much room to get into Perl's fancier aspects. Next month, I'll give you a more substantial taste of its data structures and control flow, as well as listing some useful Perl resources. If you can't wait to get started, however, just bop over to www.perl.{org,com}...

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn't writing this column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.