11

Running awk and gawk

In this chapter:
- *Invoking awk*
- *Command-Line Options*
- *Other Command-Line Arguments*
- *The AWKPATH Environment Variable*
- *Obsolete Options and/or Features*
- *Known Bugs in gawk*

This chapter covers how to run *awk*, both POSIX-standard and *gawk*-specific command-line options, and what *awk* and *gawk* do with non-option arguments. It then proceeds to cover how *gawk* searches for source files, obsolete options and/or features, and known bugs in *gawk*. This chapter rounds out the discussion of *awk* as a program and as a language.

While a number of the options and features described here were discussed in passing earlier in the book, this chapter provides the full details.

Invoking awk

There are two ways to run *awk*—with an explicit program or with one or more program files. Here are templates for both of them; items enclosed in [...] in these templates are optional:

```
awk [options] -f progfile [--] file ...
awk [options] [--] 'program' file ...
```

Besides traditional one-letter POSIX-style options, *gawk* also supports GNU long options.

It is possible to invoke *awk* with an empty program:

```
awk '' datafile1 datafile2
```

Doing so makes little sense, though; *awk* exits silently when given an empty program. If *––lint* has been specified on the command line, *gawk* issues a warning that the program is empty.

9 October 2001 01:43

Command-Line Options

Options begin with a dash and consist of a single character. GNU-style long options consist of two dashes and a keyword. The keyword can be abbreviated, as long as the abbreviation allows the option to be uniquely identified. If the option takes an argument, then the keyword is either immediately followed by an equals sign (=) and the argument's value, or the keyword and the argument's value are separated by whitespace. If a particular option with a value is given more than once, it is the last value that counts.

Each long option for *gawk* has a corresponding POSIX-style option. The long and short options are interchangeable in all contexts. The options and their meanings are as follows:

`-F` *`fs`*
`--field-separator` *`fs`*
: Sets the `FS` variable to *fs* (see the section "Specifying How Fields Are Separated" in Chapter 3, *Reading Input Files*).

`-f` *`source-file`*
`--file` *`source-file`*
: Indicates that the *awk* program is to be found in *source-file* instead of in the first non-option argument.

`-v` *`var=val`*
`--assign` *`var=val`*
: Sets the variable *var* to the value *val before* execution of the program begins. Such variable values are available inside the `BEGIN` rule (see the section "Other Command-Line Arguments" later in this chapter).

  The *–v* option can only set one variable, but it can be used more than once, setting another variable each time, like this: `awk -v foo=1 -v bar=2 ...`.

Using *–v* to set the values of the built-in variables may lead to surprising results. *awk* will reset the values of those variables as it needs to, possibly ignoring any predefined value you may have given.

`-mf` *`N`*
`-mr` *`N`*
: Sets various memory limits to the value *N*. The `f` flag sets the maximum number of fields and the `r` flag sets the maximum record size. These two flags and the *–m* option are from the Bell Laboratories research version of Unix *awk*. They are provided for compatibility but otherwise ignored by *gawk*, since

gawk has no predefined limits. (The Bell Laboratories *awk* no longer needs these options; it continues to accept them to avoid breaking old programs.)

`-W` *`gawk-opt`*
: Following the POSIX standard, implementation-specific options are supplied as arguments to the *–W* option. These options also have corresponding GNU-style long options. Note that the long options may be abbreviated, as long as the abbreviations remain unique. The full list of *gawk*-specific options is provided next.

`--` Signals the end of the command-line options. The following arguments are not treated as options even if they begin with `-`. This interpretation of *––* follows the POSIX argument parsing conventions.

  This is useful if you have filenames that start with `-`, or in shell scripts, if you have filenames that will be specified by the user that could start with `-`.

The previous list described options mandated by the POSIX standard, as well as options available in the Bell Laboratories version of *awk*. The following list describes *gawk*-specific options:

`-W compat`, `-W traditional`, `--compat`, `--traditional`
: Specifies *compatibility mode*, in which the GNU extensions to the *awk* language are disabled, so that *gawk* behaves just like the Bell Laboratories research version of Unix *awk*. *––traditional* is the preferred form of this option. See the section "Extensions in gawk Not in POSIX awk" in Appendix A, *The Evolution of the awk Language*, which summarizes the extensions. Also see the section "Downward Compatibility and Debugging" in Appendix C, *Implementation Notes*.

`-W copyright`
`--copyright`
: Print the short version of the General Public License and then exit.

`-W copyleft`
`--copyleft`
: Just like *––copyright*.

`-W dump-variables[=`*`file`*`]`
`--dump-variables[=`*`file`*`]`
: Prints a sorted list of global variables, their types, and final values to *file*. If no *file* is provided, *gawk* prints this list to the file named *awkvars.out* in the current directory.

Having a list of all global variables is a good way to look for typographical errors in your programs. You would also use this option if you have a large program with a lot of functions, and you want to be sure that your functions don't inadvertently use global variables that you meant to be local. (This is a particularly easy mistake to make with simple variable names like `i`, `j`, etc.)

`-W gen-po`
`--gen-po`
Analyzes the source program and generates a GNU `gettext` Portable Object file on standard output for all string constants that have been marked for translation. See Chapter 9, *Internationalization with gawk*, for information about this option.

`-W help`, `-W usage`, `--help`, `--usage`
Prints a "usage" message summarizing the short and long style options that *gawk* accepts and then exits.

`-W lint[=fatal]`
`--lint[=fatal]`
Warn about constructs that are dubious or nonportable to other *awk* implementations. Some warnings are issued when *gawk* first reads your program. Others are issued at runtime, as your program executes. With an optional argument of `fatal`, lint warnings become fatal errors. This may be drastic, but its use will certainly encourage the development of cleaner *awk* programs.

`-W lint-old`
`--lint-old`
Warns about constructs that are not available in the original version of *awk* from Version 7 Unix (see the section "Major Changes Between V7 and SVR3.1" in Appendix A).

`-W non-decimal-data`
`--non-decimal-data`
Enable automatic interpretation of octal and hexadecimal values in input data (see the section "Allowing Nondecimal Input Data" in Chapter 10, *Advanced Features of gawk*).

This option can severely break old programs. Use with care.

`-W posix`
`--posix`
Operates in strict POSIX mode. This disables all *gawk* extensions (just like *--traditional*) and adds the following additional restrictions:

- `\x` escape sequences are not recognized (see the section "Escape Sequences" in Chapter 2, *Regular Expressions*).
- Newlines do not act as whitespace to separate fields when `FS` is equal to a single space (see the section "Examining Fields" in Chapter 3).
- Newlines are not allowed after `?` or `:` (see the section "Conditional Expressions" in Chapter 5, *Expressions*).
- The synonym `func` for the keyword `function` is not recognized (see the section "Function Definition Syntax" in Chapter 8, *Functions*).
- The `**` and `**=` operators cannot be used in place of `^` and `^=` (see the section "Arithmetic Operators" in Chapter 5, and also see the section "Assignment Expressions" in Chapter 5).
- Specifying `-Ft` on the command line does not set the value of `FS` to be a single tab character (see the section "Specifying How Fields Are Separated" in Chapter 3).
- The `fflush` built-in function is not supported (see the section "Input/Output Functions" in Chapter 8).

If you supply both *--traditional* and *--posix* on the command line, *--posix* takes precedence. *gawk* also issues a warning if both options are supplied.

`-W profile[=`*`file`*`]`
`--profile[=`*`file`*`]`
Enable profiling of *awk* programs (see the section "Profiling Your awk Programs" in Chapter 10). By default, profiles are created in a file named *awkprof.out*. The optional *file* argument allows you to specify a different filename for the profile file.

When run with *gawk*, the profile is just a "pretty printed" version of the program. When run with *pgawk*, the profile contains execution counts for each statement in the program in the left margin, and function call counts for each function.

`-W re-interval`
`--re-interval`
Allow interval expressions (see the section "Regular Expression Operators" in Chapter 2) in regexps. Because interval expressions were traditionally not available in *awk*, *gawk* does not provide them by default. This prevents old *awk* programs from breaking.

`-W source` *`program-text`*
`--source` *`program-text`*
: Allows you to mix source code in files with source code that you enter on the command line. Program source code is taken from the *program-text*. This is particularly useful when you have library functions that you want to use from your command-line programs (see the section "The AWKPATH Environment Variable" later in this chapter).

`-W version`
`--version`
: Prints version information for this particular copy of *gawk*. This allows you to determine if your copy of *gawk* is up to date with respect to whatever the Free Software Foundation is currently distributing. It is also useful for bug reports (see the section "Reporting Problems and Bugs" in Appendix B, *Installing gawk*).

As long as program text has been supplied, any other options are flagged as invalid with a warning message but are otherwise ignored.

In compatibility mode, as a special case, if the value of *fs* supplied to the *–F* option is `t`, then `FS` is set to the tab character (`"\t"`). This is true only for *––traditional* and not for *––posix* (see the section "Specifying How Fields Are Separated" in Chapter 3).

The *–f* option may be used more than once on the command line. If it is, *awk* reads its program source from all of the named files, as if they had been concatenated together into one big file. This is useful for creating libraries of *awk* functions. These functions can be written once and then retrieved from a standard place, instead of having to be included into each individual program. (As mentioned in the section "Function Definition Syntax" in Chapter 8, function names must be unique.)

Library functions can still be used, even if the program is entered at the terminal, by specifying `-f /dev/tty`. After typing your program, type Ctrl-d (the end-of-file character) to terminate it. (You may also use `-f -` to read program source from the standard input but then you will not be able to also use the standard input as a source of data.)

Because it is clumsy using the standard *awk* mechanisms to mix source file and command-line *awk* programs, *gawk* provides the *––source* option. This does not require you to pre-empt the standard input for your source code; it allows you to easily mix command-line and library source code (see the section "The AWKPATH Environment Variable" later in this chapter).

If no *–f* or *––source* option is specified, then *gawk* uses the first nonoption command-line argument as the text of the program source code.

If the environment variable POSIXLY_CORRECT exists, then *gawk* behaves in strict POSIX mode, exactly as if you had supplied the *--posix* command-line option. Many GNU programs look for this environment variable to turn on strict POSIX mode. If *--lint* is supplied on the command line and *gawk* turns on POSIX mode because of POSIXLY_CORRECT, then it issues a warning message indicating that POSIX mode is in effect. You would typically set this variable in your shell's startup file. For a Bourne-compatible shell (such as bash), you would add these lines to the *.profile* file in your home directory:

```
POSIXLY_CORRECT=true
export POSIXLY_CORRECT
```

For a *csh*-compatible shell, you would add this line to the *.login* file in your home directory:

```
setenv POSIXLY_CORRECT true
```

Having POSIXLY_CORRECT set is not recommended for daily use, but it is good for testing the portability of your programs to other environments.

Other Command-Line Arguments

Any additional arguments on the command line are normally treated as input files to be processed in the order specified. However, an argument that has the form *var=value*, assigns the value *value* to the variable *var*—it does not specify a file at all. (This was discussed earlier in the section "Assigning Variables on the Command Line" in Chapter 5.)

All these arguments are made available to your *awk* program in the `ARGV` array (see the section "Built-in Variables" in Chapter 6, *Patterns, Actions, and Variables*). Command-line options and the program text (if present) are omitted from `ARGV`. All other arguments, including variable assignments, are included. As each element of `ARGV` is processed, *gawk* sets the variable `ARGIND` to the index in `ARGV` of the current element.

The distinction between filename arguments and variable-assignment arguments is made when *awk* is about to open the next input file. At that point in execution, it checks the filename to see whether it is really a variable assignment; if so, *awk* sets the variable instead of reading a file.

Therefore, the variables actually receive the given values after all previously specified files have been read. In particular, the values of variables assigned in this fashion are *not* available inside a `BEGIN` rule (see the section "The BEGIN and END Special Patterns" in Chapter 6), because such rules are run before *awk* begins scanning the argument list.

The variable values given on the command line are processed for escape sequences (see the section "Escape Sequences" in Chapter 2).

In some earlier implementations of *awk*, when a variable assignment occurred before any filenames, the assignment would happen *before* the `BEGIN` rule was executed. *awk*'s behavior was thus inconsistent; some command-line assignments were available inside the `BEGIN` rule, while others were not. Unfortunately, some applications came to depend upon this "feature." When *awk* was changed to be more consistent, the *–v* option was added to accommodate applications that depended upon the old behavior.

The variable assignment feature is most useful for assigning to variables such as `RS`, `OFS`, and `ORS`, which control input and output formats before scanning the datafiles. It is also useful for controlling state if multiple passes are needed over a datafile. For example:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 mydata pass=2 mydata
```

Given the variable assignment feature, the *–F* option for setting the value of `FS` is not strictly necessary. It remains for historical compatibility.

The AWKPATH Environment Variable

In most *awk* implementations, you must supply a precise path name for each program file, unless the file is in the current directory. But in *gawk*, if the filename supplied to the *–f* option does not contain a `/`, then *gawk* searches a list of directories (called the *search path*), one by one, looking for a file with the specified name.

The search path is a string consisting of directory names separated by colons. *gawk* gets its search path from the AWKPATH environment variable. If that variable does not exist, *gawk* uses a default path, `.:/usr/local/share/awk`.* (Programs written for use by system administrators should use an AWKPATH variable that does not include the current directory, ".".)

The search path feature is particularly useful for building libraries of useful *awk* functions. The library files can be placed in a standard directory in the default path and then specified on the command line with a short filename. Otherwise, the full filename would have to be typed for each file.

---

* Your version of *gawk* may use a different directory; it will depend upon how *gawk* was built and installed. The actual directory is the value of `$(datadir)` generated when *gawk* was configured. You probably don't need to worry about this, though.

By using both the *––source* and *–f* options, your command-line *awk* programs can use facilities in *awk* library files (see Chapter 12, *A Library of awk Functions*). Path searching is not done if *gawk* is in compatibility mode. This is true for both *––traditional* and *––posix*. See the section "Command-Line Options" earlier in this chapter.

If you want files in the current directory to be found, you must include the current directory in the path, either by including `.` explicitly in the path or by writing a null entry in the path. (A null entry is indicated by starting or ending the path with a colon or by placing two colons next to each other (`::`).) If the current directory is not included in the path, then files cannot be found in the current directory. This path search mechanism is identical to the shell's.

Starting with Version 3.0, if AWKPATH is not defined in the environment, *gawk* places its default search path into `ENVIRON["AWKPATH"]`. This makes it easy to determine the actual search path that *gawk* will use from within an *awk* program.

While you can change `ENVIRON["AWKPATH"]` within your *awk* program, this has no effect on the running program's behavior. This makes sense: the AWKPATH environment variable is used to find the program source files. Once your program is running, all the files have been found, and *gawk* no longer needs to use AWKPATH.

Obsolete Options and/or Features

For Version 3.1 of *gawk*, there are no deprecated command-line options from the previous version of *gawk*. The use of `next file` (two words) for `nextfile` was deprecated in *gawk* 3.0 but still worked. Starting with Version 3.1, the two-word usage is no longer accepted.

The process-related special files described in the section "Special Files for Process-Related Information" in Chapter 4, *Printing Output*, work as described, but are now considered deprecated. *gawk* prints a warning message every time they are used. (Use `PROCINFO` instead; see the section "Built-in Variables That Convey Information" in Chapter 6.) They will be removed from the next release of *gawk*.

Known Bugs in gawk

- The *–F* option for changing the value of `FS` (see the section "Command-Line Options" earlier in this chapter) is not necessary given the command-line variable assignment feature; it remains only for backward compatibility.
- Syntactically invalid single-character programs tend to overflow the parse stack, generating a rather unhelpful message. Such programs are surprisingly difficult to diagnose in the completely general case, and the effort to do so really is not worth it.