In this chapter:

3

Reading Input Files

In the typical *awk* program, all input is read either from the standard input (by default, this is the keyboard, but often it is a pipe from another command) or from files whose names you specify on the *awk* command line. If you specify input files, *awk* reads them in order, processing all the data from one before going on to the next. The name of the current input file can be found in the built-in variable `FILENAME` (see the section "Built-in Variables" in Chapter 6, *Patterns, Actions, and Variables*).

The input is read in units called *records*, and is processed by the rules of your program one record at a time. By default, each record is one line. Each record is automatically split into chunks called *fields*. This makes it more convenient for programs to work on the parts of a record.

On rare occasions, you may need to use the `getline` command. The `getline` command is valuable, both because it can do explicit input from any number of files, and because the files used with it do not have to be named on the *awk* command line (see the section "Explicit Input with getline" later in this chapter).

How Input Is Split into Records

The *awk* utility divides the input for your *awk* program into records and fields. *awk* keeps track of the number of records that have been read from the current input file. This value is stored in a built-in variable called `FNR`. It is reset to zero when a new file is started. Another built-in variable, `NR`, is the total number of input records read so far from all datafiles. It starts at zero, but is never automatically reset to zero.

Records are separated by a character called the *record separator*. By default, the record separator is the newline character. This is why records are, by default, single lines. A different character can be used for the record separator by assigning the character to the built-in variable RS.

Like any other variable, the value of RS can be changed in the *awk* program with the assignment operator, = (see the section “Assignment Expressions” in Chapter 5, *Expressions*). The new record-separator character should be enclosed in quotation marks, which indicate a string constant. Often the right time to do this is at the beginning of execution, before any input is processed, so that the very first record is read with the proper separator. To do this, use the special BEGIN pattern (see the section “The BEGIN and END Special Patterns” in Chapter 6). For example:

```
awk 'BEGIN { RS = "/" }
     { print $0 }' BBS-list
```

changes the value of RS to "/", before reading any input. This is a string whose first character is a slash; as a result, records are separated by slashes. Then the input file is read, and the second rule in the *awk* program (the action with no pattern) prints each record. Because each print statement adds a newline at the end of its output, this *awk* program copies the input with each slash changed to a newline. Here are the results of running the program on *BBS-list*:

```
$ awk 'BEGIN { RS = "/" }
>       { print $0 }' BBS-list
aardvark     555-5553     1200
300          B
alpo-net     555-3412     2400
1200
300     A
barfly       555-7685     1200
300          A
bites        555-1675     2400
1200
300     A
camelot      555-0542     300               C
core         555-2912     1200
300          C
fooey        555-1234     2400
1200
300     B
foot         555-6699     1200
300          B
macfoo       555-6480     1200
300          A
sdace        555-3430     2400
1200
300     A
sabafoo      555-2127     1200
300          C

$
```

Note that the entry for the camelot BBS is not split. In the original datafile (see the section "Datafiles for the Examples" in Chapter 1, *Getting Started with awk*), the line looks like this:

```
camelot      555-0542     300              C
```

It has one baud rate only, so there are no slashes in the record, unlike the others that have two or more baud rates. In fact, this record is treated as part of the record for the core BBS; the newline separating them in the output is the original newline in the datafile, not the one added by *awk* when it printed the record!

Another way to change the record separator is on the command line, using the variable-assignment feature (see the section "Other Command-Line Arguments" in Chapter 11, *Running awk and gawk*):

```
awk '{ print $0 }' RS="/" BBS-list
```

This sets RS to / before processing *BBS-list*.

Using an unusual character such as / for the record separator produces correct behavior in the vast majority of cases. However, the following (extreme) pipeline prints a surprising 1:

```
$ echo | awk 'BEGIN { RS = "a" } ; { print NF }'
1
```

There is one field, consisting of a newline. The value of the built-in variable NF is the number of fields in the current record.

Reaching the end of an input file terminates the current input record, even if the last character in the file is not the character in RS. (d.c.)

The empty string "" (a string without any characters) has a special meaning as the value of RS. It means that records are separated by one or more blank lines and nothing else. See the section "Multiple-Line Records" later in this chapter for more details.

If you change the value of RS in the middle of an *awk* run, the new value is used to delimit subsequent records, but the record currently being processed, as well as records already processed, are not affected.

After the end of the record has been determined, *gawk* sets the variable RT to the text in the input that matched RS. When using *gawk*, the value of RS is not limited to a one-character string. It can be any regular expression (see Chapter 2, *Regular Expressions*). In general, each record ends at the next string that matches the regular expression; the next record starts at the end of the matching string. This general rule is actually at work in the usual case, where RS contains just a newline: a record ends at the beginning of the next matching string (the next newline in the input), and the following record starts just after the end of this string (at the first character of the following line). The newline, because it matches RS, is not part of either record.

When RS is a single character, RT contains the same single character. However, when RS is a regular expression, RT contains the actual input text that matched the regular expression.

The following example illustrates both of these features. It sets RS equal to a regular expression that matches either a newline or a series of one or more uppercase letters with optional leading and/or trailing whitespace:

```
$ echo record 1 AAAA record 2 BBBB record 3 |
> gawk 'BEGIN { RS = "\n|( *[[:upper:]]+ *)" }
>             { print "Record =", $0, "and RT =", RT }'
Record = record 1 and RT =  AAAA
Record = record 2 and RT =  BBBB
Record = record 3 and RT =

$
```

The final line of output has an extra blank line. This is because the value of RT is a newline, and the print statement supplies its own terminating newline. See the section "A Simple Stream Editor" in Chapter 13, *Practical awk Programs*, for a more useful example of RS as a regexp and RT.

The use of RS as a regular expression and the RT variable are *gawk* extensions; they are not available in compatibility mode (see the section "Command-Line Options" in Chapter 11). In compatibility mode, only the first character of the value of RS is used to determine the end of the record.

Examining Fields

When *awk* reads an input record, the record is automatically *parsed* or separated by the interpreter into chunks called *fields*. By default, fields are separated by *whitespace*, like words in a line. Whitespace in *awk* means any string of one or more spaces, tabs, or newlines;* other characters, such as formfeed, vertical tab, etc. that are considered whitespace by other languages, are *not* considered whitespace by *awk*.

The purpose of fields is to make it more convenient for you to refer to these pieces of the record. You don't have to use them—you can operate on the whole record if you want—but fields are what make simple *awk* programs so powerful.

* In POSIX *awk*, newlines are not considered whitespace for separating fields.

RS = "\0" Is Not Portable

There are times when you might want to treat an entire datafile as a single record. The only way to make this happen is to give `RS` a value that you know doesn't occur in the input file. This is hard to do in a general way, such that a program always works for arbitrary input files.

You might think that for text files, the NUL character, which consists of a character with all bits equal to zero, is a good value to use for `RS` in this case:

```
BEGIN { RS = "\0" }  # whole file becomes one record?
```

gawk in fact accepts this, and uses the NUL character for the record separator. However, this usage is *not* portable to other *awk* implementations.

All other *awk* implementations[*] store strings internally as C-style strings. C strings use the NUL character as the string terminator. In effect, this means that `RS = "\0"` is the same as `RS = ""`. (d.c.)

The best way to treat a whole file as a single record is to simply read the file in, one record at a time, concatenating each record onto the end of the previous ones.

A dollar-sign (`$`) is used to refer to a field in an *awk* program, followed by the number of the field you want. Thus, `$1` refers to the first field, `$2` to the second, and so on. (Unlike the Unix shells, the field numbers are not limited to single digits. `$127` is the one hundred twenty-seventh field in the record.) For example, suppose the following is a line of input:

```
This seems like a pretty nice example.
```

Here the first field, or `$1`, is `This`, the second field, or `$2`, is `seems`, and so on. Note that the last field, `$7`, is `example.`. Because there is no space between the `e` and the `.`, the period is considered part of the seventh field.

`NF` is a built-in variable whose value is the number of fields in the current record. *awk* automatically updates the value of `NF` each time it reads a record. No matter how many fields there are, the last field in a record can be represented by `$NF`. So, `$NF` is the same as `$7`, which is `example.`. If you try to reference a field beyond the last one (such as `$8` when the record has only seven fields), you get the empty string. (If used in a numeric operation, you get zero.)

[*] At least that we know about.

The use of `$0`, which looks like a reference to the "zero-th" field, is a special case: it represents the whole input record when you are not interested in specific fields. Here are some more examples:

```
$ awk '$1 ~ /foo/ { print $0 }' BBS-list
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
macfoo       555-6480     1200/300          A
sabafoo      555-2127     1200/300          C
```

This example prints each record in the file *BBS-list* whose first field contains the string `foo`. The operator `~` is called a *matching operator* (see the section "How to Use Regular Expressions" in Chapter 2); it tests whether a string (here, the field `$1`) matches a given regular expression.

By contrast, the following example looks for `foo` in *the entire record* and prints the first field and the last field for each matching input record:

```
$ awk '/foo/ { print $1, $NF }' BBS-list
fooey B
foot B
macfoo A
sabafoo C
```

Non-constant Field Numbers

The number of a field does not need to be a constant. Any expression in the *awk* language can be used after a `$` to refer to a field. The value of the expression specifies the field number. If the value is a string, rather than a number, it is converted to a number. Consider this example:

```
awk '{ print $NR }'
```

Recall that `NR` is the number of records read so far: one in the first record, two in the second, etc. So this example prints the first field of the first record, the second field of the second record, and so on. For the twentieth record, field number 20 is printed; most likely, the record has fewer than 20 fields, so this prints a blank line. Here is another example of using expressions as field numbers:

```
awk '{ print $(2*2) }' BBS-list
```

awk evaluates the expression `(2*2)` and uses its value as the number of the field to print. The `*` sign represents multiplication, so the expression `2*2` evaluates to four. The parentheses are used so that the multiplication is done before the `$` operation; they are necessary whenever there is a binary operator in the field-number expression. This example, then, prints the hours of operation (the fourth field) for every line of the file *BBS-list*. (All of the *awk* operators are listed, in order of decreasing precedence, in the section "Operator Precedence (How Operators Nest)" in Chapter 5.)

If the field number you compute is zero, you get the entire record. Thus, `$(2-2)` has the same value as `$0`. Negative field numbers are not allowed; trying to reference one usually terminates the program. (The POSIX standard does not define what happens when you reference a negative field number. *gawk* notices this and terminates your program. Other *awk* implementations may behave differently.)

As mentioned earlier in the section "Examining Fields," *awk* stores the current record's number of fields in the built-in variable `NF` (also see the section "Built-in Variables" in Chapter 6). The expression `$NF` is not a special feature—it is the direct consequence of evaluating `NF` and using its value as a field number.

Changing the Contents of a Field

The contents of a field, as seen by *awk*, can be changed within an *awk* program; this changes what *awk* perceives as the current input record. (The actual input is untouched; *awk* never modifies the input file.) Consider the following example and its output:

```
$ awk '{ nboxes = $3 ; $3 = $3 - 10
>        print nboxes, $3 }' inventory-shipped
13 3
15 5
15 5
...
```

The program first saves the original value of field three in the variable `nboxes`. The `-` sign represents subtraction, so this program reassigns field three, `$3`, as the original value of field three minus ten: `$3 - 10`. (See the section "Arithmetic Operators" in Chapter 5.) Then it prints the original and new values for field three. (Someone in the warehouse made a consistent mistake while inventorying the red boxes.)

For this to work, the text in field `$2` must make sense as a number; the string of characters must be converted to a number for the computer to do arithmetic on it. The number resulting from the subtraction is converted back to a string of characters that then becomes field three. See the section "Conversion of Strings and Numbers" in Chapter 5.

When the value of a field is changed (as perceived by *awk*), the text of the input record is recalculated to contain the new field where the old one was. In other words, `$0` changes to reflect the altered field. Thus, this program prints a copy of the input file, with 10 subtracted from the second field of each line:

```
$ awk '{ $2 = $2 - 10; print $0 }' inventory-shipped
Jan 3 25 15 115
Feb 5 32 24 226
Mar 5 24 34 228
...
```

It is also possible to also assign contents to fields that are out of range. For example:

```
$ awk '{ $6 = ($5 + $4 + $3 + $2)
>        print $6 }' inventory-shipped
168
297
301
...
```

We've just created `$6`, whose value is the sum of fields `$2`, `$3`, `$4`, and `$5`. The `+` sign represents addition. For the file *inventory-shipped*, `$6` represents the total number of parcels shipped for a particular month.

Creating a new field changes *awk*'s internal copy of the current input record, which is the value of `$0`. Thus, if you do `print $0` after adding a field, the record printed includes the new field, with the appropriate number of field separators between it and the previously existing fields.

This recomputation affects and is affected by `NF` (the number of fields; see the section "Examining Fields" earlier in this chapter). It is also affected by a feature that has not been discussed yet: the *output field separator*, `OFS`, used to separate the fields (see the section "Output Separators" in Chapter 4, *Printing Output*). For example, the value of `NF` is set to the number of the highest field you create.

Note, however, that merely *referencing* an out-of-range field does *not* change the value of either `$0` or `NF`. Referencing an out-of-range field only produces an empty string. For example:

```
if ($(NF+1) != "")
    print "can't happen"
else
    print "everything is normal"
```

should print `everything is normal`, because `NF+1` is certain to be out of range. (See the section "The if-else Statement" in Chapter 6 for more information about *awk*'s `if-else` statements. See the section "Variable Typing and Comparison Expressions" in Chapter 5 for more information about the `!=` operator.)

It is important to note that making an assignment to an existing field changes the value of `$0` but does not change the value of `NF`, even when you assign the empty string to a field. For example:

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""
>                       print $0; print NF }'
a::c:d
4
```

The field is still there; it just has an empty value, denoted by the two colons between `a` and `c`. This example shows what happens if you create a new field:

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""; $6 = "new"
>                       print $0; print NF }'
a::c:d::new
6
```

The intervening field, `$5`, is created with an empty value (indicated by the second pair of adjacent colons), and `NF` is updated with the value six.

Decrementing `NF` throws away the values of the fields after the new value of `NF` and recomputes `$0`. (d.c.) Here is an example:

```
$ echo a b c d e f | awk '{ print "NF =", NF;
>                            NF = 3; print $0 }'
NF = 6
a b c
```

Some versions of *awk* don't rebuild `$0` when `NF` is decremented. Caveat emptor.

Specifying How Fields Are Separated

The *field separator*, which is either a single character or a regular expression, controls the way *awk* splits an input record into fields. *awk* scans the input record for character sequences that match the separator; the fields themselves are the text between the matches.

In the examples that follow, we use the small box (□) to represent spaces in the output. If the field separator is `oo`, then the following line:

```
moo goo gai pan
```

is split into three fields: `m`, `□g`, and `□gai□pan`. Note the leading spaces in the values of the second and third fields.

The field separator is represented by the built-in variable `FS`. Shell programmers take note: *awk* does *not* use the name `IFS` that is used by the POSIX-compliant shells (such as the Unix Bourne shell, *sh*, or *bash*).

The value of `FS` can be changed in the *awk* program with the assignment operator, `=` (see the section "Assignment Expressions" in Chapter 5). Often the right time to do this is at the beginning of execution before any input has been processed, so that the very first record is read with the proper separator. To do this, use the special `BEGIN` pattern (see the section “The BEGIN and END Special Patterns” in Chapter 6). For example, here we set the value of `FS` to the string `","`:

```
awk ’BEGIN { FS = "," } ; { print $2 }’
```

Given the input line:

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

this *awk* program extracts and prints the string `□29□Oak□St.`.

Sometimes the input data contains separator characters that don’t separate fields the way you thought they would. For instance, the person’s name in the example we just used might have a title or suffix attached, such as:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

The same program would extract `□LXIX`, instead of `□29□Oak□St.`. If you were expecting the program to print the address, you would be surprised. The moral is to choose your data layout and separator characters carefully to prevent such problems. (If the data is not in a form that is easy to process, perhaps you can massage it first with a separate *awk* program.)

Fields are normally separated by whitespace sequences (spaces, tabs, and newlines), not by single spaces. Two spaces in a row do not delimit an empty field. The default value of the field separator `FS` is a string containing a single space, `" "`. If *awk* interpreted this value in the usual way, each space character would separate fields, so two spaces in a row would make an empty field between them. The reason this does not happen is that a single space as the value of `FS` is a special case—it is taken to specify the default manner of delimiting fields.

If `FS` is any other single character, such as `","`, then each occurrence of that character separates two fields. Two consecutive occurrences delimit an empty field. If the character occurs at the beginning or the end of the line, that too delimits an empty field. The space character is the only single character that does not follow these rules.

Using Regular Expressions to Separate Fields

The previous section discussed the use of single characters or simple strings as the value of `FS`. More generally, the value of `FS` may be a string containing any regular expression. In this case, each match in the record for the regular expression separates fields. For example, the assignment:

```
FS = ", \t"
```

makes every area of an input line that consists of a comma followed by a space and a tab into a field separator.

For a less trivial example of a regular expression, try using single spaces to separate fields the way single commas are used. `FS` can be set to `"[ ]"` (left bracket, space, right bracket). This regular expression matches a single space and nothing else (see Chapter 2).

There is an important difference between the two cases of `FS = " "` (a single space) and `FS = "[ \t\n]+"` (a regular expression matching one or more spaces, tabs, or newlines). For both values of `FS`, fields are separated by *runs* (multiple adjacent occurrences) of spaces, tabs, and/or newlines. However, when the value of `FS` is `" "`, *awk* first strips leading and trailing whitespace from the record and then decides where the fields are. For example, the following pipeline prints `b`:

```
$ echo ' a b c d ' | awk '{ print $2 }'
b
```

However, this pipeline prints `a` (note the extra spaces around each letter):

```
$ echo ' a  b  c  d ' | awk 'BEGIN { FS = "[ \t\n]+" }
>                                  { print $2 }'
a
```

In this case, the first field is *null* or empty.

The stripping of leading and trailing whitespace also comes into play whenever `$0` is recomputed. For instance, study this pipeline:

```
$ echo '   a b c d' | awk '{ print; $2 = $2; print }'
   a b c d
a b c d
```

The first `print` statement prints the record as it was read, with leading whitespace intact. The assignment to `$2` rebuilds `$0` by concatenating `$1` through `$NF` together, separated by the value of `OFS`. Because the leading whitespace was ignored when finding `$1`, it is not part of the new `$0`. Finally, the last `print` statement prints the new `$0`.

Making Each Character a Separate Field

There are times when you may want to examine each character of a record separately. This can be done in *gawk* by simply assigning the null string (`""`) to `FS`. In this case, each individual character in the record becomes a separate field. For example:

```
$ echo a b | gawk 'BEGIN { FS = "" }
>                  {
```

```
>                       for (i = 1; i <= NF; i = i + 1)
>                           print "Field", i, "is", $i
>                   }'
Field 1 is a
Field 2 is
Field 3 is b
```

Traditionally, the behavior of `FS` equal to `""` was not defined. In this case, most versions of Unix *awk* simply treat the entire record as only having one field. (d.c.) In compatibility mode (see the section "Command-Line Options" in Chapter 11), if `FS` is the null string, then *gawk* also behaves this way.

Setting FS from the Command Line

`FS` can be set on the command line. Use the *–F* option to do so. For example:

```
awk -F, 'program' input-files
```

sets `FS` to the `,` character. Notice that the option uses an uppercase *–F* instead of a lowercase *–f*, which specifies a file containing an *awk* program. Case is significant in command-line options: the *–F* and *–f* options have nothing to do with each other. You can use both options at the same time to set the `FS` variable *and* get an *awk* program from a file.

The value used for the argument to *–F* is processed in exactly the same way as assignments to the built-in variable `FS`. Any special characters in the field separator must be escaped appropriately. For example, to use a `\` as the field separator on the command line, you would have to type:

```
# same as FS = "\\"
awk -F\\\\ '...' files ...
```

Because `\` is used for quoting in the shell, *awk* sees `-F\\`. Then *awk* processes the `\\` for escape characters (see the section "Escape Sequences" in Chapter 2), finally yielding a single `\` to use for the field separator.

As a special case, in compatibility mode (see the section "Command-Line Options" in Chapter 11) if the argument to *–F* is `t`, then `FS` is set to the tab character. If you type `-F\t` at the shell, without any quotes, the `\` gets deleted, so *awk* figures that you really want your fields to be separated with tabs and not `t`s. Use `-v FS="t"` or `-F"[t]"` on the command line if you really do want to separate your fields with `t`s.

For example, let's use an *awk* program file called *baud.awk* that contains the pattern `/300/` and the action `print $1`:

```
/300/   { print $1 }
```

Let's also set `FS` to be the `-` character and run the program on the file *BBS-list*. The following command prints a list of the names of the bulletin boards that operate at 300 baud and the first three digits of their phone numbers:

```
$ awk -F- -f baud.awk BBS-list
aardvark     555
alpo
barfly       555
bites        555
camelot      555
core         555
fooey        555
foot         555
macfoo       555
sdace        555
sabafoo      555
```

Note the second line of output. The second line in the original file looked like this:

```
alpo-net     555-3412     2400/1200/300     A
```

The - as part of the system's name was used as the field separator, instead of the - in the phone number that was originally intended. This demonstrates why you have to be careful in choosing your field and record separators.

Perhaps the most common use of a single character as the field separator occurs when processing the Unix system password file. On many Unix systems, each user has a separate entry in the system password file, one line per user. The information in these lines is separated by colons. The first field is the user's logon name and the second is the user's (encrypted or shadow) password. A password file entry might look like this:

```
arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/bash
```

The following program searches the system password file and prints the entries for users who have no password:

```
awk -F: '$2 == ""' /etc/passwd
```

Field-Splitting Summary

The following list summarizes how fields are split, based on the value of `FS` (== means "is equal to"):

`FS == " "`
: Fields are separated by runs of whitespace. Leading and trailing whitespace are ignored. This is the default.

`FS ==` *`any other single character`*
: Fields are separated by each occurrence of the character. Multiple successive occurrences delimit empty fields, as do leading and trailing occurrences. The character can even be a regexp metacharacter; it does not need to be escaped.

`FS == regexp`
Fields are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty fields.

`FS == ""`
Each individual character in the record becomes a separate field. (This is a *gawk* extension; it is not specified by the POSIX standard.)

Changing FS Does Not Affect the Fields

According to the POSIX standard, *awk* is supposed to behave as if each record is split into fields at the time it is read. In particular, this means that if you change the value of `FS` after a record is read, the value of the fields (i.e., how they were split) should reflect the old value of `FS`, not the new one.

However, many implementations of *awk* do not work this way. Instead, they defer splitting the fields until a field is actually referenced. The fields are split using the *current* value of `FS`! (d.c.) This behavior can be difficult to diagnose. The following example illustrates the difference between the two methods (the *sed*[*] command prints just the first line of */etc/passwd*):

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

which usually prints:

```
root
```

on an incorrect implementation of *awk*, while *gawk* prints something like:

```
root:nSijPlPhZZwgE:0:0:Root:/:
```

Reading Fixed-Width Data

This section discusses an advanced feature of *gawk*. If you are a novice *awk* user, you might want to skip it on the first reading.

gawk Version 2.13 introduced a facility for dealing with fixed-width fields with no distinctive field separator. For example, data of this nature arises in the input for old Fortran programs where numbers are run together, or in the output of programs that did not anticipate the use of their output as input for other programs.

An example of the latter is a table where all the columns are lined up by the use of a variable number of spaces and *empty fields are just spaces*. Clearly, *awk*'s

[*] The *sed* utility is a "stream editor." Its behavior is also defined by the POSIX standard.

normal field splitting based on FS does not work well in this case. Although a portable *awk* program can use a series of substr calls on $0 (see the section “String-Manipulation Functions” in Chapter 8, *Functions*), this is awkward and inefficient for a large number of fields.

The splitting of an input record into fixed-width fields is specified by assigning a string containing space-separated numbers to the built-in variable FIELDWIDTHS. Each number specifies the width of the field, *including* columns between fields. If you want to ignore the columns between fields, you can specify the width as a separate field that is subsequently ignored. It is a fatal error to supply a field width that is not a positive number. The following data is the output of the Unix *w* utility. It is useful to illustrate the use of FIELDWIDTHS:

```
10:06pm  up 21 days, 14:04,  23 users
User     tty       login  idle   JCPU   PCPU  what
hzuo     ttyV0     8:58pm            9      5  vi p24.tex
hzang    ttyV3     6:37pm    50                -csh
eklye    ttyV5     9:53pm            7      1  em thes.tex
dportein ttyV6     8:17pm  1:47                -csh
gierd    ttyD3    10:00pm     1                elm
dave     ttyD4     9:47pm            4      4  w
brent    ttyp0    26Jun91  4:46  26:46   4:41  bash
dave     ttyq4    26Jun9115days     46     46  wnewmail
```

The following program takes the above input, converts the idle time to number of seconds, and prints out the first two fields and the calculated idle time:

This program uses a number of *awk* features that haven’t been introduced yet.

```
BEGIN  { FIELDWIDTHS = "9 6 10 6 7 7 35" }
NR > 2 {
    idle = $4
    sub(/^  */, "", idle)   # strip leading spaces
    if (idle == "")
        idle = 0
    if (idle ~ /:/) {
        split(idle, t, ":")
        idle = t[1] * 60 + t[2]
    }
    if (idle ~ /days/)
        idle *= 24 * 60 * 60

    print $1, $2, idle
}
```

Running the program on the data produces the following results:

```
hzuo      ttyV0  0
hzang     ttyV3  50
eklye     ttyV5  0
dportein  ttyV6  107
gierd     ttyD3  1
dave      ttyD4  0
brent     ttyp0  286
dave      ttyq4  1296000
```

Another (possibly more practical) example of fixed-width input data is the input from a deck of balloting cards. In some parts of the United States, voters mark their choices by punching holes in computer cards. These cards are then processed to count the votes for any particular candidate or on any particular issue. Because a voter may choose not to vote on some issue, any column on the card may be empty. An *awk* program for processing such data could use the `FIELDWIDTHS` feature to simplify reading the data. (Of course, getting *gawk* to run on a system with card readers is another story!)

Assigning a value to `FS` causes *gawk* to use `FS` for field splitting again. Use `FS = FS` to make this happen, without having to know the current value of `FS`. In order to tell which kind of field splitting is in effect, use `PROCINFO["FS"]` (see the section "Built-in Variables That Convey Information" in Chapter 6). The value is `"FS"` if regular field splitting is being used, or it is `"FIELDWIDTHS"` if fixed-width field splitting is being used:

```
if (PROCINFO["FS"] == "FS")
    regular field splitting ...
else
    fixed-width field splitting ...
```

This information is useful when writing a function that needs to temporarily change `FS` or `FIELDWIDTHS`, read some records, and then restore the original settings (see the section "Reading the User Database" in Chapter 12, *A Library of awk Functions*, for an example of such a function).

Multiple-Line Records

In some databases, a single line cannot conveniently hold all the information in one entry. In such cases, you can use multiline records. The first step in doing this is to choose your data format.

One technique is to use an unusual character or string to separate records. For example, you could use the formfeed character (written `\f` in *awk*, as in C) to separate them, making each record a page of the file. To do this, just set the variable `RS` to `"\f"` (a string containing the formfeed character). Any other character could equally well be used, as long as it won't be part of the data in a record.

Another technique is to have blank lines separate records. By a special dispensation, an empty string as the value of `RS` indicates that records are separated by one or more blank lines. When `RS` is set to the empty string, each record always ends at the first blank line encountered. The next record doesn't start until the first nonblank line that follows. No matter how many blank lines appear in a row, they all act as one record separator. (Blank lines must be completely empty; lines that contain only whitespace do not count.)

You can achieve the same effect as `RS = ""` by assigning the string `"\n\n+"` to `RS`. This regexp matches the newline at the end of the record and one or more blank lines after the record. In addition, a regular expression always matches the longest possible sequence when there is a choice (see the section "How Much Text Matches?" in Chapter 2). So the next record doesn't start until the first nonblank line that follows—no matter how many blank lines appear in a row, they are considered one record separator.

There is an important difference between `RS = ""` and `RS = "\n\n+"`. In the first case, leading newlines in the input datafile are ignored, and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done. (d.c.)

Now that the input is separated into records, the second step is to separate the fields in the record. One way to do this is to divide each of the lines into fields in the normal manner. This happens by default as the result of a special feature. When `RS` is set to the empty string, the newline character *always* acts as a field separator. This is in addition to whatever field separations result from `FS`.

The original motivation for this special exception was probably to provide useful behavior in the default case (i.e., `FS` is equal to `" "`). This feature can be a problem if you really don't want the newline character to separate fields, because there is no way to prevent it. However, you can work around this by using the `split` function to break up the record manually (see the section "String-Manipulation Functions" in Chapter 8).

Another way to separate fields is to put each field on a separate line: to do this, just set the variable `FS` to the string `"\n"`. (This simple regular expression matches a single newline.) A practical example of a datafile organized this way might be a mailing list, where each entry is separated by blank lines. Consider a mailing list in a file named *addresses*, which looks like this:

```
Jane Doe
123 Main Street
Anywhere, SE 12345-6789
```

```
John Smith
456 Tree-lined Avenue
Smallville, MW 98765-4321
...
```

A simple program to process this file is as follows:

```
# addrs.awk --- simple mailing list program

# Records are separated by blank lines.
# Each line is one field.
BEGIN { RS = "" ; FS = "\n" }

{
    print "Name is:", $1
    print "Address is:", $2
    print "City and State are:", $3
    print ""
}
```

Running the program produces the following output:

```
$ awk -f addrs.awk addresses
Name is: Jane Doe
Address is: 123 Main Street
City and State are: Anywhere, SE 12345-6789

Name is: John Smith
Address is: 456 Tree-lined Avenue
City and State are: Smallville, MW 98765-4321

...
```

See the section "Printing Mailing Labels" in Chapter 13 for a more realistic program that deals with address lists. The following list summarizes how records are split, based on the value of `RS`:

`RS == "\n"`
: Records are separated by the newline character (`\n`). In effect, every line in the datafile is a separate record, including blank lines. This is the default.

`RS ==` *`any single character`*
: Records are separated by each occurrence of the character. Multiple successive occurrences delimit empty records.

`RS == ""`
: Records are separated by runs of blank lines. The newline character always serves as a field separator, in addition to whatever value `FS` may have. Leading and trailing newlines in a file are ignored.

RS == *regexp*

Records are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty records. (This is a *gawk* extension it is not specified by the POSIX standard.)

In all cases, *gawk* sets RT to the input text that matched the value specified by RS.

Explicit Input with getline

So far we have been getting our input data from *awk*'s main input stream—either the standard input (usually your terminal, sometimes the output from another program) or from the files specified on the command line. The *awk* language has a special built-in command called getline that can be used to read input under your explicit control.

The getline command is used in several different ways and should *not* be used by beginners. The examples that follow the explanation of the getline command include material that has not been covered yet. Therefore, come back and study the getline command *after* you have reviewed the rest of this book and have a good knowledge of how *awk* works.

The getline command returns one if it finds a record and zero if it encounters the end of the file. If there is some error in getting a record, such as a file that cannot be opened, then getline returns −1. In this case, *gawk* sets the variable ERRNO to a string describing the error that occurred.

In the following examples, command stands for a string value that represents a shell command.

Using getline with No Arguments

The getline command can be used without arguments to read input from the current input file. All it does in this case is read the next input record and split it up into fields. This is useful if you've finished processing the current record, but want to do some special processing on the next record *right now*. For example:

```
{
    if ((t = index($0, "/*")) != 0) {
        # value of `tmp' will be "" if t is 1
        tmp = substr($0, 1, t - 1)
        u = index(substr($0, t + 2), "*/")
        while (u == 0) {
            if (getline <= 0) {
                m = "unexpected EOF or error"
                m = (m ": " ERRNO)
                print m > "/dev/stderr"
                exit
            }
            t = -1
            u = index($0, "*/")
        }
        # substr expression will be "" if */
        # occurred at end of line
        $0 = tmp substr($0, u + 2)
    }
    print $0
}
```

This *awk* program deletes all C-style comments (`/* ... */`) from the input. By replacing the `print $0` with other statements, you could perform more complicated processing on the decommented input, such as searching for matches of a regular expression. (This program has a subtle problem—it does not work if one comment ends and another begins on the same line.)

This form of the `getline` command sets `NF`, `NR`, `FNR`, and the value of `$0`.

The new value of `$0` is used to test the patterns of any subsequent rules. The original value of `$0` that triggered the rule that executed `getline` is lost. By contrast, the `next` statement reads a new record but immediately begins processing it normally, starting with the first rule in the program. See the section "The next Statement" in Chapter 6.

Using getline into a Variable

You can use `getline` *`var`* to read the next record from *awk*'s input into the variable *var*. No other processing is done. For example, suppose the next line is a comment or a special string, and you want to read it without triggering any rules. This form of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of *awk* never sees it. The following example swaps every two lines of input:

```
{
    if ((getline tmp) > 0) {
        print tmp
        print $0
    } else
        print $0
}
```

It takes the following list:

```
wan
tew
free
phore
```

and produces these results:

```
tew
wan
phore
free
```

The `getline` command used in this way sets only the variables `NR` and `FNR` (and of course, *var*). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

Using getline from a File

Use `getline < file` to read the next record from *file*. Here *file* is a string-valued expression that specifies the filename. `< file` is called a *redirection* because it directs input to come from a different place. For example, the following program reads its input record from the file *secondary.input* when it encounters a first field with a value equal to 10 in the current input file:

```
{
    if ($1 == 10) {
        getline < "secondary.input"
        print
    } else
        print
}
```

Because the main input stream is not used, the values of `NR` and `FNR` are not changed. However, the record it reads is split into fields in the normal manner, so the values of `$0` and the other fields are changed, resulting in a new value of `NF`.

According to POSIX, `getline < expression` is ambiguous if *expression* contains unparenthesized operators other than `$`; for example, `getline < dir "/" file` is ambiguous because the concatenation operator is not parenthesized. You should write it as `getline < (dir "/" file)` if you want your program to be portable to other *awk* implementations. (It happens that *gawk* gets it right, but you should not rely on this. Parentheses make it easier to read.)

Using getline into a Variable from a File

Use `getline` *`var`* `<` *`file`* to read input from the file *file*, and put it in the variable *var*. As above, *file* is a string-valued expression that specifies the file from which to read.

In this version of `getline`, none of the built-in variables are changed and the record is not split into fields. The only variable changed is *var*. For example, the following program copies all the input files to the output, except for records that say `@include` *`filename`*. Such a record is replaced by the contents of the file *filename*:

```
{
    if (NF == 2 && $1 == "@include") {
        while ((getline line < $2) > 0)
            print line
        close($2)
    } else
        print
}
```

Note here how the name of the extra input file is not built into the program; it is taken directly from the data, specifically from the second field on the `@include` line.

The `close` function is called to ensure that if two identical `@include` lines appear in the input, the entire specified file is included twice. See the section "Closing Input and Output Redirections" in Chapter 4.

One deficiency of this program is that it does not process nested `@include` statements (i.e., `@include` statements in included files) the way a true macro preprocessor would. See the section "An Easy Way to Use Library Functions" in Chapter 13 for a program that does handle nested `@include` statements.

Using getline from a Pipe

The output of a command can also be piped into `getline`, using *`command`* `| getline`. In this case, the string *command* is run as a shell command and its output is piped into *awk* to be used as input. This form of `getline` reads one record at a time from the pipe. For example, the following program copies its input to its output, except for lines that begin with `@execute`, which are replaced by the output produced by running the rest of the line as a shell command:

```
{
    if ($1 == "@execute") {
        tmp = substr($0, 10)
        while ((tmp | getline) > 0)
            print
        close(tmp)
    } else
        print
}
```

The close function is called to ensure that if two identical @execute lines appear in the input, the command is run for each one. See the section "Closing Input and Output Redirections" in Chapter 4. Given the input:

```
foo
bar
baz
@execute who
bletch
```

the program might produce:

```
foo
bar
baz
arnold     ttyv0   Jul 13 14:22
miriam     ttyp0   Jul 13 14:23     (murphy:0)
bill       ttyp1   Jul 13 14:23     (murphy:0)
bletch
```

Notice that this program ran the command *who* and printed the previous result. (If you try this program yourself, you will of course get different results, depending upon who is logged in on your system.)

This variation of getline splits the record into fields, sets the value of NF, and recomputes the value of $0. The values of NR and FNR are not changed.

According to POSIX, *expression* | getline is ambiguous if *expression* contains unparenthesized operators other than $—for example, "echo " "date" | getline is ambiguous because the concatenation operator is not parenthesized. You should write it as ("echo " "date") | getline if you want your program to be portable to other *awk* implementations.

Using getline into a Variable from a Pipe

When you use *command* | getline *var*, the output of *command* is sent through a pipe to getline and into the variable *var*. For example, the following program reads the current date and time into the variable current_time, using the *date* utility, and then prints it:

```
BEGIN {
    "date" | getline current_time
    close("date")
    print "Report printed on " current_time
}
```

In this version of getline, none of the built-in variables are changed and the record is not split into fields.

Using getline from a Coprocess

Input into getline from a pipe is a one-way operation. The command that is started with *command* | getline only sends data *to* your *awk* program.

On occasion, you might want to send data to another program for processing and then read the results back. *gawk* allows you start a *coprocess*, with which two-way communications are possible. This is done with the |& operator. Typically, you write data to the coprocess first and then read results back, as shown in the following:

```
print "some query" |& "db_server"
"db_server" |& getline
```

which sends a query to *db_server* and then reads the results.

The values of NR and FNR are not changed, because the main input stream is not used. However, the record is split into fields in the normal manner, thus changing the values of $0, of the other fields, and of NF.

Coprocesses are an advanced feature. They are discussed here only because this is the section on getline. See the section "Two-Way Communications with Another Process" in Chapter 10, *Advanced Features of gawk*, where coprocesses are discussed in more detail.

Using getline into a Variable from a Coprocess

When you use *command* |& getline *var*, the output from the coprocess *command* is sent through a two-way pipe to getline and into the variable *var*.

In this version of getline, none of the built-in variables are changed and the record is not split into fields. The only variable changed is *var*.

Points to Remember About getline

Here are some miscellaneous points about getline that you should bear in mind:

- When getline changes the value of $0 and NF, *awk* does *not* automatically jump to the start of the program and start testing the new record against every pattern. However, the new record is tested against any subsequent rules.
- Many *awk* implementations limit the number of pipelines that an *awk* program may have open to just one. In *gawk*, there is no such limit. You can open as many pipelines (and coprocesses) as the underlying operating system permits.

- An interesting side effect occurs if you use `getline` without a redirection inside a `BEGIN` rule. Because an unredirected `getline` reads from the command-line datafiles, the first `getline` command causes *awk* to set the value of `FILENAME`. Normally, `FILENAME` does not have a value inside `BEGIN` rules, because you have not yet started to process the command-line datafiles. (d.c.) (See the section "The BEGIN and END Special Patterns" in Chapter 6; also see the section "Built-in Variables That Convey Information" in Chapter 6.)

Summary of getline Variants

Table 3-1 summarizes the eight variants of `getline`, listing which built-in variables are set by each one.

Table 3-1. getline Variants and What They Set

Variant	Effect
`getline`	Sets `$0`, `NF`, `FNR`, and `NR`
`getline` *`var`*	Sets *var*, `FNR`, and `NR`
`getline <` *`file`*	Sets `$0` and `NF`
`getline` *`var`* `<` *`file`*	Sets *var*
`command` `| getline`	Sets `$0` and `NF`
`command` `| getline` *`var`*	Sets *var*
`command` `|& getline`	Sets `$0` and `NF`[a]
`command` `|& getline` *`var`*	Sets *var*[a]

[a] This is a *gawk* extension.