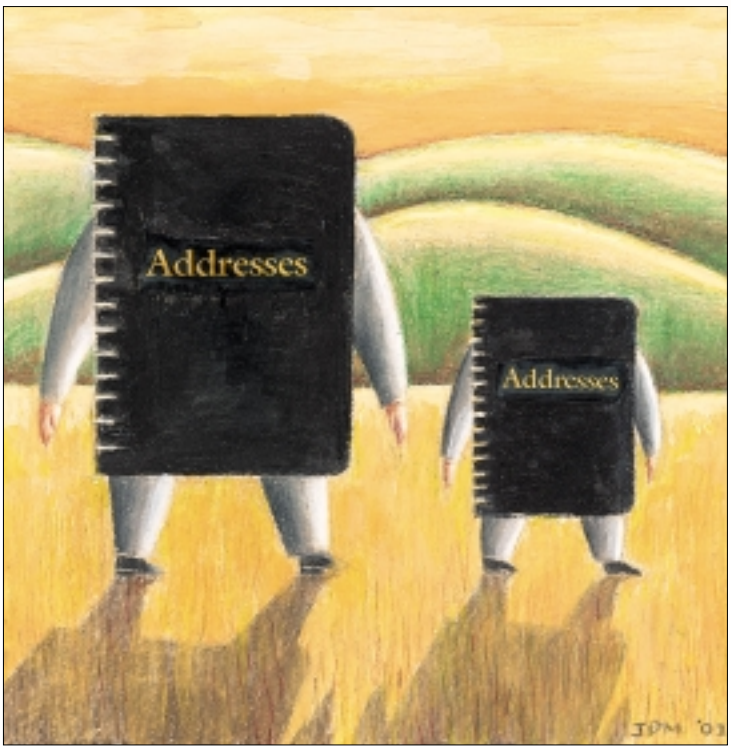
by Jeffreys Copeland and Haemer



JANE MARINSKY

Return to sender,
address unknown,
no such number,
no such zone.
– Otis Blackwell
and Winfield Scott,
Return to Sender

Through my own
fault I may find
You're no longer
living at this address
Please excuse the
lack of news.
– Elvis Costello,
The Letter Home

Son of Address Book

Those of you who have been reading us for a long time will remember that one of the first things we discussed in our columns was address books. We provided two alternate implementations, one based on *troff*'s bibliography filter *bib*, the other based on a set of utilities for handling block paragraphs. (*RS/Magazine*, May, June, July 1995.) See `http://alumni.caltech.edu/~copeland/work` for these; they unfortunately don't appear at `http://search.cpg.com`. Since it is the block paragraph implementation we've continued to use over the years, the experiment we'll conduct this month will be based on that version.

To review, our address book consists of a series of paragraphs separated by blank lines, such as:

```
Jeff Haemer
960 Ithaca Dr
Boulder  CO 80303
#h: 303-499-2619
#w: 303-443-7227 x16
#f: 303-443-7107
%jsh@usenix.org
>xmas-card
```

There are other variant address formats which we'll explore shortly, but that's the basic form. Each non-address line has a tag to help our shell scripts find it.

"What," you ask impatiently, "is this month's experiment?" Continuing from our "switch everything around" exercise of last month, one of us recently found himself using Microsoft Outlook as a mail client. But Outlook isn't just a mail client, it also makes a great floor wax, er, manages your address book. Long-time readers will know that our intrinsic masochism doesn't extend to retyping all our address book entries into a new application; we'd much rather expend the skull sweat building software to import the data for us.

So, how do we import into Outlook? We need to do a little reverse engineering. We can start by entering a couple of addresses into Outlook by hand, and then exporting them into a flat text file. (Click on "File," "Import and Export," and then "Export to a File.") We have several possible output formats, but the simplest one, the flat text version, is called a CSV file, or comma-separated-value file. CSV files are also used as a transmission format for a number of other programs, such as the Excel spreadsheet.

What does a CSV file look like? It's a series of very long lines, which makes it a great candidate to process with Perl. Each item is surrounded by quotes, and (as implied by the format name) separated from the next by a comma. The first line is a list of the field names:

```
"Title","First Name",
 "Middle Name","Last Name",
 "Suffix","Company","Department",
 "Job Title","Business Street",
 ....
 "Web Page"
```

Then, we have lines like,

```
"Dr","Jeffrey","Sherman",
 "Haemer","PhD","QMS",
 "Boulder R&D","Proj Mgr",
 ....
```

(We are, of course, folding these lines so they'll fit on the page; read them as though they were strung out.)

Hashing Over Our Example

From the data we've got here, we realized that the simplest thing to do is keep an array of the fields in the correct order, and then have a hash indexed by the field names containing the data elements. This lets us use a fairly simple output fragment:

```
foreach( @fields ) {
 my $x = defined($fields{$_}) ?
  $fields{$_} : "";
 $x =~ s/\s+$//;
 push(@outputline, $x);
 $fields{$_} = undef;
}
print "\"", join('","',@outputline), "\"\r\n";
```

Notice that we're pushing empty elements into the `output line` array for undefined elements in the hash. Also notice that, because we're expecting our output to be read by a Windows program, we end our line with both `CR` and `LF` characters.

So, how to build the real program? We can begin in the usual way: shebang line, RCS id, comment, turn on `strict` checking:

```
#! /usr/local/bin/perl -w
# $Id: CSV,v 1.5 2001/07/06 16:56:43 jeff Exp $
# convert address entries from our text
# database into a CSV file suitable
# for import into Outlook

use strict;
```

Once we've got the beginning of the program, we can list (in order) the individual field names that Outlook uses.

```
# define all the field names
my @fields = (
 "Title", "First Name", "Middle Name",
 "Last Name", "Suffix", "Company",
 "Department", "Job Title",
 "Business Street", "Business Street 2",
 "Business Street 3", "Business City",
 "Business State", "Business Postal Code",
 "Business Country",
 "Home Street", "Home Street 2",
 "Home Street 3", "Home City", "Home State",
 "Home Postal Code", "Home Country",
 "Other Street", "Other Street 2",
 "Other Street 3", "Other City",
 "Other State", "Other Postal Code",
 "Other Country",
 "Assistant's Phone",
 "Business Fax", "Business Phone",
 "Business Phone 2", "Callback",
 "Car Phone", "Company Main Phone",
 "Home Fax", "Home Phone", "Home Phone 2",
 "ISDN", "Mobile Phone",
 "Other Fax", "Other Phone", "Pager",
 "Primary Phone", "Radio Phone",
 "TTY/TDD Phone", "Telex", "Account",
 "Anniversary", "Assistant's Name",
 "Billing Information", "Birthday",
 "Categories", "Children",
 "Directory Server",
 "E-mail Address", "E-mail Display Name",
 "E-mail 2 Address", "E-mail 2 Display Name",
 "E-mail 3 Address", "E-mail 3 Display Name",
 "Gender", "Government ID Number", "Hobby",
 "Initials", "Internet Free Busy",
 "Keywords", "Language", "Location",
 "Manager's Name", "Mileage", "Notes",
 "Office Location",
 "Organizational ID Number", "PO Box",
 "Priority", "Private", "Profession",
 "Referred By", "Sensitivity", "Spouse",
 "User 1", "User 2", "User 3", "User 4",
 "Web Page"
);
```

It appears that "Kitchen Sink" is missing. We need to define the globals. We'll need an array in which to assemble the output line, and then the hash for the individual fields.

```
# we also need some global variables,
# including a hash for the fields
my @outputline;
my %fields;
```

As we explained earlier, the first line of the output is merely a reprise of the field names. This allows Outlook to know how to store the fields that are about to arrive on the following lines.

```
# begin by printing the title line
push(@outputline, @fields);
print "\"", join('","',@outputline), "\"\r\n";
```

Of course, we also need to read each address chunk as a separate record, by setting `$/`.

```
# read the input a paragraph at a time
$/ = "";
```

We loop through all the paragraphs in our input.

```
# now process each paragraph
while( <> ) {
 chomp;
```

We have occasional index markers in our address file of the form

```
- AAAA
```

so that we can navigate within the file a little more easily. Yes, this is idiosyncratic to our address book, so your mileage may vary. Of course, we don't need to include these lines in output, and we can just ignore them on input.

```
# remove index tabs
 /^- [A-Z]{4}/ && next;
```

In the main loop, we clear the `outputline` and `fields`, and send the record to a parsing routine.

```
# do each remaining record
 @outputline = ();
 %fields = ();
 convert_one($_);
```

We've already talked about how to do the output. We grab each item out of the hash, even the undefined ones.

```
# now cycle through the hash
# and assemble the output line
 foreach( @fields ) {
   my $x = defined($fields{$_}) ?
    $fields{$_} : "";
 $x =~ s/\s+$//;
 push(@outputline, $x);
 $fields{$_} = undef;
 }

 # print it out
 print "\"", join('","',@outputline), "\"\r\n";
}
```

So, that's it, we're …

What? We haven't shown you `convert_one`? Oh, all right.

The Service Routine

We start by splitting the input up into an array of individual lines.

```
sub convert_one {
 # split the input paragraph into
 # individual lines:  the grep elides
 # tags and alternate addresses
 my @addrlines = grep !/^[[!]/, split /\n/;
```

The first line is a name, and we need to split it up into first, last, and suffix. Our typical entry would be something like `Tom & Lydia Reilly`. We won't be pedantic and try to find the title (since we don't use them in our personal address book), nor isolate the middle name. For a couple, we'll cheat and treat both names as the given name.

```
# separate name into first and last
 $addrlines[0] =~
  s/\s([a-z\s]*\S+)\s*(,\s*\S+\s*)*$//;
 $fields{"First Name"} = $`;
 $fields{"Last Name"} = $1;
 if( length($2) ) {
  $fields{"Suffix"} = $2;
  $fields{"Suffix"} =~ s/,*\s*//;
 }
 $fields{"First Name"} =~ s/^(-)*//;
```

Two interesting points we should note here: First, the nomenclature (pardon the pun) is "first name" and "last name," not "given name" and "family name"–for our Asian correspondents, it's often the case that first name will be family name, as in Mao Tse-tung. Second, once we're done, we can toss the name line away:

```
# now we're done with the name line,
#  so get rid of it
 shift @addrlines;
```

We'll loop through the remaining lines in the address entry, populating the hash as we go. We keep track of the field we're populating in the variable `$fld`, and the last field we populated in `$lastfld`.

```
# now we proceed to go through
# the rest of the lines
 my $fld = "";
 my $lastfld;
 foreach (@addrlines) {
  $lastfld = $fld;
  $fld = "";
```

How do we proceed from there? Hint: it's a big case statement.

```
SWITCH: {
 # first handle the odd cases:
 $fld = "Spouse", s/^&\s*//,
  last SWITCH
   if( /^&/ );

 $fld = "Children", s/^\(&\s*(.*)\)/$1/,
  last SWITCH
   if( /^\(&/ );
```

Notice that this is a standard Perl `if` statement, which is one Perl answer to C's switch statement; `last SWITCH` skips to the end of the block named `SWITCH`. We use compound statements to process two common types of lines in our address book entries: spouse on a separate line (typically when they have different surnames), and children, as in the example

```
Mary Louise Copeland
& Doug Morgan
(& Graham & Amy)
```

Spouse is preceeded by an ampersand; children are preceeded by an ampersand but enclosed in parentheses. Again, your address book may be formatted slightly differently: adjust the software accordingly.

We can also recognize the patterns for phone numbers easily.

```
if( s/^#h:\s*([\d-]+).*/$1/ ) {
 $fld = "Home Phone";
 $fld = "Home Phone 2"
  if( defined($fields{$fld}) );
 last SWITCH;
 }
```

The Outlook format allows for multiple home phone numbers, so we step through them every time we find a line tagged #h. Notice that we skip any annotations, such as

```
#h: 619-483-2232 beach house
```

We actually keep the first and last home phone number for each entry. Are there better suggestions?

Business phone numbers are potentially odd, since they can have various extensions, so we need to do a bit of a hand-wave on the line once we have captured the number. We see if we have detected an extension, and append it to the resulting line.

```
if( s/^#w:\s*([\d-]+)(.*)/$1/ ) {
 $fld = "Business Phone";
 $fld = "Business Phone 2"
  if( defined($fields{$fld}) );
 # did we have an extension?
 (my $ext = $2) =~ s/.*(x\d+).*/ $1/;
 $_ .= $ext  if( length($ext) );
 last SWITCH;
 }
```

Fax numbers and cell phones get handled in a similar fashion. (Again, a design decision: how do we file the fax number we have? Home fax first? Business fax first? "Other" is sufficiently ambiguous.)

```
if( s/^#f:\s*([\d-]+).*/$1/ ) {
 $fld = "Home Fax";
 $fld = "Other Fax"
  if( defined($fields{$fld}) );
 last SWITCH;
 }

 if( s/^#m:\s*([\d-]+).*/$1/ ) {
  $fld = "Mobile Phone";
  last SWITCH;
 }
```

How many e-mail addresses do you have? How many do your friends have? (Notice that like all good programs, ours work even with `/dev/null` as input, since we're too busy at work to have any friends.) In our text address book e-mail addresses are preceeded with percent signs.

```
if( s/^%\s*([^\s]+).*/$1/ ) {
 $fld = "E-mail Address";
 $fld = "E-mail 2 Address"
  if( defined($fields{$fld}) );
 $fld = "E-mail 3 Address"
  if( defined($fields{$fld}) );
 last SWITCH;
 }
```

We also have category markers in our address book. In the Outlook form, these are separated by semicolons in the "Categories" field.

```
if( />/ ) {
 s/^>//;
 s/\s+>/;/g;
 s/$/;/;
 $fld = "Categories";
 last SWITCH;
 }
```

Foreign Legion

Now we need to be a little tricky. Occasionally, we will trip over a line that doesn't have a special tag. In the normal course of events, this would be a street address line, but it is all capitals. This is the convention in our text address book for a country name. If we've hit the country name, the immediately preceeding line must have been the city and province. We will take it as two special cases. The Canadian case is easier, because Canadian provinces have two-letter abbreviations, just like states in the U.S.

```
# we've got a Canadian address
# (why special?  there are US-style
# abbreviations for provinces)
 if( /^CANADA/ ) {
  # split the last line into city, etc,
  # clearing that line in the process
  $fields{$lastfld} =~
  s/([\w\s]+),*\s+
  ([A-Z][A-Z])\s+
  ([A-Z\d\s-]+)//x;
  $fields{"Home City"} = $1;
  $fields{"Home State"} = $2;
  $fields{"Home Postal Code"} = $3;
  $fld = "Home Country";
  last SWITCH;
  }
```

Because we know what the last field is–that is, where we left the city and province–we can recover them, clearing the last field in the process of parsing it. For other countries, the same trick applies, except that usually the province is spelled out.

```
# we've got another country....
```

```
# almost the same strategy
 if( /^[A-Z ]+$/ ) {
  # split the last line into city, etc,
  # clearing that line in the process
  $fields{$lastfld} =~
  s/([\w\s]+),\s+
  ([\w\s]+)\s+
  ([A-Z\d\s-]+)//x;
  $fields{"Home City"} = $1;
  $fields{"Home State"} = $2;
  $fields{"Home Postal Code"} = $3;
  $fld = "Home Country";
  last SWITCH;
  }
```

It might have been easier to recognize that we have an address with a country name at the outset and use a completely different case statement to parse the address. On the other hand, this doesn't add too much complexity, and prevents us from duplicating a lot of code.

In the U.S., the city-state line is frighteningly easy to recognize: some text, followed by a space and two letters, followed by five digits. Notice that we parse up the line by stripping off the state and zip code, leaving the city remaining on the line. We also recognize the line by finding a five digit zip code, but can parse and store a nine digit one. We can fool ourselves if we have friends living on Northeast twelve thousand and second Street, or `NE 12002nd St`, but even in Portland and Seattle, which appear to number their streets to infinity, this is unlikely.

```
# city, state zip
if( /\s[A-Z]{2}\s+\d{5}.*/ ) {
  s/\s([A-Z][A-Z])\s+([\d-]+).*//;
  $fields{"Home State"} = $1;
  $fields{"Home Postal Code"} = $2;
  $fld = "Home City";
  last SWITCH;
}
```

If we've fallen through to here, we've got a simple address line, like "123 Melody Lane." We populate them into the hash one after another.

```
# by default, we've got an address line
$fld = "Home Street";
$fld = "Home Street 2"
  if( defined($fields{$fld}) );
$fld = "Home Street 3"
  if( defined($fields{$fld}) );
 }
 $fields{$fld} = defined($fields{$fld}) ?
  $fields{$fld} . $_ : $_;
 }
}
```

That's it, the end of our service routine `convert_one`, and the end of our program.

Once we've run this program over our address book and regenerated it in CSV form, we can import it into Outlook by reversing the steps we took to discover the format. In other words, click on "File," "Import and Export," "Import from another file or program."

Summary

Why did we bother doing this? We've argued long and hard against storing vital data like our address books in proprietary programs, but given that we are using Outlook as a mail client, having our address book handy in the same application makes some amount of sense. (On the other hand, CSV is a sufficiently open format that there are a number of Perl modules for handling it.) We did this as our software exercise this month because it gave us an excuse to work with Perl hashes. We shared it because there are a number of similar conversions we may be interested in doing–importing addresses into the desktop application for the Palm handheld is remarkably similar, for example. We also realize that our address parsing is a little idiosyncratic. We could have made life a little easier if we'd used the `Lingua::EN::AddressParse` module from CPAN (see `http://www.cpan.org`), but we'll leave that project as an exercise for the reader.

Since we're talking about Windows and UNIX files again, we should mention something pointed out by astute reader Calvin Hilton: Calvin noticed that our Palm software from January and April's columns was failing on Windows. He tracked it down to his Perl implementation helpfully replacing `\n` with `\r\n` . Of course, this wreaks all sorts of havoc on any data structures that just happen to contain a counted string of 10 (that's hex `0xA`, also known as newline) bytes. Using `binmode` on the output filehandle tells Perl that the file in question isn't really a text file, and prevents that from happening.

We keep threatening to talk about test data. Indeed, we've been doing so since January. Next time, we may actually do it. Until then, happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at Minolta-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` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.