MAY 9, 2019

A macro processor scans input text for defined symbols — the macros — and replaces that text by other text, or possibly by other symbols. For instance, a macro processor can convert one language into another.

If you're a C programmer, you know cpp, the C preprocessor, a simple macro processor. m4 is a powerful macro processor that's been part of Unix for some 30 years, but it's almost unknown — except for special purposes, such as generating the sendmail.cf file. It's worth knowing because you can do things with m4 that are hard to do any other way.

The GNU version of m4 has some extensions from the original V7 version. (You'll see some of them.) As of this writing, the latest GNU version was 1.4.2, released in August 2004. Version 2.0 is under development.

While you won't become an m4 wizard in three pages (or in six, as the discussion of m4continues next month), but you can master the basics. So, let's dig in.

Simple Macro Processing

A simple way to do macro substitution is with tools like sed and cpp. For instance, the command sed's/XPRESIDENTX/President Bush/' reads lines of text, changing every occurrence of XPRESIDENTX to President Bush. sed can also test and branch, for some rudimentary decision-making.

As another example, here's a C program with a cpp macro named ABSDIFF() that accepts two arguments, a and b.

```
#define ABSDIFF(a, b)
((a)>(b) ? (a)-(b) : (b)-(a))
```

Given that definition, cpp will replace the code...

```
diff = ABSDIFF(v1, v2);
```

... with

```
diff = ((v1)>(v2) ? (v1)-(v2) : (v2)-(v1));
```

v1 replaces a everywhere, and v2 replace b. ABSDIFF() saves typing — and the chance for error.

Introducing m4

Unlike sed and other languages, m4 is designed specifically for macro processing. m4manipulates files, performs arithmetic, has functions for handling strings, and can do much more.

m4 copies its input (from files or standard input) to standard output. It checks each token (a name, a quoted string, or any single character that's not a part of either a name or a string) to see if it's the name of a macro. If so, the token is replaced by the macro's value, and then that text is pushed back onto the input to be rescanned. (If you're new to m4, this repeated scanning may surprise you, but it's one key to m4 s power.) Quoting text, like ` text', prevents expansion. (See the section on "Quoting.")

m4 comes with a number of predefined macros, or you can write your own macros by calling the define() function. A macro can have multiple arguments– up to 9 in original m4, and an unlimited number in GNU m4. Macro arguments are substituted before the resulting text is rescanned.

Here's a simple example (saved in a file named foo.m4):

```
one
define(`one', `ONE')dnl
one
define(`ONE', `two')dnl
one ONE oneONE
`one'
```

The file defines two macros named one and ONE. It also has four

lines of text. If you feed the file to m4 using m4 foo.m4, m4 produces:

```
one
ONE
two two oneONE
one
```

Here's what's happening:

*Line 1 of the input, which is simply the characters one and a newline, doesn't match any macro (so far), so it's copied to the output as-is.

*Line 2 defines a macro named one(). (The opening parenthesis before the arguments must come just after define with no whitespace between.) From this point on, any input

string one will be replaced with ONE. (The dnl is explained below.)

*Line 3, which is again the characters one and a newline, is affected by the just-defined macro one(). So, the text one is converted to the text ONE and a newline.

*Line 4 defines a new macro named ONE(). Macro names are case-sensitive.

*Line 5 has three space-separated tokens. The first two are one and ONE. The first is converted to ONE by the macro named one(), then both are converted to two by the macro named ONE(). Rescanning doesn't find any additional matches (there's no macro named two()), so the first two words are output as two two. The rest of line 5 (a space, oneONE, and a newline) doesn't match a macro so it's output as-is. In other words, a macro name is only recognized when it's surrounded by non-alphanumerics.

*Line 6 contains the text one inside a pair of quotes, then a newline. (As you've seen, the opening quote is a backquote or grave accent; the closing quote is a single quote or acute accent.) Quoted text doesn't match any macros, so it's output as-is: one. Next comes the final newline.

Input text is copied to the output as-is and that includes newlines. The built-in dnlfunction, which stands for "delete to new line," reads and discards all characters up to and including the next newline. (One of its uses is to put comments into an m4 file.) Without dnl, the newline after each of our calls to define would be output as-is. We could demonstrate that by editing foo.m4 to remove the two dnl s. But, to stretch things a bit, let's use sed to remove those two calls from the file and pipe the result to m4:

```
$ sed 's/dnl//' foo.m4 | m4
one

ONE

two two oneONE
one
```

If you compare this example to the previous one, you'll see that there

are two extra newlines at the places where dnl used to be.

Let's summarize. You've seen that input is read from the first character to the last. Macros affect input text only after they're defined. Input tokens are compared to macro names and, if they match, replaced by the macro's value. Any input modified by a macro is pushed back onto the input and is rescanned for possible modification. Other text (that isn't modified by a macro) is passed to the output as-is.

Quoting

Any text surrounded by `` ' '' (a grave accent and an acute accent) isn't expanded immediately. Whenever `m4` evaluates something, it strips off one level of quotes. When you define a macro, you'll often want to quote the arguments — but not always. `Listing One` has a demo. It uses `m4` interactively, typing text to its standard input.

```
Listing One: Quoting demonstration
$ m4
define(A, 100)dnl
define(B, A)dnl
define(C, `A')dnl
dumpdef(`A', `B', `C')dnl
A: 100
B: 100
C: A
dumpdef(A, B, C)dnl
stdin:5: m4: Undefined name 100
stdin:5: m4: Undefined name 100
stdin:5: m4: Undefined name 100
A B C
100 100 100
CTRL-D
$
```

The listing starts by defining three macros `A`, `B`, and `C`. `A` has the value `100`. So does `B`: because its argument `A` isn't quoted, `m4` replaces `A` with `100` before assigning that value to `B`. While defining `C`, though, quoting the argument means that its value becomes literal `A`.

You can see the values of macros by calling the built-in function `dumpdef` with the names of the macros. As expected, `A` and `B` have the value `100`, but `C` has `A`.

In the second call to `dumpdef`, the names are not quoted, so each name is expanded to `100`before `dumpdef` sees them. That explains the error messages, because there's no macro named `100`. In the same way, if we simply enter the macro names, the three tokens are scanned repeatedly, and they all end up as `100`.

You can change the quoting characters at any time by calling `changequote`. For instance, in text containing lots of quote marks, you could call `changequote({,})dnl` to change the quoting characters to curly braces. To restore the defaults, simply call `changequote` with no arguments.

In general, for safety, it's a good idea to quote all input text that isn't a macro call. This avoids `m4` interpreting a literal word as a call to a macro. Another way to avoid this problem is by using the GNU `m4` option `--prefix-builtins` or `-P`. It changes all built-in macro names to be prefixed by `m4_`. (The option doesn't affect user-defined macros.) So, under this option, you'd write `m4_dnl` and `m4_define` instead of `dnl` and `define`, respectively.

Keep quoting and rescanning in mind as you use `m4`. Not to be tedious, but remember that `m4 does` rescan its input. For some in-depth tips, see "Web Paging: Tips and Hints on `m4`Quoting" by R.K. Owen, Ph.D., at `http://owen.sj.ca.us/rkowen/howto/webpaging/m4tipsquote.html`.

Decisions and Math

`m4` can do arithmetic with its built-in functions `eval`, `incr`, and `decr`. `m4` doesn't support loops directly, but you can combine recursion and the decision macro `ifelse` to write loops.

Let's start with an example adapted from the file `/usr/share/doc/m4/examples/debug.m4`(on a Debian system). It defines the macro `countdown()`. Evaluating the macro with an argument of `5` — as in `countdown(5)` — outputs the text `5, 4, 3, 2, 1, 0, Liftoff!`.

```
$ cat countdown.m4
define(`countdown', `$1, ifelse(eval($1 > 0),
1, `countdown(decr($1))', `Liftoff!')')dnl
countdown(5)
$ m4 countdown.m4
5, 4, 3, 2, 1, 0, Liftoff!
```

The `countdown()` macro has a single argument. It's broken across two

`lines`. That's fine in `m4` because macro arguments are delimited by

parentheses which don't have to be on the same line. Here's the

argument without its surrounding quotes:

```
$1, ifelse(eval($1 > 0), 1,
`countdown(decr($1))', `Liftoff!')
)
```

`$1` expands to the macro's first argument. When `m4` evaluates that

`countdown` macro with an argument of `5`, the result is:

```
5, ifelse(eval(5 > 0), 1,
`countdown(decr(5))', `Liftoff!')
```

The leading "`5, `" is plain text that's output as-is as the first number in the countdown. The rest of the argument is a call to `ifelse`. `Ifelse` compares its first two arguments. If they're equal, the third argument is evaluated; otherwise, the (optional) fourth argument is evaluated.

Here, the first argument to `ifelse`, `eval(5> 0)`, evaluates as 1 (logical" true") if the test is true (if 5 is greater than 0). So the first two arguments are equal, and `m4` evaluates `countdown(decr(5))`. This starts the recursion by calling `countdown(4)`.

Once we reach the base condition of `countdown(0)`, the test `eval(0> 0)` fails and the `ifelse`call evaluates `` `Liftoff!' ``. (If recursion is new to you, you can read about it in books on computer science and programming techniques.)

Note that, with more than four arguments, `ifelse` can work like a `case` or `switch` in other languages. For instance, in `ifelse(a,b,c,d,e,f,g)`, if `a` matches `b`, then `c`; else if `d` matches `e`then `f`; else `g`.

The `m4 info` file shows more looping and decision techniques, including a macro named `forloop()` that implements a nestable for-loop.

This section showed some basic math operations. (The `info` file shows more.) You've seen that you can quote a single macro argument that contains a completely separate string (in this case, a string that prints a number, then runs `ifelse` to do some more work). This one-line example (broken onto two lines here) is a good hint of `m4`' s power. It's a mimimalist language, for sure, and you'd be right to complain about its tricky evaluation in a global environment, leaving lots of room for trouble if you aren't careful. But you might find this expressive little language to be challenging enough that it's addictive.

Building Web Pages

Let's wrap up this `m4` introduction with a typical use: feeding an input file to a set of macros to generate an output file. Here, the macro file `html.m4` defines three macros: `_startpage()`, `_ul()`, and `_endpage()`. (The names start with underscore characters to help prevent false matches with non-macro text. For instance, `_ul()` won't match the HTML tag <ul& gt;.) The `_startpage()` macro accepts one argument: the page title, which is also copied into a level-1 heading that appears at the start of the page. The `_ul()` macro makes an HTML unordered list. Its arguments (an unlimited number) become the list items. And `_endpage()` makes the closing HTML text, including a "last change" date taken from the Linux `date` utility.

`Listing Two` shows the input file, and `Listing Three` is the HTML output. The `m4` macros that do all the work are shown in `Listing Four`. (Both the input file and the macros are available online at

http://www.linux-mag.com/downloads/2005-02/power.)

```
Listing Two: webpage.m4h, an" unexpanded" web page
_startpage(`Sample List')
_ul(`First item', `Second item',
`Third item, longer than the first two')
_endpage
```

```
Listing Three: An m4- generated web page
$ m4 html.m4 webpage.m4h > list.html
$ cat list.html
<html>
<head>
<title>Sample List</title>
</head>
<body>
<h1>Sample List</h1>
<ul>
<li>First item</li>
<li>Second item</li>
<li>Third item, longer than the first two</li>
</ul>

<p>Last change: Fri Jan 14 15:32:06 MST 2005
</p>
</body>
</html>
```

In `Listing Four`, both `_startpage()` and `_endpage()` are straightforward. The `esyscmd`macro is one of the many `m4` macros we haven't covered — it runs a Linux command line, then uses the command's output as input to `m4`. The `_ul()` macro outputs opening and closing HTML `<ul>` tags, passing its arguments to the `_listitems()` macro via `$@`, which expands into the quoted list of arguments.

`_listitems()` is similar to the `countdown()` macro shown earlier: `_listitems()` makes a recursive loop. At the base condition (the end of recursion), when `$#` (the number of arguments) is 0, the empty third argument means that `ifelse` does nothing. Or, if there's one argument (`$#` is 1), `ifelse` simply outputs the last list item inside a pair of `<li>` tags. Otherwise, there's more than one argument, so the macro starts by outputting the first argument inside `<li>` tags, then calls `_listitems()` recursively to output the other list items. The argument to the recursive call is `shift($@)`. The `m4` `shift` macro returns its list of arguments without its first argument — which, here, is all of the arguments we haven't processed yet.

Notice the nested quoting: some of the arguments inside the (quoted) definition of `_listitems()` are quoted themselves. This delays interpretation until the macro is called. (`m4` tracing, which we'll cover next month, can help you see what's happening.)

```
Listing Four: html.m4, macros to generate an HTML page from Listing Two
define(`_startpage', `
<head>
<title>$1</title>
</head>
<body>
```

```
<h1>$1</h1>')dnl
dnl
define(`_endpage', `
<p>Last change: esyscmd(date)</p>
</body>
</html>')dnl
dnl
define(`_listitems', `ifelse($#, 0, ,
$#, 1, `<li>$1</li>',
`<li>$1</li>
_listitems(shift($@))')')dnl
define(`_ul', `<ul>
_listitems($@)
</ul>')dnl
```

This month, let's dig deeper into m4 and look at included files, diversions, frozen files, and debugging and tracing. Along the way, we'll see some of the rough edges of m4's minimalist language and explore workarounds. Before we start, though, here's a warning from the GNU m4 info page:

Some people[find] m4 to be fairly addictive. They first use m4 for simple problems, then take bigger and bigger challenges, learning how to write complex m4 sets of macros along the way. Once really addicted, users pursue writing of sophisticated m4 applications even to solve simple problems, devoting more time debugging their m4 scripts than doing real work. Beware that m4 may be dangerous for the health of compulsive programmers.

So take a deep breath… Good. Now let's dig in again!

Included Files

m4's built-in include() macro takes m4's input from a named file until the end of that file, when the previous input resumes. sinclude() works like include() except that it won't complain if the included file doesn't exist.

If an included file isn't in the current directory, GNU m4 searches the directories specified with the -Icommand-line option, followed by any directories in the colon-separated M4PATH environment variable.

Including files is often used to read in other m4 code, but can also be used to read plain text files. However, if you're reading plain text files, watch out for files that contain text that can confuse m4, such as quotes, commas, and parentheses. One way to work around that problem and read the contents of a random file is by using changequote() to temporarily override the quoting characters and also replacing include() with esyscmd(), which filters the file through a Linux utility like tr or sed.

Listing One has a contrived example that shows one way to read /etc/hosts, replacing parentheses with square brackets and commas with dashes.

Listing One: A filtering version of the m4 include() macro

```
% cat readfile.m4
dnl readfile: display file named on
dnl command line in -Dfile=
dnl converting () to [] and , to -
dnl
file `file on 'esyscmd(`hostname')
changequote({,})dnl
esyscmd({tr '(),' '[]-' <} file)dnl

That's all.
changequote
% cat /etc/hosts
127.0.0.1       localhost
216.123.4.56    foo.bar.com        foo

# Following lines are for `IPv6'
# (added automatically, we hope)
::1     ip6-localhost ip6-loopback
...

% m4 -Dfile=/etc/hosts readfile.m4
/etc/hosts file on foo

127.0.0.1       localhost
234.123.4.56    foo.bar.com        foo

# Following lines are for `IPv6'
# [added automatically- we hope]
::1     ip6-localhost ip6-loopback
...

That's all.
```

The option -D or --define lets you define a macro from the command line, before any input files are read. (Later, we'll see an cleaner way to read text from arbitrary files with GNU m4's undivert().)

Diversions: An Overview

Normally, all output is written directly to m4's standard output. But you can use the divert() macro to collect output into temporary storage places. This is one of m4's handiest features.

The argument to divert() is typically a stream number, the ID of the diversion that should get the output from now on.

*Diversion 0 is the default. Text written to diversion 0 goes to m4's standard output. If you've been diverting text to another stream, you can

call divert(0) or just divert to resume normal output.
*Text written to diversions 1, 2, and so on is held until m4 exits or until you call undivert(). (More about that in a moment.)
*Any text written to diversion -1 isn't emitted. Instead, diversion- 1 is "nowhere," like the Linux pseudo-file /dev/null. It's often used to comment code and to define macros without using the pesky dnl macro at the ends of lines.
*The divnum macro outputs the current diversion number.
Standard m4 supports diversions- 1 through 9, while GNU m4 can handle a essentially unlimited number of diversions. The latter version of m4 holds diverted text in memory until it runs out of memory and then moves the largest chunks of data to temporary files. (So, in theory, the number of diversions in GNU m4 is limited to the number of available file descriptors.)
All diversions 1, 2,…, are output at the end of processing in ascending order of stream number. To output diverted text sooner, simply call undivert() with the stream number. undivert() outputs text from a diversion and then empties the diversion. So, immediately calling undivert() again on the same diversion outputs nothing.
"Undiverted" text is output to the current diversion, which isn't always the standard output! You can use this to move text from one diversion to another.
Output from a diversion is not rescanned for macros.

Diverse Diversions

Before looking at the more-obvious uses of numbered diversions, let's look at a few surprising ones.
As was mentioned, diversion- 1 discards output. One of the most irritating types of m4 output is the newline chanacters after macro definitions. You can stop them by calling dnl after each define, but you can also stop them by defining macros after a call to divert(-1).
Here are two examples. This first example, nl, doesn't suppress the newline from define…

```
 `The result is:'
 define(`name', `value')
 name
```

… but the next example, nonl, does, by defining the macro inside a diversion:

```
 `The result is:'
 divert(-1)
 define(`name', `value')
 divert(0)dnl
 name
```

Let's compare the nl and nonl versions.

```
$ m4 nl
The result is:

value
$ m4 nonl
The result is:
value
```

The second divert() ends with dnl, which eats the the following newline. Adding the argument (0), which is actually the default, lets you write dnl without a space before it (which would otherwise be output). You can use divert``dnl instead, because an empty quoted string (``) is another way to separate the divert and dnl macro calls.
Of course, that trick is more reasonably done around a group of several define s. You can also write comments inside the same kind of diversion. This is an easy way to write blocks of comments without putting dnl at the start of each line. Just remember that macros are recognized inside the diversion (even though they don't make output). So, the following code increments i twice:

```
divert(-1)
Now we run define(`i', incr(i)):
define(`i', incr(i))
divert``dnl
```

dnl can start comments, and that works on even the oldest versions of m4. Generally, # is also a comment character. If you put it at the start of the comment above, as in #Now…, then i won't be incremented.
Before seeing the "obvious" uses of diversions, here's one last item from the bag of diversion tricks. GNU m4 lets you output a file's contents by calling undivert() instead of include(). The advantage is that, like undiverting a diversion, "undiverting" a file doesn't scan the file's contents for macros. This lets you avoid the really ugly workaround showed in Listing One. With GNU m4, you could have written simply:

```
undivert(`/etc/hosts')
```

Diversions as Diversions

The previous section showed some offbeat uses of divert(). Now let's see a more obvious use: splitting output into parts and reassembling those parts in a different order.
Listing Two, Three, and Four show a HTML generator that outputs the text of each top-level heading in two places: in a table of contents at the start of the web page, and again, later, in the body of the web page. The table of contents includes links to the actual headings later in the document, which will have an anchor (an HTML id).

Listing Two has the file, htmltext.m4, with the macro calls. Listing Three shows the HTML output from the macros (which omits the blank lines, because HTML parsers ignore them). Listing Four shows the macros, which call include() to bring in the htmltext.m4 file at the proper place. (Blank lines have been added to the macros to make the start and end of each macro more obvious.)

Listing Two: m4 macro calls: the htmltext.m4 file

```
 _h1(`First heading')
 _p(`The first paragraph.')
 _h1(`Second heading')
 _p(`The second paragraph.
 Yadda yadda yadda')
 _h1(`Third heading')
 _p(`The third paragraph.')
```

Listing Three: HTML output from the htmltext.m4 file

```
 <strong>Table of contents:</strong>
 <ol>
 <li><a href="#H1_1">First heading</a></li>
 <li><a href="#H1_2">Second heading</a></li>
 <li><a href="#H1_3">Third heading</a></li>
 </ol>
 <h1 id="H1_1">First heading</h1>
 <p>
 The first paragraph.
 </p>
 <h1 id="H1_2">Second heading</h1>
 <p>
 The second paragraph.
 Yadda yadda yadda
 </p>
 <h1 id="H1_3">Third heading</h1>
 <p>
 The third paragraph.
 </p>
```

Listing Four: The m4 code that makes the HTML in Listing Three

```
 define(`_h1count', 0)

 define(`_h1', `divert(9)
 define(`_h1count', incr(_h1count))
 <li><a href="`#'H1`_'_h1count">$1</a></li>
 divert(1)
 <h1 id="H1`_'_h1count">$1</h1>
```

```
divert')

define(`_p', `divert(1)
<p>
$1
</p>
divert')

include(`htmltext.m4')

<strong>Table of contents:</strong>
<ol>
undivert(9)
</ol>
undivert(1)
```

Let's look at the code in Listing Four.

*The _h1count macro sets the number used at the end of each HTML id. It's incremented by a define call inside the _h1 macro.

*The _h1 (heading level 1) macro starts by calling divert(9). The code used diversion 9 to store the HTML for the table of contents. After incrementing _h1count, the macro outputs a list item surrounded by <li> and tags. (The <ol> tags come later: when the code undiverts diversion 9.) Notice that the # is quoted to keep it from being treated as an m4 comment character. In the same way, the underscore is quoted (`_'), since it's used as part of the HTML id string (for instance, href="#H1_2").

A final call to divert switches output back to the normal diversion 0, which is m4's standard output.

*The _p (paragraph) macro is straightforward. It stores a pair of <p> tags with the first macro argument in-between in diversion 1.

*A call to include() brings in the file htmltext.m4 (Listing Two). This could have done this in several other ways, on the m4 command line, for instance.

*Finally, the call undivert(9) outputs the table of contents surrounded by a pair of ordered-list tags, followed by the headers and paragraphs from undivert(1).

This example shows one use of diversions: to output text in more than one way. Another common use – in sendmail, for instance – is gathering various text into "bunches" by its type or purpose.

Frozen Files

Large m4 applications can take time to load. GNU m4 supports a feature called frozen files that speeds up loading of common base files. For instance, if your common definitions are stored in a file named common.m4,you can pre-process that file to create a frozen file containing the m4 state information:

```
$ m4 -F common.m4f common.m4
```

Then, instead of using m4 common.m4, you use m4 -R common.m4f for faster access to the commondefinitions.

Frozen files work in a majority of cases, but there are gotchas. Be sure to read the m4 info file (type info m4) before you use this feature.

Debugging and Tracing

m4's recursion and quoting can make debugging a challenge. A thorough understanding of m4 helps, of course, and the techniques shown in the next section are worth studying. Here are some built-in debugging techniques:

*To see a macro definition, use dumpdef(), which was covered last month. dumpdef() shows you what's left after the initial layer of quoting is stripped off of a macro definition and any substitutions are made.

*The traceon() macro traces the execution of the macros you name as arguments, or, without a list of macros, it traces all macros. The trace output shows the depth of expansion, which is typically 1, but can be greater if a macro contains macro calls. Use traceoff to stop tracing.

*The debugmode() macro gives you a lot of control over debugging output. It accepts a string of flags, which are described in the m4 info file. You can also specify debugging flags on the command line with -d or --debug. These flags also affect the output of dumpdef() and traceon().

More about m4

Last month and this, you've seen some highlights of m4. If you have the GNU version of m4, its info page (info m4) is a good place to learn more.

R.K. Owen's quoting page (http://owen.sj.ca.us/rkowen/howto/webpaging/m4tipsquote.html) has lots of tips about — what else — quoting in m4. His site also has other m4 information and examples.

Ken Turner's technical report "CSM-126: Exploiting the m4 Macro Language," available from http://www.cs.stir.ac.uk/research/publications/techreps/previous.html, shows a number of m4 techniques.

*A Google search for m4 macro turns up a variety of references. To find example code, try a search with an m4-specific macro name, like m4 dnl and m4 divert -motorway. (In Google, the -motorway avoids matches of the British road named the M4. You can also add -sendmail to skip sendmail- specific information.)

*Mailing lists about m4 are at http://savannah.gnu.org/mail/?group= m4.

Happy m4 hacking!

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.

Get the Free Newsletter!

Subscribe to