4

Printing Output

In this chapter:

One of the most common programming actions is to *print*, or output, some or all of the input. Use the `print` statement for simple output, and the `printf` statement for fancier formatting. The `print` statement is not limited when computing *which* values to print. However, with two exceptions, you cannot specify *how* to print them—how many columns, whether to use exponential notation or not, and so on. (For the exceptions, see the section "Output Separators" and the section "Controlling Numeric Output with print" later in this chapter.) For printing with specifications, you need the `printf` statement (see the section "Using printf Statements for Fancier Printing" later in this chapter).

Besides basic and formatted printing, this chapter also covers I/O redirections to files and pipes, introduces the special filenames that *gawk* processes internally, and discusses the `close` built-in function.

The print Statement

The `print` statement is used to produce output with simple, standardized formatting. Specify only the strings or numbers to print, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:

```
print item1, item2, ...
```

The entire list of items may be optionally enclosed in parentheses. The parentheses are necessary if any of the item expressions uses the > relational operator;

otherwise, it could be confused with a redirection (see the section "Redirecting Output of print and printf" later in this chapter).

The items to print can be constant strings or numbers, fields of the current record (such as `$1`), variables, or any *awk* expression. Numeric values are converted to strings and then printed.

The simple statement `print` with no items is equivalent to `print $0`: it prints the entire current record. To print a blank line, use `print ""`, where `""` is the empty string. To print a fixed piece of text, use a string constant, such as `"Don't Panic"`, as one item. If you forget to use the double-quote characters, your text is taken as an *awk* expression, and you will probably get an error. Keep in mind that a space is printed between any two items.

Examples of print Statements

Each `print` statement makes at least one line of output. However, it isn't limited to only one line. If an item value is a string that contains a newline, the newline is output along with the rest of the string. A single `print` statement can make any number of lines this way.

The following is an example of printing a string that contains embedded newlines (the `\n` is an escape sequence, used to represent the newline character; see the section "Escape Sequences" in Chapter 2, *Regular Expressions*):

```
$ awk 'BEGIN { print "line one\nline two\nline three" }'
line one
line two
line three
```

The next example, which is run on the *inventory-shipped* file, prints the first two fields of each input record, with a space between them:

```
$ awk '{ print $1, $2 }' inventory-shipped
Jan 13
Feb 15
Mar 15
...
```

A common mistake in using the `print` statement is to omit the comma between two items. This often has the effect of making the items run together in the output, with no space. The reason for this is that juxtaposing two string expressions in *awk* means to concatenate them. Here is the same program, without the comma:

```
$ awk '{ print $1 $2 }' inventory-shipped
Jan13
Feb15
Mar15
...
```

To someone unfamiliar with the *inventory-shipped* file, neither example's output makes much sense. A heading line at the beginning would make it clearer. Let's add some headings to our table of months (`$1`) and green crates shipped (`$2`). We do this using the `BEGIN` pattern (see the section "The BEGIN and END Special Patterns" in Chapter 6, *Patterns, Actions, and Variables*) so that the headings are only printed once:

```
awk 'BEGIN {  print "Month Crates"
              print "----- ------" }
           {  print $1, $2 }' inventory-shipped
```

When run, the program prints the following:

```
Month Crates
----- ------
Jan 13
Feb 15
Mar 15
...
```

The only problem, however, is that the headings and the table data don't line up! We can fix this by printing some spaces between the two fields:

```
awk 'BEGIN { print "Month Crates"
             print "----- ------" }
           { print $1, "     ", $2 }' inventory-shipped
```

Lining up columns this way can get pretty complicated when there are many columns to fix. Counting spaces for two or three columns is simple, but any more than this can take up a lot of time. This is why the `printf` statement was created (see the section "Using printf Statements for Fancier Printing" later in this chapter); one of its specialties is lining up columns of data.

You can continue either a `print` or `printf` statement simply by putting a newline after any comma (see the section "awk Statements Versus Lines" in Chapter 1, *Getting Started with awk*).

Output Separators

As mentioned previously, a `print` statement contains a list of items separated by commas. In the output, the items are normally separated by single spaces. However, this doesn't need to be the case; a single space is only the default. Any string of characters may be used as the *output field separator* by setting the built-in variable `OFS`. The initial value of this variable is the string `" "`—that is, a single space.

The output from an entire `print` statement is called an *output record*. Each `print` statement outputs one output record, and then outputs a string called the *output record separator* (or `ORS`). The initial value of `ORS` is the string `"\n"`; i.e., a newline character. Thus, each `print` statement normally makes a separate line.

In order to change how output fields and records are separated, assign new values to the variables `OFS` and `ORS`. The usual place to do this is in the `BEGIN` rule (see the section "The BEGIN and END Special Patterns" in Chapter 6), so that it happens before any input is processed. It can also be done with assignments on the command line, before the names of the input files, or using the *–v* command-line option (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*). The following example prints the first and second fields of each input record, separated by a semicolon, with a blank line added after each newline:

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
>             { print $1, $2 }' BBS-list
aardvark;555-5553

alpo-net;555-3412

barfly;555-7685
...
```

If the value of `ORS` does not contain a newline, the program's output is run together on a single line.

Controlling Numeric Output with print

When the `print` statement is used to print numeric values, *awk* internally converts the number to a string of characters and prints that string. *awk* uses the `sprintf` function to do this conversion (see the section "String-Manipulation Functions" in Chapter 8, *Functions*). For now, it suffices to say that the `sprintf` function accepts a *format specification* that tells it how to format numbers (or strings), and that there are a number of different ways in which numbers can be formatted. The different format specifications are discussed more fully in the section "Format-Control Letters" later in this chapter.

The built-in variable `OFMT` contains the default format specification that `print` uses with `sprintf` when it wants to convert a number to a string for printing. The default value of `OFMT` is `"%.6g"`. The way `print` prints numbers can be changed by supplying different format specifications as the value of `OFMT`, as shown in the following example:

```
$ awk 'BEGIN {
>   OFMT = "%.0f"  # print numbers as integers (rounds)
>   print 17.23, 17.54 }'
17 18
```

According to the POSIX standard, *awk*'s behavior is undefined if `OFMT` contains anything but a floating-point conversion specification. (d.c.)

Using printf Statements for Fancier Printing

For more precise control over the output format than what is normally provided by `print`, use `printf`. `printf` can be used to specify the width to use for each item, as well as various formatting choices for numbers (such as what output base to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point). This is done by supplying a string, called the *format string*, that controls how and where to print the other arguments.

Introduction to the printf Statement

A simple `printf` statement looks like this:

```
printf format, item1, item2, ...
```

The entire list of arguments may optionally be enclosed in parentheses. The parentheses are necessary if any of the item expressions use the `>` relational operator; otherwise, it can be confused with a redirection (see the section "Redirecting Output of print and printf" later in this chapter).

The difference between `printf` and `print` is the *format* argument. This is an expression whose value is taken as a string; it specifies how to output each of the other arguments. It is called the *format string*.

The format string is very similar to that in the ISO C library function `printf`. Most of *format* is text to output verbatim. Scattered among this text are *format specifiers*—one per item. Each format specifier says to output the next item in the argument list at that place in the format.

The `printf` statement does not automatically append a newline to its output. It outputs only what the format string specifies. So if a newline is needed, you must include one in the format string. The output separator variables `OFS` and `ORS` have no effect on `printf` statements. For example:

```
$ awk 'BEGIN {
>    ORS = "\nOUCH!\n"; OFS = "+"
>    msg = "Dont Panic!"
>    printf "%s\n", msg
> }'
Dont Panic!
```

Here, neither the `+` nor the `OUCH!` appear when the message is printed.

Format-Control Letters

A format specifier starts with the character `%` and ends with a *format-control letter*—it tells the `printf` statement how to output one item. The format-control letter specifies what *kind* of value to print. The rest of the format specifier is made up of optional *modifiers* that control *how* to print the value, such as the field width. Here is a list of the format-control letters:

`%c` This prints a number as an ASCII character; thus, `printf "%c", 65` outputs the letter `A`. (The output for a string value is the first character of the string.)

`%d`, `%i`
These are equivalent; they both print a decimal integer. (The `%i` specification is for compatibility with ISO C.)

`%e`, `%E`
These print a number in scientific (exponential) notation; for example:

```
printf "%4.3e\n", 1950
```

prints `1.950e+03`, with a total of four significant figures, three of which follow the decimal point. (The `4.3` represents two modifiers, discussed in the next section.) `%E` uses `E` instead of `e` in the output.

`%f` This prints a number in floating-point notation. For example:

```
printf "%4.3f", 1950
```

prints `1950.000`, with a total of four significant figures, three of which follow the decimal point. (The `4.3` represents two modifiers, discussed in the next section.)

`%g`, `%G`
These print a number in either scientific notation or in floating-point notation, whichever uses fewer characters; if the result is printed in scientific notation, `%G` uses `E` instead of `e`.

`%o` This prints an unsigned octal integer.

`%s` This prints a string.

`%u` This prints an unsigned decimal integer. (This format is of marginal use, because all numbers in *awk* are floating-point; it is provided primarily for compatibility with C.)

`%x`, `%X`
These print an unsigned hexadecimal integer; `%X` uses the letters `A` through `F` instead of `a` through `f`.

`%%` This isn't a format-control letter, but it does have meaning—the sequence `%%` outputs one `%`; it does not consume an argument and it ignores any modifiers.

When using the integer format-control letters for values that are outside the range of a C `long` integer, *gawk* switches to the `%g` format specifier. Other versions of *awk* may print invalid values or do something else entirely. (d.c.)

Modifiers for printf Formats

A format specification can also include *modifiers* that can control how much of the item's value is printed, as well as how much space it gets. The modifiers come between the `%` and the format-control letter. We will use the small box "□" in the following examples to represent spaces in the output. Here are the possible modifiers, in the order in which they may appear:

N`$` An integer constant followed by a `$` is a *positional specifier*. Normally, format specifications are applied to arguments in the order given in the format string. With a positional specifier, the format specification is applied to a specific argument, instead of what would be the next argument in the list. Positional specifiers begin counting with one. Thus:

```
printf "%s %s\n", "don't", "panic"
printf "%2$s %1$s\n", "panic", "don't"
```

prints the famous friendly message twice.

At first glance, this feature doesn't seem to be of much use. It is in fact a *gawk* extension, intended for use in translating messages at runtime. See the section "Rearranging printf Arguments" in Chapter 9, *Internationalization with gawk*, which describes how and why to use positional specifiers. For now, we will not use them.

`-` The minus sign, used before the *width* modifier (see later in this list), says to left-justify the argument within its specified width. Normally, the argument is printed right-justified in the specified width. Thus:

```
printf "%-4s", "foo"
```

prints `foo□`.

space

For numeric conversions, prefix positive values with a space and negative values with a minus sign.

`+` The plus sign, used before the *width* modifier (see later in this list), says to always supply a sign for numeric conversions, even if the data to format is positive. The `+` overrides the space modifier.

`#` Use an "alternate form" for certain control letters. For `%o`, supply a leading zero. For `%x` and `%X`, supply a leading `0x` or `0X` for a nonzero result. For `%e`, `%E`, and `%f`, the result always contains a decimal point. For `%g` and `%G`, trailing zeros are not removed from the result.

`0` A leading `0` (zero) acts as a flag that indicates that output should be padded with zeros instead of spaces. This applies even to non-numeric output formats. (d.c.) This flag only has an effect when the field width is wider than the value to print.

width

width is a number specifying the desired minimum width of a field. Inserting any number between the `%` sign and the format-control character forces the field to expand to this width. The default way to do this is to pad with spaces on the left. For example:

```
printf "%4s", "foo"
```

prints `□foo`.

The value of *width* is a minimum width, not a maximum. If the item value requires more than *width* characters, it can be as wide as necessary. Thus, the following:

```
printf "%4s", "foobar"
```

prints `foobar`.

Preceding the *width* with a minus sign causes the output to be padded with spaces on the right, instead of on the left.

`.`*prec*

A period followed by an integer constant specifies the precision to use when printing. The meaning of the precision varies by control letter:

`%e`, `%E`, `%f`

Number of digits to the right of the decimal point.

`%g`, `%G`

Maximum number of significant digits.

`%d`, `%i`, `%o`, `%u`, `%x`, `%X`
Minimum number of digits to print.

`%s` Maximum number of characters from the string that should print.

Thus, the following:

```
    printf "%.4s", "foobar"
```

prints `foob`.

The C library `printf`'s dynamic *width* and *prec* capability (for example, `"%*.*s"`) is supported. Instead of supplying explicit *width* and/or *prec* values in the format string, they are passed in the argument list. For example:

```
w = 5
p = 3
s = "abcdefg"
printf "%*.*s\n", w, p, s
```

is exactly equivalent to:

```
s = "abcdefg"
printf "%5.3s\n", s
```

Both programs output `␣␣abc`. Earlier versions of *awk* did not support this capability. If you must use such a version, you may simulate this feature by using concatenation to build up the format string, like so:

```
w = 5
p = 3
s = "abcdefg"
printf "%" w "." p "s\n", s
```

This is not particularly easy to read but it does work.

C programmers may be used to supplying additional `l`, `L`, and `h` modifiers in `printf` format strings. These are not valid in *awk*. Most *awk* implementations silently ignore these modifiers. If *––lint* is provided on the command line (see the section "Command-Line Options" in Chapter 11), *gawk* warns about their use. If *––posix* is supplied, their use is a fatal error.

Examples Using printf

The following is a simple example of how to use `printf` to make an aligned table:

```
awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
```

This command prints the names of the bulletin boards (`$1`) in the file *BBS-list* as a string of 10 characters that are left-justified. It also prints the phone numbers (`$2`) next on the line. This produces an aligned two-column table of names and phone numbers, as shown here:

```
$ awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
aardvark   555-5553
alpo-net   555-3412
barfly     555-7685
bites      555-1675
camelot    555-0542
core       555-2912
fooey      555-1234
foot       555-6699
macfoo     555-6480
sdace      555-3430
sabafoo    555-2127
```

In this case, the phone numbers had to be printed as strings because the numbers are separated by a dash. Printing the phone numbers as numbers would have produced just the first three digits: `555`. This would have been pretty confusing.

It wasn't necessary to specify a width for the phone numbers because they are last on their lines. They don't need to have spaces after them.

The table could be made to look even nicer by adding headings to the tops of the columns. This is done using the `BEGIN` pattern (see the section "The BEGIN and END Special Patterns" in Chapter 6) so that the headers are only printed once, at the beginning of the *awk* program:

```
awk 'BEGIN { print "Name      Number"
             print "----      ------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

The above example mixed `print` and `printf` statements in the same program. Using just `printf` statements can produce the same results:

```
awk 'BEGIN { printf "%-10s %s\n", "Name", "Number"
             printf "%-10s %s\n", "----", "------" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Printing each column heading with the same format specification used for the column elements ensures that the headings are aligned just like the columns.

The fact that the same format specification is used three times can be emphasized by storing it in a variable, like this:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Name", "Number"
             printf format, "----", "------" }
     { printf format, $1, $2 }' BBS-list
```

At this point, it would be a worthwhile exercise to use the `printf` statement to line up the headings and table data for the *inventory-shipped* example that was covered earlier in the section on the `print` statement (see the section "The print Statement" earlier in this chapter).

Redirecting Output of print and printf

So far, the output from `print` and `printf` has gone to the standard output, usually the terminal. Both `print` and `printf` can also send their output to other places. This is called *redirection*.

A redirection appears after the `print` or `printf` statement. Redirections in *awk* are written just like redirections in shell commands, except that they are written inside the *awk* program.

There are four forms of output redirection: output to a file, output appended to a file, output through a pipe to another command, and output to a coprocess. They are all shown for the `print` statement, but they work identically for `printf`:

`print` *`items`* `>` *`output-file`*

This type of redirection prints the items into the output file named *output-file*. The filename *output-file* can be any expression. Its value is changed to a string and then used as a filename (see Chapter 5, *Expressions*).

When this type of redirection is used, the *output-file* is erased before the first output is written to it. Subsequent writes to the same *output-file* do not erase *output-file*, but append to it. (This is different from how you use redirections in shell scripts.) If *output-file* does not exist, it is created. For example, here is how an *awk* program can write a list of BBS names to one file named *name-list*, and a list of phone numbers to another file named *phone-list*:

```
$ awk '{ print $2 > "phone-list"
>        print $1 > "name-list" }' BBS-list
$ cat phone-list
555-5553
555-3412
...
$ cat name-list
aardvark
alpo-net
...
```

Each output file contains one name or number per line.

`print` *`items`* `>>` *`output-file`*

This type of redirection prints the items into the pre-existing output file named *output-file*. The difference between this and the single-> redirection is that the old contents (if any) of *output-file* are not erased. Instead, the *awk* output is appended to the file. If *output-file* does not exist, then it is created.

`print` *`items`* `|` *`command`*

It is also possible to send output to a program through a pipe instead of into a file. This type of redirection opens a pipe to *command* and writes the values of *items* through this pipe to a process created to execute *command*.

The redirection argument *command* is actually an *awk* expression. Its value is converted to a string whose contents give the shell command to be run. For example, the following produces two files, one unsorted list of BBS names, and one list sorted in reverse alphabetical order:

```
awk '{ print $1 > "names.unsorted"
       command = "sort -r > names.sorted"
       print $1 | command }' BBS-list
```

The unsorted list is written with an ordinary redirection, while the sorted list is written by piping through the *sort* utility.

The next example uses redirection to mail a message to the mailing list `bug-system`. This might be useful when trouble is encountered in an *awk* script run periodically for system maintenance:

```
report = "mail bug-system"
print "Awk script failed:", $0 | report
m = ("at record number " FNR " of " FILENAME)
print m | report
close(report)
```

The message is built using string concatenation and saved in the variable `m`. It's then sent down the pipeline to the *mail* program. (The parentheses group the items to concatenate—see the section "String Concatenation" in Chapter 5.)

The `close` function is called here because it's a good idea to close the pipe as soon as all the intended output has been sent to it. See the section "Closing Input and Output Redirections" later in this chapter for more information.

This example also illustrates the use of a variable to represent a *file* or *command*—it is not necessary to always use a string constant. Using a variable is generally a good idea, because *awk* requires that the string value be spelled identically every time.

`print` *`items`* `|&` *`command`*

This type of redirection prints the items to the input of *command*. The difference between this and the single-`|` redirection is that the output from *command* can be read with `getline`. Thus *command* is a coprocess, which works together with, but subsidiary to, the *awk* program.

This feature is a *gawk* extension, and is not available in POSIX *awk*. See the section "Two-Way Communications with Another Process" in Chapter 10, *Advanced Features of gawk*, for a more complete discussion.

Redirecting output using `>`, `>>`, `|`, or `|&` asks the system to open a file, pipe, or coprocess only if the particular *file* or *command* you specify has not already been written to by your program or if it has been closed since it was last written to.

It is a common error to use > redirection for the first `print` to a file, and then to use >> for subsequent output:

```
# clear the file
print "Don't panic" > "guide.txt"
...
# append
print "Avoid improbability generators" >> "guide.txt"
```

This is indeed how redirections must be used from the shell. But in *awk*, it isn't necessary. In this kind of case, a program should use > for all the `print` statements, since the output file is only opened once.

As mentioned earlier (see the section "Points to Remember About getline" in Chapter 3, *Reading Input Files*), 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. *gawk* allows a program to open as many pipelines as the underlying operating system permits.

Piping into sh

A particularly powerful way to use redirection is to build command lines and pipe them into the shell, *sh*. For example, suppose you have a list of files brought over from a system where all the filenames are stored in uppercase, and you wish to rename them to have names in all lowercase. The following program is both simple and efficient:

```
{ printf("mv %s %s\n", $0, tolower($0)) | "sh" }

END { close("sh") }
```

The `tolower` function returns its argument string with all uppercase characters converted to lowercase (see the section "String-Manipulation Functions" in Chapter 8). The program builds up a list of command lines, using the *mv* utility to rename the files. It then sends the list to the shell for execution.

Special Filenames in gawk

gawk provides a number of special filenames that it interprets internally. These filenames provide access to standard file descriptors, process-related information, and TCP/IP networking.

Special Files for Standard Descriptors

Running programs conventionally have three input and output streams already available to them for reading and writing. These are known as the *standard input*, *standard output*, and *standard error output*. These streams are, by default, connected to your terminal, but they are often redirected with the shell, via the `<`, `<<`, `>`, `>>`, `>&`, and `|` operators. Standard error is typically used for writing error messages; the reason there are two separate streams, standard output and standard error, is so that they can be redirected separately.

In other implementations of *awk*, the only way to write an error message to standard error in an *awk* program is as follows:

```
print "Serious error detected!" | "cat 1>&2"
```

This works by opening a pipeline to a shell command that can access the standard error stream that it inherits from the *awk* process. This is far from elegant, and it is also inefficient, because it requires a separate process. So people writing *awk* programs often don't do this. Instead, they send the error messages to the terminal, like this:

```
print "Serious error detected!" > "/dev/tty"
```

This usually has the same effect but not always: although the standard error stream is usually the terminal, it can be redirected; when that happens, writing to the terminal is not correct. In fact, if *awk* is run from a background job, it may not have a terminal at all. Then opening */dev/tty* fails.

gawk provides special filenames for accessing the three standard streams, as well as any other inherited open files. If the filename matches one of these special names when *gawk* redirects input or output, then it directly uses the stream that the filename stands for. These special filenames work for all operating systems that *gawk* has been ported to, not just those that are POSIX-compliant:

/dev/stdin
: The standard input (file descriptor 0).

/dev/stdout
: The standard output (file descriptor 1).

/dev/stderr
: The standard error output (file descriptor 2).

/dev/fd/N
: The file associated with file descriptor *N*. Such a file must be opened by the program initiating the *awk* execution (typically the shell). Unless special pains are taken in the shell from which *gawk* is invoked, only descriptors 0, 1, and 2 are available.

The filenames */dev/stdin*, */dev/stdout*, and */dev/stderr* are aliases for */dev/fd/0*, */dev/fd/1*, and */dev/fd/2*, respectively. However, they are more self-explanatory. The proper way to write an error message in a *gawk* program is to use */dev/stderr*, like this:

```
print "Serious error detected!" > "/dev/stderr"
```

Note the use of quotes around the filename. Like any other redirection, the value must be a string. It is a common error to omit the quotes, which leads to confusing results.

Special Files for Process-Related Information

gawk also provides special filenames that give access to information about the running *gawk* process. Each of these "files" provides a single record of information. To read them more than once, they must first be closed with the `close` function (see the section "Closing Input and Output Redirections" later in this chapter). The filenames are:

/dev/pid
: Reading this file returns the process ID of the current process, in decimal form, terminated with a newline.

/dev/ppid
: Reading this file returns the parent process ID of the current process, in decimal form, terminated with a newline.

/dev/pgrpid
: Reading this file returns the process group ID of the current process, in decimal form, terminated with a newline.

/dev/user
: Reading this file returns a single record terminated with a newline. The fields are separated with spaces. The fields represent the following information:

  `$1` The return value of the `getuid` system call (the real user ID number).

  `$2` The return value of the `geteuid` system call (the effective user ID number).

  `$3` The return value of the `getgid` system call (the real group ID number).

  `$4` The return value of the `getegid` system call (the effective group ID number).

  If there are any additional fields, they are the group IDs returned by the `getgroups` system call. (Multiple groups may not be supported on all systems.)

These special filenames may be used on the command line as datafiles, as well as for I/O redirections within an *awk* program. They may not be used as source files with the *–f* option.

The special files that provide process-related information are now considered obsolete and will disappear entirely in the next release of *gawk*. *gawk* prints a warning message every time you use one of these files. To obtain process-related information, use the `PROCINFO` array. See section "Built-in Variables That Convey Information" in Chapter 6.

Special Files for Network Communications

Starting with Version 3.1 of *gawk*, *awk* programs can open a two-way TCP/IP connection, acting as either a client or a server. This is done using a special filename of the form:

```
/inet/protocol/local-port/remote-host/remote-port
```

The *protocol* is one of `tcp`, `udp`, or `raw`, and the other fields represent the other essential pieces of information for making a networking connection. These filenames are used with the `|&` operator for communicating with a coprocess (see the section "Two-Way Communications with Another Process" in Chapter 10). This is an advanced feature, mentioned here only for completeness. See Chapter 14, *Internetworking with gawk*, for an in-depth discussion with many examples.

Special Filename Caveats

Here is a list of things to bear in mind when using the special filenames that *gawk* provides:

- Recognition of these special filenames is disabled if *gawk* is in compatibility mode (see the section "Command-Line Options" in Chapter 11).
- As mentioned earlier, the special files that provide process-related information are now considered obsolete and will disappear entirely in the next release of *gawk*. *gawk* prints a warning message every time you use one of these files. To obtain process-related information, use the `PROCINFO` array. See the section "Built-in Variables" in Chapter 6.

- Starting with Version 3.1, *gawk always* interprets these special filenames.* For example, using `/dev/fd/4` for output actually writes on file descriptor 4, and not on a new file descriptor that is `dup`'ed from file descriptor 4. Most of the time this does not matter; however, it is important to *not* close any of the files related to file descriptors 0, 1, and 2. Doing so results in unpredictable behavior.

Closing Input and Output Redirections

If the same filename or the same shell command is used with `getline` more than once during the execution of an *awk* program (see the section "Explicit Input with getline" in Chapter 3), the file is opened (or the command is executed) the first time only. At that time, the first record of input is read from that file or command. The next time the same file or command is used with `getline`, another record is read from it, and so on.

Similarly, when a file or pipe is opened for output, the filename or command associated with it is remembered by *awk*, and subsequent writes to the same file or command are appended to the previous writes. The file or pipe stays open until *awk* exits.

This implies that special steps are necessary in order to read the same file again from the beginning, or to rerun a shell command (rather than reading more output from the same command). The `close` function makes these things possible:

```
close(filename)
```

or:

```
close(command)
```

The argument *filename* or *command* can be any expression. Its value must *exactly* match the string that was used to open the file or start the command (spaces and other "irrelevant" characters included). For example, if you open a pipe with this:

```
"sort -r names" | getline foo
```

then you must close it with this:

```
close("sort -r names")
```

* Older versions of *gawk* would interpret these names internally only if the system did not actually have a a */dev/fd* directory or any of the other special files listed earlier. Usually this didn't make a difference, but sometimes it did; thus, it was decided to make *gawk*'s behavior consistent on all systems and to have it always interpret the special filenames itself.

Once this function call is executed, the next `getline` from that file or command, or the next `print` or `printf` to that file or command, reopens the file or reruns the command. Because the expression that you use to close a file or pipeline must exactly match the expression used to open the file or run the command, it is good practice to use a variable to store the filename or command. The previous example becomes the following:

```
sortcom = "sort -r names"
sortcom | getline foo
...
close(sortcom)
```

This helps avoid hard-to-find typographical errors in your *awk* programs. Here are some of the reasons for closing an output file:

- To write a file and read it back later on in the same *awk* program. Close the file after writing it, then begin reading it with `getline`.
- To write numerous files, successively, in the same *awk* program. If the files aren't closed, eventually *awk* may exceed a system limit on the number of open files in one process. It is best to close each one when the program has finished writing it.
- To make a command finish. When output is redirected through a pipe, the command reading the pipe normally continues to try to read input as long as the pipe is open. Often this means the command cannot really do its work until the pipe is closed. For example, if output is redirected to the *mail* program, the message is not actually sent until the pipe is closed.
- To run the same program a second time, with the same arguments. This is not the same thing as giving more input to the first run! For example, suppose a program pipes output to the *mail* program. If it outputs several lines redirected to this pipe without closing it, they make a single message of several lines. By contrast, if the program closes the pipe after each line of output, then each line makes a separate message.

If you use more files than the system allows you to have open, *gawk* attempts to multiplex the available open files among your datafiles. *gawk*'s ability to do this depends upon the facilities of your operating system, so it may not always work. It is therefore both good practice and good portability advice to always use `close` on your files when you are done with them. In fact, if you are using a lot of pipes, it is essential that you close commands when done. For example, consider something like this:

```
{
    ...
    command = ("grep " $1 " /some/file | my_prog -q " $3)
    while ((command | getline) > 0) {
        process output of command
    }
    # need close(command) here
}
```

Using close's Return Value

In many versions of Unix *awk*, the `close` function is actually a statement. It is a syntax error to try and use the return value from `close`: (d.c.)

```
command = "..."
command | getline info
retval = close(command)  # syntax error in most Unix awks
```

gawk treats `close` as a function. The return value is −1 if the argument names something that was never opened with a redirection, or if there is a system problem closing the file or process. In these cases, *gawk* sets the built-in variable `ERRNO` to a string describing the problem.

In *gawk*, when closing a pipe or coprocess, the return value is the exit status of the command. Otherwise, it is the return value from the system's `close` or `fclose` C functions when closing input or output files, respectively. This value is zero if the close succeeds, or −1 if it fails.

The return value for closing a pipeline is particularly useful. It allows you to get the output from a command as well as its exit status.

For POSIX-compliant systems, if the exit status is a number above 128, then the program was terminated by a signal. Subtract 128 to get the signal number:

```
exit_val = close(command)
if (exit_val > 128)
    print command, "died with signal", exit_val - 128
else
    print command, "exited with code", exit_val
```

Currently, in *gawk*, this only works for commands piping into `getline`. For commands piped into from `print` or `printf`, the return value from `close` is that of the library's `pclose` function.

This example creates a new pipeline based on data in *each* record. Without the call to `close` indicated in the comment, *awk* creates child processes to run the commands, until it eventually runs out of file descriptors for more pipelines.

Even though each command has finished (as indicated by the end-of-file return status from `getline`), the child process is not terminated;[*] more importantly, the file descriptor for the pipe is not closed and released until `close` is called or *awk* exits.

`close` will silently do nothing if given an argument that does not represent a file, pipe or coprocess that was opened with a redirection.

When using the `|&` operator to communicate with a coprocess, it is occasionally useful to be able to close one end of the two-way pipe without closing the other. This is done by supplying a second argument to `close`. As in any other call to `close`, the first argument is the name of the command or special file used to start the coprocess. The second argument should be a string, with either of the values `"to"` or `"from"`. Case does not matter. As this is an advanced feature, a more complete discussion is delayed until the section "Two-Way Communications with Another Process" in Chapter 10, which discusses it in more detail and gives an example.

[*] The technical terminology is rather morbid. The finished child is called a "zombie," and cleaning up after it is referred to as "reaping."