In this chapter:

- *Naming Library Function Global Variables*
- *General Programming*
- *Datafile Management*
- *Processing Command-Line Options*
- *Reading the User Database*
- *Reading the Group Database*

12

A Library of awk Functions

The section "User-Defined Functions" in Chapter 8, *Functions*, describes how to write your own *awk* functions. Writing functions is important, because it allows you to encapsulate algorithms and program tasks in a single place. It simplifies programming, making program development more manageable, and making programs more readable.

One valuable way to learn a new programming language is to *read* programs in that language. To that end, this chapter and Chapter 13, *Practical awk Programs*, provide a good-sized body of code for you to read, and hopefully, to learn from.

This chapter presents a library of useful *awk* functions. Many of the sample programs presented later in this book use these functions. The functions are presented here in a progression from simple to complex.

The section "Extracting Programs from Texinfo Source Files" in Chapter 13 presents a program that you can use to extract the source code for these example library functions and programs from the Texinfo source for this book. (This has already been done as part of the *gawk* distribution.)

If you have written one or more useful, general-purpose *awk* functions and would like to contribute them to the author's collection of *awk* programs, see the section "How to Contribute" in the Preface for more information.

The programs in this chapter and in Chapter 13 freely use features that are *gawk*-specific. Rewriting these programs for different implementations of *awk* is pretty straightforward.

Diagnostic error messages are sent to */dev/stderr*. Use `| "cat 1>&2"`, instead of `> "/dev/stderr"` if your system does not have a */dev/stderr*, or if you cannot use *gawk*.

A number of programs use `nextfile` (see the section "Using gawk's nextfile Statement" in Chapter 6, *Patterns, Actions, and Variables*) to skip any remaining input in the input file. The section "Implementing nextfile as a Function" later in this chapter shows you how to write a function that does the same thing.

Finally, some of the programs choose to ignore upper- and lowercase distinctions in their input. They do so by assigning one to `IGNORECASE`. You can achieve almost the same effect* by adding the following rule to the beginning of the program:

```
# ignore case
{ $0 = tolower($0) }
```

Also, verify that all regexp and string constants used in comparisons use only lowercase letters.

Naming Library Function Global Variables

Due to the way the *awk* language evolved, variables are either *global* (usable by the entire program) or *local* (usable just by a specific function). There is no intermediate state analogous to `static` variables in C.

Library functions often need to have global variables that they can use to preserve state information between calls to the function—for example, `getopt`'s variable `_opti` (see the section "Processing Command-Line Options" later in this chapter). Such variables are called *private*, since the only functions that need to use them are the ones in the library.

When writing a library function, you should try to choose names for your private variables that will not conflict with any variables used by either another library function or a user's main program. For example, a name like `i` or `j` is not a good choice, because user programs often use variable names like these for their own purposes.

The example programs shown in this chapter all start the names of their private variables with an underscore (_). Users generally don't use leading underscores in their variable names, so this convention immediately decreases the chances that the variable name will be accidentally shared with the user's program.

---

* The effects are not identical. Output of the transformed record will be in all lowercase, while `IGNORECASE` preserves the original contents of the input record.

In addition, several of the library functions use a prefix that helps indicate what function or set of functions use the variables—for example, `_pw_byname` in the user database routines (see the section "Reading the User Database" later in this chapter). This convention is recommended, since it even further decreases the chance of inadvertent conflict among variable names. Note that this convention is used equally well for variable names and for private function names as well.*

As a final note on variable naming, if a function makes global variables available for use by a main program, it is a good convention to start that variable's name with a capital letter—for example, `getopt`'s `Opterr` and `Optind` variables (see the section "Processing Command-Line Options" later in this chapter). The leading capital letter indicates that it is global, while the fact that the variable name is not all capital letters indicates that the variable is not one of *awk*'s built-in variables, such as `FS`.

It is also important that *all* variables in library functions that do not need to save state are, in fact, declared local.† If this is not done, the variable could accidentally be used in the user's program, leading to bugs that are very difficult to track down:

```
function lib_func(x, y,    l1, l2)
{
    ...
    use variable some_var   # some_var should be local
    ...                     # but is not by oversight
}
```

A different convention, common in the Tcl community, is to use a single associative array to hold the values needed by the library function(s), or "package." This significantly decreases the number of actual global names in use. For example, the functions described in the section "Reading the User Database" later in this chapter might have used array elements `PW_data["inited"]`, `PW_data["total"]`, `PW_data["count"]`, and `PW_data["awklib"]`, instead of `_pw_inited`, `_pw_awklib`, `_pw_total`, and `_pw_count`.

The conventions presented in this section are exactly that: conventions. You are not required to write your programs this way—we merely recommend that you do so.

---

* While all the library routines could have been rewritten to use this convention, this was not done, in order to show how my own *awk* programming style has evolved and to provide some basis for this discussion.

† *gawk*'s *––dump–variables* command-line option is useful for verifying this.

General Programming

This section presents a number of functions that are of general programming use.

Implementing nextfile as a Function

The `nextfile` statement, presented in the section "Using gawk's nextfile Statement" in Chapter 6, is a *gawk*-specific extension—it is not available in most other implementations of *awk*. This section shows two versions of a `nextfile` function that you can use to simulate *gawk*'s `nextfile` statement if you cannot use *gawk*.

A first attempt at writing a `nextfile` function is as follows:

```
# nextfile --- skip remaining records in current file
# this should be read in before the "main" awk program

function nextfile()    { _abandon_ = FILENAME; next }
_abandon_ == FILENAME  { next }
```

Because it supplies a rule that must be executed first, this file should be included before the main program. This rule compares the current datafile's name (which is always in the `FILENAME` variable) to a private variable named `_abandon_`. If the filename matches, then the action part of the rule executes a `next` statement to go on to the next record. (The use of _ in the variable name is a convention. It is discussed more fully in the section "Naming Library Function Global Variables" earlier in this chapter.)

The use of the `next` statement effectively creates a loop that reads all the records from the current datafile. The end of the file is eventually reached and a new datafile is opened, changing the value of `FILENAME`. Once this happens, the comparison of `_abandon_` to `FILENAME` fails, and execution continues with the first rule of the "real" program.

The `nextfile` function itself simply sets the value of `_abandon_` and then executes a `next` statement to start the loop.

This initial version has a subtle problem. If the same datafile is listed *twice* on the command line, one right after the other or even with just a variable assignment between them, this code skips right through the file a second time, even though it should stop when it gets to the end of the first occurrence. A second version of `nextfile` that remedies this problem is shown here:

```
# nextfile --- skip remaining records in current file
# correctly handle successive occurrences of the same file
# this should be read in before the "main" awk program

function nextfile()   { _abandon_ = FILENAME; next }
```

```
_abandon_ == FILENAME {
    if (FNR == 1)
        _abandon_ = ""
    else
        next
}
```

The `nextfile` function has not changed. It makes `_abandon_` equal to the current filename and then executes a `next` statement. The `next` statement reads the next record and increments `FNR` so that `FNR` is guaranteed to have a value of at least two. However, if `nextfile` is called for the last record in the file, then *awk* closes the current datafile and moves on to the next one. Upon doing so, `FILENAME` is set to the name of the new file and `FNR` is reset to one. If this next file is the same as the previous one, `_abandon_` is still equal to `FILENAME`. However, `FNR` is equal to one, telling us that this is a new occurrence of the file and not the one we were reading when the `nextfile` function was executed. In that case, `_abandon_` is reset to the empty string, so that further executions of this rule fail (until the next time that `nextfile` is called).

If `FNR` is not one, then we are still in the original datafile and the program executes a `next` statement to skip through it.

An important question to ask at this point is: given that the functionality of `nextfile` can be provided with a library file, why is it built into *gawk*? Adding features for little reason leads to larger, slower programs that are harder to maintain. The answer is that building `nextfile` into *gawk* provides significant gains in efficiency. If the `nextfile` function is executed at the beginning of a large datafile, *awk* still has to scan the entire file, splitting it up into records, just to skip over it. The built-in `nextfile` can simply close the file immediately and proceed to the next one, which saves a lot of time. This is particularly important in *awk*, because *awk* programs are generally I/O-bound (i.e., they spend most of their time doing input and output, instead of performing computations).

Assertions

When writing large programs, it is often useful to know that a condition or set of conditions is true. Before proceeding with a particular computation, you make a statement about what you believe to be the case. Such a statement is known as an *assertion*. The C language provides an `<assert.h>` header file and corresponding `assert` macro that the programmer can use to make assertions. If an assertion fails, the `assert` macro arranges to print a diagnostic message describing the condition that should have been true but was not, and then it kills the program. In C, using `assert` looks this:

```
#include <assert.h>

int myfunc(int a, double b)
{
    assert(a <= 5 && b >= 17.1);
    ...
}
```

If the assertion fails, the program prints a message similar to this:

```
prog.c:5: assertion failed: a <= 5 && b >= 17.1
```

The C language makes it possible to turn the condition into a string for use in printing the diagnostic message. This is not possible in *awk*, so this `assert` function also requires a string version of the condition that is being tested. Following is the function:

```
# assert --- assert that a condition is true. Otherwise exit.

function assert(condition, string)
{
    if (! condition) {
        printf("%s:%d: assertion failed: %s\n",
            FILENAME, FNR, string) > "/dev/stderr"
        _assert_exit = 1
        exit 1
    }
}

END {
    if (_assert_exit)
        exit 1
}
```

The `assert` function tests the `condition` parameter. If it is false, it prints a message to standard error, using the `string` parameter to describe the failed condition. It then sets the variable `_assert_exit` to one and executes the `exit` statement. The `exit` statement jumps to the `END` rule. If the `END` rules finds `_assert_exit` to be true, it exits immediately.

The purpose of the test in the `END` rule is to keep any other `END` rules from running. When an assertion fails, the program should exit immediately. If no assertions fail, then `_assert_exit` is still false when the `END` rule is run normally, and the rest of the program's `END` rules execute. For all of this to work correctly, *assert.awk* must be the first source file read by *awk*. The function can be used in a program in the following way:

```
function myfunc(a, b)
{
    assert(a <= 5 && b >= 17.1, "a <= 5 && b >= 17.1")
    ...
}
```

If the assertion fails, you see a message similar to the following:

```
mydata:1357: assertion failed: a <= 5 && b >= 17.1
```

There is a small problem with this version of `assert`. An `END` rule is automatically added to the program calling `assert`. Normally, if a program consists of just a `BEGIN` rule, the input files and/or standard input are not read. However, now that the program has an `END` rule, *awk* attempts to read the input datafiles or standard input (see the section "Startup and cleanup actions" in Chapter 6), most likely causing the program to hang as it waits for input.

There is a simple workaround to this: make sure the `BEGIN` rule always ends with an `exit` statement.

Rounding Numbers

The way `printf` and `sprintf` (see the section "Using printf Statements for Fancier Printing" in Chapter 4, *Printing Output*) perform rounding often depends upon the system's C `sprintf` subroutine. On many machines, `sprintf` rounding is "unbiased," which means it doesn't always round a trailing `.5` up, contrary to naive expectations. In unbiased rounding, `.5` rounds to even, rather than always up, so 1.5 rounds to 2 but 4.5 rounds to 4. This means that if you are using a format that does rounding (e.g., `"%.0f"`), you should check what your system does. The following function does traditional rounding; it might be useful if your *awk*'s `printf` does unbiased rounding:

```
# round.awk --- do normal rounding

function round(x,   ival, aval, fraction)
{
    ival = int(x)    # integer part, int() truncates

    # see if fractional part
    if (ival == x)   # no fraction
        return x
```

```
    if (x < 0) {
        aval = -x     # absolute value
        ival = int(aval)
        fraction = aval - ival
        if (fraction >= .5)
            return int(x) - 1   # -2.5 --> -3
        else
            return int(x)       # -2.3 --> -2
    } else {
        fraction = x - ival
        if (fraction >= .5)
            return ival + 1
        else
            return ival
    }
}

# test harness
{ print $0, round($0) }
```

The Cliff Random Number Generator

The Cliff random number generator* is a very simple random number generator that "passes the noise sphere test for randomness by showing no structure." It is easily programmed in less than 10 lines of *awk* code:

```
# cliff_rand.awk --- generate Cliff random numbers

BEGIN { _cliff_seed = 0.1 }

function cliff_rand()
{
    _cliff_seed = (100 * log(_cliff_seed)) % 1
    if (_cliff_seed < 0)
        _cliff_seed = - _cliff_seed
    return _cliff_seed
}
```

This algorithm requires an initial "seed" of 0.1. Each new value uses the current seed as input for the calculation. If the built-in `rand` function (see the section "Numeric Functions" in Chapter 8) isn't random enough, you might try using this function instead.

Translating Between Characters and Numbers

One commercial implementation of *awk* supplies a built-in function, `ord`, which takes a character and returns the numeric value for that character in the machine's character set. If the string passed to `ord` has more than one character, only the first one is used.

---

* *http://mathworld.wolfram.com/CliffRandomNumberGenerator.html.*

The inverse of this function is `chr` (from the function of the same name in Pascal), which takes a number and returns the corresponding character. Both functions are written very nicely in *awk*; there is no real reason to build them into the *awk* interpreter:

```
# ord.awk --- do ord and chr

# Global identifiers:
#    _ord_:        numerical values indexed by characters
#    _ord_init:    function to initialize _ord_

BEGIN    { _ord_init() }

function _ord_init(    low, high, i, t)
{
    low = sprintf("%c", 7) # BEL is ascii 7
    if (low == "\a") {    # regular ascii
        low = 0
        high = 127
    } else if (sprintf("%c", 128 + 7) == "\a") {
        # ascii, mark parity
        low = 128
        high = 255
    } else {        # ebcdic(!)
        low = 0
        high = 255
    }

    for (i = low; i <= high; i++) {
        t = sprintf("%c", i)
        _ord_[t] = i
    }
}
```

Some explanation of the numbers used by `chr` is worthwhile. The most prominent character set in use today is ASCII. Although an 8-bit byte can hold 256 distinct values (from 0 to 255), ASCII only defines characters that use the values from 0 to 127.* In the now distant past, at least one minicomputer manufacturer used ASCII, but with mark parity, meaning that the leftmost bit in the byte is always 1. This means that on those systems, characters have numeric values from 128 to 255. Finally, large mainframe systems use the EBCDIC character set, which uses all 256 values. While there are other character sets in use on some older systems, they are not really worth worrying about:

* ASCII has been extended in many countries to use the values from 128 to 255 for country-specific characters. If your system uses these extensions, you can simplify `_ord_init` to simply loop from 0 to 255.

```
function ord(str,    c)
{
    # only first character is of interest
    c = substr(str, 1, 1)
    return _ord_[c]
}

function chr(c)
{
    # force c to be numeric by adding 0
    return sprintf("%c", c + 0)
}

#### test code ####
# BEGIN    \
# {
#    for (;;) {
#        printf("enter a character: ")
#        if (getline var <= 0)
#            break
#        printf("ord(%s) = %d\n", var, ord(var))
#    }
# }
```

An obvious improvement to these functions is to move the code for the `_ord_init` function into the body of the `BEGIN` rule. It was written this way initially for ease of development. There is a "test program" in a `BEGIN` rule, to test the function. It is commented out for production use.

Merging an Array into a String

When doing string processing, it is often useful to be able to join all the strings in an array into one long string. The following function, `join`, accomplishes this task. It is used later in several of the application programs (see Chapter 13).

Good function design is important; this function needs to be general but it should also have a reasonable default behavior. It is called with an array as well as the beginning and ending indices of the elements in the array to be merged. This assumes that the array indices are numeric—a reasonable assumption since the array was likely created with `split` (see the section "String-Manipulation Functions" in Chapter 8):

```
# join.awk --- join an array into a string

function join(array, start, end, sep,    result, i)
{
    if (sep == "")
        sep = " "
    else if (sep == SUBSEP) # magic value
        sep = ""
```

```
    result = array[start]
    for (i = start + 1; i <= end; i++)
        result = result sep array[i]
    return result
}
```

An optional additional argument is the separator to use when joining the strings back together. If the caller supplies a nonempty value, `join` uses it; if it is not supplied, it has a null value. In this case, `join` uses a single blank as a default separator for the strings. If the value is equal to `SUBSEP`, then `join` joins the strings with no separator between them. `SUBSEP` serves as a "magic" value to indicate that there should be no separation between the component strings.*

Managing the Time of Day

The `systime` and `strftime` functions described in the section "Using gawk's Timestamp Functions" in Chapter 8 provide the minimum functionality necessary for dealing with the time of day in human readable form. While `strftime` is extensive, the control formats are not necessarily easy to remember or intuitively obvious when reading a program.

The following function, `gettimeofday`, populates a user-supplied array with preformatted time information. It returns a string with the current time formatted in the same way as the *date* utility:

```
# gettimeofday.awk --- get the time of day in a usable format

# Returns a string in the format of output of date(1)
# Populates the array argument time with individual values:
#    time["second"]       -- seconds (0 - 59)
#    time["minute"]       -- minutes (0 - 59)
#    time["hour"]         -- hours (0 - 23)
#    time["althour"]      -- hours (0 - 12)
#    time["monthday"]     -- day of month (1 - 31)
#    time["month"]        -- month of year (1 - 12)
#    time["monthname"]    -- name of the month
#    time["shortmonth"]   -- short name of the month
#    time["year"]         -- year modulo 100 (0 - 99)
#    time["fullyear"]     -- full year
#    time["weekday"]      -- day of week (Sunday = 0)
#    time["altweekday"]   -- day of week (Monday = 0)
#    time["dayname"]      -- name of weekday
#    time["shortdayname"] -- short name of weekday
#    time["yearday"]      -- day of year (0 - 365)
#    time["timezone"]     -- abbreviation of timezone name
#    time["ampm"]         -- AM or PM designation
#    time["weeknum"]      -- week number, Sunday first day
#    time["altweeknum"]   -- week number, Monday first day
```

* It would be nice if *awk* had an assignment operator for concatenation. The lack of an explicit operator for concatenation makes string operations more difficult than they really need to be.

```
function gettimeofday(time,    ret, now, i)
{
    # get time once, avoids unnecessary system calls
    now = systime()

    # return date(1)-style output
    ret = strftime("%a %b %d %H:%M:%S %Z %Y", now)

    # clear out target array
    delete time

    # fill in values, force numeric values to be
    # numeric by adding 0
    time["second"]       = strftime("%S", now) + 0
    time["minute"]       = strftime("%M", now) + 0
    time["hour"]         = strftime("%H", now) + 0
    time["althour"]      = strftime("%I", now) + 0
    time["monthday"]     = strftime("%d", now) + 0
    time["month"]        = strftime("%m", now) + 0
    time["monthname"]    = strftime("%B", now)
    time["shortmonth"]   = strftime("%b", now)
    time["year"]         = strftime("%y", now) + 0
    time["fullyear"]     = strftime("%Y", now) + 0
    time["weekday"]      = strftime("%w", now) + 0
    time["altweekday"]   = strftime("%u", now) + 0
    time["dayname"]      = strftime("%A", now)
    time["shortdayname"] = strftime("%a", now)
    time["yearday"]      = strftime("%j", now) + 0
    time["timezone"]     = strftime("%Z", now)
    time["ampm"]         = strftime("%p", now)
    time["weeknum"]      = strftime("%U", now) + 0
    time["altweeknum"]   = strftime("%W", now) + 0

    return ret
}
```

The string indices are easier to use and read than the various formats required by `strftime`. The `alarm` program presented in the section "An Alarm Clock Program" in Chapter 13 uses this function. A more general design for the `gettimeofday` function would have allowed the user to supply an optional timestamp value to use instead of the current time.

Datafile Management

This section presents functions that are useful for managing command-line datafiles.

Noting Datafile Boundaries

The `BEGIN` and `END` rules are each executed exactly once at the beginning and end of your *awk* program, respectively (see the section "The BEGIN and END Special Patterns" in Chapter 6). We (the *gawk* authors) once had a user who mistakenly thought that the `BEGIN` rule is executed at the beginning of each datafile and the `END` rule is executed at the end of each datafile. When informed that this was not the case, the user requested that we add new special patterns to *gawk*, named `BEGIN_FILE` and `END_FILE`, that would have the desired behavior. He even supplied us the code to do so.

Adding these special patterns to *gawk* wasn't necessary; the job can be done cleanly in *awk* itself, as illustrated by the following library program. It arranges to call two user-supplied functions, `beginfile` and `endfile`, at the beginning and end of each datafile. Besides solving the problem in only nine (!) lines of code, it does so *portably*; this works with any implementation of *awk*:

```
# transfile.awk
#
# Give the user a hook for filename transitions
#
# The user must supply functions beginfile() and endfile()
# that each take the name of the file being started or
# finished, respectively.

FILENAME != _oldfilename \
{
    if (_oldfilename != "")
        endfile(_oldfilename)
    _oldfilename = FILENAME
    beginfile(FILENAME)
}

END   { endfile(FILENAME) }
```

This file must be loaded before the user's "main" program, so that the rule it supplies is executed first.

This rule relies on *awk*'s `FILENAME` variable that automatically changes for each new datafile. The current filename is saved in a private variable, `_oldfilename`. If `FILENAME` does not equal `_oldfilename`, then a new datafile is being processed and it is necessary to call `endfile` for the old file. Because `endfile` should only be called if a file has been processed, the program first checks to make sure that `_oldfilename` is not the null string. The program then assigns the current filename to `_oldfilename` and calls `beginfile` for the file. Because, like all *awk* variables, `_oldfilename` is initialized to the null string, this rule executes correctly even for the first datafile.

The program also supplies an `END` rule to do the final processing for the last file. Because this `END` rule comes before any `END` rules supplied in the "main" program, `endfile` is called first. Once again the value of multiple `BEGIN` and `END` rules should be clear.

This version has same problem as the first version of `nextfile` (see the section "Implementing nextfile as a Function" earlier in this chapter). If the same datafile occurs twice in a row on the command line, then `beginfile` and `endfile` are not executed at the end of the first pass and at the beginning of the second pass. The following version solves the problem:

```
# ftrans.awk --- handle data file transitions
#
# user supplies beginfile() and endfile() functions

FNR == 1 {
    if (_filename_ != "")
        endfile(_filename_)
    _filename_ = FILENAME
    beginfile(FILENAME)
}

END  { endfile(_filename_) }
```

The section "Counting Things" in Chapter 13 shows how this library function can be used and how it simplifies writing the main program.

Rereading the Current File

Another request for a new built-in function was for a `rewind` function that would make it possible to reread the current file. The requesting user didn't want to have to use `getline` (see the section "Explicit Input with getline" in Chapter 3, *Reading Input Files*) inside a loop.

However, as long as you are not in the `END` rule, it is quite easy to arrange to immediately close the current input file and then start over with it from the top. For lack of a better name, we'll call it `rewind`:

```
# rewind.awk --- rewind the current file and start over

function rewind(    i)
{
    # shift remaining arguments up
    for (i = ARGC; i > ARGIND; i--)
        ARGV[i] = ARGV[i-1]

    # make sure gawk knows to keep going
    ARGC++
```

```
        # make current file next to get done
        ARGV[ARGIND+1] = FILENAME

        # do it
        nextfile
    }
```

This code relies on the `ARGIND` variable (see the section "Built-in Variables That Convey Information" in Chapter 6), which is specific to *gawk*. If you are not using *gawk*, you can use ideas presented in the section "Noting Datafile Boundaries" earlier in this chapter to either update `ARGIND` on your own or modify this code as appropriate.

The `rewind` function also relies on the `nextfile` keyword (see the section "Using gawk's nextfile Statement" in Chapter 6). See the section "Implementing nextfile as a Function" earlier in this chapter for a function version of `nextfile`.

Checking for Readable Datafiles

Normally, if you give *awk* a datafile that isn't readable, it stops with a fatal error. There are times when you might want to just ignore such files and keep going. You can do this by prepending the following program to your *awk* program:

```
# readable.awk --- library file to skip over unreadable files

BEGIN {
    for (i = 1; i < ARGC; i++) {
        if (ARGV[i] ~ /^[A-Za-z_][A-Za-z0-9_]*=.*/ \
            || ARGV[i] == "-")
            continue    # assignment or standard input
        else if ((getline junk < ARGV[i]) < 0) # unreadable
            delete ARGV[i]
        else
            close(ARGV[i])
    }
}
```

In *gawk*, the `getline` won't be fatal (unless *--posix* is in force). Removing the element from `ARGV` with `delete` skips the file (since it's no longer in the list).

Treating Assignments as Filenames

Occasionally, you might not want *awk* to process command-line variable assignments (see the section "Assigning Variables on the Command Line" in Chapter 5, *Expressions*). In particular, if you have filenames that contain an `=` character, *awk* treats the filename as an assignment, and does not process it.

Some users have suggested an additional command-line option for *gawk* to disable command-line assignments. However, some simple programming with a library file does the trick:

```
# noassign.awk --- library file to avoid the need for a
# special option that disables command-line assignments

function disable_assigns(argc, argv,    i)
{
    for (i = 1; i < argc; i++)
        if (argv[i] ~ /^[A-Za-z_][A-Za-z_0-9]*=.*/)
            argv[i] = ("./" argv[i])
}

BEGIN {
    if (No_command_assign)
        disable_assigns(ARGC, ARGV)
}
```

You then run your program this way:

```
awk -v No_command_assign=1 -f noassign.awk -f yourprog.awk *
```

The function works by looping through the arguments. It prepends `./` to any argument that matches the form of a variable assignment, turning that argument into a filename.

The use of `No_command_assign` allows you to disable command-line assignments at invocation time, by giving the variable a true value. When not set, it is initially zero (i.e., false), so the command-line arguments are left alone.

Processing Command-Line Options

Most utilities on POSIX compatible systems take options, or "switches," on the command line that can be used to change the way a program behaves. *awk* is an example of such a program (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*). Often, options take *arguments*; i.e., data that the program needs to correctly obey the command-line option. For example, *awk*'s *–F* option requires a string to use as the field separator. The first occurrence on the command line of either –– or a string that does not begin with – ends the options.

Modern Unix systems provide a C function named `getopt` for processing command-line arguments. The programmer provides a string describing the one-letter options. If an option requires an argument, it is followed in the string with a colon. `getopt` is also passed the count and values of the command-line arguments and is called in a loop. `getopt` processes the command-line arguments for option

letters. Each time around the loop, it returns a single character representing the next option letter that it finds, or `?` if it finds an invalid option. When it returns −1, there are no options left on the command line.

When using `getopt`, options that do not take arguments can be grouped together. Furthermore, options that take arguments require that the argument is present. The argument can immediately follow the option letter, or it can be a separate command-line argument.

Given a hypothetical program that takes three command-line options, *−a*, *−b*, and *−c*, where *−b* requires an argument, all of the following are valid ways of invoking the program:

```
prog -a -b foo -c data1 data2 data3
prog -ac -bfoo -- data1 data2 data3
prog -acbfoo data1 data2 data3
```

Notice that when the argument is grouped with its option, the rest of the argument is considered to be the option's argument. In this example, *−acbfoo* indicates that all of the *−a*, *−b*, and *−c* options were supplied, and that `foo` is the argument to the *−b* option.

`getopt` provides four external variables that the programmer can use:

`optind`
: The index in the argument value array (`argv`) in which the first nonoption command-line argument can be found.

`optarg`
: The string value of the argument to an option.

`opterr`
: Usually `getopt` prints an error message when it finds an invalid option. Setting `opterr` to zero disables this feature. (An application might want to print its own error message.)

`optopt`
: The letter representing the command-line option.

The following C fragment shows how `getopt` might process command-line arguments for *awk*:

```
int
main(int argc, char *argv[])
{
    ...
    /* print our own message */
    opterr = 0;
    while ((c = getopt(argc, argv, "v:f:F:W:")) != -1) {
        switch (c) {
```

```
            case 'f':    /* file */
                ...
                break;
            case 'F':    /* field separator */
                ...
                break;
            case 'v':    /* variable assignment */
                ...
                break;
            case 'W':    /* extension */
                ...
                break;
            case '?':
            default:
                usage();
                break;
            }
        }
        ...
    }
```

As a side point, *gawk* actually uses the GNU `getopt_long` function to process both normal and GNU-style long options (see the section "Command-Line Options" in Chapter 11).

The abstraction provided by `getopt` is very useful and is quite handy in *awk* programs as well. Following is an *awk* version of `getopt`. This function highlights one of the greatest weaknesses in *awk*, which is that it is very poor at manipulating single characters. Repeated calls to `substr` are necessary for accessing individual characters (see the section "String-Manipulation Functions" in Chapter 8).*

The discussion that follows walks through the code a bit at a time:

```
# getopt.awk --- do C library getopt(3) function in awk

# External variables:
#    Optind -- index in ARGV of first nonoption argument
#    Optarg -- string value of argument to current option
#    Opterr -- if nonzero, print our own diagnostic
#    Optopt -- current option letter

# Returns:
#    -1     at end of options
#    ?      for unrecognized option
#    <c>    a character representing the current option

# Private Data:
#    _opti  -- index in multi-flag option, e.g., -abc
```

* This function was written before *gawk* acquired the ability to split strings into single characters using `""` as the separator. We have left it alone, since using `substr` is more portable.

The function starts out with a list of the global variables it uses, what the return values are, what they mean, and any global variables that are "private" to this library function. Such documentation is essential for any program, and particularly for library functions.

The `getopt` function first checks that it was indeed called with a string of options (the `options` parameter). If `options` has a zero length, `getopt` immediately returns −1:

```
function getopt(argc, argv, options,    thisopt, i)
{
    if (length(options) == 0)    # no options given
        return -1

    if (argv[Optind] == "--") {  # all done
        Optind++
        _opti = 0
        return -1
    } else if (argv[Optind] !~ /^-[^: \t\n\f\r\v\b]/) {
        _opti = 0
        return -1
    }
```

The next thing to check for is the end of the options. A −− ends the command-line options, as does any command-line argument that does not begin with a `-`. `Optind` is used to step through the array of command-line arguments; it retains its value across calls to `getopt`, because it is a global variable.

The regular expression that is used, `/^-[^: \t\n\f\r\v\b]/`, is perhaps a bit of overkill; it checks for a `-` followed by anything that is not whitespace and not a colon. If the current command-line argument does not match this pattern, it is not an option, and it ends option processing:

```
    if (_opti == 0)
        _opti = 2
    thisopt = substr(argv[Optind], _opti, 1)
    Optopt = thisopt
    i = index(options, thisopt)
    if (i == 0) {
        if (Opterr)
            printf("%c -- invalid option\n",
                thisopt) > "/dev/stderr"
        if (_opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return "?"
    }
```

The _opti variable tracks the position in the current command-line argument (argv[Optind]). If multiple options are grouped together with one - (e.g., *–abx*), it is necessary to return them to the user one at a time.

If _opti is equal to zero, it is set to two, which is the index in the string of the next character to look at (we skip the -, which is at position one). The variable thisopt holds the character, obtained with substr. It is saved in Optopt for the main program to use.

If thisopt is not in the options string, then it is an invalid option. If Opterr is nonzero, getopt prints an error message on the standard error that is similar to the message from the C version of getopt.

Because the option is invalid, it is necessary to skip it and move on to the next option character. If _opti is greater than or equal to the length of the current command-line argument, it is necessary to move on to the next argument, so Optind is incremented and _opti is reset to zero. Otherwise, Optind is left alone and _opti is merely incremented.

In any case, because the option is invalid, getopt returns ?. The main program can examine Optopt if it needs to know what the invalid option letter actually is. Continuing on:

```
        if (substr(options, i + 1, 1) == ":") {
            # get option argument
            if (length(substr(argv[Optind], _opti + 1)) > 0)
                Optarg = substr(argv[Optind], _opti + 1)
            else
                Optarg = argv[++Optind]
            _opti = 0
        } else
            Optarg = ""
```

If the option requires an argument, the option letter is followed by a colon in the options string. If there are remaining characters in the current command-line argument (argv[Optind]), then the rest of that string is assigned to Optarg. Otherwise, the next command-line argument is used (-xFOO versus -x FOO). In either case, _opti is reset to zero, because there are no more characters left to examine in the current command-line argument. Continuing:

```
        if (_opti == 0 || _opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return thisopt
    }
```

Finally, if `_opti` is either zero or greater than the length of the current command-line argument, it means this element in `argv` is through being processed, so `Optind` is incremented to point to the next element in `argv`. If neither condition is true, then only `_opti` is incremented, so that the next option letter can be processed on the next call to `getopt`.

The `BEGIN` rule initializes both `Opterr` and `Optind` to one. `Opterr` is set to one, since the default behavior is for `getopt` to print a diagnostic message upon seeing an invalid option. `Optind` is set to one, since there's no reason to look at the program name, which is in `ARGV[0]`:

```
BEGIN {
    Opterr = 1    # default is to diagnose
    Optind = 1    # skip ARGV[0]

    # test program
    if (_getopt_test) {
        while ((_go_c = getopt(ARGC, ARGV, "ab:cd")) != -1)
            printf("c = <%c>, optarg = <%s>\n",
                _go_c, Optarg)
        printf("non-option arguments:\n")
        for (; Optind < ARGC; Optind++)
            printf("\tARGV[%d] = <%s>\n",
                Optind, ARGV[Optind])
    }
}
```

The rest of the `BEGIN` rule is a simple test program. Here is the result of two sample runs of the test program:

```
$ awk -f getopt.awk -v _getopt_test=1 -- -a -cbARG bax -x
c = <a>, optarg = <>
c = <c>, optarg = <>
c = <b>, optarg = <ARG>
non-option arguments:
        ARGV[3] = <bax>
        ARGV[4] = <-x>

$ awk -f getopt.awk -v _getopt_test=1 -- -a -x -- xyz abc
c = <a>, optarg = <>
x -- invalid option
c = <?>, optarg = <>
non-option arguments:
        ARGV[4] = <xyz>
        ARGV[5] = <abc>
```

In both runs, the first –– terminates the arguments to *awk*, so that it does not try to interpret the *–a*, etc., as its own options. Several of the sample programs presented in Chapter 13 use `getopt` to process their arguments.

Reading the User Database

The PROCINFO array (see the section "Built-in Variables" in Chapter 6) provides access to the current user's real and effective user and group ID numbers, and if available, the user's supplementary group set. However, because these are numbers, they do not provide very useful information to the average user. There needs to be some way to find the user information associated with the user and group ID numbers. This section presents a suite of functions for retrieving information from the user database. See the section "Reading the Group Database" later in this chapter for a similar suite that retrieves information from the group database.

The POSIX standard does not define the file where user information is kept. Instead, it provides the <pwd.h> header file and several C language subroutines for obtaining user information. The primary function is getpwent, for "get password entry." The "password" comes from the original user database file, */etc/passwd*, which stores user information, along with the encrypted passwords (hence the name).

While an *awk* program could simply read */etc/passwd* directly, this file may not contain complete information about the system's set of users.* To be sure you are able to produce a readable and complete version of the user database, it is necessary to write a small C program that calls getpwent. getpwent is defined as returning a pointer to a struct passwd. Each time it is called, it returns the next entry in the database. When there are no more entries, it returns NULL, the null pointer. When this happens, the C program should call endpwent to close the database. Following is *pwcat*, a C program that "cats" the password database:

```
/*
 * pwcat.c
 *
 * Generate a printable version of the password database
 */

#include <stdio.h>
#include <pwd.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct passwd *p;
```

* It is often the case that password information is stored in a network database.

```
        while ((p = getpwent()) != NULL)
            printf("%s:%s:%d:%d:%s:%s:%s\n",
                p->pw_name, p->pw_passwd, p->pw_uid,
                p->pw_gid, p->pw_gecos, p->pw_dir, p->pw_shell);

        endpwent();
        exit(0);
    }
```

If you don't understand C, don't worry about it. The output from *pwcat* is the user database, in the traditional */etc/passwd* format of colon-separated fields. The fields are:

Login name
: The user's login name.

Encrypted password
: The user's encrypted password. This may not be available on some systems.

User-ID
: The user's numeric user ID number.

Group-ID
: The user's numeric group ID number.

Full name
: The user's full name, and perhaps other information associated with the user.

Home directory
: The user's login (or "home") directory (familiar to shell programmers as `$HOME`).

Login shell
: The program that is run when the user logs in. This is usually a shell, such as bash.

A few lines representative of *pwcat*'s output are as follows:

```
$ pwcat
root:3Ov02d5VaUPB6:0:1:Operator:/:/bin/sh
nobody:*:65534:65534::/:
daemon:*:1:1::/:
sys:*:2:2::/:/bin/csh
bin:*:3:3::/bin:
arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
miriam:yxaay:112:10:Miriam Robbins:/home/miriam:/bin/sh
andy:abcca2:113:10:Andy Jacobs:/home/andy:/bin/sh
...
```

With that introduction, following is a group of functions for getting user information. There are several functions here, corresponding to the C functions of the same names:

```
# passwd.awk --- access password file information

BEGIN {
    # tailor this to suit your system
    _pw_awklib = "/usr/local/libexec/awk/"
}

function _pw_init(    oldfs, oldrs, olddol0, pwcat, using_fw)
{
    if (_pw_inited)
        return

    oldfs = FS
    oldrs = RS
    olddol0 = $0
    using_fw = (PROCINFO["FS"] == "FIELDWIDTHS")
    FS = ":"
    RS = "\n"

    pwcat = _pw_awklib "pwcat"
    while ((pwcat | getline) > 0) {
        _pw_byname[$1] = $0
        _pw_byuid[$3] = $0
        _pw_bycount[++_pw_total] = $0
    }
    close(pwcat)
    _pw_count = 0
    _pw_inited = 1
    FS = oldfs
    if (using_fw)
        FIELDWIDTHS = FIELDWIDTHS
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where *pwcat* is stored. Because it is used to help out an *awk* library routine, we have chosen to put it in */usr/local/libexec/awk*; however, you might want it to be in a different directory on your system.

The function `_pw_init` keeps three copies of the user information in three associative arrays. The arrays are indexed by username (`_pw_byname`), by user-id number (`_pw_byuid`), and by order of occurrence (`_pw_bycount`). The variable `_pw_inited` is used for efficiency; `_pw_init` needs only to be called once.

Because this function uses `getline` to read information from *pwcat*, it first saves the values of `FS`, `RS`, and `$0`. It notes in the variable `using_fw` whether field splitting with `FIELDWIDTHS` is in effect or not. Doing so is necessary, since these functions could be called from anywhere within a user's program, and the user may have his own way of splitting records and fields.

The `using_fw` variable checks `PROCINFO["FS"]`, which is `"FIELDWIDTHS"` if field splitting is being done with `FIELDWIDTHS`. This makes it possible to restore the correct field-splitting mechanism later. The test can only be true for *gawk*. It is false if using `FS` or on some other *awk* implementation.

The main part of the function uses a loop to read database lines, split the line into fields, and then store the line into each array as necessary. When the loop is done, `_pw_init` cleans up by closing the pipeline, setting `_pw_inited` to one, and restoring `FS` (and `FIELDWIDTHS` if necessary), `RS`, and `$0`. The use of `_pw_count` is explained shortly.

The `getpwnam` function takes a username as a string argument. If that user is in the database, it returns the appropriate line. Otherwise, it returns the null string:

```
function getpwnam(name)
{
    _pw_init()
    if (name in _pw_byname)
        return _pw_byname[name]
    return ""
}
```

Similarly, the `getpwuid` function takes a user-id number argument. If that user number is in the database, it returns the appropriate line. Otherwise, it returns the null string:

```
function getpwuid(uid)
{
    _pw_init()
    if (uid in _pw_byuid)
        return _pw_byuid[uid]
    return ""
}
```

The `getpwent` function simply steps through the database, one entry at a time. It uses `_pw_count` to track its current position in the `_pw_bycount` array:

```
function getpwent()
{
    _pw_init()
    if (_pw_count < _pw_total)
        return _pw_bycount[++_pw_count]
    return ""
}
```

The `endpwent` function resets `_pw_count` to zero, so that subsequent calls to `getpwent` start over again:

```
function endpwent()
{
    _pw_count = 0
}
```

A conscious design decision in this suite was made that requires each subroutine to call `_pw_init` to initialize the database arrays. The overhead of running a separate process to generate the user database, and the I/O to scan it, are only incurred if the user's main program actually calls one of these functions. If this library file is loaded along with a user's program, but none of the routines are ever called, then there is no extra runtime overhead. (The alternative is move the body of `_pw_init` into a `BEGIN` rule, which always runs *pwcat*. This simplifies the code but runs an extra process that may never be needed.)

In turn, calling `_pw_init` is not too expensive, because the `_pw_inited` variable keeps the program from reading the data more than once. If you are worried about squeezing every last cycle out of your *awk* program, the check of `_pw_inited` could be moved out of `_pw_init` and duplicated in all the other functions. In practice, this is not necessary, since most *awk* programs are I/O-bound, and it clutters up the code.

The *id* program in the section "Printing out User Information" in Chapter 13 uses these functions.

Reading the Group Database

Much of the discussion presented in the previous section applies to the group database as well. Although there has traditionally been a well-known file (*/etc/group*) in a well-known format, the POSIX standard only provides a set of C library routines (`<grp.h>` and `getgrent`) for accessing the information. Even though this file may exist, it likely does not have complete information. Therefore, as with the user database, it is necessary to have a small C program that generates the group database as its output.

grcat, a C program that "cats" the group database, is as follows:

```
/*
 * grcat.c
 *
 * Generate a printable version of the group database
 */

#include <stdio.h>
#include <grp.h>
```

```
int
main(argc, argv)
int argc;
char **argv;
{
    struct group *g;
    int i;

    while ((g = getgrent()) != NULL) {
        printf("%s:%s:%d:", g->gr_name, g->gr_passwd,
                g->gr_gid);
        for (i = 0; g->gr_mem[i] != NULL; i++) {
            printf("%s", g->gr_mem[i]);
            if (g->gr_mem[i+1] != NULL)
                putchar(',');
        }
        putchar('\n');
    }
    endgrent();
    exit(0);
}
```

Each line in the group database represents one group. The fields are separated with colons and represent the following information:

Group name
: The group's name.

Group password
: The group's encrypted password. In practice, this field is never used; it is usually empty or set to `*`.

Group-ID
: The group's numeric group ID number; this number is unique within the file.

Group member list
: A comma-separated list of usernames. These users are members of the group. Modern Unix systems allow users to be members of several groups simultaneously. If your system does, then there are elements `"group1"` through `"group`*N*`"` in `PROCINFO` for those group ID numbers. (Note that `PROCINFO` is a *gawk* extension; see the section "Built-in Variables" in Chapter 6.)

Here is what running *grcat* might produce:

```
$ grcat
wheel:*:0:arnold
nogroup:*:65534:
daemon:*:1:
kmem:*:2:
staff:*:10:arnold,miriam,andy
other:*:20:
...
```

Here are the functions for obtaining information from the group database. There are several, modeled after the C library functions of the same names:

```
# group.awk --- functions for dealing with the group file

BEGIN    \
{
    # Change to suit your system
    _gr_awklib = "/usr/local/libexec/awk/"
}

function _gr_init(    oldfs, oldrs, olddol0, grcat, using_fw, n, a, i)
{
    if (_gr_inited)
        return

    oldfs = FS
    oldrs = RS
    olddol0 = $0
    using_fw = (PROCINFO["FS"] == "FIELDWIDTHS")
    FS = ":"
    RS = "\n"

    grcat = _gr_awklib "grcat"
    while ((grcat | getline) > 0) {
        if ($1 in _gr_byname)
            _gr_byname[$1] = _gr_byname[$1] "," $4
        else
            _gr_byname[$1] = $0
        if ($3 in _gr_bygid)
            _gr_bygid[$3] = _gr_bygid[$3] "," $4
        else
            _gr_bygid[$3] = $0

        n = split($4, a, "[ \t]*,[ \t]*")
        for (i = 1; i <= n; i++)
            if (a[i] in _gr_groupsbyuser)
                _gr_groupsbyuser[a[i]] = \
                    _gr_groupsbyuser[a[i]] " " $1
            else
                _gr_groupsbyuser[a[i]] = $1

        _gr_bycount[++_gr_count] = $0
    }
    close(grcat)
    _gr_count = 0
    _gr_inited++
    FS = oldfs
    if (using_fw)
        FIELDWIDTHS = FIELDWIDTHS
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where *grcat* is stored. Because it is used to help out an *awk* library routine, we have chosen to put it in */usr/local/libexec/awk*. You might want it to be in a different directory on your system.

These routines follow the same general outline as the user database routines (see the section "Reading the User Database" earlier in this chapter). The `_gr_inited` variable is used to ensure that the database is scanned no more than once. The `_gr_init` function first saves `FS`, `FIELDWIDTHS`, `RS`, and `$0`, and then sets `FS` and `RS` to the correct values for scanning the group information.

The group information is stored is several associative arrays. The arrays are indexed by group name (`_gr_byname`), by group ID number (`_gr_bygid`), and by position in the database (`_gr_bycount`). There is an additional array indexed by username (`_gr_groupsbyuser`), which is a space-separated list of groups to which each user belongs.

Unlike the user database, it is possible to have multiple records in the database for the same group. This is common when a group has a large number of members. A pair of such entries might look like the following:

```
tvpeople:*:101:johnny,jay,arsenio
tvpeople:*:101:david,conan,tom,joan
```

For this reason, `_gr_init` looks to see if a group name or group ID number is already seen. If it is, then the usernames are simply concatenated onto the previous list of users. (There is actually a subtle problem with the code just presented. Suppose that the first time there were no names. This code adds the names with a leading comma. It also doesn't check that there is a `$4`.)

Finally, `_gr_init` closes the pipeline to *grcat*, restores `FS` (and `FIELDWIDTHS` if necessary), `RS`, and `$0`, initializes `_gr_count` to zero (it is used later), and makes `_gr_inited` nonzero.

The `getgrnam` function takes a group name as its argument, and if that group exists, it is returned. Otherwise, `getgrnam` returns the null string:

```
function getgrnam(group)
{
    _gr_init()
    if (group in _gr_byname)
        return _gr_byname[group]
    return ""
}
```

The `getgrgid` function is similar, it takes a numeric group ID and looks up the information associated with that group ID:

```
function getgrgid(gid)
{
    _gr_init()
    if (gid in _gr_bygid)
        return _gr_bygid[gid]
    return ""
}
```

The `getgruser` function does not have a C counterpart. It takes a username and returns the list of groups that have the user as a member:

```
function getgruser(user)
{
    _gr_init()
    if (user in _gr_groupsbyuser)
        return _gr_groupsbyuser[user]
    return ""
}
```

The `getgrent` function steps through the database one entry at a time. It uses `_gr_count` to track its position in the list:

```
function getgrent()
{
    _gr_init()
    if (++_gr_count in _gr_bycount)
        return _gr_bycount[_gr_count]
    return ""
}
```

The `endgrent` function resets `_gr_count` to zero so that `getgrent` can start over again:

```
function endgrent()
{
    _gr_count = 0
}
```

As with the user database routines, each function calls `_gr_init` to initialize the arrays. Doing so only incurs the extra overhead of running *grcat* if these functions are used (as opposed to moving the body of `_gr_init` into a `BEGIN` rule).

Most of the work is in scanning the database and building the various associative arrays. The functions that the user calls are themselves very simple, relying on *awk*'s associative arrays to do work.

The *id* program in the section "Printing out User Information" in Chapter 13 uses these functions.