by Jeffreys Copeland and Haemer

Looking Through Our Mail

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

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

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

Letters, We get letters,
We get stacks and stacks of letters …
– Perry Como

Recently, we received a letter from a corporate attorney advising us that the courts had ordered his client to produce all of our correspondence with that client. We were surprised by this–wouldn't you be?–if for no other reason than we didn't think we'd had any. But with as much mail as we get, who could tell?

Curious, we tried using `grep` to look through all our old, saved mail, but the strings we were looking for were very common. Worse, `grep` really didn't give us what we wanted: mail messages. Stepping back a moment, we realized that we needed a tool that understood the semantics of mail. In fact, we reasoned, this wasn't even a limited-utility tool. Questions like, "Where the heck are those messages from Charlie about the Viper project?" are pretty frequent.

How long would it take to write such a tool from scratch? All morning, as it turned out. What's more, doing it is a good illustration of how to build tools quickly, so in this column, we'll walk you through the process.

As a side benefit, the next time you know of someone who's trying to look through large volumes of electronic mail, you'll be prepared to help with a line-by-line explanation of this tool.

Design by Analogy

Start by asking yourself how the tool should look to the user. Even if you're making something that no one else will ever use, design the interface as though you're writing it for someone else; chances are, six months later, you'll forget why you made the choices you did. A good rule of thumb is to make the tool look as much as possible like something else familiar to you. Because "theft" is a poor choice of words around lawyers, we like the phrase "design by analogy."

Here, we want something `grep`-like, so we'll try to make the interface look like `grep`'s. We'll even call our tool `mgrep`, for *mailgrep*, to keep things simple. This decision means that in two years, after we've forgotten what we did, when we sit down to look for mail messages containing the string `zzazz` in the mailboxes `Feb.mbox` and `Mar.mbox`, we can start out trying `mgrep zzazz Feb.mbox Mar.mbox` and not be surprised with the result. Also, a little attention to detail now will let us pull out all messages containing zzazz or ZZAZZ or Zzazz with mgrep -i zzazz Feb.mbox Mar.mbox. Why -i? Because that's the flag grep uses for case-insensitive matches.

Moreover, this saves us from having to design all the features and options either from scratch or at once. For example, if we do a bare-bones implementation now, and a year later we want to add an option that means "report all mailboxes that contain messages with this pattern," we won't have to convene a design committee. All we need to do is glance through the grep man page, note the -l option (which prints only the names of files containing lines that match the pattern), and write code to add a -l flag to mgrep.

More Theft

Man is a tool-using animal… Without tools he is nothing, with tools he is all. – "Sartor Resartus," Thomas Carlyle

On to implementation. Our goal is to get something working quickly, so Perl seems like a good tool to use because we know Perl–familiarity is never a factor to be ignored. What's more, this is a text-processing problem–one of Perl's strengths. If we're going to use Perl, our first impulse is to continue to steal by raiding the Comprehensive Perl Archive Network (CPAN). Oh, sorry. Make that, "…build on the work of others."

We'll begin by perusing the modules list at http://www.perl.com/CPAN/modules/. Well, not actually *there*, because going to http://www.perl.com/CPAN/ takes you to a multiplexer that automatically throws you to the nearest mirror site. This trick gets you good performance without making you memorize a lot of URLs. (Note: The trailing slash is very important. Without it, you don't get the multiplexer.) Once there, we quickly find our way to http://www.perl.com/CPAN/modules/00modlist.long.html, the current module list, and begin looking. Modules are Perl's rough equivalent of Ada packages or C++ classes: language extensions, often object-oriented, to handle specific problems.

What will we need? Something to handle argument parsing would be nice, so we don't have to handcraft our emulation of grep's flags. A search for "option" quickly yields an entire section of the CPAN, which begins like that shown in Listing 1.

Getopt::Std ("Implements basic getopt and

Listing 1

```
12) Option, Argument, Parameter and Configuration File Processing

Name            DSLI    Description                                    Info
---------       ----    --------------------------------------------   ---------
Getopt::
::EvaP          Mdpr    Long/short options, multilevel help            LUSOL
::Gnu           adcf    GNU form of long option handling               WSCOT
::Help          bdpf    Yet another getopt, has help and defaults      IANPX
::Long          Supf    Advanced option handling                       JV
::Mixed         Rdpf    Supports both long and short options           CJM
::Regex         ad      Option handling using regular expressions      JARW
::Simple        RdpO    A simplified interface to Getopt::Long         RSAVAGE +
::Std           Supf    Implements basic getopt and getopts            P5P
::Tabular       adpr    Table-driven argument parsing with help text   GWARD +
```

Listing 2

```
19) Mail and Usenet News

Name            DSLI    Description                                    Info
-------------   ----    -----------------------------------------     --------
Mail::
::Address       adpf    Manipulation of electronic mail addresses      GBARR
::Alias         adpO    Reading/Writing/expanding of mail aliases      GBARR
::Cap           adpO    Parse mailcap files as specified in RFC1524    GBARR
::Field         RdpO    Base class for handling mail header fields     GBARR +
::Folder        adpO    Base-class for mail folder handling            KJOHNSON
::Header        RdpO    Manipulate mail RFC822 compliant headers       GBARR +
::Internet      adpO    Functions for RFC822 address manipulations     GBARR
::MH            adcr    MH mail interface                              MRG

::Mailer        adpO    Simple mail agent interface (see Mail::Send)   GBARR
::POP3Client    bdpO    Support for clients of POP3 servers            SDOWD
::Send          adpO    Simple interface for sending mail              GBARR
::Util          adpf    Mail utilities (for by some Mail::* modules)   GBARR
```

Listing 3

```
tar -zxvf MailTools-1.1003.tar.gz   # unpack the archive
cd MailTools-1.1003                 # enter the source directory


perl Makefile.PL                    # build a Makefile for your system
make                                # build the package
make test                           # test (!) it
make install                        # install it
```

`getopts`") should do the trick, working to match the familiar POSIX.1 call, `getopt()`. What's more, the "S" in the second column means that this module is a standard part of the Perl 5 distribution, so we don't even have to pull a copy off the archive. What else? Well, Section 19 looks pretty good (see Listing 2).

Most of these are packaged up in a single `tar` file, by Graham Barr, called "MailTools," so we pull over the most recent version, `MailTools-1.1003.tar.gz`. After we unpack it (assuming we have GNU `tar` with a decompressor–the `-z` flag–otherwise, we can pipe `gzcat` into `tar`), building and installing the module and its documentation requires only following the instructions in the `README` (see Listing 3).

A little manual page perusal reveals that this is enough. A small amount of work–some of it cut-and-paste, which one of our colleagues calls "snarf-and-barf "–gives us the code shown in Listing 4.

but have found that if we collect most of our declarations in one place then it's easier to notice when we're using several different variables to do almost the same thing. We also like to declare our scalar, hash and array variables in separate statements, but that's idiosyncrasy, not Perl.

Lines 10 through 16 process the command-line arguments and give them default values. After the call to `Getopt::Std::getopts()`, all option values are contained in the hash `%opt_args`, and the only things left in `@ARGV` are the pattern to search for and the file names–no muss, no fuss, nothing to tidy up. The loop beginning on line 14 gives any unselected options the value zero. (For the really nitpicky, we are aware that it also sets `$opt_arg{':'}` to zero: meaningless, but harmless.)

Lines 17 through 23 actually interpret some of the options. As long as we are looking for mail messages that contain strings, why not let users specify what part of the mail message to look in? As it turns out, the `Mail::Internet` module lets us get

Exegesis

Lines 1 through 4 are boilerplate: They guarantee that the script is interpreted by a version of Perl that has enough features to support it; they provide an RCS ID string, to let us know what revision of our code we're dealing with; and turn on lots of warnings, both at compile time and runtime, to prevent us from wasting time debugging really stupid mistakes. We are trying to minimize development time, not running time.

Lines 5, 6 and 7 pull in the three modules from which we'll be using functions. Lines 8 and 9 declare variables. (Line 4 tells the compiler to complain about undeclared variables, which helps catch typos.) We could declare them as we use them

Listing 4

```
1  #!/usr/local/bin/perl -w
2  # $Id: mgrep,v 1.4 1997/12/27 00:11:13 jsh Exp $

3  require 5.004;
4  use strict;

5  use Getopt::Std;
6  use Mail::Util qw(read_mbox);
7  use Mail::Internet;

8  my ($options, $parts, $pattern, $usage);
9  my %opt_args;

10 # parse args and check for proper invocation
11 $usage = "usage: $0 [-b|-h|-H Header_field|-W] [-i] [-v] pattern [mailbox ...]";
12 $options = 'H:Wbhiv';
13 getopts $options, opt_args or die $usage;
14 foreach (split //, $options) {
15   $opt_args{$_} ||= 0;
16 }

17 $parts = $opt_args{'b'} + $opt_args{'h'} + ($opt_args{'H'} ? 1 : 0);
18 $parts < 2 or die $usage;
19 $opt_args{'W'} = ! $parts;

20 $pattern = shift;
21 if ($opt_args{'i'}) {
```

```
22   $pattern = "(?i)$pattern";
23 }

24 if (@ARGV == 1) {push @ARGV, "/dev/stdin";}
25 while (@ARGV) {
26   my $mbox = shift;
27   $^W = 0;
28   my @msgs = read_mbox $mbox or die "Can't read $mbox:$!";
29   $^W = 1;
30   foreach (@msgs) {
31     my ($tgt, $mail);
32     $mail = Mail::Internet->new($_);

33     my $head = $mail->head;
34     my $body = $mail->body;

35     if ($opt_args{'W'}) {     # the default
36         $tgt = [@$body, @{$head->header}];
37     } elsif ($opt_args{'b'}) {
38         $tgt = $body
39     } elsif ($opt_args{'h'}) {
40         $tgt = $head->header;
41     } elsif ($opt_args{'H'}) {
42         $tgt = [ $head->get($opt_args{'H'}) ];
43     } else {
44         die $usage;
45     }

46     $mail->print
47         if ( grep /$pattern/, @$tgt xor $opt_args{'v'} );
48   }
49 }

50 =head1 NAME
51 mgrep - look through mailboxes for messages containing a string

52 =head1 SYNOPSIS

53   mgrep [-bhiv] pattern [mailbox ...]

54 =head1 DESCRIPTION

55 =over 2

56 I<mgrep> looks for mail messages containing a pattern,
57 and prints the resulting messages on standard out.

58 By default looks in both header and body for the specified pattern.

59 When redirected to a file, the result is another mailbox,
60 which can, in turn, be handled by standard User Agents,
61 such as I<elm>,
62 or even used as input for another instance of I<mgrep>.

63 =back

64 =head1 OPTIONS AND ARGUMENTS

65 Many of the options and arguments are analogous to those of grep.

66 =over 8

67 =item B<pattern>
```

each of these, separately, so we use the options `-b`, `-h`, `-H` and `-W` to tell `mgrep` to look in the body, header, specific header field or whole message, respectively. (The same flags given to `grep` aren't all that interesting, so we'll use the letters for something more meaningful.) Lines 17 and 18 enforce a prohibition against mixing these options and make `-W` the default.

Line 20 grabs the pattern. This pattern can be any Perl regular expression, which means that our tool will actually be a little easier to use than `grep`, or `egrep`, which only understand POSIX regular expressions. Lines 21 through 23 implement the `-i` (case-insensitive matching) option, with Perl 5's new syntax for regular expression extensions. Unlike the syntax `/pattern/`*i*, which specifies case-insensitivity at compile time, putting `(?i)` at the beginning of a pattern makes case-insensitivity part of the pattern itself, so you can specify case sensitivity at runtime.

Lines 24, 27 and 29 are hacks, impelled by the current implementation of `Mail::Util::read_mbox()`. Lines 27 and 29, which bracket the call, are there to temporarily turn off the `-w` flag and block a complaint about the internals of `read_mbox()`.

Line 24 lets `mgrep` read from standard input if no files are named in the argument list. Here again, the normal Perl idiom, `while(<>)`, is unavailable because of a detail of the implementation of `read_mbox()`. This brings up an important point: We could have avoided having to put in these three hacks, by writing our own replacement. But how much work would that be?

Lines 26 and 28 grab the mailbox named on the command line and transform the mailbox into an internal form–an array of individual mail messages–for processing. The remainder of the program loops through that array, looking inside each message for the pattern, and printing the requested messages.

Line 32 turns an individual mail message into an object with methods listed in the `Mail::Internet` module, and lines 33 and 34 use these methods to extract the header and body.

Lines 35 through 45 use other methods from the same module to create an array of text lines to search for the pattern. What gets stuffed into the array depends on the value of the `-b`, `-h`, `-H` and `-W` flags, but the end result is a reference to the target array, `$tgt`. (You might think that lines 43 through 45 are superfluous. You might think you could even prove, mathematically, that earlier code

```
68 The pattern to search for in the mail message.
69 May be any Perl regular expression,
70 but should be quoted on the command line
71 to protect against globbing (shell expansion).

72 =item B<mailbox>

73 Mailboxes must be traditional, UNIX C</bin/mail> mailbox format.
74 If no mailbox is specified, takes input from stdin.

75 =item B<-b>

76 Look only in the bodies of mail messages.

77 =item B<-h>

78 Look only in the headers of mail messages.

79 =item B<-H>

80 Look in the specified header of mail messages.
81 Field names are case-insensitive.

82 =item B<-i>

83 Make the search case-insensitive (by analogy to I<grep -i>).

84 =item B<-v>

85 Invert the sense of the search, (by analogy to I<grep -v>).

86 =item B<-W>

87 Look through the entire mail message (default)

88 =back

89 =head1 EXAMPLE

90   find . -name '*mbox' -print | xargs mgrep -i alstadt > /tmp/alstadt.mbox

91 This finds every file whose name ends in C<mbox>
92 under the current directory, searches each for messages containing
93 the strings "alstadt," "ALSTADT," "Alstadt," etc.,
94 and puts a copy of everything it finds into C</tmp/alstadt.mbox>

95   find . -name '*mbox' -print | xargs mgrep -H to brother

96 This searches the same set of files for messages containing
97 the string "brother" in the "To:" field.

98 =head1 AUTHOR

99  Jeffrey S. Haemer, <jsh@boulder.qms.com>

100 =head1 SEE ALSO

101 elm(1), mail(1), grep(1), perl(1), printmail(1), Mail::Internet(3)
102 Crocker, D. H.,
103 Standard for the Format of Arpa Internet Text Messages, RFC822.

104 =cut
```

has guaranteed that one of these flags has to be set. Sure it has. We put stuff like this in because experience tells us it always saves us a lot of debugging time.) Whew.

Now, if you look back, you'll see that most of what we've done so far is really just argument handling. We've tried to do things sturdily, so that when the code gets this far, it's likely to look for what we think we are asking for. Moreover, we've tried to do things professionally enough that if the code *doesn't* get this far, it fails cleanly. Even so, it's taken us only 45 lines of code and comments.

But what about the real work? Oh, you mean the remainder of the program: lines 46 and 47. Line 47 uses Perl's built-in `grep` function, which searches an array of lines for a Perl regular expression, to impose selection criteria on each message, for example, the `xor` implements the `-v` flag. Line 46 uses one of the mail-message methods to print any selected messages in RFC 822 format. Done.

Lines 50 through 104, more than half the total number of lines in the file, are documentation. Even though we've designed this so that you shouldn't need to look at a man page very often, that doesn't keep us from writing one. As is usual for Perl utilities, the documentation is in POD (plain old documentation) form. Not only does POD documentation live in the same file as the code it describes–it's normally ignored by `perl`–but tools that come with the standard distribution let you transform such code-documentation chimeras into a variety of attractive documents, including flat text, UNIX man pages and Web pages. The CPAN even has a module that will let the code part use the documentation part to generate runtime usage and help messages.

And that, gentle reader, is that. We'll be back next month with more amazing programmer tricks. Until then, happy trails.