Randal Schwartz's Perls of Wisdom

RANDAL L. SCHWARTZ

Apress®

Randal Schwartz's Perls of Wisdom

ISBN (pbk): 1-59059-323-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Mike Schilli
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editors: Ami Knox, Nicole LeClerc
Production Manager: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Indexer: Kevin Broccoli
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for the complete listings contained in this book is available to readers at http://www.apress.com in the Downloads section.

Taint So Easy, Is It?

Unix Review, Column 33 (August 2000)

Randal's Note The `Scalar::Util` module (in the core in recent versions of Perl) lets us test whether a value is tainted, but that module hadn't been written when I wrote this article. Otherwise, this is all good sound advice on how to use one of the unique Perl features to increase relative security.

If you've been reading my columns for any length of time, you've probably seen me mention "taint mode," usually briefly while I'm describing a "hash-bang" line of something like

```
#!/usr/bin/perl -Tw
```

which turns on warnings (the `-w`) and taint mode (the `-T`). But what *is* taint mode?

Taint mode is a security feature of Perl, and includes two levels of operation. First, while taint mode is in effect, some operations are forbidden. One of these is that `$ENV{PATH}` cannot contain any world-writable directories when firing off a child process (like with backticks or `system`). Should your program attempt an unsafe action, the program aborts (via `die`) immediately, before the action has a chance to create a potential security violation. You could have included code to check this yourself, but having Perl perform the checks ensures a consistency and a "best practices" level of competence that you may not have the capability or resources to include explicitly.

The second level of operation is much more interesting and unique to Perl (amongst all the popular languages I know of), in which Perl keeps track of a "distrust" of each scalar value in the program. Every item of data coming from input sources (command-line arguments, environment variables, locale information, some system calls, and all file input) is marked "tainted."

For example, the following operations all generate tainted data:

```
$t1 = <STDIN>;
$t2 = $ENV{USER};
$t3 = $ARGV[2];
@t4 = <*.txt>;
```

In each of these examples, the data has come "from the outside world," and is therefore treated as potentially dangerous. Once data is tainted, the taint propagates to any data *derived* from the tainted data:

```
$t5 = $t4[0];
$t6 = "/home/$t2";
chomp($t1);
@x = ("help", "me", $t3, "please");
```

Note that tainting is on a per-scalar basis. So `$x[2]` is tainted, not the entire array `@x`.

Once data is marked tainted, nearly any attempt to use the data to affect the outside world will be blocked, causing an immediate `die` with a taint violation. For example, invoking `rename` where either the source name or destination name is tainted is considered dangerous. This permits normal operations:

```
rename $x[0], $x[1];
```

But not operations that involve tainted data (recall that `$x[2]` is tainted from earlier):

```
rename $x[0], $x[2];
```

What this means is that data that comes in from the outside world cannot trivially affect the outside world as well. Why is this important?

Well, the typical use of taint mode is to enable programs that act on behalf of other users to operate in a safer manner. For example, a `setuid` or `setgid` program borrows the privileges of its owner for the duration of execution, allowing an ordinary user to act as `root` (or some other user) for a selected set of operations. Or a CGI program, executing as the web server ID (typically `nobody`), is acting with that user's privileges on behalf of a request from any web client, generally without direct access to the server except through the web server.

In both of these cases, it's important that input data be checked so as not to permit the user who invokes the program from borrowing the privileges of the executing user ID to perform unintended actions.

For example, it'd be pretty dangerous to rename a file based on the input from a CGI form:

```
use CGI qw(param);
...
my $source = param('source');
my $dest = param('destination');
rename $source, $dest;
```

Now perhaps the author of this CGI script believed that since the form contained only radio buttons or pop-up menus that were clearly defined this would be a safe program. But in reality, a person with intent to damage or break in could just as easily invoke this script passing arbitrary data in `source` and `destination`, and potentially rename any file to which the web userid has access!

With taint mode enabled, the CGI parameters (having been derived from either reading `STDIN` or an environment variable) are marked tainted, and therefore the `rename` operation would fail before it has committed potential damage. (To enable taint mode on a CGI script, just include `-T` in the `#!` line, as shown earlier.) And that's exactly the safest thing to do here.

But obviously, there are times when input data must in fact legitimately affect the outside world. Here's where the next feature of taint mode comes in. As a sole exception, the results of a regular expression memory reference (usually accessed as the numeric variables like `$1` and `$2` and so on) are never tainted, even though the match may have been performed on tainted data. This gives us the "carefully guarded gate in the fence," when used properly. For example:

```
my $source = param('source');
unless ($source =~ /^(gilligan|skipper|professor)$/) {
  die "unexpected source $source\n";
}
$safe_source = $1;
```

Here, $source is expected to be one of gilligan, skipper, or professor. If not, we'll die before executing the next statement, which copies the captured memory into $safe_source. (Note the parens in the regular expression match are performing double duty, needed for both proper precedence regarding the vertical bar and the beginning and ending of string anchors, as well as having the side effect of setting up the first back-reference memory. Sometimes, you get lucky.)

The value of $safe_source is now legitimate to be used in the rename operation earlier, as it came from a regular expression memory, and not directly from input data. In fact, we could even have assigned it back over $source (a common thing to do):

```
$source = $1; # source now untainted
```

Of course, we'd have to perform a similar operation on $destination to complete the operation.

So, if someone attempts to give us an incorrect value for the source parameter, like ginger, the program aborts. Certainly, this program would have aborted with or without taint mode, but in taint mode it works only because we *added* the extra code to perform a regular expression match, during which we needed to think about what the possible legal values for the string might have been.

And that brings up the next point: we typically can't perform an explicit match against a known list of values. More often, the data is a user-specified value that needs to fit a general description, but again, regular expressions are pretty good at matching many things.

So, let's say the $source there came from a text field box, rather than a pop-up menu, permitting an arbitrary string. How do we pass that along to the rename operator? Well, first we have to decide what a legitimate string might be. For example, let's restrict to filenames that contain only \w-matching characters, including a dot (as long as the dot is not the first character). That'd be like this:

```
$source = param('source');
$source =~ /^(\w[\w.]*)$/ or die;
$source = $1;
```

Once again, if the string is not as expected, we die. And only if we haven't died will we continue on to use $1, which has now been verified to be a name of the form that we expect.

Note that it's *very* important to test the result of the regular expression match, because $1 (and the other memory variables) is set only when you have a successful regular expression match. Otherwise, you get an earlier match, and that's definitely bad news:

```
## bad code do not use ##
$param('source') =~ /^(\w[\w.]*)$/;
$source = $1;
## bad code do not use ##
```

A slightly more compact way of writing this correctly might be

```
my ($source) = param('source') =~ /^(\w[\w.]*)$/
  or die "bad source";
```

Here, I'm using $1 implicitly as the list context result of the regular expression match, and declaring the variable that will hold it, and checking for errors, all in one compact statement.

The regular expression pattern should be as restrictive as you can get. For example, if you use something like `/(.*)/s`, you've effectively removed any of the benefits of taint mode for that particular data, making it potentially possible for someone to hijack your program in unintended ways.

So, I hope this gives you a bit of insight into how to use taint mode, and why it is useful. If this column "taint" enough for you, I suggest you check out the `perlsec` manpage (perhaps using the command `perldoc perlsec` at a prompt). Until next time, enjoy your new security knowledge.

Discovering Incomprehensible Documentation

Linux Magazine, Column 18 (November 2000)

■ **Randal's Note** The Perl Power Tools project was abandoned, and restarted as an updated project by Casey West (at `http://ppt.caseywest.com`), so the URLs listed in the article are now wrong. Also, had I written this today, I'd probably again use my `File::Finder` instead of the awkward `File::Find` there.

Ahh, manpages. Some of them are great. But a few of them are just, well, incomprehensible.

So I was sitting back a few days ago, wondering if there was a way to locate the ugly ones for some sort of award, and then I remembered that I had seen a neat module called `Lingua::EN::Fathom` that could compute various statistics about a chunk of text or a file, including the relative readability indices, such as the "Fog" index. The "Fog" index is interesting in particular because it was originally calibrated to be an indication of the "grade level," with 1.0 being "first grade text" and 12.0 as "high school senior." At least, that's the way I remember it.

While I don't believe in the irrational religion applied to these indices sometimes ("We shall have no documentation with a Fog factor of higher than 10.0"), I do think they are an indicator that something is amiss.

So, in an hour or so, I hacked out a program that wanders through all of the manpages in my `MANPATH`, extracts the text information (discarding the troff markup), and sorts them by Fog index for my amusement. Since I brought together a lot of different technologies and CPAN modules to do it, I thought I'd share this program with you, found in Listing 1-1, later.

Line 1 gives the path to Perl, along with turning on warnings. I usually don't trigger any warnings, but it's nice to run with the safetys on occasionally.

Line 2 enables the normal compiler restrictions, requiring me to declare all variables, quote all quoted strings (rather than using barewords), and prevents me from using those hard-to-debug symbolic references.

Line 3 ensures that each `print` operation on `STDOUT` results in an immediate I/O operation. Normally, we'd like `STDOUT` to be buffered to minimize the number of system calls, but since this program produces a trace of what's happening, I would kinda like to know that it's happening *while* it is happening, not after we got 8000 bytes to throw from a buffer. As an aside, some people would prefer that I use `$| = 1;` here because it would be clearer. But I find the `$|++` form easier to type, and I saw Larry do it once, so it must be blessed.

Line 6 provides the only configuration variable for this program: the location of the memory to be used between invocations. Running the text analysis on the data each time is expensive (especially while I was testing the report generator at the bottom of the program), so I'm keeping a file in my home directory to hold the results. The filename will have an extension appended to it, depending on the chosen DBM library.

Line 9 is what got me started on this program: a module from the CPAN to compute readability scores. As this is not part of the standard distribution, you'll need to install this yourself.

Line 10 provides the two constants I needed for the later DBM association.

Line 11 pulls in the "multilevel database" adaptor. MLDBM wraps the fetch and store routines for a DBM-tied hash so that any reference (meaning a data structure) is first "serialized." The result is that a complex data structure is turned into a simple byte string during storage, and when retrieved, the reverse occurs, so that we get a similar data structure again. There are interesting limitations, but none of them got in my way for this program.

The args to the `use` indicate that we want to use `DB_File` as our DBM, and `Storable` as our serializer. `DB_File` is found in the Perl distribution, but you must have installed "Berkeley DB" before building Perl for this to be useful. Replace that with `SDBM` if you can't find `DB_File`. `Storable` is also found in the CPAN, and is my preferred serializer for its robustness and speed. `Data::Dumper` can also be used here, with the advantage that it's the default.

Line 12 selects the ever-popular `File::Find` module (included in the distribution) to recurse downward through the `man` directories to scout out the manpage files.

Line 13 enables simple trapping of signals with a trivial `die` operation. I found that without this, if I pressed Control-C too early in the process, none of my database would be updated, which makes sense after I thought about it. (An interrupt stops everything, not even giving Perl a chance to write the data to the file by closing the database cleanly.)

Line 15 associates `%DB` with the multilevel database named in `$DATAFILE`. The remaining parameters are passed to the underlying `DB_File` tie operation, and select the creation as needed of the database, and the permissions to give to the file if needed.

Line 17 sets up a global `@manpages` variable to hold the manpages found by the subroutine in lines 20 through 23.

Lines 19 through 24 walk through the directories named in my `MANPATH`, looking for manpages. First, the `MANPATH` is split on colons, then each element is suffixed with slash-period-slash. As far as `File::Find` is concerned, this doesn't change the starting directories, but the presence of this marker is needed later to distinguish the prefix directory from the location within that directory, as we'll see in line 29.

The anonymous subroutine starting in line 19 is called repeatedly by `File::Find`'s `find` routine. The full name of each file can be found in `$File::Find::name`, while `$_` is set up properly together with the current directory to perform file stat tests. The conditions I'm using here declare that we're looking for a plain file (not symbolic link) that isn't named `whatis`, and that it not be too big or too small. If it's a go, the name gets stuffed at the end of `@manpages`.

Line 26 creates the text analyzer object. I humored myself at the time by calling it `$fat`, which originally was a shortened form of "fathom." As I write this text the next day, I can't remember why I found that funny. I guess it's meta-funny.

And now for the first big loop, in lines 28 to 48. This is where we've got the list of manpages, and it's time to go see just how awful they are.

Line 29 pulls apart the `$dir`, which is the original element of my `MANPATH`, from the `$file`, which is the path below that directory. This is possible because we included the slash-dot-slash marker in the middle of the path during the filename searching, and necessary because

the troff commands of the manpages presume that the current directory is at the top of the manpage tree during processing, particularly for the `.so` command, which can bring in another manpage like an include file.

Line 30 refixes the name to avoid the marker, and line 31 shows us our progress with that updated name.

Lines 32 through 36 keep us from rescanning the same file. First, the modification timestamp is grabbed into `$mtime`. Next, we check the existing database entry (if any) to see if the recorded modification time from a previous run is the same as the modification time we've just seen. If they're the same, we've already done this file on some prior run, and we can skip this one altogether. If not, we gotta get our hands dirty on it instead.

Line 38 is where this program spends most of its time. We have a `deroff` command that reads a troff file, and removes most of the troff embedded control sequences and commands. While it's not perfect, it's fairly useful, and close enough for this demonstration. And we need to be in that parent directory so that relative filenames work; that's handled with the simple one-liner shell command inside the backquotes.

"But wait," you may ask, "I don't have a `deroff`!" Never fear. I ran into the same problem myself. A quick search on the net (thank you, `www.google.com`!) revealed that this had been one of the already completed commands in the *Perl Power Tools* project, archived at `http://language.perl.com/ppt/`. So, I downloaded the `.tar.gz` file from that page, extracted the pure Perl implementation of `deroff`, and installed it quite nicely. Yes, there's a few open-source C versions out there, but I didn't want to futz around.

Line 39 detects a failure in the attempt to deroff the text, and moves along if something broke. Nothing went wrong in the hundreds of files I analyzed, but ya never know.

Line 40 is where this program does some heavy CPU on its own. The text of the deroff'ed manpage is crunched, looking for the various statistics, including our readability scores. There didn't appear to be any error return possible from this call, so I didn't try to detect one.

Line 42 creates the `%info` data structure to hold the attributes of this particular file that we want to store into the database. We'll start with the modification time that we fetched earlier, to ensure that later passes will go "Hey, I've already seen this version."

Lines 43 through 45 use the `$fat` object to access the three scores, via the `fog`, `flesch`, and `kincaid` methods. I've used a nice trick here: an "indirect method call," where the name of the method comes from a variable. The result is as if I had said

```
$info{"fog"} = $fat->fog();
$info{"flesch"} = $fat->flesch();
$info{"kincaid"} = $fat->kincaid();
```

but with a lot less typing. (That is, until just now, to illustrate the point.)

Line 46 stores the information into the database. The value is a reference to the hash, but the `MLDBM` triggers a serialization, so that the actual DBM value stored is just a byte string that can be reconverted into a similar data structure upon access. And in fact, an access already potentially occurred up in line 33. The access to `$DB{$name}` fetched a byte string from the disk database, which was then reconverted into a hashref so that the subsequent access to the hashref element with a key of `mtime` would succeed.

Line 47 lets us know we did the deed for this file, and are moving on.

And that completes the data gathering phase, so it's now time to do the report, as indicated in line 50.

Line 54 is a quick trick with interesting performance consequences. The hash of %DB acts like a normal hash, but actually involves two levels of tied data structures. This can be quite slow, especially when performing repeated accesses for sorting. So, we *copy* the entire database as an in-memory hash in one brief operation, to the %db hash. Now we can use %db in the same way we would have used %DB, but without the same access expense. Of course, since it's a copy, we can't change the real database, but that's not needed here.

Lines 55 to 57 sort the database by the key specified in $kind, defined in line 52. We've got a descending numeric sort, to put the worst offenders first. A simple printf makes the columns line up nicely.

And my output from running this program looks something like Listing 1-2,shown later. Yeah, that first file ranked in at a whopping "grade 167" education to read it. In theory. And about 5th grade or 6th grade for the simplest few. As a comparison, the text of this column (before editing) came out at around 13.3 on the Fog index. Hmm. I hope you all made it through high school! Until next time, keep your sentences short, and to the point. Enjoy!

Listing 1-1

```
=0=     ##### LISTING ONE #####
=1=     #!/usr/bin/perl -w
=2=     use strict;
=3=     $|++;
=4=
=5=     ## config
=6=     my $DATAFILE = "/home/merlyn/.manfog";
=7=     ## end config
=8=
=9=     use Lingua::EN::Fathom;
=10=    use Fcntl qw(O_CREAT O_RDWR);
=11=    use MLDBM qw(DB_File Storable);
=12=    use File::Find;
=13=    use sigtrap qw(die normal-signals);
=14=
=15=    tie my %DB, 'MLDBM', $DATAFILE, O_CREAT|O_RDWR, 0644 or die ↵
    "Cannot tie: $!";
=16=
=17=    my @manpages;
=18=
=19=    find sub {
=20=      return unless -f and not -l and $_ ne "whatis";
=21=      my $size = -s;
=22=      return if $size < 80 or $size > 16384;
=23=      push @manpages, $File::Find::name;
=24=    }, map "$_/./", split /:/, $ENV{MANPATH};
=25=
=26=    my $fat = Lingua::EN::Fathom->new;
=27=
=28=    for my $name (@manpages) {
```

```
=29=      next unless my ($dir, $file) = $name =~ m{(.*?)/\./(.*)}s;
=30=      $name = "$dir/$file";
=31=      print "$name ==> ";
=32=      my $mtime = (stat $name)[9];
=33=      if (exists $DB{$name} and $DB{$name}{mtime} == $mtime) {
=34=        print "... already computed\n";
=35=        next;
=36=      }
=37=
=38=      my $text = `cd $dir && deroff $file`;
=39=      (print "cannot deroff: exit status $?"), next if $?;
=40=      $fat->analyse_block($text);
=41=
=42=      my %info = ( mtime => $mtime );
=43=      for my $meth (qw(fog flesch kincaid)) {
=44=        $info{$meth} = $fat->$meth();
=45=      }
=46=      $DB{$name} = \%info;
=47=      print "... done\n";
=48=    }
=49=
=50=    print "final report:\n\n";
=51=
=52=    my $kind = "fog";
=53=
=54=    my %db = %DB;                    # speed up the cache
=55=    for my $page (sort { $db{$b}{$kind} <=> $db{$a}{$kind} } keys %db) {
=56=      printf "%10.3f %s\n", $db{$page}{$kind}, $page;
=57=    }
```

Listing 1-2

```
=0=     ##### LISTING 1-2 #####
=1=     final report:
=2=
=3=        167.341 /usr/lib/perl5/5.00503/man/man3/WWW::Search::Euroseek.3
=4=        154.020 /usr/lib/perl5/5.00503/man/man3/GTop.3
=5=         65.528 /usr/lib/perl5/5.00503/man/man3/Tk::X.3
=6=         56.616 /usr/man/man1/mh-chart.1
=7=         45.591 /usr/man/man1/tar.1
=8=         40.133 /usr/lib/perl5/5.00503/man/man3/Bio::SeqFeatureI.3
=9=         39.012 /usr/lib/perl5/5.00503/man/man3/XML::BMEcat.3
=10=        37.714 /usr/lib/perl5/5.00503/man/man3/less.3
=11=        37.200 /usr/lib/perl5/5.00503/man/man3/Business::UPC.3
=12=        36.809 /usr/lib/perl5/5.00503/man/man3/Number::Spell.3
=13=    [...many lines omitted...]
=14=         7.179 /usr/man/man1/tiffsplit.1
```

```
=15=        7.174 /usr/lib/perl5/5.00503/man/man3/Tie::NetAddr::IP.3
=16=        7.018 /usr/lib/perl5/5.00503/man/man3/DaCart.3
=17=        6.957 /usr/man/man3/form_driver.3x
=18=        6.899 /usr/man/man7/samba.7
=19=        6.814 /usr/lib/perl5/5.00503/man/man3/Array::Reform.3
=20=        6.740 /usr/lib/perl5/5.00503/man/man3/Net::GrpNetworks.3
=21=        6.314 /usr/man/man5/rcsfile.5
=22=        6.210 /usr/lib/perl5/5.00503/man/man3/Network::IPv4Addr.3
=23=        6.002 /usr/lib/perl5/5.00503/man/man3/Net::IPv4Addr.3
=24=        5.881 /usr/man/man8/kbdrate.8
=25=        5.130 /usr/lib/perl5/5.00503/man/man3/Net::Netmask.3
```

What Is That, Anyway?

Unix Review, Column 36 (February 2001)

Randal's Note Another article on `File::Find`, and identifying things, first with the built-in type switches, and then with `File::MMagic`, which knows enough to peer inside the contents of files to pick out things of interest.

So, you've got a directory full of mixed stuff, or maybe an entire tree of directories. Just what's behind each of those names? Are they directories, symbolic links, or just plain files? And if they're files, are they text files or binary files? And if they're binary files, are they images, executables, or some random garbage?

Perl has many built-in operators to make getting lists of names easy, and also for figuring out what you really have once you have a name.

For example, let's find all the subdirectories within the current directory:

```
for my $name (glob '*') {
  next unless -d $name;
  print "one directory is $name\n";
}
```

Here, the `glob` operator expands to all the non-dot-prefixed names within the current directory, and the `-d` operator returns true for all those names that are directories.

What if we wanted to do this recursively? We need to step outside of the core Perl, but not very far away. A core-included module called `File::Find` takes care of nearly all of our recursive directory processing problems. Let's find all directories below the current directory:

```
use File::Find;
find sub {
  return unless -d $_;
  print "one directory is $File::Find::name\n";
}, ".";
```

The find subroutine takes a subroutine reference (called a "coderef"), here provided with the anonymous subroutine constructor. Each name found below . (specified on the last line of this snippet) will trigger an invocation of this subroutine, with $File::Find::name set to the full name, and $_ set to the basename (with the working directory already selected to the directory in which the name is located).

If you run this, you'll see that each directory is typically shown two or more times! Once as a name within its parent directory, once as the name of . when we're in the directory, and perhaps one or more times for each of the subdirectories contained within the directory. So how do we eliminate that? Well, just rejecting "dot" and "dot-dot" in the subroutine will do nicely:

```
use File::Find;
find sub {
  return if $_ eq "." or $_ eq "..";
  return unless -d $_;
  print "one directory is $File::Find::name\n";
}, ".";
```

There. We'll keep moving forward from this as our base, because rejecting the meta-links of dot and dot-dot is generally a useful thing.

What about all the symbolic links? Can we find those? Sure! That's the -l operator:

```
use File::Find;
find sub {
  return if $_ eq "." or $_ eq "..";
  return unless -l $_;
  print "one symlink is $File::Find::name\n";
}, ".";
```

Cool! But where do they point? That's the readlink operator, as in

```
use File::Find;
find sub {
  return if $_ eq "." or $_ eq "..";
  return unless -l $_;
  my $dest = readlink($_);
  print "one symlink is $File::Find::name, pointing to $dest\n";
}, ".";
```

We can skip the -l test by knowing that any non-symlink will automatically return undef on the readlink, as in

```
use File::Find;
my @search = @ARGV;
@search = qw(.) unless @search;
find sub {
  return if $_ eq "." or $_ eq "..";
  return unless defined (my $dest = readlink($_));
  print "one symlink is $File::Find::name, pointing to $dest\n";
}, @search;
```

I've also made it simpler to run this on different directories by passing them on the command line.

So, what do we have left? We can notice and skip over directories and symbolic links. How about files? Files are where the real action is located. And some of them are text-like, and some of them are binary-like. Although even those lines are blurry: you could argue that XML is really just a text-like binary format, and a Microsoft Word document is clearly text inside a binary-like format.

But back to what Perl can help with, first. Let's add the `-T` operator to distinguish those text files:

```
use File::Find;
my @search = @ARGV;
@search = qw(.) unless @search;
find sub {
  return if -d $_ or -l $_;
  return unless -T $_;
  print "One text file is $File::Find::name\n";
}, @search;
```

And that's pretty cool. Just a list of text files. But this actually doesn't tell us too much. What we might really want is a list of all the Perl scripts. What can tell us that? Well, the Unix command called `file` can peer inside the contents of a file to figure out what it is. Let's invoke that on each file:

```
use File::Find;
my @search = @ARGV;
@search = qw(.) unless @search;
find sub {
  return if -d $_ or -l $_;
  my $file_said = `file $_`;
  if ($file_said =~ /perl/) {
    print "$File::Find::name: $file_said";
  }
}, @search;
```

Hey, look at that. Now we're pulling out just the names that `file` insists are possibly Perl programs. But this program will slow to a crawl on a large tree. We're reinvoking the `file` command individually on every file in the tree.

There's a couple of ways to go from here to speed it up. I could save all the filenames to invoke file once at the end of the program:

```
use File::Find;
my @search = @ARGV;
@search = qw(.) unless @search;
my @list;
find sub {
  return if -d $_ or -l $_;
  push @list, $File::Find::name;
```

```
}, @search;
for (`file @list`) {
  if (/perl/) {
    print;
  }
}
```

And yes, that sped it up considerably faster, but now we don't get the results until the end of the tree walk, and we'll run into problems if the number of arguments exceeds a comfortable limit for `file`.

But there's another way. Out in the CPAN (at places such as `http://search.cpan.org`), we can find the `File::MMagic` module, which apparently is a Perl module derived from the `file` command created for the PPT project originally based on code written for Apache to implement the `mod_mime` module, to emulate the standard `file` command. Wow. And now I'm going to write a recursive controllable `file`-like program on top of that. Will the reuse ever stop? (I hope not!)

So, what we need from this module is the method called `checktype_filename`, which returns back a MIME type (like `text/plain` or `image/jpeg`), and perhaps a semicolon and some additional information. So let's find all the Perl scripts quickly. First, after a little playing around, I see that the string I'm looking for has "executable" followed by a space, then something ending in "perl" followed by a space and then "script". That's a simple regular expression, so I'll add that at the right place:

```
use File::Find;
use File::MMagic;
my $mm = File::MMagic->new;
my @search = @ARGV;
@search = qw(.) unless @search;
my @list;
find sub {
  return if -d $_ or -l $_;
  my $type = $mm->checktype_filename($_);
  next unless $type =~ /executable \S+\/perl script/;
  print "$File::Find::name: $type\n";
}, @search;
```

Now I know what programs to look at when I upgrade, to see which modules they all use. (Hmm. Sounds like an idea for another column. I'll note that.)

And one last fun one. Let's find all the images in the tree, and then call `Image::Size` (also found in the CPAN) on them to see their respective sizes. Just a few more tweaks:

```
use File::Find;
use File::MMagic;
use Image::Size;
my $mm = File::MMagic->new;
my @search = @ARGV;
@search = qw(.) unless @search;
my @list;
find sub {
```

```
    return if -d $_ or -l $_;
    my $type = $mm->checktype_filename($_);
    next unless $type =~ /^image\//;
    print "$File::Find::name: $type: ";
    my ($x, $y, $imgtype) = imgsize($_);
    if (defined $x) {
      print "$imgtype: $x x $y\n";
    } else {
      print "error: $imgtype\n";
    }
  }, @search;
```

And as it turns out, I could have left the `File::MMagic` out of this program, since `Image::Size` can cheerfully inform me when it wasn't called on an image, but you know the old Perl motto: There's More Than One Way To Do It!

So, next time someone asks you "What do you have?," I hope you can answer them with a nice short Perl program now. Until next time, enjoy!

Speeding Up Your Perl Programs

Unix Review, Column 49 (November 2003)

How fast does your Perl run? Hopefully, most of the time, the answer is "fast enough." But sometimes it isn't. How do we speed up a Perl program? How do we know what's making it unacceptably slow?

The first rule of speed optimization is "don't." Certainly, don't write things that waste time, but first aim for clarity and maintainability. With today's processor speeds far exceeding the "supercomputers" of yesterday, sometimes that's "fast enough."

But how do we know if something is "fast enough"? The first easy cut is to just check total runtime. That's usually as simple as putting "time" in front of your program invocation, and looking at the result (which may vary depending on your operating system and choice of shell). For example, here's a run of one of my frequently invoked programs, preceding the command `get.mini` with `time`:

```
[localhost:~] merlyn% time get.mini
authors/01mailrc.txt.gz ... up to date
modules/02packages.details.txt.gz ... up to date
modules/03modlist.data.gz ... up to date
67.540u 3.290s 1:40.62 70.3%        0+0k 1585+566io 0pf+0w
```

So, this tells me that this program took about 70 CPU seconds over a 100-second timespan, essentially doing nothing useful for me except verifying that my local mirror of the CPAN is up to date. (Generally, to get sensible numbers, I have to do this on an otherwise idle system.)

Hmm. I wonder where it's actually spending all of its time? We can get the next level of information by invoking the built-in Perl profiler on the script. Simply include `-d:DProf` as a Perl command switch, and Perl will automatically record where the program is spending its time into a file called `tmon.out` in the current directory:

```
[localhost:~] merlyn% perl -d:DProf `which get.mini`
authors/01mailrc.txt.gz ... up to date
modules/02packages.details.txt.gz ... up to date
modules/03modlist.data.gz ... up to date
[localhost:~] merlyn% ls -l tmon.out
-rw-r--r--    1 merlyn   staff    44991232 Sep  8 08:46 tmon.out
[localhost:~] merlyn%
```

(I have to use `which` here because `perl` won't look in my shell's search path for the script.) The execution under the profiler slows the program down a bit, but otherwise doesn't affect any functionality. We now need to summarize the raw data of `tmon.out` with the `dprofpp` (also included with Perl):

```
[localhost:~] merlyn% dprofpp
Total Elapsed Time = 99.50810 Seconds
  User+System Time = 70.79810 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 14.9   10.60 12.739 545807   0.0000 0.0000  URI::_generic::authority
 14.8   10.54 12.002 461833   0.0000 0.0000  URI::_generic::path
 13.1   9.296 19.853 293900   0.0000 0.0001  URI::new
 12.8   9.108 39.082  83971   0.0001 0.0005  URI::_generic::abs
 9.82   6.953 12.677 461842   0.0000 0.0000  URI::_scheme
 8.16   5.778  6.024 293900   0.0000 0.0000  URI::_init
 7.58   5.368 91.054  41987   0.0001 0.0022  main::my_mirror
 6.39   4.522  4.521 293900   0.0000 0.0000  URI::implementor
 5.78   4.093 32.267  41984   0.0001 0.0008  URI::_generic::rel
 5.07   3.588  3.588 251910   0.0000 0.0000  URI::_generic::_check_path
 4.26   3.016  3.016  83976   0.0000 0.0000  File::Spec::Unix::canonpath
 3.92   2.775 15.390  83968   0.0000 0.0002  URI::http::canonical
 3.48   2.465 14.186 377874   0.0000 0.0000  URI::scheme
 3.09   2.185  9.973  83968   0.0000 0.0001  URI::_server::canonical
 2.25   1.596  3.419  41988   0.0000 0.0001  File::Spec::Unix::catdir
[localhost:~] merlyn%
```

Now, this is some interesting information that we can use. We took 70 CPU seconds. This is just like the previous run, so the profiling didn't significantly alter our execution. The first routine listed, `URI::_generic::authority`, was called half a million times, contributing about 10 CPU seconds to the total. (There are other switches to `dprofpp`, but the default options seem to be the most useful most of the time.)

If we could make `URI::_generic::authority` twice as fast, we'd save about 5 seconds on the run. Even making `URI::new` run twice as fast would save us 5 seconds on the run as well, even though it's called half as frequently. Generally, the important thing here is to note total CPU for a routine, not the number of times it is invoked, or the CPU spent on an individual invocation. Although, I do find it interesting that I had to create 293,000 `URI` objects: perhaps there's some redesign of my algorithm to avoid that. Good algorithm design is important, and is both a science and an art.

The other thing to note is that most of the routines that are burning CPU are in libraries, not in my code. So, without changing the code of a library, I need to call library routines more efficiently or less frequently if I want to speed this up.

Then we have the bigger picture to consider. I invoke this program maybe a dozen times a day. If I worked hard to reduce this program to 35 CPU seconds instead of 70 CPU seconds, I'll save about 5 CPU minutes a day. How long would I have to work to optimize it to that level? And how many times would I have to invoke the program before the money I spent on the (usually otherwise idle) CPU being saved is equal to my billing rate for the time I spent?

This is why we usually don't worry about speed. CPUs are cheap. Programmer time is expensive.

But let's push it the other way for a minute. Suppose this program had to be invoked once a minute continuously. At 70 CPU seconds per run, we've just run out of processor (not to mention that the real-time speed was even longer). So we're forced to optimize the program, or buy a bigger machine, or run the program on multiple machines (if possible).

Armed with the data from the profiling run, we might want to tackle the `URI::new` routine. But rather than tweaking the code a bit, and rerunning the program overall, it's generally more effective to benchmark various versions of the code for given inputs, in isolation from the rest of the program.

For example, suppose we wanted to speed up the part of the routine that determines the URI's scheme separate from the rest of the URL. Three possible strategies come to mind for me initially: a regular expression match, a `split`, or an `index/substr`, implemented roughly as follows:

```
sub re_match {
  my $str = "http://www.stonehenge.com/perltraining/";;
  my ($scheme, $rest) = $str =~ /(.*?):(.*)/;
}
sub split_it {
  my $str = "http://www.stonehenge.com/perltraining/";;
  my ($scheme, $rest) = split /:/, $str, 2;
}
sub index_substr {
  my $str = "http://www.stonehenge.com/perltraining/";;
  my $pos = index($str, ":");
  my $scheme = substr($str, 0, $pos-1);
  my $rest = substr($str, $pos+1);
}
```

We now have three candidates. Which one is faster? For that, we can use the built-in `Benchmark` module, as follows:

```
use Benchmark qw(cmpthese);
my $URI = "http://www.stonehenge.com/perltraining/";;
sub re_match {
  my $str = $URI;
  my ($scheme, $rest) = $str =~ /(.*?):(.*)/;
}
sub split_it {
```

```
    my $str = $URI;
    my ($scheme, $rest) = split /:/, $str, 2;
  }
  sub index_substr {
    my $str = $URI;
    my $pos = index($str, ":");
    my $scheme = substr($str, 0, $pos-1);
    my $rest = substr($str, $pos+1);
  }
  cmpthese(-1, {
    re_match => \&re_match,
    split_it => \&split_it,
    index_substr => \&index_substr,
  });
```

which when run results in

```
                  Rate     re_match     split_it index_substr
re_match      131022/s           --         -37%         -46%
split_it      208777/s          59%           --         -14%
index_substr  242811/s          85%          16%           --
```

The `-1` on the `cmpthese` call says "run each of these for roughly 1 CPU second." Note that `index_substr` seems to be the clear winner here, even though there are more things for me to type and get correct.

One thing to watch out for in benchmarking is optimizing for atypical data. For example, the URI I picked is probably typical, but what happens if the URL is longer instead?

```
my $URI = "http://www.stonehenge.com/merlyn/"; .
  "Pictures/Trips/2003/03-06-PerlWhirlMacMania/" .
  "Day-0-Pearl-Harbor/?show=14";
```

Now our results show that `index_substr` is even better!

```
                  Rate     re_match     split_it index_substr
re_match      117108/s           --         -41%         -49%
split_it      199110/s          70%           --         -13%
index_substr  229682/s          96%          15%           --
```

Given those two extremes, I'd say that `index_substr` is probably the right tool for this particular task, all other things being equal.

Armed with Perl's built-in profiler and the `Benchmark` module, I can usually fine-tune my routines, and get the speed I want out of my Perl programs. If that's not enough, I could take those frequently called expensive routines and code them in a lower-level language instead. But usually that won't be needed. Until next time, enjoy!

Deep Copying, Not Deep Secrets

Unix Review, Column 30 (February 2000)

One of the modules I encountered in the CPAN had a problem with creating multiple objects. Each object got created fine, but on further use simultaneously in the program, some of the data from one object mysteriously shows up in the second one, even if the first one is freed!

Upon inspection, I found that the object was being initialized from a *shallow copy* of a template, and I told the author that he needed to *deep copy* the template instead. He was puzzled by the suggestion, and if you aren't familiar with these two terms, I bet you are a little confused now as well.

What is deep copying and why do we need it? Let's start with a simple example, and work back to the problem I posed a moment ago.

For example, let's grab all the password information into a simple hash, with the key being the username, and the value being an arrayref pointing to the nine-element value returned by `getpwent()`. On a first cut, we quickly hack out something like this:

```
while (@x = getpwent()) {
  $info{$x[0]} = \@x;
}
for (sort keys %info) {
  print "$_ => @{$info{$_}}\n"
}
```

What? Where did all the data go? We stored a reference to the data into the hash value. Well, maybe this will make it clearer:

```
while (@x = getpwent()) {
  $info{$x[0]} = \@x;
  print "$info{$x[0]}\n";
}
```

On my machine, this printed `ARRAY(0x80ce7fc)` dozens of times, once for every user on the system. So what does that mean? It means that we are reusing the same array repeatedly, and therefore we don't have dozens of arrayrefs pointing to distinct arrays, we have a single array with many pointers to the same data. So on the last pass through the gathering loop, `@x` is emptied, and therefore all the array references are pointing to the identical empty data.

This is because we made a *shallow copy* of the reference to `@x` into the hash value, which is a copy of only the top-level pointer, but not the contents. What we really needed was not only a copy of the reference, but a copy of to what the reference pointed. That's simple enough here:

```
while (@x = getpwent()) {
  $info{$x[0]} = [@x];
}
for (sort keys %info) {
  print "$_ => $info{$_} => @{$info{$_}}\n"
}
```

And now notice, we've got a distinct arrayref for each hash element, pointing to an independent copy of the nine elements of the array originally contained in @x. This worked because we created a new anonymous arrayref with the expression [@x], which also gives this anonymous array an initial value made of copies of the elements of @x.

So that's a basic *deep copy*: copying not only the top-level pointer, but also all the things within the data structure to maintain complete independence.

Actually there was one other way to ensure unique subelements in this example, and I'll show it for completeness lest my Perl hacking friends get irritated. You don't need to copy anything if you just generate the data in a distinct array in the first place:

```
while (my @x = getpwent()) {
  $info{$x[0]} = \@x;
}
for (sort keys %info) {
  print "$_ => $info{$_} => @{$info{$_}}\n"
}
```

Here, each pass through the loop starts with a brand-new completely distinct lexical @x rather than reusing the old existing variable. So when a reference is taken to it, and it falls out of scope at the bottom of the loop, the variable automatically remains behind as an anonymous variable.

But let's get back to deep copying. Here's another example. Let's suppose Fred and Barney are sharing a house:

```
$place = {
  Street => "123 Shady Lane",
  City => "Granite Junction",
};
$fred = {
  First_name => "Fred",
  Last_name => "Flintstone",
  Address => $place,
};
$barney = {
  First_name => "Barney",
  Last_name => "Rubble",
  Address => $place,
};
```

Now note that $fred->{Address}{City} is "Granite Junction" just as we might expect it, as is $barney->{Address}{City}. But we've done a shallow copy from $place into both of the Address element values. This means that there's not two copies of the data, but just one. We can see this when we change one of the values. Let's let Fred move to his own place:

```
$fred->{Address}{Street} = "456 Palm Drive";
$fred->{Address}{City} = "Bedrock";
```

Looks safe enough. But what happened to Barney? He moved along with Fred!

```
print "@{$barney->{Address}}{qw(Street City)}\n";
```

This prints Fred's new address! Why did that happen? Once again, the assignment of `$place` as the address in both cases made a shallow copy: both data structures shared a common pointer to the common street and city data. Again, a deep copy would have helped:

```
$place = {
  Street => "123 Shady Lane",
  City => "Granite Junction",
};
$fred = {
  First_name => "Fred",
  Last_name => "Flintstone",
  Address => {%$place},
};
$barney = {
  First_name => "Barney",
  Last_name => "Rubble",
  Address => {%$place},
};
$fred->{Address}{Street} = "456 Palm Drive";
$fred->{Address}{City} = "Bedrock";
print "@{$barney->{Address}}{qw(Street City)}\n";
```

There . . . now each `Address` field is a completely disconnected copy, so when we update one, the other stays pure. This works because just like the `[@x]` construct, we are creating a new independent anonymous hash and taking a reference to it.

But what if `$place` was itself a deeper structure? That is, suppose the street address was made up of a number and a name:

```
$place = {
  Street => {
    Number => 123,
    Name => "Shady Lane",
  },
  City => "Granite Junction",
};
$fred = {
  First_name => "Fred",
  Last_name => "Flintstone",
  Address => {%$place},
};
$barney = {
  First_name => "Barney",
  Last_name => "Rubble",
  Address => {%$place},
};
```

We've now done something that's not quite a deep copy, but also not a shallow copy. Certainly, the hash at `$fred->{Address}` is different from `$barney->{Address}`. But they both contain a value that is identical to the `$place->{Street}` hashref! So if we move Fred just down the street:

```
$fred->{Address}{Street}{Number} = 456;
```

then Barney moves along with him again! Now, we could fix this problem by applying the logic for copying the address one more time to the street structure:

```
$fred = {
  First_name => "Fred",
  Last_name => "Flintstone",
  Address => {
    Street => {%{$place->{Street}}},
    City => $place->{City},
  },
};
```

But as you can see, it's getting more and more convoluted. And what if we change `City` to be another structure, or added another level to `Street`? Bleh.

Fortunately, we can write a simple general-purpose deep copier with a recursive subroutine. Here's a simple little deep copy routine:

```
sub deep_copy {
  my $this = shift;
  if (not ref $this) {
    $this;
  } elsif (ref $this eq "ARRAY") {
    [map deep_copy($_), @$this];
  } elsif (ref $this eq "HASH") {
    +{map { $_ => deep_copy($this->{$_}) } keys %$this};
  } else { die "what type is $_?" }
}
```

This subroutine expects a single item: the top of a tree of hashrefs and listrefs and scalars. If the item is a scalar, it is simply returned, since a shallow copy of a scalar is also a deep copy. If it's an arrayref, we create a new anonymous array from the data. However, each element of this array could itself be a data structure, so we need a deep copy of it. The solution is straightforward: simply call `deep_copy` on each item. Similarly, a new hashref is constructed by copying each element, including a deep copy of its value. (The hash key is always a simple scalar, so it needs no copy, although that would have been easy enough to add.) To see it work, let's give some data:

```
$place = {
  Street => {
    Number => 123,
    Name => [qw(Shady Lane)],
  },
```

```
    City => "Granite Junction",
    Zip => [97007, 4456],
  };
  $place2 = $place;
  $place3 = {%$place};
  $place4 = deep_copy($place);
```

Hmm. How do we see what we've done, and what's being shared? Let's add a call to the standard library module, `Data::Dumper`:

```
  use Data::Dumper;
  $Data::Dumper::Indent = 1;
  print Data::Dumper->Dump(
    [$place, $place2, $place3, $place4],
    [qw(place place2 place3 place4)]
  );
```

And that generates on my system

```
  $place = {
    'City' => 'Granite Junction',
    'Zip' => [
      97007,
      4456
    ],
    'Street' => {
      'Name' => [
        'Shady',
        'Lane'
      ],
      'Number' => 123
    }
  };
  $place2 = $place;
  $place3 = {
    'City' => 'Granite Junction',
    'Zip' => $place->{'Zip'},
    'Street' => $place->{'Street'}
  };
  $place4 = {
    'City' => 'Granite Junction',
    'Zip' => [
      97007,
      4456
    ],
    'Street' => {
      'Number' => 123,
      'Name' => [
```

```
        'Shady',
        'Lane'
      ]
    }
};
```

Hey, look at that. `Data::Dumper` let me know that `$place2` is a shallow copy of `$place`, while `$place3` is an intermediately copied value. Notice the elements of `$place` inside `$place3`. And since `$place4` contains no previously seen references, we know it's a completely independent deep copy. Success! (The ordering of the hash elements is inconsistent, but that's immaterial and undetectable in normal use.)

Now, this simple `deep_copy` routine will break if there are recursive data pointers (references that point to already seen data higher in the tree). For that, you might look at the `dclone` method of the `Storable` module, found in the CPAN.

So, when you use = with a structure, be sure you know what you are doing. You may need a deep copy instead of that shallow copy. For further information, check out your online documentation with `perldoc perldsc` and `perldoc perllol` and even `perldoc perlref` and `perldoc perlreftut` for the basics. Until next time, enjoy!

Mirroring Your Own Mini-CPAN

Linux Magazine, Column 42 (November 2002)

The Comprehensive Perl Archive Network, known as "the CPAN," is the "one-stop shopping center" for all things Perl. This 1.2 GB archive contains over 13,000 modules for inclusion in your programs, as well as scripts, documentation, many non-Unix Perl binaries, and other interesting things.

Although there's nearly always a good, fast CPAN archive nearby when you are connected to the net, sometimes you're connected to the net at different speeds (like quickly at work, but slowly at home, or vice versa), or not at all. And what do you do then when you're like me, at 30,000 feet jetting off to yet another conference or customer site, and you realize you need a module that you haven't yet installed on your laptop? (This is especially an issue when a deadline for a magazine column looms close.)

Well, for the past year or so, I've been mirroring the entire CPAN to my laptop, thanks to the permission and cooperation of the owner of one of the major archive sites (and a few carefully constructed `rsync` commands). But at a recent conference, someone said, "Hey, can you just burn that onto a CD for me?," and I was stuck. The current CPAN exceeds the size of a CD-ROM, even though only a small portion of the files are needed for module installation!

So that got me thinking. If I brought down only the files that were needed by `CPAN.pm` to perform the installation of the latest release of a module, how big would that be? And the answer was wonderfully surprising: a bit more than 200 MB, which easily fits on a CD-ROM.

Unfortunately, I didn't see any clean, easy-to-use, efficient "mirror only the latest modules of the CPAN" program out there, so I wrote my own, which I present in Listing 1-3 later.

Lines 1 through 3 start nearly every long program that I write, enabling warnings, compiler restrictions, and disabling buffering on `STDOUT`.

Lines 5 through 17 form the configuration section of this program. There's really only three things to set here.

`$REMOTE` is the URL prefix leading to the nearest CPAN archive. The uncommented value is the main United States CPAN archive. The next value is the Finland archive, which also happens to be the master archive. If you want the most up-to-date sources, they're here. And because I was initially developing this program at the annual SAGE-AU conference in Australia, the value following that is the Australian CPAN archive. Finally, I have a *complete* CPAN archive on my laptop's disk already, so I can point to that with a `file:` URL as well, as shown by the fourth value.

That's the source, and we need to define a destination, and that's in `$LOCAL`. This is a simple Unix path. If you're on a non-Unix system, you can specify this in the local directory syntax, since we'll be using the cross-platform `File::Spec` library to manipulate this path. And, as the comment warns, this program owns the contents of that directory, and is free to delete anything it sees fit, so keep that in mind as you are specifying the path.

Finally, a simple true/false `$TRACE` flag decides whether this program is noisy by default or quiet by default. The noise is limited to actual activity, and reassures me during execution that something is happening.

Next, from lines 20 to 30, we'll pull in the necessary modules. The standard Perl bundle gives us the `dirname`, `catfile`, and `find` routines. The optional CPAN-installable `LWP` library gives us the `URI` object module and the `mirror` routine (and some associated status values). And `Compress::Zlib` lets us expand the gzip-compressed index file so we know what distributions are needed for the mirror.

Once we've got everything set up, it's time to transfer everything needed for a typical operation of the core `CPAN` module (described by `perldoc CPAN` in a typical Perl installation). First, we need the index files, defined in lines 34 to 36. We'll call `my_mirror` on each of those, defined later. For now, we'll presume that this creates or refreshes each of those files below the `$LOCAL`-identified directory.

The `02packages.details.txt.gz` file is a flat text file with a short header that contains the path to each distribution for each module in the CPAN. However, this file is gzip-compressed, so we need to expand the file to process the contents. Stealing the example out of the `Compress::Zlib` manpage nearly directly, lines 40 to 52 expand this file and extract the necessary information.

Line 40 constructs the filename in a platform-independent way by using the `catfile` routine. Note that we're actually passing three parameters. The first parameter is the value of `$LOCAL`, which serves as the starting point, from which we descend further to the subdirectory called `modules`, and thence finally to a file within that directory called `02packages.details.txt.gz`. I've tested this only on Unix, but I'll presume that the program is portable, because I've used the portable functions.

Line 41 takes this constructed path, and creates a `Compress::Zlib` object, which can be asked to deliver the uncompressed file line-by-line. If that fails, we're in an unrecoverable state, and we'll abort.

The data contains a header, delimited by a blank line, so we need to skip over all the data up to and including that blank line. We'll do this by setting a flag to an initial `1` value in line 42. Line 43 reads a line at a time into `$_`, stopping when there is no more data (or there's an I/O error). Lines 44 to 47 look for the end of the header as long as we're still in the header. A header ends on a line that doesn't contain a nonblank character, hence the `unless`.

If we make it to line 49, we're staring at a standard line from the index, which looks something like

```
Parse::RecDescent 1.80 D/DC/DCONWAY/Parse-RecDescent-1.80.tar.gz
```

The first column is the module name (here `Parse::RecDescent`), and is not very interesting to us. Neither is the second column, which is the current version number. But the third column contains (the unique part of) the path to the distribution for this module, and that's what the `CPAN` module will be looking for, and what we need to mirror.

Note that many module names will share the same common distribution file, so we'll need logic to avoid downloading duplicates. We'll defer that problem to the `my_mirror` subroutine.

A few of the modules are listed as belonging to a core Perl distribution. To avoid mirroring the various Perl distributions (and wasting space in our mirror), we'll skip over them in line 50. The regular expression is somewhat ad-hoc, but seems to do the right thing.

Line 51 mirrors the requested distribution into our local mirror. The `1` parameter says "If it already exists, it's up to date," and is an optimization based on external knowledge that a given distribution will never be updated in place. Rather, a new file will be created with a new version number. Of course, like any optimization, we do this with some hesitation and a bit of caution.

Once we've passed through the entire module list, we need to delete any outdated modules. A CPAN contributor has the option of leaving older versions of modules in the CPAN, or deleting them. We need to keep track of everything that is current, and delete anything not mentioned, in order to keep in sync with the master archive.

And that's it, as line 57 confirms.

But of course, that's not the whole story. We need to manage the mirroring. There are two steps to mirroring: fetching the files, and throwing away anything left over. These need to share a common hash, which we'll define as a closure variable inside a `BEGIN` block starting in line 59. The `%mirrored` hash in line 62 is keyed by the filename, and has a value of 1 to indicate that the file has been at least checked for existence, and 2 to indicate that it has been mirrored from the remote site and brought up to date. At the end of the run, any files that aren't either 1 or 2 for values are deleted files or temp files, and should be deleted from our mirror.

The `my_mirror` routine starting in line 64 does the hard work. The two parameters are the partial URL path and the "skip if present" flag.

In line 68, we use the `URI` module to construct the full URL, based on the `$REMOTE` value and the partial path. Line 69 constructs the local file path, based on `$LOCAL` and the partial path as well. The task for the remainder of the subroutine is to make the local file be up to date with respect to the remote URL.

Line 70 manages the checksum file. Each distribution is checksummed to ensure proper complete transfer. We'll first pretend that the checksum file doesn't need updating, but later remove that assumption if we end up transferring the distribution file.

Starting in line 72, we look at what to do to bring this file up to date. If `$skip_if_present` is true, then we'll never worry about the remote timestamp being out of sync. If the file is present, it's good enough, noted by the `-f` flag in line 72. Line 74 records that the file was at least checked for existence, so we don't delete it during the cleanup phase.

If `$skip_if_present` is not true, or the file doesn't exist, then it's time to do a full mirror on this distribution. We'll note that in line 77. Line 79 creates the directory to receive the file. (I would argue that `LWP` should do this for me, but that's not the way it works.) The `$TRACE` value causes a series of `mkdir` command lines to be traced to the output; otherwise, this operation is silent. Line 80 also puts out some noise if `$TRACE` is set: note the absence of a newline, because we're going to follow on with a result status.

Line 81 is where the real work happens. We'll call `mirror` to bring the remote URL to the local file. This is done in such a way that the existing modification timestamp (if any) is noted

and respected, minimizing the load on the remote server. And the file is actually written into a temp file, and then renamed only when the transfer is complete, thus ensuring that other users of this directory will not see partially transferred files at normal locations. (If one of these transfers aborts midway, the cleanup phase at the end of this program will delete the partial transfer.) The modification time is also updated to that of the remote data, so that a later mirror will again note that the file is up to date.

The result of `mirror` is an HTTP status value. If it's `RC_OK`, then we've got a new version of the remote file. In this case, the checksum file may now be out of date: we can't merely check for its existence, so we'll flag that by setting the variable to 0 in line 84.

If the response is `RC_NOT_MODIFIED`, then we already had an up-to-date version of the file, and the remote server has informed us of such without even sending us a new version. In that case, we end up in line 90, finishing out the tracing message if needed.

However, if the status is neither of these, then something wrong has happened, and we'll generate a `warn` noting the status, and abort any further operation on this path by returning from the subroutine.

Once the distribution has been transferred, it's time to grab the checksum file. If the path is a distribution (checked in line 94), we'll compute the path to the `CHECKSUMS` file in lines 95 and 96. We must be careful to perform URL calculations here, not native path calculations. And, to keep the algorithm easy, we need to compute the path relative to the original CPAN mirror base, not a full path. Thankfully, this is also trivial with the `URI` module.

In line 97, if we're not already looking at a `CHECKSUMS` file, we need to call back to ourselves to transfer the file. This is a clean tail recursion, so I could have simply used a `goto` or a loop, but the subroutine call seemed easier and clearer at the time. If the checksum might already be up to date, it will merely be checked for its presence. If a transfer has taken place, a full mirror call will be issued instead.

Finally, we have the cleanup phase routine. We'll start at `$LOCAL` using the `File::Find` recursion. If a file exists, and it's not noted as such in the `%mirrored` hash (line 105), then we remove it (line 107).

And there you have it. Set up the configuration, and let it rip. On the first execution, you will want to be on a fast link (or a relatively unloaded time of day), because it downloads about 200 megabytes of data. After that, it's about 2–5 minutes per (average) day on a 28.8 link, which is completely tolerable for me from my hotel room when I'm on the road. And don't forget: you're downloading only installable modules, not the rest of the CPAN.

To use this mini-CPAN mirror with `CPAN.pm`, you'll need to enter at the CPAN prompt

```
o conf urllist unshift file://$LOCAL
o conf commit
reload index
```

Here, `$LOCAL` is replaced by the value you've set in `$LOCAL` but specified as a URL path (forward slashes for directory delimiters, and percent-escaped unusual characters). That's because `CPAN.pm` is expecting a URL, not a file path.

At the risk of repeating myself: this won't make CPAN installations any faster, unless you happen to be a road warrior like me, needing to do CPAN installations when you are on a very slow net link (or no link at all). Of course, you could burn a daily CD for your friends, and "hand them a CPAN archive on a disk," providing a gateway between your bandwidth and the sneakernet. At least you won't be worrying trying to figure out how to fit the full 1.2+ GB CPAN on a CD-ROM! Until next time, enjoy!

Listing 1-3

```
=1=     #!/usr/bin/perl -w
=2=     use strict;
=3=     $|++;
=4=
=5=     ### CONFIG
=6=
=7=     my $REMOTE = "http://www.cpan.org/";;
=8=     # my $REMOTE = "http://fi.cpan.org/";;
=9=     # my $REMOTE = "http://au.cpan.org/";;
=10=    # my $REMOTE = "file://Users/merlyn/MIRROR/CPAN/";;
=11=
=12=    ## warning: unknown files below this dir are deleted!
=13=    my $LOCAL = "/Users/merlyn/MIRROR/MINICPAN/";
=14=
=15=    my $TRACE = 1;
=16=
=17=    ### END CONFIG
=18=
=19=    ## core -
=20=    use File::Path qw(mkpath);
=21=    use File::Basename qw(dirname);
=22=    use File::Spec::Functions qw(catfile);
=23=    use File::Find qw(find);
=24=
=25=    ## LWP -
=26=    use URI ();
=27=    use LWP::Simple qw(mirror RC_OK RC_NOT_MODIFIED);
=28=
=29=    ## Compress::Zlib -
=30=    use Compress::Zlib qw(gzopen $gzerrno);
=31=
=32=    ## first, get index files
=33=    my_mirror($_) for qw(
=34=                         authors/01mailrc.txt.gz
=35=                         modules/02packages.details.txt.gz
=36=                         modules/03modlist.data.gz
=37=                        );
=38=
=39=    ## now walk the packages list
=40=    my $details = catfile($LOCAL, qw(modules ↵
        02packages.details.txt.gz));
=41=    my $gz = gzopen($details, "rb") or die "Cannot open details: ↵
        $gzerrno";
=42=    my $inheader = 1;
=43=    while ($gz->gzreadline($_) > 0) {
```

```
=44=      if ($inheader) {
=45=        $inheader = 0 unless /\S/;
=46=        next;
=47=      }
=48=
=49=      my ($module, $version, $path) = split;
=50=      next if $path =~ m{/perl-5};  # skip Perl distributions
=51=      my_mirror("authors/id/$path", 1);
=52=    }
=53=
=54=    ## finally, clean the files we didn't stick there
=55=    clean_unmirrored();
=56=
=57=    exit 0;
=58=
=59=    BEGIN {
=60=      ## %mirrored tracks the already done, keyed by filename
=61=      ## 1 = local-checked, 2 = remote-mirrored
=62=      my %mirrored;
=63=
=64=      sub my_mirror {
=65=        my $path = shift;           # partial URL
=66=        my $skip_if_present = shift; # true/false
=67=
=68=        my $remote_uri = URI->new_abs($path, $REMOTE)->as_string; ⏎
          # full URL
=69=        my $local_file = catfile($LOCAL, split "/", $path);⏎
          # native absolute file
=70=        my $checksum_might_be_up_to_date = 1;
=71=
=72=        if ($skip_if_present and -f $local_file) {
=73=          ## upgrade to checked if not already
=74=          $mirrored{$local_file} = 1 unless $mirrored{$local_file};
=75=        } elsif (($mirrored{$local_file} || 0) < 2) {
=76=          ## upgrade to full mirror
=77=          $mirrored{$local_file} = 2;
=78=
=79=          mkpath(dirname($local_file), $TRACE, 0711);
=80=          print $path if $TRACE;
=81=          my $status = mirror($remote_uri, $local_file);
=82=
=83=          if ($status == RC_OK) {
=84=            $checksum_might_be_up_to_date = 0;
=85=            print " ... updated\n" if $TRACE;
=86=          } elsif ($status != RC_NOT_MODIFIED) {
=87=            warn "\n$remote_uri: $status\n";
=88=            return;
```

```
=89=          } else {
=90=            print " ... up to date\n" if $TRACE;
=91=          }
=92=        }
=93=
=94=        if ($path =~ m{^authors/id}) { # maybe fetch CHECKSUMS
=95=          my $checksum_path =
=96=            URI->new_abs("CHECKSUMS", $remote_uri)->rel($REMOTE);
=97=          if ($path ne $checksum_path) {
=98=            my_mirror($checksum_path, $checksum_might_be_up_to_date);
=99=          }
=100=       }
=101=     }
=102=
=103=     sub clean_unmirrored {
=104=       find sub {
=105=         return unless -f and not $mirrored{$File::Find::name};
=106=         print "$File::Find::name ... removed\n" if $TRACE;
=107=         unlink $_ or warn "Cannot remove $File::Find::name: $!";
=108=       }, $LOCAL;
=109=     }
=110=   }
```