m4 *Macro Processor* 6

Overview

m4 is a general-purpose macro processor that can be used to preprocess C and assembly language programs. Besides the straightforward replacement of one string of text by another, m4 lets you perform

- Integer arithmetic
- File inclusion
- Conditional macro expansion
- String and substring manipulation

You can use built-in macros to perform these tasks or you can define your own macros. Built-in and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process.

The basic operation of m4 is to read every alphanumeric token (string of letters and digits) and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is replaced onto the input to be rescanned. Macros can be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

Macro calls have the general form

```
name(arg1, arg2, ..., argn)
```

If a macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that appear in the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is returned to the input stream and rescanned. This is explained in the following paragraphs.

You invoke `m4` with a command of the form

```
$ m4 file file file
```

Each argument file is processed in order. If there are no arguments or if an argument is a hyphen, the standard input is read. If you are eventually going to compile the `m4` output, use a command like this:

```
$ m4 file1.m4 > file1.c
```

You can use the `-D` option to define a macro on the `m4` command line. Suppose you have two similar versions of a program. You might have a single `m4` input file capable of generating the two output files. That is, `file1.m4` could contain lines such as

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

Your `makefile` for the program might look like this:

```
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

You can use the `-U` option to "undefine" `VER`. If `file1.m4` contains

```
if(VER, 1, do_something)
if(VER, 2, do_something)
ifndef(VER, do_something)
```

then your `makefile` would contain

```
file0.0.c : file1.m4
m4 -UVER file1.m4 > file1.0.c
...
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

`m4` *Macros*

Defining Macros

The primary built-in `m4` macro is `define()`, which is used to define new macros. The following input

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. The defined string must be alphanumeric and must begin with a letter (an underscore is considered as a letter). The defining string is any text that contains balanced parentheses; it may stretch over multiple lines.

As a typical example

```
define(N, 100)
...
if (i > N)
```

defines `N` to be `100` and uses the *symbolic constant* `N` in a later `if` statement.

As noted, the left parenthesis must immediately follow the word `define` to signal that `define()` has arguments. If the macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. In the previous example, then, `N` is a macro with no arguments.

A macro name is only recognized as such if it appears surrounded by non-alphanumeric characters. In the following example the variable `NNN` is unrelated to the defined macro `N` even though the variable contains `N`s.

```
define(N, 100)
...
if (NNN > 100)
```

`m4` expands macro names into their defining text as soon as possible. So

```
define(N, 100)
define(M, N)
```

defines `M` to be `100` because the string `N` is immediately replaced by `100` as the arguments of `define(M, N)` are collected. To put this another way, if `N` is redefined, `M` keeps the value `100`.

There are two ways to avoid this result. The first, which is specific to the situation described here, is to change the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now `M` is defined to be the string `N`, so when the value of `M` is requested later, the result will always be the value of `N` at that time. The `M` will be replaced by `N` which will be replaced by 100.

Quoting

The more general solution is to delay the expansion of the arguments of `define()` by quoting them. Any text surrounded by left and right single quotes is not expanded immediately, but has the quotes stripped off as the arguments are collected. The value of the quoted string is the string stripped of the quotes.

Therefore, the following defines `M` as the string `N`, not `100`.

```
define(N, 100)
define(M, `N')
```

The general rule is that `m4` always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word `define` is to appear in the output, the word must be quoted in the input:

```
`define' = 1;
```

It's usually best to quote the arguments of a macro to ensure that what you are assigning to the macro name actually gets assigned. To redefine `N`, for example, you delay its evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

Otherwise the `N` in the second definition is immediately replaced by `100`.

```
define(N, 100)
...
define(N, 200)
```

The effect is the same as saying:

```
define(100, 200)
```

Note that this statement will be ignored by `m4` since only things that look like names can be defined.

If left and right single quotes are not convenient, the quote characters can be changed with the built-in macro `changequote()`:

```
changequote([, ])
```

In this example the macro makes the "quote" characters the left and right brackets instead of the left and right single quotes. The quote symbols can be up to five characters long. The original characters can be restored by using `changequote()` without arguments:

```
changequote
```

`undefine()` removes the definition of a macro or built-in macro:

```
undefine('N')
```

Here the macro removes the definition of `N`. Be sure to quote the argument to `undefine()`. Built-ins can be removed with `undefine()` as well:

```
undefine('define')
```

Note that once a built-in is removed or redefined, its original definition cannot be reused. Macros can be renamed with `defn()`. Suppose you want the built-in `define()` to be called `XYZ()`. You specify

```
define(XYZ, defn('define'))
undefine('define')
```

After this, `XYZ()` takes on the original meaning of `define()`. So

```
XYZ(A, 100)
```

defines `A` to be `100`.

The built-in `ifdef()` provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')
ifdef('u3b', 'define(wordsize,32)')
```

The `ifdef()` macro permits three arguments. If the first argument is defined, the value of `ifdef()` is the second argument. If the first argument is not defined, the value of `ifdef()` is the third argument:

```
ifdef('unix', on UNIX, not on UNIX)
```

If there is no third argument, the value of `ifdef()` is null.

Arguments

So far you have been given information about the simplest form of macro processing, that is, replacing one string with another (fixed) string. Macros can also be defined so that different invocations have different results. In the

replacement text for a macro (the second argument of its `define()`), any occurrence of `$`*n* is replaced by the *n*th argument when the macro is actually used. So the macro `bump()`, defined as

```
define(bump, $1 = $1 + 1)
```

is equivalent to `x = x + 1` for `bump(x)`.

A macro can have as many arguments as you want, but only the first nine are accessible individually, `$1` through `$9`. `$0` refers to the macro name itself. As noted, arguments that are not supplied are replaced by null strings, so a macro can be defined that concatenates its arguments:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

That is, `cat(x, y, z)` is equivalent to `xyz`. Arguments `$4` through `$9` are null since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained, so

```
define(a, b c)
```

defines `a` to be `b c`.

Arguments are separated by commas. A comma "protected" by parentheses does not terminate an argument. The following example has two arguments, `a` and `(b,c)`. You can specify a comma or parenthesis as an argument by quoting it. :

```
define(a, (b,c))
```

In the following example, `$(**` is replaced by a list of the arguments given to the macro in a subsequent invocation. The listed arguments are separated by commas. So

```
define(a, 1)
define(b, 2)
define(star, `$(**')
star(a, b)
```

gives the result `1,2`. So does

```
star(‘a’, ‘b’)
```

because `m4` strips the quotes from `a` and `b` as it collects the arguments of `star()`, then expands `a` and `b` when it evaluates `star()`.

`$@` is identical to `$(**` except that each argument in the subsequent invocation is quoted. That is,

```
define(a, 1)
define(b, 2)
define(at, ‘$@’)
at(‘a’, ‘b’)
```

gives the result `a,b` because the quotes are put back on the arguments when `at()` is evaluated.

`$#` is replaced by the number of arguments in the subsequent invocation. So

```
define(sharp, ‘$#’)
sharp(1, 2, 3)
```

gives the result `3`,

```
sharp()
```

gives the result `1`, and

```
sharp
```

gives the result `0`.

The built-in `shift()` returns all but its first argument. The other arguments are quoted and returnedto the input with commas in between. The simplest case

```
shift(1, 2, 3)
```

gives `2,3`. As with `$@`, you can delay the expansion of the arguments by quoting them, so

```
define(a, 100)
define(b, 200)
shift(‘a’, ‘b’)
```

gives the result `b` because the quotes are put back on the arguments when `shift()` is evaluated.

Arithmetic Built-Ins

`m4` provides three built-in macros for doing integer arithmetic. `incr()` increments its numeric argument by 1. `decr()` decrements by 1. So to handle the common programming situation in which a variable is to be defined as “one more than N” you would use

```
define(N, 100)
define(N1, ‘incr(N)’)
```

That is, `N1` is defined as one more than the current value of `N`.

The more general mechanism for arithmetic is a built-in macro called `eval()`, which is capable of arbitrary arithmetic on integers. Its operators, in decreasing order of precedence, are

```
+ - (unary)
(**(**
(** / %
+ -
== != < <= > >=
! ~
&
| ^
&&
||
```

Parentheses may be used to group operations where needed. All the operands of an expression given to `eval()` must ultimately be numeric. The numeric value of a true relation (like `1 > 0`) is 1, and false is 0. The precision in `eval()` is 32 bits.

As a simple example, you can define `M` to be `2(**(**N+1` with

```
define(M, `eval(2(**(**N+1)')
```

Then the sequence

```
define(N, 3)
M(2)
```

gives `9` as the result.

File Inclusion

A new file can be included in the input at any time with the built-in macro `include()`:

```
include(filename)
```

inserts the contents of *filename* in place of the macro and its argument. The value of `include()` (its replacement text) is the contents of the file. If needed, the contents can be captured in definitions and so on.

A fatal error occurs if the file named in `include()` cannot be accessed. To get some control over this situation, the alternate form `sinclude()` ("silent include") can be used. This built-in says nothing and continues if the file named cannot be accessed.

Diversions

`m4` output can be diverted to temporary files during processing, and the collected material can be output on command. `m4` maintains nine of these diversions, numbered 1 through 9. If the built-in macro `divert(`*n*`)` is used, all subsequent output is appended to a temporary file referred to as *n*. Diverting to this file is stopped by the `divert()` or `divert(0)` macros, which resume the normal output process.

Diverted text is normally placed at the end of processing in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in `undivert()` brings back all diversions in numerical

order; `undivert()` with arguments brings back the selected diversions in the order given. Undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of `undivert()` is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in `divnum()` returns the number of the currently active diversion. The current output stream is 0 during normal processing.

System Commands

Any program can be run by using the `syscmd()` built-in. The following example invokes the operating system `date` command. Normally, `syscmd()` would be used to create a file for a subsequent `include()`.

```
syscmd(date)
```

To make it easy to name files uniquely, the built-in `maketemp()` replaces a string of `XXXXX` in the argument with the process ID of the current process.

Conditional Testing

Arbitrary conditional testing is performed with the built-in `ifelse()`. In its simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, `ifelse()` returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called `compare()` can be defined as one that compares two strings and returns `yes` or `no`, if they are the same or different:

```
define(compare, `ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent evaluation of `ifelse()` from occurring too early. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

`ifelse()` can actually have any number of arguments and provides a limited form of branched decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*.

String Manipulation

The `len()` macro returns the length of the string (number of characters) in its argument. So

```
len(abcdef)
```

is `6`, and

```
len((a,b))
```

is `5`.

The `substr()` macro can be used to produce substrings of strings. So

```
substr(s, i, n)
```

returns the substring of *s* that starts at the *i*th position (origin 0) and is *n* characters long. If *n* is omitted, the rest of the string is returned. When you input the following example:

```
substr('now is the time',1)
```

it returns the following string:

```
ow is the time
```

If *i* or *n* are out of range, various things happen.

The `index(`*s1*`,` *s2*`)` macro returns the index (position) in *s1* where the string *s2* occurs, -1 if it does not occur. As with `substr()`, the origin for strings is 0.

`translit()` performs character transliteration [character substitution] and has the general form

```
translit(s, f, t)
```

that modifies *s* by replacing any character in *f* by the corresponding character in *t*.

Using the following input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*.

Therefore, the following would delete vowels from `s`:

```
translit(s, aeiou)
```

The macro `dnl()` deletes all characters that follow it, up to and including the next newline. It is useful mainly for removing empty lines that otherwise would clutter `m4` output. The following input, for example, results in a newline at the end of each line that is not part of the definition:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

So the new-line is copied into the output where it may not be wanted. When you add `dnl()` to each of these lines, the newlines will disappear. Another method of achieving the same result is to type:

```
divert(-1)
define(...)
...
divert
```

6

Printing

The built-in macro `errprint()` writes its arguments on the standard error file. An example would be

```
errprint(‘fatal error’)
```

`dumpdef()` is a debugging aid that dumps the current names and definitions of items specified as arguments. If no arguments are given, then all current names and definitions are printed.

Summary of Built-In `m4` *Macros*

Table 6-1 Summary of Built-In `m4` Macros

Built-In m4 Macros	**Description**
`changequote(L, R)`	Change left quote to L, right quote to R
`changecom`	Change left and right comment markers from the default # and newline
`decr`	Return the value of the argument decremented by 1
`define`(*name*, *stuff*)	Define *name* as *stuff*
`defn`(‘*name*’)	Return the quoted definition of the argument(s)
`divert`(*number*)	Divert output to stream *number*
`divnum`	Return number of currently active diversions
`dnl`	Delete up to and including newline
`dumpdef`(‘*name*’, ‘*name*’, . . .)	Dump specified definitions
`errprint`(*s*, *s*, . . .)	Write arguments *s* to standard error
`eval`(*numeric expression*)	Evaluate *numeric expression*
`ifdef`(‘*name*’, *true string*, *false string*)	Return *true string* if *name* is defined, *false* string if *name* is not defined
`ifelse`(*a*, *b*, *c*, *d*)	If *a* and *b* are equal, return *c*, else return *d*
`include`(*file*)	Include contents of *file*
`incr`(*number*)	Increment *number* by 1
`index`(*s1*, *s2*)	Return position in *s1* where *s2* occurs, or –1 if *s2* does not work
`len`(*string*)	Return length of *string*
`maketemp(. . .XXXXX. . .)`	Make a temporary file
`m4 exit`	Cause immediate exit from `m4`
`m4 wrap`	Argument 1 will be returned to the input stream at final EOF
`popdef`	Remove current definition of argument(s)
`pushdef`	Save any previous definition (similar to define)
`shift`	Return all but first argument(s)
`sinclude`(*file*)	Include contents of *file* — ignore and continue if *file* not found
`substr`(*string*, *position*, *number*)	Return substring of *string* starting at *position* and *number* characters long
`syscmd`(*command*)	Run *command* in the system
`sysval`	Return code from the last call to `syscmd`
`traceoff`	Turn off trace globally and for any macros specified

Table 6-1 Summary of Built-In `m4` Macros (Continued)

Built-In m4 Macros	Description
`traceon`	Turn on tracing for all macros, or with arguments, turn on tracing for named macros
`translit` (*string*, *from*, *to*)	Transliterate characters in *string* from the set specified by from to the set specified by *to*
`undefine` (‘*name*’)	Remove *name* from the list of definitions
`undivert` (*number,number,. . .*)	Append diversion *number* to the current diversion