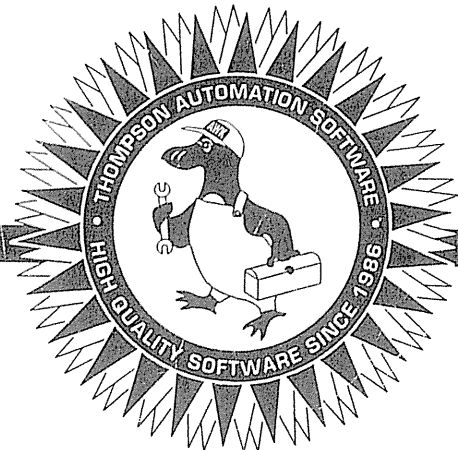


# Thompson Automation Software

New Version 5.0!

## TAWK COMPILER



# Table of Contents

---

Chapter 1: Introduction to TAWK .....	1
TAWK Program Installation .....	1
PATH variable .....	1
Optional AWKPATH variable .....	2
Optional TMP variable .....	2
History of TAWK .....	2
Chapter 2: TAWK Program Format .....	3
Running a TAWK Program .....	4
The Two Most Common Mistakes .....	5
TAWK Language Elements .....	5
Numbers .....	5
Strings .....	6
Regular Expressions .....	7
Simple Variables .....	7
Arrays .....	8
Fields .....	8
Chapter 3: Basic File Processing .....	9
Types of Program Blocks .....	9
BEGIN and INIT Blocks .....	10
Pattern-Action Blocks .....	10
Pattern-Range Blocks .....	10
BEGINFILE and ENDFILE Blocks .....	11
EXIT, END and TERM Blocks .....	11
More Examples of Program Blocks .....	12
Skipping Records or Files .....	14
The “next” statement .....	15
The “exit” and “abort” statements .....	15
Chapter 4: Program Arguments .....	16
Command Line Processing Rules .....	17
Program Arguments and the Automatic Input Loop .....	17
Command Line Variable Assignments .....	18
Chapter 5: Advanced Program Input/Output .....	19
Program Output .....	19
Print Statement .....	19
OFS and ORS Variables .....	19
Printf Statement .....	19
Printing to Files and Pipes .....	20
Reading Data from Files .....	21
Getline Statement .....	21
Specifying a Different Input Record Format .....	23
Communicating with the User .....	24
Closing files and pipes .....	24
Chapter 6: TAWK Expressions .....	25
Summary .....	25
Using strings and numbers .....	26
Converting strings to numbers .....	26
Converting numbers to strings .....	26

Arithmetic operators .....	27
Increment and Decrement Operators.....	28
String concatenation.....	28
Assignment.....	28
Assignment Short-Hand .....	29
Comparison operators .....	29
Comparing strings with numbers.....	30
Comparing Uninitialized Variables.....	30
Comparing Fields.....	30
The Difference Between = and == .....	31
Logical AND, OR, NOT .....	31
The Difference Between Logical And Bitwise Operators .....	32
Pattern Matching Operators .....	32
Conditional Expression .....	33
Chapter 7: Functions .....	34
Summary.....	34
Discussion.....	34
Function Arguments.....	35
Details of Arrays as Function Arguments.....	37
Chapter 8: Arrays .....	39
Arrays of varying dimensions .....	40
Array Assignment .....	40
The “in” keyword.....	41
Arrays and “for” loops. ....	41
Array Sorting.....	42
Sorting Examples .....	42
Preserving duplicate keys.....	43
Sorting on multiple keys.....	43
Alternate Sort Orders. ....	43
Keeping an array in order.....	44
Arrays as Arbitrary Structures.....	44
Common Mistakes .....	46
Old Style Arrays.....	46
Chapter 9: Fields .....	48
Summary.....	48
Discussion.....	48
Assignment to fields.....	48
Other methods to obtain fields: .....	49
Using the split or splitp functions.....	49
Using the match function .....	49
Using the pack and unpack functions .....	49
Examples.....	49
Chapter 10: Regular Expression Patterns .....	51
Regular Expression Matching Patterns .....	51
Regular Expression Flags.....	52
Regular Expression Tutorial .....	53
Efficiency Issues .....	55
Chapter 11: TAWK Flow Control Statements .....	56
The “if” statement .....	56
TRUE and FALSE .....	57
The “while” loop.....	57
The “do” loop.....	58

The "for" loop .....	58
The "break" and "continue" statements .....	59
Empty statement.....	60
Chapter 12: Variable Declarations .....	61
Summary.....	61
Global Variables .....	61
Module Local Variables:.....	61
Function Local Variables .....	61
Examples.....	61
Undeclared Variables .....	61
Uninitialized Variables .....	62
Restrictions on Command Line Variable Assignments.....	62
Chapter 13: Using the "awk" Program.....	63
Specifying the program. ....	63
AWK Program Options.....	63
Additional Command Line Processing by AWK .....	64
Chapter 14: Using the TAWK Compiler.....	66
TAWK Compiler Options in Alphabetic Order .....	66
Config File .....	69
Long Command Lines And Response Files .....	69
Libraries .....	69
AWKPATH Environment Variable .....	70
Temporary Files .....	70
Incremental Compilation.....	71
Chapter 15: Using the TAWK Debugger.....	72
Starting the Debugger. ....	72
Basic Debugger Concepts .....	72
Tracing Your Program .....	73
Stop Points .....	73
Viewing Variables.....	73
Leaving the Debugger.....	74
Chapter 16: Calling External Functions .....	75
Declaring External Functions in TAWK.....	75
Using Dynamic Linking.....	79
Using Static Linking (DOS Version) .....	80
Advanced TAWK Programming Topics.....	83
What is the TAWK Executable File? .....	83
Win32 Call-Back Functions .....	84
Sharing File Handles Between TAWK and C.....	84
Additional Information.....	85
Chapter 17: TAWK Built-In Functions.....	86
Chapter 18: TAWK Built-In Variables .....	131
Appendix 1: Details of Number Representation.....	147
Round-Off Error Suppression .....	147
Appendix 2: Compatibility with AWK .....	149
Common AWK Incompatibilities: .....	149
Index to Functions and Variables .....	150



## Chapter 1: Introduction to TAWK

Chapter 1: The TAWK Compiler is available for several different operating systems. You should install the version for the operating system you intend to use. The various versions are:

Win32 Version	Runs on Windows NT or Windows 95, but not Windows 3.1.
DOS Version	Runs on MS-DOS or PC-DOS. Use this version for Windows 3.1.
UNIX Version	Runs on the Sun Solaris Operating System. A version for SunOS is in development.
OS/2 Version	Runs on OS/2 version 1.3 or higher. Includes both 32-bit runtime modules for OS/2 2 or OS/2 WARP, and 16-bit runtime modules that can build dual mode (DOS and OS/2) programs, or programs for OS/2 version 1.3

## TAWK Program Installation

Read the file named "readme.txt" on the installation disk first. This file may provide additional installation instructions and other information about the latest release.

To install TAWK, choose or create a directory where you want to place the software. Change to this directory and run the "install" command on the installation disk.

If you are upgrading your TAWK Compiler from a previous version, we recommend installing the new version of TAWK in the same directory as the old version without deleting the old version. This will insure that your previously compiled TAWK programs keep working. The detailed explanation is as follows: By default, compiled TAWK programs use a runtime file that resides in the TAWK installation directory. TAWK programs that are compiled with the default options need this file when they run. The name of the TAWK runtime file is `awkr???exe`, where `???` stands for the version number, so you can have multiple versions on your system simultaneously. By installing the new version of the TAWK Compiler in the same directory as the old, the old runtime files will be preserved, but all the other files that come with the TAWK Compiler will be brought up to date. Alternatively, you could recompile your TAWK programs with the new version of the TAWK Compiler, or use the TAWK Compiler `-xe` option (or other `-x` option) to create a stand-alone `.exe` program that does not require the runtime file.

Additionally, if you are upgrading and you use multiple operating systems, you should upgrade all different operating system versions at the same time. Otherwise, you will not be able to cross compile.

### PATH variable.

We recommend that you change your PATH environment variable to include the directory where TAWK is installed.

Under DOS, the PATH environment variable is set in the `autoexec.bat` file. For example, your "autoexec.bat" file might include:

```
set PATH=C:\dos;C:\windows;C:\usr\bin;C:\tawk
```

Under OS/2 you set the PATH in the "config.sys" file instead. Under Win32 (Windows NT or Windows 95), environment variables are set by one of the programs in the control panel. Under UNIX the PATH is typically set in your ".profile" or ".kshrc" files for Shell users, or ".cshrc" file for C-Shell users.

PC users may need to restart the computer to make changes to configuration files take effect.

Changing the PATH is not strictly necessary. As an alternative you could invoke the TAWK programs using a full path each time. When TAWK runs it needs to find various files from its installation directory, but it looks for these files first in the directory from which it was invoked, then along the PATH variable, then along the AWKPATH variable.

### Optional AWKPATH variable

The AWKPATH environment variable can optionally be set to a path where TAWK will search for TAWK program source files. If a program source file specified to the TAWK Compiler is not found in the current directory, then each directory in the AWKPATH variable is searched. The format of the AWKPATH variable is the same as the PATH variable, that is, a list of directories separated by colons (UNIX) or semi-colons (DOS or OS/2.)

### Optional TMP variable

TAWK creates temporary files in the directory indicated by the TMP environment variable. If the TMP variable is not set, then temporary files are created in the current directory. Temporary files created by TAWK are always removed before TAWK exits.

## History of TAWK

The TAWK programming language is a major new revision of the AWK language by Thompson Automation Software. It is the result of over 5 years of development with user feedback and participation. TAWK includes a compiler, a debugger, and extensive new functionality for both general purpose and data-processing applications. Two interfaces are included for TAWK: the TAWK Compiler is used to generate executables; the "awk" program is a UNIX-compatible interface for the TAWK Compiler that allows a TAWK program to be compiled and executed in a single step.

The original AWK programming language was developed by Alfred Aho, Peter Weinberger and Brian Kernighan at AT&T in 1977. The name "AWK" was derived from the author's initials. A revision of the AWK language called "New AWK" or "nawk" for short, was released in 1985 for UNIX based systems and added user-defined functions and some text processing functions.

We use the term "TAWK" to distinguish this product from previous versions of awk because, in terms of suitability for general purpose programming tasks, TAWK is qualitatively superior to awk. For example, TAWK includes bit and structure manipulation, sorting, type conversions, and many other operations that were notably absent from awk. The TAWK Compiler produces high speed executables, while the formerly interpreted awk language was sometimes painfully slow. TAWK also permits very large TAWK programs to be developed and executed, and will even run programs larger than available memory under DOS by transparently using disk space for program storage.

TAWK retains backward compatibility with AWK and is still one of the easiest to use general purpose programming languages. But TAWK is also suitable for the development of commercial programming applications that were formerly the exclusive domain of more traditional languages like Pascal, C or C++.

## Chapter 2: TAWK Program Format

Program blocks are used to control the flow of your program from beginning to end. A TAWK program consists of one or more program blocks in the following general format:

```
trigger { statements }

trigger { statements }
...
```

Each program block consists of a *trigger* and a list of program *statements* that are executed when the trigger occurs. The trigger may be a keyword (like **BEGIN**) or a pattern (discussed in the next chapter.) The opening curly brace must appear on the same line as the *trigger* and the *statements* may appear on the same line or on following lines.

The format of the simplest type of TAWK program is:

```
BEGIN { statements }
```

This is called a **BEGIN** block and is one of several different types of program blocks to be discussed. The list of statements in the **BEGIN** block is executed when the program begins. The simplest TAWK program is:

```
BEGIN { print "hello world" }
```

This program executes the statement: `print "hello world"`, which prints out the string "hello world".

The case of letters (upper-case versus lower-case) is **IMPORTANT!** The following are incorrect:

```
BEGIN { PRINT "HELLO WORLD" }      # WRONG

begin { print "hello world" }      # WRONG
```

Comments in a program start with the character `#` and continue to the end of the line, as shown above. To enhance your program's clarity you can also insert extra spaces or tabs between language elements, and you can insert blank lines before or after statements.

```
# This is the same program as above
BEGIN      { # But it is harder to read.
print      "hello world"
}          # So you may wish to follow the
# programming style of our examples.
```

To enhance program clarity it is a good idea to use a consistent style. We usually indent statements enclosed in curly braces and place the closing curly brace (`}`) at the same level of indentation as the **BEGIN** keyword or other statement that it goes with, as in the example below. Large TAWK programs may have multiple levels of nested parentheses, and using a consistent style makes such programs easier to read.

Multiple statements may be placed on separate lines, or separated by semi-colons, or both. For example, you can write:

```
BEGIN {
  print "Hello World"
  print "hello again"
}
```

You can use a semicolon instead of a new line:

```
BEGIN {
  print "hello world"; print "hello again"
}
```

Or both:

```
BEGIN {
    print "hello world";
    print "hello again";
}
```

A very long statement can be continued onto multiple lines by placing a back-slash (\) at the end of every line but the last. For example:

```
BEGIN {
    print \
        "hello",
        "world"
}
```

The back-slash is optional if TAWK can figure out that the statement obviously continues onto the next line. In the above example, no backslash was required after the comma because TAWK knows that any line that ends with a comma must continue onto the next line. However, you could have put a backslash there, too. When in doubt, use a backslash.

## Running a TAWK Program.

There are two ways to execute a TAWK program. The "awk" program allows you to compile and execute your TAWK program in a single step and is easier to use for small programs. The TAWK Compiler allows you to compile your TAWK program into an executable and is suitable for larger programs or whenever you need to generate an executable. Both of these programs are covered in full detail in later chapters. Here is an introduction to get you started:

To use the "awk" program, use a plain text editor to place your TAWK program in a file, and then invoke "awk" to run the program. For example, let's say the contents of file myprog.awk is:

```
BEGIN {
    print "hello world"
}
```

Enter the following command to execute this TAWK program. The -f option (which must be lower-case -f, not -F) tells TAWK to execute the program file myprog.awk.

```
awk -f myprog.awk
```

You will get the following output:

```
hello world
```

Notes:

- 1) You must use a plain text editor to create the file, not a word processor. Word processors (like WordPerfect or Microsoft Word) typically include text formatting codes in the file, in addition to the text.
- 2) If your computer says something like "Bad command or file name", it means the operating system could not find the awk program. Make sure that the awk program (called awk.exe under DOS or OS/2) is in one of the directories listed in your PATH environment variable.
- 3) If you get unexpected results, try entering awk without arguments:

```
awk
```

You should get a copyright message something like this:

```
TAWK Version X.XX (DOS)
Copyright 1994 Thompson Automation, Inc.
All Rights Reserved.
Enter TAWK Program followed by a blank line:
```

If you don't get this message, you are not using Thompson Automation's TAWK program. To get back to your normal prompt, just press the Enter key a couple of times.

## The Two Most Common Mistakes

- 1) Your program must go in a program block, typically a BEGIN block. If you just type something like this:

```
a = 1    # WRONG
```

You will get an error message, but TAWK will still try to execute this program because it resembles a pattern-action block with the action statements missing.

- 2) Under DOS or OS/2, don't enter your TAWK program like this:

```
awk 'program'    # Wrong under DOS or OS/2!
```

This works fine under UNIX or if you are using the Thompson Toolkit Shell for DOS or OS/2. But the standard DOS and OS/2 command interpreters will process any `<>|` or `&` characters in your *program* as redirection symbols, so programs that contain these characters will fail. You will also quickly run into the command line length limit under DOS.

## TAWK Language Elements

The fundamental elements of the TAWK language are:

Language Element	Examples
numbers	1492    0x1f    1.7E3
strings	"this is a string"
regular expression pattern	/Sam Houston/
simple variables	x, NF, whatever
array variables	y[10], z[1]["a"], ENVIRON["PATH"]
Fields	\$0, \$1, \$2, ...

## Numbers

You can enter a number in various ways:

```
1492    # Integer
14.92   # Floating point
1.492E3 # Scientific notation
0x2a    # Hexadecimal notation
```

The "E" in scientific notation can be read "times ten to the". So "1.492E3" is 1.492 times ten to the third power or 1492, and 3.1E-2 is 3.1 times ten to the minus second power or 0.031. A lower case "e" can be used instead of "E" in such numbers. The following are all the same number:

```
1492.0    1.492e3    +1.492E+03    14.92e2
```

You can not put commas in numbers in TAWK. If your numeric data contains commas, your program will have to remove them before using the numbers. The **gsub** function (described in the Function Reference) can easily remove the commas from a number.

Numbers may be entered in hexadecimal notation using the special prefix 0x or 0X (that is a numeric zero, not the letter oh), followed by up to eight hexadecimal digits chosen from the set: 0-9, a-f, or A-F. For example, 0xa is 10, 0x10 is 16, and 0x20 is 32.

TAWK stores numbers internally as either integers or double precision floating point numbers, whichever is appropriate. If an integer calculation overflows, the result is automatically converted to floating point format to maintain maximum accuracy. TAWK will use the computer's hardware floating point unit (FPU) if it has one, otherwise TAWK performs floating point operations using software subroutines. Details of number representation are covered in the appendix.

## Strings

Strings are used in TAWK to store character data. A string may be specified in a TAWK program as ASCII characters surrounded by quotes, as in:

```
BEGIN {
    x = "hello world"
}
```

Strings appearing in your TAWK program this way are called "literal strings." A literal string may contain "escape sequences" to represent special characters, such as a quote character. Escape sequences are recognized only when strings are entered as literal strings in a program source file. When you read strings into your program any other way, for example, from a data file, the data is just stored in the string "as is."

Escape sequences in strings consist of a backslash followed by one or more characters as shown in the table below. When an escape sequence appears in a literal string in a TAWK program, the escape sequence is replaced by the character indicated in the table.

**String Escape Sequences**

Character Sequence	Meaning in String
\a	audible alert (beep)
\b	backspace
\f	form-feed
\n	newline (Moves cursor to next line)
\r	carriage-return (Moves cursor to start of current line)
\t	tab
\v	vertical tab
\"	quote character
\\	Back-slash character
\ddd	octal representation of character code ddd (ddd represents an octal number.)
\xdd	hex representation of character code dd (dd represents a hexadecimal number.)

The form-feed and vertical-tab characters usually have no special meaning on IBM compatible computers but are used on many other computer systems to cause vertical cursor motion.

To enter a quote in your string, precede the quote with a backslash. For example:

```
x = "hello \"real\" world"
```

The escape sequence \" is replaced by ", and the string that TAWK stores in memory for variable x looks like this:

```
hello "real" world
```

To enter a backslash, you must double the backslash. This is commonly used for DOS path names, for example:

```
path = "C:\\usr\\doug\\doc\\tutorial"
```

In this case, the string that TAWK stores in memory looks like this:

```
C:\usr\doug\doc\tutorial
```

Strings are not limited to storing ASCII characters; you can store any arbitrary data in a string. To enter an arbitrary character into a string, use either \ddd, where ddd represents one to three octal digits specifying the character code, or use \xdd, where dd represents one or two hex digits specifying the character code. For example: \x21 represents the ASCII character code for the

exclamation mark, and `\x00` represents a nul (zero) character. There is nothing special about the nul (zero) character in TAWK; it can appear in any string or input or output file.

A backslash followed by any character other than those in the escape sequence table above causes TAWK to print a warning message, and the backslash and the character are replaced by just the character.

A very long string can be continued on to multiple lines using a backslash like this:

```
print "now is the time for all good \
people to come to the aid of their party"
```

Be careful not to indent the continuation of the string, otherwise the spaces or tabs will be included in the string. Also be sure there are no spaces or tabs or other characters after the backslash; the backslash must be the last character on the line:

```
# INCORRECT:
print "four score \      # NO comment here!
and seven years ago"
```

## Regular Expressions

TAWK uses a special notation to specify patterns that are used to match characters in a string. This notation is derived from technical literature on text processing, and is called "regular expression" notation. A regular expression may be specified in a TAWK program by enclosing the regular expression in forward slashes /like this/. For example:

```
/hello world/
```

The regular expression above will match the string: "hello world", or: "I said hello world again" but not: "hello" or "Hello world".

The regular expression may be followed by optional flag characters, for example, the "i" flag makes the pattern case-insensitive:

```
/case insensitive matching/i
```

Special characters are used in regular expressions to indicate more complicated patterns to be matched. In certain contexts a string can be interpreted as a regular expression. An entire chapter is devoted to regular expressions.

## Simple Variables

A variable name consists of letters, digits, and underscores, and may not start with a digit. For example:

```
rows          # ok
with_110_coronets # ok
76_trombones  # NOT a variable
```

Your variables can store any type of data, including numbers or strings. When a variable is used in an expression in a context that requires a number or string, TAWK automatically converts the value to the required type, either number or string. For example:

```
BEGIN {
  y = "4"
  x = 3 + y
  print x
}
```

The result in variable x is 7. The value of variable y, which was a string, was automatically converted to a number. In this example it was obvious what TAWK should do, but some cases of automatic conversion are more complicated. For example, when the number 3.1 is converted to a string, should it be "3.1" or "3.100" or 3.1E+000"? And how should TAWK react if you try to compare a number to a string? TAWK has special rules for these cases covered in the chapter on expressions.

Uninitialized variables (variables that have not yet been assigned a value) have a special value. When used in an expression, an uninitialized variable appears to have the string value "" or the numeric value 0, whichever is appropriate.

Variable declarations are optional. Variable declarations are unnecessary in small TAWK programs consisting of a single program source file, but are needed in larger programs where variables are to be shared among multiple TAWK program source files. Variable declarations are covered in a separate chapter.

Some variable names are built-in to TAWK and have a pre-defined special meaning. These built-in variable names are reserved, and may not be used for any other purpose. With only a few exceptions these variable names are all upper-case, making them easy to spot in a program. The "Built-In Variables" chapter includes a complete list of built-in variables and their meaning.

## Arrays

An array is represented by a variable name followed by one or more array indices enclosed in square brackets. Using a variable as an array makes the variable into an array. For example:

```
BEGIN {  
    x[3] = 3  
    another[2][9] = "hi"  
    print x[3], another[2][9] # This prints: 3 "hi"  
}
```

Each unique array index refers to a unique element in the array. Each element in the array can hold any type of data. Unlike other languages, the array index is a string, allowing arrays to be used as a kind of data-base index. If an integer is given as an array index, it is converted to a string. As a consequence, the elements of an array do not need to be contiguous, that is, you can make assignments to `x[1]` and `x[1000]` without automatically creating array elements `x[2]` through `x[999]`. TAWK provides a special `for` loop statement to access each element of an array in turn. TAWK can also optionally sort the elements of an array by the array indices. An entire chapter is devoted to ways to use arrays.

## Fields

TAWK can automatically break the current input record into fields. A field is a single recognizable piece of information in the current input record. Fields are referenced in a TAWK program using the symbol: `$n`, where `n` is an integer or an expression that evaluates to an integer. The first field is `$1`, the second is `$2`, etc. `$0` denotes the entire current input record.

You can specify the field type with the built-in variables: `FS` and `FPAT`. The default field type specifies that fields are separated by sequences of spaces or tabs, that is, the fields are the words in the record. If the current input record is:

```
Pat Thompson      Hiking      1/1/86  165
```

Then it consists of five fields as follows:

```
$1 = "Pat"  
$2 = "Thompson"  
$3 = "Hiking"  
$4 = "1/1/86"  
$5 = "165"
```

An entire chapter is devoted to fields.



## Chapter 3: Basic File Processing

Like other programming languages, TAWK has a full range of functions allowing you to open, read and write to files by programming these operations yourself. This type of File I/O gives you complete control and is discussed in a later chapter.

However, TAWK has an easier way to process files. If your data files consist of records, TAWK provides an "Automatic Input Loop" as a convenient way to create the most common type of file processing program. You can use an Automatic Input Loop if your program meets the following constraints:

- a) The input data to be processed consists of sequential records. For example, in text files the "records" are simply the lines in the text file, and in a data-base file the records are the entries in the data-base.
- b) The "record" format must be one of the built-in types that TAWK recognizes: either using a record separator or using a fixed-length format. For example, the default record format specifies that the record separator is the newline character, which is a fancy way to say that the records are the lines in an ordinary text file.

If your program meets these constraints, then you can use an Automatic Input Loop and let TAWK take care of opening and closing the files for you.

The input data files to be processed can be specified on the command line when your TAWK program is run. If no files are specified on the command line, your program will obtain data from its standard input, which could be from another command via a "pipe", from a file that is specified using redirection (like this: < filename) or could be typed interactively from the terminal. (See Program Arguments, below, for additional discussion.)

You don't have to include any explicit instructions in your TAWK program to create an Automatic Input Loop. Rather, your TAWK program will automatically use an Automatic Input Loop if it contains a Pattern-Action block, a Pattern-Range block, or an END block.

### Types of Program Blocks

The following lists all the types of program blocks:

**BEGIN { statements }**

The *statements* are executed at the beginning of the program.

**INIT { statements }**

The *statements* are executed at the beginning of the program, but before any BEGIN blocks are executed.

**EXIT { statements }**

The *statements* are executed at the end of the program, after any END blocks..

**TERM { statements }**

The *statements* are executed at the end of the program, but after any END or EXIT blocks.

These blocks create an Automatic Input Loop:

**pattern { statements }**

This is called a pattern-action block. The *statements* are executed when the *pattern* matches the current input record.

**pattern1, pattern2 { statements }**

This is called a pattern-range block. The *statements* are executed for each input record starting with a record that matches the pattern expression: *pattern1* and continuing until a record is found that matches *pattern2*. *Pattern1* and *pattern2* may both match the same line, in which case the range is a single line.

**BEGINFILE { statements }**

The *statements* are executed each time TAWK starts processing a new file.

**ENDFILE { statements }**

The *statements* are executed each time after TAWK finishes processing a file.

**END { statements }**

The *statements* are executed at the end of the program. This is similar to an EXIT block, except that your program will automatically use an Automatic Input Loop if it contains an END block.

## BEGIN and INIT Blocks

We have already seen the BEGIN block in chapter 1: the statements in the BEGIN block are executed when the program begins. If your program does not need an Automatic Input Loop, you can place the entire program in a BEGIN block. Many TAWK programs are contained entirely in a BEGIN block.

The INIT block is like a BEGIN block, but all INIT blocks in all program files are executed before any BEGIN blocks. This is not important in small programs, but INIT blocks are useful in large programs consisting of many files to perform initialization operations that must occur before the main part of the program gets going.

## Pattern-Action Blocks

A pattern-action block has the following form:

```
pattern { statements }
```

This block states that TAWK should use an Automatic Input Loop and execute the specified *statements* for each input record that matches the *pattern*. The Automatic Input Loop reads each sequential record, one at a time, from the input (either from specified data files or from the program's standard input) into the current input record (\$0) and then evaluates the pattern part of each pattern-action block and executes the statements associated with each pattern that matches the input record. There can be as many pattern-action blocks in your program as you require; the patterns are tested in the order that you place them in your program.

The pattern can be any valid TAWK expression. The statements are executed if the pattern expression evaluates to TRUE. (In TAWK a value of 0 or "" is considered FALSE and any other value is TRUE.) In the pattern expression a regular expression */like this/* is treated as TRUE if the regular expression matches the current input record.

There are two degenerate cases: If the pattern is omitted the statements are executed for each record. If the { statements } (including the curly braces) are omitted then TAWK supplies a default action of: { print } which causes the current input record (\$0) to be printed.

### Examples of pattern-action blocks:

```
# This prints all lines that contain "this"
/this/ { print }

# This is the same program:
/this/

# This numbers the input lines:
# (FNR is a built-in variable holding the number of
# records processed from the current data file.)
{ print FNR ":" $0 }

# This prints the first line of each data file:
FNR == 1 { print }
```

## Pattern-Range Blocks

A pattern-range block includes two patterns separated by a comma that specify a range of input records:

```
pattern1, pattern2 { statements }
```

This block states that TAWK should use an Automatic Input Loop and execute the specified *statements* for each input record starting with an input record that matches *pattern1* and continuing until an input record is found that matches *pattern2*. If *pattern2* matches the same input record as *pattern1*, then the range is that single record. After *pattern2* is found TAWK starts looking for *pattern1* again on the next record. If *pattern2* is never found after *pattern1*, all remaining input records are matched.

As with a pattern-action block, any TAWK expression may be used for *pattern1* or *pattern2*.

Examples:

```
# This prints all ranges of lines starting with
# a line that contains "this" through a line
# that contains "that".
/this/,/that/ { print }

# This is the same program:
/this/,/that/

# This prints the first 5 lines of each file:
FNR == 1, FNR == 5 { print }
```

## BEGINFILE and ENDFILE Blocks

The BEGINFILE block is executed after TAWK's Automatic Input Loop opens a file but before the first record is read in. This block is useful if you need to seek to a particular position in the file before you start processing it. Compare a BEGINFILE block to the following pattern-action statement:

```
FNR == 1 { statements }
```

FNR is the number of records in the current file, so this pattern-action block is executed after the first record of each file is read in. In contrast, the BEGINFILE block is executed before the first record is read in, and will be executed even if the file is empty and contains no records. There is also a slight speed advantage to using a BEGINFILE block because TAWK tests each pattern-action statement for each record. The BEGINFILE block is a good place to print a banner when you start processing a new file, for example:

```
BEGINFILE {
    print "Now Processing File:", FILENAME > stderr
}
```

The ENDFILE block is executed after all records have been read from the file, but before the file is closed. After executing the ENDFILE block, if the file is a normal file (not a device like the terminal) TAWK tries one more time to read records from the file, and if it succeeds continues reading from the same file rather than going on to the next. This allows you to seek to a new position in the ENDFILE block and allow TAWK to continue processing records at the new position. In this case, TAWK will execute the ENDFILE block more than once on the same file, and in fact may process the same file forever if you do not provide some way to terminate it. For example, here is some code to force TAWK to process each file twice without closing and reopening the file:

```
ENDFILE {
    # flag is a variable we set when we are
    # processing the file the second time around.
    if (flag) { flag = 0; }
    else { fseek(FILENAME,0); flag = 1; }
}
```

## EXIT, END and TERM Blocks

These three types of blocks allow you to specify code to be executed just prior to program termination. The EXIT block is the most useful, while the END and TERM blocks are designed for special purposes. The code in the EXIT block is executed when:

- 1) All input from an Automatic Input Loop has been read and processed, or
- 2) An exit statement is executed to prematurely terminate the program.

For example, here is a program to print the number of words in a file:

```
# Count the number of words:
# (NF is a built-in variable holding the number
# of fields in the current record.)
{ nwords = nwords + NF }
EXIT { print "There were", nwords, "words." }
```

The END block is similar to an EXIT block, except that it forces your program to use an Automatic Input Loop, even if no Pattern-Action, BEGINFILE or ENDFILE blocks are included in the program. If there is an END block but no Pattern-Action Blocks, TAWK simply reads in all the records and discards them before processing the END block. For example, suppose we

wanted only to count the number of lines in a file. We can use TAWK's built-in variable: NR that counts the total number of lines processed, so we do not need any pattern-action blocks at all. The following is the complete program:

```
# This prints how many records were processed:
END { print "There were", NR, "records." }
```

If you wanted to write the above program using an EXIT block, you would have to include a dummy pattern-action block, which does not do anything, but forces the program to use an automatic input loop. However, the above example will execute more quickly than the following example:

```
# Same result as previous program, above:
{ ; } # This is a do-nothing pattern action block
EXIT { print "There were", NR, "records." }
```

The TERM block is similar to an EXIT block, except that it is executed even if your program is aborted. Your program will be aborted if it executes the **abort** built-in function, or if it receives a catchable signal, for example, a user interrupt, that would normally kill your program and that is not otherwise handled by your TAWK program. (Your TAWK program can specify custom handling of signals using the SIGNAL array.) Use a TERM block only for clean-up actions, such as removing temporary files, that must be accomplished no matter how your program ends.

TAWK attempts to detect most program termination events so that it can execute the TERM block(s). Under UNIX, TAWK will detect most catchable signals, including the hangup, interrupt and quit signals. Under Win32 (Windows NT or Windows 95), TAWK will also detect window close, shutdown, and log-off events. Therefore, your program will not respond to a window close (or other) event until the TERM block(s) finish. Note that TAWK will only execute the TERM block(s) once, even if multiple signals are generated. Therefore, users can generally not use a Control-C interrupt to kill a TAWK program that is executing its TERM blocks. To keep your users happy, you should not perform any lengthy processing in a TERM block.

Note the differences between the exit and abort statements. The "exit" statement causes the code in the END, EXIT and TERM block(s) to be executed before the program actually terminates. The "abort" statement causes only the code in the TERM block(s) to be executed before the program terminates.

Your program may contain any number of END, EXIT and/or TERM blocks. The order of execution is: all END blocks in all program files are executed, then all EXIT blocks in all program files, then all TERM blocks in all program files.

## More Examples of Program Blocks

To demonstrate some example programs we will assume the following input file containing employee names and information. This file is a text file, and the records that we wish to process are simply the lines in the file.

Example Datafile

Fname	Lname	Hobby	Hire-Date	Salary
Pat	Thompson	Hiking	1/1/86	165
Craig	Ralston	boating	1/1/90	190
Mike	Smith	bird-watching	2/17/92	140
Doug	Brown	hiking	9/1/92	150
Don	Harvey	hacking	3/26/91	160
Jim	Patterson	Iron-Smithing	4/19/89	180

To try these programs yourself, place this data in a file called "datafile" (or any other name you choose) and specify this filename on the TAWK command line after the -f progname option. For example:

```
awk -f progname.awk datafile
```

Suppose we want to find an employee whose hobby is hiking. After all, Thompson Automation Software is located in the beautiful Pacific Northwest! The following simple program prints out every line that matches the pattern "hiking". The **print** command without any arguments prints the current record. These programs use TAWK's default record type, which is just to treat each line in the file as a separate record.

```
/hiking/ { print }
```

Because the default action is { print }, it can be omitted in this simple case making the program amazingly short:

```
/hiking/      # This is a complete TAWK program!
```

In both these cases the output will be:

```
Doug Brown      hiking      9/1/92   150
```

The above program works like this: TAWK creates an Automatic Input Loop to run the above program. The Automatic Input Loop reads each record, one at a time, from the input data file(s). For each record, the pattern-action block is tested to see if the pattern matches the current record. If the pattern `/hiking/` is found in the current record, then the statement **print** is executed, which prints the current record.

A program can contain any number of pattern-action blocks, which are tested in the order they occur in the program file. In the above example, we missed Pat, whose hobby was listed as "Hiking", because it is capitalized and we were looking for lower case "hiking". Here is a corrected program:

```
/hiking/ { print }
/Hiking/ { print }
```

This program contains two pattern-action blocks, one for each of the patterns for which we are searching. We could also look for both "hiking" and "Hiking" at the same time using the `|` symbol in the regular expression to separate alternatives to be matched:

```
/Hiking|hiking/ { print }
```

Yet another way to solve the problem would be to add the "i" flag to the end of the regular expression, which causes it to do a case-insensitive pattern match. This would print records containing "hiking" or "Hiking" or "HIKING", etc.

```
/hiking/i { print }
```

For all of the above the output is:

```
Pat Thompson      Hiking      1/1/86   165
Doug Brown        hiking      9/1/92   150
```

Notice that the records come out in the same order in which they appear in the data file. This is simply a consequence of the records being processed serially from the beginning of the file to the end.

The pattern-expression can be any TAWK expression. Complex patterns are often built up using TAWK's logical operators:

#### Logical Operators in TAWK

<u>Symbol</u>	<u>Meaning</u>
&&	AND
	OR
!	NOT

For example, to find employees with hobbies of "hiking" or "Hiking" we could have used the following program:

```
/hiking/ || /Hiking/ { print }
```

To find all employees that do NOT hike, we could use this:

```
! ( /hiking/ || /Hiking/ ) { print }
```

The output will be:

```
Fname  Lname      Hobby      Hire-Date  Salary
Craig  Ralston     boating     1/1/90     190
Mike   Smith       bird-watching 2/17/92    140
Don    Harvey      hacking      3/26/91    160
Jim    Patterson   Iron-Smithing 4/19/89    180
```

A very long pattern can be continued onto additional lines by ending all but the first with a backslash. For example:

```
/hiking/ | | \
/hacking/ | | \
/boating/ { print }
```

### Examples Using Fields

Suppose we wanted to print Mike Smith's record. We might use the following:

```
/Smith/ { print $0 }
```

The output is:

```
Mike Smith      bird-watching 2/17/92 1   140
Jim Patterson   Iron-Smithing  4/19/89 3   180
```

Oops! Since we were searching for "Smith" any where in the line, we also printed out Jim Patterson whose hobby is Iron-Smithing. We can get the correct result using the following program:

```
$2 == "Smith" { print $0 }
```

Which outputs:

```
Mike Smith      bird-watching 2/17/92   140
```

The expression `$2 == "Smith"` tests the second field to see if it is "Smith", which is TRUE only for Mike Smith's record, which is printed. While we are at it, we better test the first name too, just in case we later add another employee named Smith.

```
$1 == "Mike" && $2 == "Smith" { print $0 }
```

The `&&` stands for logical AND. This expression is TRUE only when the first field is "Mike" and the second field is "Smith".

The `~` operator is a pattern matching operator that returns TRUE if the string on its left side matches the pattern on its right. This can be used to match fields against a pattern. For example, to find all employees whose hobby (field three) is boating we could use:

```
$3 ~ /boating/ { print $0 }
```

This will print any employee whose hobby is "boating".

### Skipping Records or Files

Normally the Automatic Input Loop processes every record in every input file specified on the command line. But sometimes you want to skip over a record or a file, or prematurely terminate your program for some other reason. The following table lists ways to skip records or files:

<u>Statement</u>	<u>Result</u>
<code>next;</code>	Causes TAWK to immediately process the next record. The remaining pattern-action blocks, if any, are skipped for the current record and the current record is discarded.
<code>exit;</code>	Skips all remaining files and goes immediately to the END block
<code>abort;</code>	Skips all remaining files and terminates the program immediately without executing the END block
<code>close(FILENAME);</code>	Closing the current file prevents TAWK from getting any more input from that file. The next input record processed will come from the next filename on the command line, if any. Unlike the <b>next</b> statement, this does not cause an immediate jump, or affect processing of the remaining pattern action blocks, if any, using the current record.

Modify ARGV, ARGC and ARGV You can also control the files that the Automatic Input Loop will process by changing these variables. This is covered in the chapter on Program Arguments.

### The “next” statement

When TAWK encounters a “next” statement, it immediately starts processing the next record. The current record is discarded and the remaining pattern-action blocks in the TAWK program, if any, are skipped. After it reads in the next record, the Automatic Input Loop starts over at the top of the program.

Example using the “next” statement:

The first record in our datafile is a header that specifies the categories of the fields (or columns) in the file. We probably do not want to print this record in reports that we generate from the datafile. The following program is similar to the previous example that prints employees that do NOT hike, except that it skips printing the first record:

```
FNR == 1 { next }
! /hiking/i { print }
```

The FNR variable is a built-in variable that holds the number of records processed from the current file. The expression FNR == 1 is TRUE only when FNR is 1, which is true only when the automatic input loop is processing the first record in the data file. The next statement is executed when the first record is encountered, which causes the program to skip the first record and go on to the next record. The output is now:

Craig Ralston	boating	1/1/90	190
Mike Smith	bird-watching	2/17/92	140
Don Harvey	hacking	3/26/91	160
Jim Patterson	Iron-Smithing	4/19/89	180

Note that the order of the two pattern-action statements is important. Because pattern-action statements are tested and executed in the order they appear in the program file, we must test for FNR == 1 first if we want to skip printing the first record.

### The “exit” and “abort” statements

The **exit** and **abort** statements cause TAWK to skip all of the remaining input records. For example:

```
if (a == 0) {
    print "Division by 0. Internal error! I quit!"
    abort(2)
}
x = y / a
...
```

The difference between the **exit** and **abort** statements is that the **exit** statement causes your program to first execute any END blocks before exiting, while **abort** leaves the program immediately. If an **exit** statement occurs in an END block, it skips the rest of the END block and skips any other unexecuted END blocks as well. An additional difference under DOS is that any output that is saved in output pipes that have not yet been closed causes normal execution of the pipe command after an **exit** statement but abandonment of the data saved in the pipe after an **abort** statement.

Both the **exit** and **abort** statements can specify an exit status for the TAWK program. In the above example the exit status was 2. If no exit status is specified, 0 is assumed. Under DOS and OS/2 the exit status must be in the range 0 to 255. Most programs follow the convention that an exit status of 0 indicates successful completion, while any other exit status indicates an error. The exit status can be tested by other programs after the TAWK program exits. See the discussion of the exit function in the Function Reference for an example showing how to use exit codes.

## Chapter 4: Program Arguments

The arguments passed to your TAWK program are available in the built-in ARGV and ARGV variables. The ARGV variable is an array with the name of the program in ARGV[0], and the program arguments in ARGV[1], ARGV[2], etc. Options that are processed by TAWK are removed from the program arguments before your program gets them. A list of the options processed by TAWK is provided in detail in later chapters on the "awk" program and the TAWK Compiler.

The ARGV variable contains the number of elements in the ARGV array. In other words, ARGV contains the number of program arguments plus one.

As an example, using the TAWK compiler:

```
awkc myprog.awk
myprog -v var=value -w -x hello world
```

Or using the "awk" program:

```
awk -f myprog.awk -v var=value -w -x hello world
```

The -f, -v and -w arguments are processed by TAWK and removed from the command line so the program arguments are:

```
ARGV[0] = C:\usr\patt\myprog.exe
ARGV[1] = -x
ARGV[2] = hello
ARGV[3] = world
ARGC = 4
```

The -x argument was passed to your program because it is not recognized by TAWK.

To pass options that TAWK recognizes (like -f or -v) to your TAWK program place them after -- in the command line. A -- in the command line tells TAWK not to examine the rest of the command line for options, and is itself removed. For example:

```
awk -f myprog.awk -- -f -v
```

or

```
awkc myprog.awk
myprog -- -f -v
```

```
ARGV[0] = C:\usr\patt\myprog.awk
ARGV[1] = -f
ARGV[2] = -v
ARGC = 3
```

Alternatively, specify the -eo option when you compile your program to prevent the compiled program from automatically recognizing any options. For example:

```
awkc -eo foo.awk
foo -v
```

Results in program: "foo" having the following program arguments:

```
ARGV[0] = C:\usr\patt\foo.exe
ARGV[1] = -v
ARGC = 2
```

Notes:

- 1) When using the "awk" program ARGV[0] ends with ".awk" instead of ".exe". Under UNIX the ARGV[0] path would use / rather than \. Under OS2 or UNIX: ARGV[0] may sometimes contain a path relative to the current directory rather than a full path. Under DOS version 2: ARGV[0] may not contain the path or may just be "awk".



- 2) You don't really need the built-in ARGV variable to determine the number of elements in the ARGV array. You could also use: `length(ARGV)`.

## Command Line Processing Rules

TAWK programs perform filename expansion on the command line arguments before placing them in the ARGV array. TAWK programs also permit quoted arguments that contain spaces. To put a quote in the command line precede it with a backslash. For example:

```
awk -f foo.awk "a quoted \" string"
```

```
ARGV[0] = C:\usr\patt\foo.awk
ARGV[1] = a quoted " string
ARGC = 2
```

Notes:

- 1) The "awk" program is UNIX compatible so it always uses UNIX compatible filename expansion rules. In contrast compiled TAWK programs use the filename expansion rules that are appropriate for the operating system. Under DOS the pattern to match any filename is `"*. *"`; under UNIX it is `"*"`.
- 2) Combined TAWK/C programs that are linked with your C compiler let the underlying C libraries handle argument processing, so your TAWK program gets whatever command line processing is provided by the C compiler vendor. For example, Borland version 3 does not support quoted strings in command line arguments.

## Program Arguments and the Automatic Input Loop

If your program uses an Automatic Input Loop the filenames to be processed come from the program arguments. The Automatic Input Loop starts after all INIT and BEGIN blocks have been processed. If `ARGC == 1` when the Automatic Input Loop starts, it indicates no filename arguments were specified so TAWK obtains its input from the program's standard input. If the standard input was not otherwise redirected, it will come from the user's terminal.

Otherwise TAWK processes the program arguments as follows: The built-in variable ARGV, which is initially 1 when your program starts, is the index in the ARGV array of the next filename that TAWK will process. When TAWK is ready to process a file it examines the elements of the ARGV array starting at element `ARGV[ARGV]` and continuing through element `ARGV[ARGC-1]`. (We subtract one from ARGC because ARGC is the number of program arguments plus one for the command name.) The following table shows the action that TAWK will take depending on the program argument:

**TAWK Program Argument Processing  
by the Automatic Input Loop**

<u>If the argument is:</u>	<u>TAWK will do this:</u>
<code>""</code> (null)	The argument is ignored;
<code>"-"</code> (a single dash)	The TAWK program's standard input will be processed by the Automatic Input Loop;
<code>name=value</code>	The variable assignment will be processed;
<code>filename</code>	The filename is opened and processed by the Automatic Input Loop.

If a filename can not be opened or a variable name is not valid TAWK prints a warning message and goes on to the next program argument. In all cases ARGV is incremented so that it always indicates the next ARGV element to be processed.

Your program can modify ARGV, ARGC or ARGV at any time. For example, if you don't want to specify the filenames on the command line your program can place them in the ARGV array (and be sure to set ARGC appropriately) before the Automatic

Input Loop starts. To create a program that uses some of its arguments as regular arguments and some as filenames you can either increment ARGV past the regular arguments, or remove them from the ARGV array by setting them to a null string ("") so they will be ignored. You can also modify ARGV, ARGV and ARGV while the Automatic Input Loop is running. For example, to process the same file again you can decrement ARGV. To skip the next file on the command line, you can increment ARGV. To force awk to process a file of your choosing you can change ARGV[ARGV]. To add additional files to be processed to the end of the argument list use: ARGV[ARGC++] = "filename". To skip all remaining files you can set ARGV = ARGV. (Hint: to terminate processing a file in the middle of the file use: close(FILENAME).)

Your program can place filename patterns in the ARGV array. If the Automatic Input Loop detects a filename pattern when it goes to use an element of the ARGV array, it processes all filenames that match that pattern.

## Command Line Variable Assignments

Variable assignments may appear on the command line of a TAWK program. If the TAWK program consists of multiple program files then only variables that are declared "global" can be set. If the TAWK program consists of a single program file then TAWK variables that are undeclared can also be set.

Variable assignments can appear on the command line in two different ways. The -v option allows you to set variables before the program starts, for example:

```
myprog -vthis=that -v these=those arguments ...
```

Sets variable "this" to the value "that" and variable "these" to the value "those" before the program starts. Note that a space after the -v is optional.

If your program uses an Automatic Input Loop then variable assignments may be intermixed with filenames in the program arguments. The variable assignment occurs when TAWK examines that program argument for processing by the Automatic Input Loop. For example:

```
myprog -va=1 b=2 file1 c=3 file2
```

This program sets variable a=1 before the program starts. It sets b=2 after the BEGIN blocks are executed but before processing file1. It sets c=3 after processing file1 but before processing file2.

## Chapter 5: Advanced Program Input/Output

### Program Output

#### Print Statement

We've already seen the simplest form of the print statement, that is, "print" followed by an expression:

```
BEGIN {
    print 3 + 4      # prints 7
    print(3 + 4)     # parentheses are optional
}
```

Normally, parentheses are required around the arguments of a function, and the parentheses must follow the function name with no intervening space. However, the print, printf, getline, and a few other functions are special exceptions to this rule. For these functions the parentheses are optional.

If no expression is specified then the current input record (\$0) is printed. To print a blank line you must explicitly specify a null string (""):

```
BEGIN {
    print           # Prints the current input record ($0)
    print $0       # Identical to: print
    print ""        # Prints a blank line
}
```

#### OFS and ORS Variables

You can have any number of comma-separated expressions in a **print** statement:

```
BEGIN {
    print 3+4, 5+6, 7+8 # prints 7 11 15
}
```

TAWK prints the result of the expressions separated by the output field separator. The output field separator is a built-in variable named OFS that defaults to a space, but you can change it with a simple assignment:

```
BEGIN {
    OFS = "..."
    print 3+4, 5+6, 7+8 # prints 7...11...15
}
```

You don't have to use the OFS variable. The above could also be accomplished by simply printing a string that is created by concatenating the individual strings that you want to print. For example:

```
BEGIN {
    print 3+4 "..." 5+6 "..." 7+8 # prints 7...11...15
}
```

However, using the OFS variable runs slightly faster. Beware that the OFS variable is also used to reconstruct \$0 whenever fields are modified. See the chapter on Fields for examples using OFS for this purpose.

The built-in variable ORS (Output Record Separator) specifies the string that ends the line that is output by **print**. The default is a newline character ("\n"), which is appropriate for text files.

#### Printf Statement

The printf statement provides formatted output capabilities. You can specify the exact format for numbers (like how many digits you want printed), the exact length for strings, and minimum/maximum printed field widths to help you line up your output in columns.

The first argument to the `printf` statement is a format string that can contain regular characters that are just printed and format specifications that begin with the percent character (%).

Some of the most often used format specifications are as follows:

<code>%c</code>	a single character
<code>%s</code>	a string of characters
<code>%f</code>	a number in floating point notation: 1.23456
<code>%e</code>	a number in exponential notation: 1.23e+012
<code>%d</code>	an integer in decimal notation

The format specifications in the format string are interpreted from left to right, and each format specification tells `printf` how to format the next argument to `printf`, starting at the first argument after the format string.

For example, both of the following print "hello world":

```
BEGIN {
    printf "hello world\n"
    printf "%s %s\n", "hello", "world"
}
```

Unlike the `print` statement, the `printf` statement does not automatically end the line with the value of the `ORS` variable. If you want to end the line, you must remember to print the `"\n"` character at the end of the line. This gives you full control of the output. Notice that we included the `"\n"` character in both of the above `printf` statements to end the line.

The `printf` format specifications may also specify the width of the output string, whether it should be truncated, whether it should be left or right justified, and many other things. The `printf` statement and its format specifiers are described in great detail, along with other functions, in the chapter on Built-In Functions.

## Printing to Files and Pipes

You can redirect the output of a `print` or `printf` statement to a file like this:

```
BEGIN {
    print "hello world" > "out.txt"
}
```

This prints "hello world" to the file "out.txt". `Print` actually takes an expression after the `>` symbol, so you could put the file name in a variable:

```
BEGIN {
    output_file = "out.txt"
    print 3 + 4 > output_file
}
```

This is a good idea if you have several `print` statements that you're redirecting to a file, and you want to specify the actual file name in only one place. If you're specifying the actual file name in the `print` statement, don't forget the quotes. Otherwise TAWK thinks it's a variable and uses the contents of the variable as the name of the file as in the last example above. If you're lucky the variable will be uninitialized, and you'll get a runtime error, making it easy to figure out your mistake. If you're not lucky, the variable will have a legal file name in it, and your output may end up in a surprising location.

If the file to which you're redirecting output already exists, it will be over-written by the first `print` to it. The second `print` will append.

```
BEGIN {
    output_file = "out.txt"
    # prints to first line of out.txt
    print 3 + 4 > output_file
    # prints to second line of out.txt
    print 5 + 6 > output_file
}
```

You can append to a file by replacing ">" with ">>" in the print or printf statement:

```
BEGIN {
    output_file = "out.txt"
    # Appends to existing file out.txt
    print 3 + 4 >> output_file
    # So does this
    print 5 + 6 >> output_file
}
```

It's really only the first **print** or **printf** command to a file that determines whether it over-writes or appends. The rest always append:

```
BEGIN {
    output_file = "out.txt"
    # Appends to existing file: out.txt
    print 3 + 4 >> output_file
    # This also appends to file out.txt:
    print 5 + 6 > output_file
}
```

You can also redirect your output to a command by replacing ">" with "|" in the print statement:

```
BEGIN {
    cmd = "sort"
    print 5 | cmd
    print 4 | cmd
    print 3 | cmd
    close(cmd)
}
```

The **close** function call is required to close the pipe and force execution of the cmd (the sort command in this case.) The above example produces the following output:

```
3
4
5
```

As with the file redirection, you can use any expression for the command. If you intend to specify the command itself in the print statement, don't forget the quotes. The following example won't work if you forget the quotes:

```
BEGIN {
    print 5 | "sort"
    print 4 | "sort"
    print 3 | "sort"
    close("sort")
}
```

## Reading Data from Files

### Getline Statement

The **getline** function is the primary method used in TAWK to read data from a file. The **getline** function has several different forms depending on whether you want to read the next record using the Automatic Input Loop, or whether you want to read from a specific file.

To read a record from the Automatic Input Loop, use:

```
getline
or
getline variable_name
```

Getline reads the next record that would be processed by the Automatic Input Loop. If a *variable\_name* is specified, the record is placed in that variable; otherwise the record is placed in the current record variable: \$0. The Automatic Input Loop gets the filenames to be processed from program's command line. As always, whenever \$0 is changed the corresponding fields: \$1, \$2, etc. and the variable: NF (Number of Fields) are also updated. The variables: NR (Total Number of Records) and FNR (Number of Records from current File) are also incremented.

To read a record from a specific file, use:

```
getline < "filename"
or
getline variable_name < "filename"
```

This reads the next record from the file: *filename*. If a *variable\_name* is specified, the record is placed in that variable; otherwise the record is placed in the current record variable: \$0. The "*filename*" can be specified as a string as shown, or can be a variable, or any TAWK expression. As a special case, if the filename is "-", the record comes from the standard input. The above form of getline does not increment the NR and FNR variables, unless the specified *filename* is the current input file being processed by the Automatic Input Loop. This form of getline also prevents the BEGINFILE and ENDFILE blocks from being executed automatically.

Getline returns 1 if a record is returned; 0 if there is no more input available; or -1 if there is an error while trying to open the file or read the record. Failure is most often caused by the specified file not being found, or already being used in a shared environment.

Getline can also read records from another command using a "pipe". This is discussed in the Built-In Functions chapter.

#### Examples:

It is VERY IMPORTANT to check the return value from **getline**. Consider the following program that simply reads the records from a file and prints them back out:

```
# NOT RECOMMENDED:
BEGIN {
    while (getline x < "filename") {
        print x
    }
}
```

What happens if the file: *filename* does not exist, or is already in use by another program? The **getline** function will return -1, which the **while** loop will consider TRUE because it is a non-zero value. So the **while** loop will continue forever in an infinite loop. To write this program to correctly handle the case where an error is encountered by **getline**, use this:

```
# RECOMMENDED:
BEGIN {
    while (getline x < "filename" > 0) {
        print x
    }
}
```

The above example is similar to the previous one, but it handles errors from the **getline** function appropriately. By testing the return value from **getline** to see if it is greater than zero, we insure that the **print** statement is only executed when **getline** successfully reads a record. Do not be confused by the < and > symbols: the < symbol is part of the syntax of the **getline** function and specifies the file to read; the > symbol tests the return value of getline to see if it is greater than 0. If you want, you can add parentheses to the statement to help clarify it as follows:

```
while ((getline x < "filename") > 0)
```

A program that uses an Automatic Input Loop can always be transformed into a program that calls **getline** directly. For example, the following programs are equivalent:

```
# This program prints lines containing "this"
/this/ { print $0 }

# This program is the same as the above.
BEGIN {
    while (getline > 0) {
        if ($0 ~ /this/) print $0
    }
}

# This program is almost the same as the above.
BEGIN {
    while (ARGI < ARGV) {
        while (getline < ARGV[ARGI] > 0) {
            if ($0 ~ /this/) print $0
        }
        ARGI++
    }
}
```

Note: the final program above is not quite identical to the other two. If no arguments are specified on the command line of a TAWK program, the Automatic Input Loop reads from the standard input. The third program above simply does nothing in this case. To truly make the program identical to the other two you would have to check for no command line arguments and add a special case as follows:

```
BEGIN {
    if (ARGC == 1) {
        # Read from standard input:
        while (getline < "-" > 0) {
            if ($0 ~ /this/) print $0
        }
    }
    else while (ARGI < ARGV) {
        # Read from files specified on command line:
        while (getline < ARGV[ARGI] > 0) {
            if ($0 ~ /this/) print $0
        }
        ARGI++
    }
}
```

## Specifying a Different Input Record Format

The `getline` function uses the current values of the variables: `RS` (Record Separator) and `RECLLEN` (Record Length) to determine the format of the next record to read. The `RS` variable defaults to a new-line character, which means that the records are just the lines in the file. You can change `RS` to be any single character, or to be a regular expression that must match the record separator. The `RECLLEN` variable specifies a maximum record length that may not be exceeded. If the record separator specified by the `RS` variable is not found in this many characters, the record is returned anyway.

The variable: `RSM` (Record Separator Matched) holds the last record separator that was found. `RSM` is set both by the `getline` function and by the Automatic Input Loop. `RSM` is useful when `RS` is set to a regular expression, so you can tell what record separator was actually matched. Another use for `RSM` is that when a record is truncated due to `RECLLEN` exceeded, `RSM` contains an empty string, meaning that no record separator was found.

To read fixed length records, use the `fread` function. This function returns the number of characters you specify, without checking for record separators. An empty string, or a string of fewer characters than you specified, is returned when you reach the end of the file. Its kind of a boring function, so we won't talk about it here. See the Built-In Functions chapter.

## Communicating with the User

You can read or write directly to the user's terminal (also called a console) using the special file name: "CON" (in DOS, OS/2, or Win32) or "/dev/tty" (in UNIX.) By using this special filename, you guarantee that the input or output will go to the user's terminal, even if the standard input or standard output of the program was redirected by the user. For example:

```
BEGIN {
    printf "Enter the filename: " > "CON"
    getline name < "CON"
    if (name !~ /[a-zA-Z0-9+@#/\[\]\+\/] {
        print "Invalid filename! Exiting!" > "CON"
        abort(2)
    }
}
```

In the above example we printed a prompt to the user's terminal using the **printf** function. Because we did not include a "\n" in the string to **printf**, the line is not automatically terminated and the cursor is left on the same line as the message: "Enter a filename: ". The **getline** function reads a line of input from the user and places the entire line in the variable: name. Every thing the user typed on the line, except the final Enter key, is returned in this variable. It is your responsibility to verify that the user's input is correct for your purposes. The final **if** statement checks to see if the user's input looks somewhat like a valid filename, and prints an error message to the terminal and terminates the program if it does not.

The **getkey** function allows you to read a single key stroke from the user's terminal. Unlike the **getline** < "CON" technique, this function does not wait for an entire line to be entered; it returns each key as it is pressed. The **kbhit** function can be used to determine if a key stroke is already waiting. These functions can be used together to design interactive programs. The example programs installed with TAWK include a pop-up menu program that uses the **kbhit** and **getkey** functions extensively.

## Closing files and pipes

There are limits on the number of files and/or pipes you can have open. TAWK prints a warning message if you attempt to open too many files or pipes simultaneously. The **close** function closes a file or pipe. To close a file, specify the name of the file. To close a pipe, specify the exact same string that was used to read to or write from the pipe. If you only read part of a file or pipe and are done with it, you should close it using the **close** function to free the operating system resources (like file descriptors) that were in use. It is not strictly necessary to close files or pipes that you read to completion (that is, you read them until **getline** returns 0), because TAWK automatically releases the operating system resources in this case.

In some versions of TAWK, all output to a pipe is saved in a temporary file until you close the pipe, at which time the command is executed. So you must close the pipe yourself if you want to force the command to run before your program is finished. An unclosed pipe is closed automatically when your program ends (unless the program is aborted), and the command will run then.



## Chapter 6: TAWK Expressions

An expression combines language elements (like strings, numbers, patterns or variables) with operators (like + or -) to produce a new result. TAWK provides all the standard arithmetic operators, as well as operators to work on strings and to match patterns.

### Summary

The following table lists the expression operators in the order of precedence, from highest at the top to lowest at the bottom. Operators appearing in each grouping have the same precedence. "Order of Precedence" means that when there are two different ways to evaluate an expression, operators higher in the table are evaluated before operators lower in the table. In the table below, A, B and C represent any expressions, and V represents a variable or array.

**Table of TAWK Expressions**

<u>Expression</u>	<u>Meaning</u>
Grouping:	
( A )	parentheses are used for grouping
Fields:	
\$A	field reference (Note 1)
Increment and Decrement Operators:	
++V or V++	auto-increment variable
V-- or --V	auto-decrement variable
Exponentiation:	
A ^ B	A to the power B (Note 1)
Other Unary Operators:	
+ A	unary plus
- A	unary minus
! A	logical NOT
Multiplicative Operators:	
A * B	multiplication
A / B	division
A % B	remainder
Additive Operators:	
A + B	addition
A - B	subtraction
String Concatentation:	
A B	string A concatenated with string B (no explicit operator is required)
Comparison (Note 4):	
A == B	TRUE if A equals B
A != B	TRUE if A is not equal to B
A < B	TRUE if A is less than B
A <= B	TRUE if A is less than or equal to B
A > B	TRUE if A is greater than B
A >= B	TRUE if A is greater than or equal to B
Pattern Matching:	
A ~ B	TRUE if A matches regular expression B
A !~ B	TRUE if A does not match regular expression B
Array Membership:	
A in V	TRUE if variable V is an array that contains an element whose index is A

Logical And:

A && B                      TRUE if both A and B are TRUE (Note 2)

Logical Or:

A || B                      TRUE if either A or B is TRUE (Note 2)

Conditional Expression:

A ? B : C                      if A is TRUE result is B, else C (Note 1)

Assignment:

V = A                      assignment (Notes 1, 3)

Notes:

1. These operators are evaluated from right to left. All other operators are evaluated from left to right. For example, A+B+C means: (A+B)+C. But A^B^C is evaluated right-to-left, so it means: A^(B^C).
2. Not only do || and && evaluate from left to right, but they stop evaluating as soon as the result is known.
3. In addition to plain assignment using "=", there are also short-hand assignment operators: += -= \*= /= %= and ^=. They are described below.
4. Most operators coerce their operands to the appropriate type, either number or string. Comparisons are an important exception: the comparison will be numeric if both arguments are numbers or lexical if both arguments are strings. If one argument is a number and one is a string, the expression is ambiguous. See the discussion of comparison below for the rules applied in this case.

## Using strings and numbers

Variables in TAWK can contain any type of data including numbers, strings, regular expression patterns, etc. When a variable is used in an expression in a context that requires a number or string, TAWK automatically converts the value to the required type: either number or string.

### Converting strings to numbers

When a string is used in an expression where a number is expected, TAWK will automatically convert the string to a number. For example:

```
x = 3 + "4"                      # Result is 7
```

In this example, addition requires numbers, so the string "4" was converted to the number 4. When converting a string to a number, TAWK skips any leading spaces or tabs, then uses whatever part of the string looks like a number. The number may include a leading + or - sign. Any valid number format may be used, including integers, exponential notation, and hexadecimal notation. For example:

```
x = 3 + "4score"                      # Result is 7
x = 3 + " 4score"                      # Result is 7
x = 3 + "1.7"                      # Result is 4.7
x = 3 + "1e2"                      # Result is 103
x = 3 + "0x10"                      # Result is 19
x = 3 + "this4"                      # Result is 3
```

In the examples above, "1e2" is a valid exponential notation for the number 100, and "0x10" is a valid hexadecimal notation for 16. The last example above shows that if the string does not begin with a number, it is treated as 0.

### Converting numbers to strings

When a number is used in an expression where a string is required, TAWK will automatically convert the number to a string. For example, the string concatenation operator requires strings, so any numbers are converted to strings:

```
x = 3 "four"                      # Result is "3four"
```

In the above example, the concatenation operator had two arguments, the number 3 and the string "four". First the number 3 was converted to the string "3", then concatenated to the string "four" to get the result.

Converting an integer to a string is straightforward because there is one obvious string representation for each integer. By contrast each floating point number has numerous representations. For example, the number 10.0 could be converted to the string "10.0" or "10.00" or "1e1", etc. By default TAWK will use either fixed point or exponential notation, whichever seems more appropriate, and will truncate the value to approximately six significant digits. For example:

```
x = 1000000 "hi"      # Result is "1000000hi"
x = 1000000000 "hi"    # Result is "1e+09hi"
x = 9.87654321 "hi"    # Result is "9.876543hi"
```

You can change the default way TAWK converts floating point numbers to strings by setting the built-in variables CONVFMT and OFMT. For example, you can request more significant digits, or force TAWK to use fixed point or exponential notation. The CONVFMT and OFMT variables are discussed in the chapter: TAWK Built-In Variables.

## Arithmetic operators

Arithmetic operators are straightforward in TAWK:

```
x = 3 + 4      # Addition: result is 7
x = 3 - 4      # Subtraction: result is -1
x = 3 * 4      # Multiplication: result is 12
x = 4 / 2      # Division: result is 2.
x = 2 ^ 3      # Exponentiation: result is 8
x = 4 % 3      # Modulo (remainder after division): result is 1
```

When you have an expression with more than one operator that could be evaluated in different orders, the operator with the higher "precedence" is evaluated first. The precedence of operators is indicated in the table above: operators higher in the table are evaluated before operators lower in the table. For example:

```
x = 3 + 4 * 5  # Multiplication first, result is 23
```

Because multiplication (\*) has a higher precedence than addition (+) this expression is evaluated as:

```
x = 3 + (4 * 5)  # Result is 23
```

When you have two operators with the same precedence, they are normally evaluated from left to right, as you would expect. (The only significant exception is exponentiation, which is customarily evaluated from right to left.) For example:

```
x = 3 - 4 + 5
```

is evaluated as:

```
x = (3 - 4) + 5  # Result is 4
```

You can use parentheses to force an explicit order of evaluation: a parenthesized expression is always evaluated first.

Division and modulo may yield floating point results. For example:

```
x = 4 / 2      # 4 divided by 2 is 2
x = 5 / 2      # 5 divided by 2 is 2.5
x = 4 % 3      # 4 modulo 3 is 1
x = 4.5 % 3    # 4.5 modulo 3 is 1.5
```

If you want an integer result, use the built-in int() function, which returns the integer part of its argument, for example:

```
x = int( 4 / 2 )  # Result is 2
x = int( 5 / 2 )  # Result is 2
```

Division by zero causes a warning message to be printed. Using the result of division by zero in another operation may cause a fatal error.

## Increment and Decrement Operators

TAWK provides a short-hand for the common task of adding one to, or subtracting one from, a variable. The ++ operator is the increment operator. It can be used in a statement by itself, as in:

```
a++      # Means a = a + 1
```

Or it can be used in an expression. In this case, it returns the incremented value if the ++ precedes the variable, or it returns the unincremented value if ++ follows the variable. For example:

```
c = x[++a]
```

is the same as

```
a = a + 1
c = x[a]
```

while

```
c = x[a++]
```

is the same as

```
c = x[a]
a = a + 1
```

The -- operator is the decrement operator and is analogous to ++, except that it decrements a variable by one.

## String concatenation

String concatenation has no explicit operator. String concatenation is implied simply by placing two expressions next to each other, optionally separated by spaces or tabs. For example, the following expression:

```
x = "one" "two"
```

combines the two strings together into a new string: "onetwo", which is assigned as the new value of the variable x.

## Assignment

Assignment uses the equals (=) operator. For example:

```
a = 0      # Assigns the number 0 to variable a
b = "this" # Assigns the string "this" to b
c = /this/ # Assigns the regular expression to c
```

Assignment can also be used as part of an expression. For example, you can say:

```
a = b = c = 0
```

instead of

```
a = 0
b = 0
c = 0
```

The way it works is that "=" returns the value it's assigning. TAWK parses assignment from right to left, so here's how it's evaluated:

```
a = (b = (c = 0))
```

You can see that c is set to 0, then the result of that, which is 0, is passed on as an expression to set b, and the result of that, still 0, is passed on to set a.

Note the = operator means assignment, but the == operator means logical comparison, which is described below. Be careful not to confuse the two.

### Assignment Short-Hand

There is a set of short-hand assignment operators that are included for compatibility with the "C" programming language. The short-hand assignment operators and their meaning are:

```
a += 3 # means: a = a + 3
a -= 3 # means: a = a - 3
a *= 3 # means: a = a * 3
a /= 3 # means: a = a / 3
a %= 3 # means: a = a % 3
a ^= 3 # means: a = a ^ 3
```

### Comparison operators

The comparison operators are

<u>Symbol</u>	<u>Meaning</u>
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

The comparison operators return 1 if the comparison is TRUE, or 0 if the comparison is FALSE. For example, the following prints "true":

```
if (3 < 4) print "true" else print "false"
```

You can also use the value of a comparison operator in an expression, for example:

```
x = 3 < 4      # Result is 1
x = 3 > 4      # Result is 0
x = 7 + (3 < 4) # Result is 8.
```

The comparison operators can compare either numbers or strings. Numbers are simply compared numerically. Strings are compared by ASCII collating sequence. For example, the following are all TRUE:

```
"a" < "b"
"A" < "a"   # A comes before a
"a" < "aa"
"ab" < "ac"
"1" < "a"
"10" < "2"  # String comparison!
```

If you want to do a case-insensitive comparison use the built-in toupper() or tolower() function, which takes a string argument and returns the same string with all characters converted to upper (lower) case. For example:

```
toupper("a string") == toupper("A String")
```

If you try to compare things that are not strings or numbers, for example, a regular expression or an array, you will get a "note" message warning you that your program is doing something suspicious. (Note messages can be turned off with the WARNINGS variable.) However, two special cases are recognized: you can compare open file descriptors, such as returned by the fopen() function. These are guaranteed to compare "equal" only if they really are the same file descriptor. Similarly, you can compare an array to another array using == or !=. The comparison is guaranteed to be "equal" only if the two arrays are the exact same array. Note that two different arrays that happen to have the same contents would be "not equal".

## Comparing strings with numbers

When you compare a string with a number, TAWK must decide whether to perform a string comparison or a numeric comparison. In the above examples, it was obvious whether the things being compared were numbers or strings, but it is usually more obscure. For example, consider the following user-defined function:

```
function compare(a,b)
{
    return a <= b
}
```

TAWK does not know, and neither does the programmer, whether this function will do a numeric or a string comparison until the function is actually called.

A good rule of thumb is that when you use a relational operator and you're not sure the arguments are going to be the same type, force the conversion yourself. An easy way to do this is to add 0 to the arguments for numeric comparison, or concatenate an empty string "" for string comparison. For example, the above could have been written:

```
# To ensure numeric comparison, do this:
return (a + 0) <= (b + 0)

# To ensure string comparison do this:
return (a "") <= (b "")
```

In the first example, a and b were both converted to numbers before the comparison was made. In the second example, they were both converted to strings. (A note: the parentheses were not needed in these examples, because both addition and string concatenation have higher precedence and are thus performed before the comparison operator (<). However, the parentheses make the examples easier to read.)

If you ignore the above advice and compare a string and a number, TAWK will determine whether to use numeric or string comparison. If the string consists only of a number, with optional leading spaces or tabs and an optional + or - sign, then TAWK converts the string to a number and performs a numeric comparison. Otherwise TAWK converts the number to a string and performs a string comparison, EVEN IF THE STRING BEGINS WITH A NUMBER! As with numeric conversion, any valid number format will be recognized, including exponential and hexadecimal notation.

For example, the following are all TRUE:

```
2 == "2"           # Numeric comparison
2 == " 2"          # Numeric comparison
2 == " +2"         # Numeric comparison
20 == "2e1"        # Numeric comparison
16 == "0x10"       # Numeric comparison

2 != "a"           # String comparison
2 != "2a"          # String comparison
```

In the last two examples, TAWK converted the 2 to a "2" and used string comparison because the string on the right did not consist solely of a number.

## Comparing Uninitialized Variables

For comparison purposes, uninitialized variables have the string value "" and the numeric value 0. So if x is an uninitialized variable, the following are all TRUE:

```
x == ""           # String comparison of "" to ""
x == 0            # Numeric comparison of 0 to 0
x != "0"         # String comparison of "" to "0"
```

## Comparing Fields

A field that consists solely of a number, optionally preceded by spaces, tabs, and an optional + or - sign, is considered to be a number for comparison purposes. So in an expression like the following, which compares field 1 in the current input record with field 2:

```
$1 < $2
```

TAWK may use either string or numeric comparison, depending on the values of the input fields when the program is run. This useful feature allows you to write programs to process input data, without having to worry over-much about whether the data will be strings or numbers.

The array elements created by the `split()` or `splitp()` functions are treated the same way.

### The Difference Between `=` and `==`

Be careful not to confuse `=` with `==`. The first is assignment and the second is logical comparison. A common error in TAWK is to use `=` instead of `==`, for example:

```
if (x == 2) ...    # CORRECT
if (x = 2) ...    # WRONG
```

The first compares the value of `x` to 2. The second assigns the value 2 to `x`, and then the value 2 is tested by the `if`. It will always be TRUE! TAWK prints a warning message if you place an assignment like this in an `if` statement or other boolean context.

However, it is legal to use an assignment in any expression, and there are times when it makes sense inside an `if` statement. For example, the following program opens the file: "filename" with the `fopen` function, which returns 0 only if it fails:

```
fd = fopen("filename", "r")
if (fd != 0) {
    print "Filename was opened OK"
}
```

We don't recommend it, but this could be re-written as:

```
if (fd = fopen("filename", "r")) {
    print "Filename was opened OK"
}
```

This is legally correct, but generates a warning message. If you really want to do this, you can tell TAWK not to print the warning message by enclosing the assignment in an extra level of parentheses, like this:

```
if ((fd = fopen("filename", "r"))) {
    print "Filename was opened OK"
}
```

### Logical AND, OR, NOT

The results of comparisons, or any other expression, can be joined with the following logical operators:

<u>Symbol</u>	<u>Meaning</u>
<code>&amp;&amp;</code>	Logical AND
<code>  </code>	Logical OR
<code>!</code>	Logical NOT

For example:

```
if (a < 4 && b < 5) ...
```

Means: if (a less than 4 AND b less than 5)

```
if (a < 4 || b < 5) ...
```

Means a less than 4 OR b less than 5

```
if ( !(a < 4) ) ...
```

Means if NOT (a less than 4), that is, it evaluates (a less than 4), then takes the logical complement (NOT) of that, which is also the same as: if (a is not less than 4).

The AND (&&) and OR (||) operators evaluate the expressions from left to right, and stop when the result is known. So (a < 4 || b < 6) does not evaluate the b < 6 part if a < 4 because it doesn't matter what value b is. This can make a difference if the part being skipped has a side effect, for example, if it contains a function call that must be executed.

## The Difference Between Logical And Bitwise Operators

The logical operators are used where a boolean expression is expected, for example, in an **if** statement. Logical operators examine their arguments and return TRUE (1) or FALSE (0) depending on whether the logical condition is true.

TAWK also has bitwise operators that are accessed by the functions and(), or(), xor() and not(). The bitwise operators examine each bit of their 32 bit integer arguments and return a 32 bit value that is the result of performing the specified operation (and, or, xor or not) on each of the corresponding 32 bits in the two arguments.

The logical operators (&&, ||, !) should be used inside **if** statements, **while** loops, or wherever you are testing a boolean condition. The bitwise operators should only be used when you need to perform a bit by bit operation on integer values.

## Pattern Matching Operators

The matching operators are used to determine if a string matches a regular expression pattern:

<u>Expression</u>	<u>Meaning</u>
x ~ /pattern/	TRUE if x matches the regular expression pattern
x !~ /pattern/	TRUE if x does not match the regular expression pattern

The match normally looks for the pattern anywhere in the string. As always the regular expression can begin with ^ to force it to start matching at the beginning of the string, or can end with \$ to force it to match all the way to the end of the string.

The pattern matching operator is extremely powerful and replaces a host of functions that you would typically find in other programming languages. For example, the following functions can be written to perform tests for upper-case, lower-case, digits, etc.:

```
# This function returns TRUE
# if the string contains only digits:
function isdigit(str) { return str ~ /^[0-9]+$ / }

# This function returns TRUE
# if the string contains only upper-case letters:
function isupper(str) { return str ~ /^[A-Z]+$ / }

# This function returns TRUE
# if the string contains only letters:
function isletter(str) {
    return str ~ /^[a-zA-Z]+$ /
}

# This function returns TRUE if the string
# contains the representation of an integer.
# (optional sign followed by one or more digits)
function isinteger(str) {
    return str ~ /^[+-]?[0-9]+$ /
}
```

The regular expression to be matched can also be specified as a string that is determined at run-time, for example:



```

BEGIN {
    letter = "[a-zA-Z]"
    number = "[0-9]"
    # This is TRUE if x is a letter or number:
    if (x ~ (letter "|" number)) ...
}

```

However, specifying the regular expression /like this/ is much faster.

The ~ operator is very similar to the match function. The following two statements are nearly identical:

```

if (x ~ pattern) ...
if (match(x,pattern)) ...

```

The only difference between these statements is that the match function sets the global variables RSTART and RLENGTH.

TAWK has another hidden use for the pattern-matching operator. Whenever a regular expression is used in the pattern part of a pattern action program block, it is actually evaluated using the ~ pattern-matching operator. In other words a pattern-action program block like this:

```

/pattern/ { statements }

```

is evaluated by TAWK as if it had been written like this:

```

$0 ~ /pattern/ { statements }

```

The symbol \$0 stands for the current record. This says that the statements are to be executed if the current record matches the specified regular expression pattern.

## Conditional Expression

A conditional expression is a shorthand that avoids using **if** and **else**. It uses two symbols: the question mark (?) and colon (:). Its general form is:

```

expr1 ? expr2 : expr3

```

It works like this: *expr1* is evaluated first. If *expr1* is TRUE (that is, non-zero) then the value of this expression is *expr2*, otherwise it is *expr3*.

```

if (a < b) c = a else c = b

```

It could be replaced with a single conditional expression:

```

c = a < b ? a : b

```

## Chapter 7: Functions

TAWK provides a large number of built-in functions. TAWK also allows user-defined functions. This chapter describes how to create and use user-defined functions.

### Summary

A user-defined function looks like this. Square brackets: [ ] indicate optional items:

```
[ scope ] function name( [ argument_list ] )
{
    [ local_variable_declaration ... ]
    [ statements ]
}
```

The function *argument\_list* looks like this:

```
[ ref ] variable [ , [ ref ] variable ] ...
```

There can be optional *local\_variable\_declarations*, each of which looks like this.

```
local variable [ = expression ] [ , ... ]
```

Function *names* may consist of letters, underscores or digits, and may not start with a digit. Built-in function names are reserved, so user-defined functions may not override built-in functions. The optional *scope*, above, is either **local** or **global**. Global functions can be called from any TAWK program file, while local functions can be called only from the TAWK program file in which they are defined. If no *scope* is specified, the default scope for functions is global.

The function may optionally have local variables indicated by a **local** keyword in the function body before any *statements*. Multiple local variables can be declared with multiple **local** keywords (for example: local a; local b; local c), or can be separated by commas after a single **local** keyword (for example: local a, b, c). Function local variables that are not given an explicit initialization *expression* are always given the “uninitialized” value to start. Values assigned to function local variables are lost when the function returns, and TAWK always automatically cleans up any local storage used by such variables.

The call to a function need not specify all function arguments. Unspecified arguments have the “uninitialized” value inside the function. The built-in `argcount()` function returns the number of actual arguments passed to the current function.

Function *arguments* are either pass-by-value (the default) or pass-by-reference (if the **ref** keyword is specified before the *argument* in the function definition.) If an argument is pass-by-value, and the actual parameter specified when the function is called is a string, an integer, or anything except an array, then changes to that argument in the function can not affect the caller of the function. If a pass-by-value argument is an array, however, changes made to the array inside the function will modify the original array. (There are obscure exceptions, discussed below.) If an argument is pass-by-reference, and the actual parameter specified when the function is called is a variable name, then any changes to this argument in the function will be made to the original variable. This allows a function to return information to the caller of the function. If the actual parameter to a pass-by-reference argument is not a variable, it is not an error: the argument is simply passed by value.

A function can return a value with the **return** statement. Functions that do not explicitly return a value automatically return the special “uninitialized” value, when used in a context that requires a return value.

### Discussion

Here is a simple user-defined function called “sum”. This function has three arguments: a, b and c. It returns the sum of the three numbers.

```
function sum(a, b, c)
{
    return a + b + c
}
```

You could optionally include a **local** or **global** keyword. Here is the same function that will be accessible only in the source file in which it appears:

```
local function sum(a, b, c)
{
    return a + b + c
}
```

Here is an example calling the function:

```
BEGIN {
    print sum(3,4,5) # prints 12
}
```

You do not need to specify all the function arguments when you call the function. Unspecified arguments are given an "uninitialized" value that will be 0 or "" inside your function, whichever is appropriate. For example:

```
BEGIN {
    print sum(3,4) # prints 7
}
```

Functions can have local variables. The following example computes the tangent of an angle specified in degrees:

```
function tandeg(x) {
    local pi = atan2(0,-1)
    x = x * pi / 180
    return sin(x) / cos(x)
}
```

In the above function `pi` is a local variable, which means that the variable can only be used inside the function. The variable is created when the function is called, and destroyed when the function returns. If a function local variable has the same name as a variable outside the function, the local variable used inside the function is a separate variable, and does not affect the global variable.

## Function Arguments

Notice that the `tandeg` function changes the value of the argument `x`. What if you called it as follows:

```
BEGIN {
    y = 45
    print tandeg(y) # prints 1
    print y # prints 45
}
```

Why didn't variable `y` get changed by function `tandeg`? The answer is that when you call a user-defined function, TAWK makes a copy of string and numeric arguments so that they'll still be the same when you return. TAWK does not copy arrays, so a function can change the values of the array elements. For example:

```
function neg_array(x) {
    for (y in x) x[y] = -x[y]
}

BEGIN {
    z[1] = 3
    z[2] = 4
    print z[1], z[2] # prints 3 4
    neg_array(z)
    print z[1], z[2] # prints -3 -4
}
```

Notice that there is no **return** in function: `neg_array`. You only need a **return** when the code calling the function is expecting a returned value. This call is not expecting a return value:

```
BEGIN {  
    neg_array(z)  
}
```

This call IS expecting a return value:

```
BEGIN {  
    w = neg_array(z)  
}
```

This latest example would set variable “w” to uninitialized because the function “neg\_array” does not return a value.

Notice also in function “neg\_array” that it is using a variable named “y” that is not an argument to the function. This will affect the value of the global variable “y”, so if you were using “y” in your calling code, it would get clobbered by this function and you could spend a while trying to figure out what went wrong with your code. That’s why it’s a good idea to make sure all variables you use in your function are either arguments, intentionally global, or declared local. Here’s how you declare a variable local:

```
function neg_array(x)  
{  
    local y  
    for (y in x) x[y] = -x[y]  
}
```

An example of an intentionally global variable might be “pi”, which you set to atan2(0,-1) once and use in many other places in your code.

There can be any number of local variables in the function, and any number of comma-separated variables listed in the local statement. The one restriction is that the local variable declarations must come before any other statements in the function. A local variable declaration can also initialize the local variable:

```
function degrees_to_radians(x)  
{  
    local y  
    local pi = atan2(0,-1)  
    for (y in x) x[y] = x[y] * pi / 180  
}
```

This converts an array of values in degrees to radians. This is more efficient because it calls the atan2 function just once.

Recursive functions are allowed in TAWK. The following recursive function computes factorials:

```
function factorial(n)  
{  
    if (n <= 1) return 1  
    else return n * factorial(n-1)  
}  
  
BEGIN {  
    print factorial(3)  
}
```

Particularly for recursive functions, you must remember to declare the variables used in the function as **local** variables.

Here’s how to make an array local to a function:

```
function example()  
{  
    local x  
    x[1] = ...  
    x[2] = ...  
    ...  
}
```

When you leave function “example”, the array “x” will go out of scope, deleting all its elements.

The **return** statement returns just one thing to the caller. What if you want to return more than one thing? One way to do this is to return an array containing the information. For example, a function might consult a data-base and return the name and address of a customer:

```
function get_customer() {
    local ret_array, cust_name, cust_addr
    ... # code goes here to set cust_name
    ... # and cust_addr
    ret_array["name"] = cust_name
    ret_array["addr"] = cust_addr
    return ret_array
}

BEGIN {
    local x = get_customer()
    print "name =", x["name"], "addr =", x["addr"]
}
```

Another way to return multiple values is to use pass-by-reference arguments. If an argument is declared with the **ref** keyword, then the function can change the values of the arguments to return those values to the caller. For example:

```
function get_customer(ref name, ref addr) {
    local cust_name, cust_addr
    ... # code goes here to set cust_name
    ... # and cust_addr
    name = cust_name
    addr = cust_addr
}

BEGIN {
    local x, y
    get_customer(x, y)
    print "name =", x, "addr =", y
}
```

Either of the above two methods solves the problem. Which method you use is a matter of personal preference. Notice that some of TAWK's built-in functions also use pass-by-reference to return multiple values to the user. For example, look at the `scr_gcp` function: this function uses pass-by-reference to return two different values: the row and column of the cursor.

## Details of Arrays as Function Arguments

This is a discussion of the technical details of array arguments, for advanced users.

When an array is passed-by-value to a function, if the function modifies the array, the modifications will normally affect the original array. This is because TAWK actually treats arrays as pointers. When an array is passed-by-value, what really happens is that the pointer to the array is passed-by-value. Therefore, if the argument is used as an array inside the function, the original array, which is being pointed to by the argument, is affected. However, if an assignment is made to the argument, then the original pointer is lost, and any further array operations will not affect the original array.

Consider the following example. The first assignment will modify the original array that was passed to `demo()`. The second assignment will change the array: `y`, but will not affect the original array.

```
function demo(x) {
    x[1] = "this"
    x = y
    x[1] = "that"
}
```

The delete function can be used on a pass-by-value array, because the delete function does not actually destroy the array, it only deletes its contents, leaving an empty array. So the following function will perform as expected: it leaves the contents of array `x` with only one element, regardless of what it contained before the function was called:

```
function hey(x) {  
    delete x  
    x[1] = "that"  
}
```

Now consider the following program that calls the hey() function defined above:

```
BEGIN {  
    x = "hello world"  
    hey(x)  
    print x[1]    # Prints "this"  
}
```

We already know that pass-by-value arrays only work because what is really passed is a pointer to an array. But in this example, x is not an array. How can this possibly work? The answer is that TAWK performs a compile-time program analysis to determine which function arguments are arrays, and inserts code to turn such function arguments into arrays just before the function is called. In the above example, just before hey() is called, the contents of x ("hello world") is discarded, and x is turned into an array. For this reason, the above code is bad form: you should not use the same variable name as both an array and a scalar value. If you printed the contents of x at the beginning of function hey(), you would find that x is already an array; it no longer contains "hello world". TAWK does not currently print a warning message for this kind of thing, but it should.

On very rare occasions, the compile-time program analysis can not determine which variables are arrays. If this happens, modifications to a pass-by-value array will not be propagated back to the caller. There are two ways to solve this problem: one is to simply declare the argument as pass-by-reference in the function definition. Alternatively, you could turn the variable into an array yourself, before calling the function. The easiest way to do this is to declare the variable as an array. The following example is crystal clear, both to you, the programmer, and, just as importantly, to the TAWK Compiler: x is an array, which is passed to hey(), which modifies it, and the modifications persist after hey() returns.

```
BEGIN {  
    local x[]  
    hey(x)  
    print x[1]    # Prints "this"  
}
```

None of the discussion above applies to pass by reference arguments. If a variable name is supplied as the actual parameter for a pass-by-reference argument, then changes to that argument inside the function ALWAYS change the original variable.

## Chapter 8: Arrays

An array reference is a variable name followed by one or more array indices enclosed in square brackets. An array is created simply by using any variable as an array. For example:

```
a[3]
b[7][9]
```

In this example, variable `a` has one subscript and is a one-dimensional array. The notation `a[3]` refers to the array element whose index is "3". Variable `b` has two indices and is a two-dimensional array. The notation `b[7][9]` refers to the array element with what you might call a row index of "7" and a column index of "9". A two-dimensional array can be thought of as a one-dimensional array, each of whose elements is itself another one-dimensional array.

Arrays in TAWK use strings for the subscripts. If you use a number as a subscript, TAWK converts it to a string. For example, this is perfectly valid:

```
a["hello"]
```

The subscript may be any expression, which is converted to a string. The following all refer to the same array element, `a["3"]`:

```
a["3"]   a[3]   a[2+1]   a[3.0]   a["3"]
```

Note however, that `a["03"]` would be a new and different array element, because the string "3" is not the same as the string "03".

There are no fixed limits on the number of subscripts an array may have, nor on the number of elements in the array. The DOS version of TAWK will automatically use extended memory, expanded memory, or disk space for array storage, so arrays are not bound by DOS's 640K memory limitation, either. However, there are practical limits on array size depending on how much memory your computer has. When the array will not fit in real memory, the access speed degrades. Under DOS, a practical limit on array size is around 150,000 elements per array. See also the discussion of sorting, below, and the `SORTTYPE` variable in the chapter on Built-In Variables.

Using an element of an array causes it to come into existence. For example,

```
x[4] = 17      # Example 1
v = x[5]       # Example 2
x[6]           # Example 3
```

Each of the above statements creates an array element. You do not have to make an assignment to create an array element: just using it creates it. So example 2 will create subscript "5" in array `x` if it did not already exist. Example 3 is a complete and valid TAWK statement; it looks like it doesn't do anything but in fact it creates subscript "6" in array `x` if it did not already exist. An array element created like this has the "uninitialized" value, which in TAWK means it will be treated as though it has the numeric value 0 or the string value "", until such time as a value is assigned to it.

You can also delete any element of an array, or an entire array, with the delete statement. Deleting an array element simply causes it to not exist any more. For example, to delete element `a[3]`:

```
delete a[3]
```

To delete all elements in array `a`, use the following. This will work even if array `a` has two or more dimensions.

```
delete a
```

Deleting array elements recovers the memory storage that was used. TAWK also automatically deletes an array if it becomes inaccessible. For example, the following user-defined function has a local variable that is used as an array:

```
function demo()
{
    local x
    x[1] = "this"
}
```

You do not need to delete array `x` before leaving this function; TAWK takes care of it. (You could also delete it yourself if you want.) It is also OK to delete an array more than once: nothing happens the second time.

## Arrays of varying dimensions

Do not use the same name for a normal variable and an array, or for two different arrays with different numbers of subscripts. For example:

```
a = 1
a[2] = 3          # NO
a[4][5] = 6       # NO

a = 1
b[2] = 3          # yes
c[4][5] = 6       # yes
```

While it is legal to change the number of array subscripts in an array, as in the first example, it is not recommended because the previous contents of the variable are lost (lost to the variable, that is: TAWK always recovers the memory storage) and you get an entirely new array. In some cases TAWK may also print a warning message to let you know you just made a mistake.

## Array Assignment

Assignment of a variable containing an array to another variable yields an additional reference to the same array. This allows array names in TAWK to be used like “pointers” are used in other languages. The following example makes variable y refer to the same array as variable x, and will print “10”:

```
x[1] = 10
y = x
print y[1]
```

Since an array assignment creates two references to the same array, the array can be manipulated using either of the variable names. In the following example, x and y refer to the same array, so changing y also changes x. This example prints “20”:

```
x[1] = 10
y = x
y[2] = 20
print x[2]
```

When an array element or an entire array is deleted using the **delete** statement, all variables referring to the same array are affected. After the **delete** statement all the variables referring to the array continue to refer to the same array, which is now empty (has no elements.) In the following example, x and y continue to refer to the same array even after the **delete** statement. This prints “20”:

```
x[1] = 10
y = x
delete x
x[2] = 20
print y[2]
```

Because a two (or more) dimensional array is really just an array of arrays, you can use the array name without all the subscripts to refer to the whole array. For example:

```
function print_row(y) { print y[1], y[2] }

BEGIN {
    x[1][1] = 11
    x[1][2] = 12
    x[2][1] = 21
    x[2][2] = 22
    print "first row"
    print_row(x[1])
    print "second row"
    print_row(x[2])
}
```



## The “in” keyword

The **in** keyword has two uses: 1) to test for the existence of an array subscript in an **if** statement; 2) to enumerate all the elements of an array using a **for** loop.

Use the **in** operator to tell if an element with a specific subscript exists. For example, assuming that array `aRay["1"]` has not been created:

```
if (1 in aRay) ...      # is FALSE
aRay[1] = 8
if (1 in aRay) ...      # is now TRUE
```

The following methods for checking the existence of an array element are NOT recommended:

```
if (aRay[1] == "") ...  # NOT recommended
if (aRay[1] == 0) ...   # NOT recommended
if (aRay[1]) ...        # NOT recommended
```

In these examples, the array element `aRay["1"]` is CREATED simply by using it, so all of these statements would create `aRay["1"]` if it did not already exist. The **in** keyword is the only safe way to check for a subscript without creating it.

To check a multi-dimensional array use multiple **in** keywords. For example, to check if `x[3][4]` exists, use this:

```
if (3 in x && 4 in x[3]) ...
```

## Arrays and “for” loops.

TAWK provides a way to enumerate the elements of an array, that is, to examine each of the array elements in turn. It uses a **for** loop with the **in** keyword, and has the following general form:

```
for (variable in array) statement
```

The *variable* must be a variable name, the *array* must be an array name, and the *statement* can be a single statement, an empty statement (indicated by a semi-colon) or a group of statements enclosed in curly braces: { }. For each element in the array, the *variable* is set to the subscript of the array element, and then the *statement(s)* are executed. The array indicies are optionally sorted before the **for** loop begins: See “Array Sorting” below.

As an example, the following code:

```
BEGIN {
  x[5] = "five"
  x[4] = "four"
  x[3] = "three"
  for (y in x) print x[y]
}
```

produces the following output:

```
three
four
five
```

Notice that the elements are retrieved and printed in a sorted order (subscript "3" first, followed by "4", followed by "5".) The order in which the array elements are retrieved is not affected by the order in which they were created.

Here is an example that shows how to go through all the subscripts in a multi-dimensional array. In this example the two-dimensional array: `x` is treated like a one-dimensional array, each of whose elements is another one-dimensional array. In this example, the first **for** loop goes through array `x`. Each element: `x[i]` is itself an array, so the second **for** loop goes through array `x[i]`, and each of those elements is referred to by `x[i][j]`.

```

BEGIN {
    x[1][1] = "hello"
    x[1][2] = "world"
    x[2][1] = "from"
    x[2][2] = "tawk"
    for (i in x) {
        for (j in x[i]) {
            print x[i][j]
        }
    }
}

```

produces the following output:

```

hello
world
from
tawk

```

## Array Sorting

When an array is used in a **for** loop with the **in** keyword, the subscripts of the array are automatically sorted, as in the above examples. If the array subscripts look like integers (like: "17") or fixed point floating point numbers (like: "1.23"), they are sorted numerically, otherwise they are sorted by ASCII collating sequence. The built-in variable: SORTTYPE can be used to turn off sorting or to change the sort order.

Notice that the INDICIES are sorted, not the VALUES in the array. If you want to sort some strings, then use them as the indicies of an array. For example, to sort the strings "this", "that" and "those", you could use:

```

BEGIN {
    x["this"]
    x["that"]
    x["those"]
    for (i in x) print i
}

```

The output will be:

```

that
this
those

```

More examples of sorting are provided below.

## Sorting Examples

To read in a file of unique lines, sort it, and print it back out, you can use this:

```

{ x[$0] }

END { for (i in x) print i }

```

The first line in this example is a *pattern-action* block with the *pattern* part missing, which means that this statement is executed for every record in the input data file(s). This first statement saves up all the input records in the array: x. When all data file(s) have been read the END block prints the records back out in sorted order.

Suppose you want to sort only on the third field. Let each array element use the third field as the index and the entire record as the value.

```

{ x[$3] = $0 }

END { for (i in x) print x[i] }

```

A column sort is just as easy. How about sorting on columns 2-6, which is five characters wide:

```
{ x[substr($0,2,5)] = $0 }
END { for (i in x) print x[i] }
```

### Preserving duplicate keys.

What if there are duplicate lines with the same key, and you need to save all the duplicates? The array indices must be unique, so if you use just the key by itself as the array index, the duplicate records will be discarded. To preserve duplicates, we will use an array index that contains the key that we want to sort on, concatenated with a unique identifier. In this example, the current record number: NR makes a convenient unique identifier, since each record appears on a separate line.

We will separate the key from the unique identifier with a period character (".") to make sure that keys that just happen to end with a number are distinguished from keys that do not. Otherwise, for example, the index used for record number 1 with the key "a1" would be indistinguishable from record number 11 with the key "a"; both would use the same index: "a11". By including the period, the indices become unique: one is "a1.1" and the other is "a.11".

```
{ x[$0 "." NR] = $0 }
END { for (i in x) print x[i] }
```

### Sorting on multiple keys.

To sort on multiple keys, use an array whose index combines all the keys into a single string. Then use a **for** loop to sort the array indices.

For example, suppose we want to sort a file where the primary sort key is in columns 2-7 (five characters), and the secondary sort key is columns 9-10 (two characters). We will also append NR to the sort keys, so that we will not discard records with duplicate identical keys.

```
{
    key1 = substr($0,2,5)
    key2 = substr($0,9,2)
    x[ key1 key2 NR ] = $0
}
END { for (i in x) print x[i] }
```

The only problem occurs when the keys are of varying lengths. For example, suppose the primary sort key is field 1 (\$1) and the secondary sort key is field 2 (\$2), and suppose they have varying lengths. Here is an example file to illustrate the problem:

```
aa    b    record 1
a     ab   record 2
```

If we just concatenate field 1 with field 2, these two records will be indistinguishable. One way to solve the problem is to separate the fields with the nul (0) character. Because the nul character is less than every other character in the ASCII chart, it causes the sort to work properly, assuming your sort keys do not contain a nul character! For example:

```
{
    x[$1 "\0" $2 "\0" NR ] = $0
}
END { for (i in x) print x[i] }
```

### Alternate Sort Orders.

The SORTTYPE variable allows you to specify several different types of sorting. For example, you can specify a straight ASCII sort, where the array indices are sorted alphabetically, or an alpha-numeric sort, where numeric array indices (in integer or decimal notation) are sorted as numbers. You can also reverse the sort order or ignore alphabetic case. SORTTYPE is described in the chapter on Built-In Variables.

What if you want specify an alternate sorting order? For example, suppose you want to sort using a character set other than English? When TAWK sorts the array indices, it sorts them in ascending order based on the ordinal value of each character. To create a custom sort order use the **translate** function to remap the characters used in the array index so that the replacement characters are arranged in ascending order. The following example translates accented characters in the record to their unaccented

equivalents before using the record as the array index. When the array is sorted, the accented and unaccented characters will be sorted into adjacent locations:

```
{
  newstr = translate($0,
    "àáâãäåèéëìíîïðóôõöùúûüåääååæææëëïíîïðóôõöùúûüåääååæææëëïííîïðóôõöùúûü",
    "aaaaaaeeeeeiioooooouuuuAAAAAAEEEEIIIIIOOOOOUUUU")
  x[newstr] = $0
}

END { for (i in x) print x[i] }
```

### Keeping an array in order.

This is a very common question. The first solution is simple: just use an integer index! For example to read in a file and then print it out again in order, just use:

```
{ x[NR] = $0 }

END { for (i in x) print x[i] }
```

Actually, you wasted your computer's time by sorting the above array. Of course, your computer may have nothing better to do, but you still could have written it this way:

```
{ x[++cnt] = $0 }

END {
  for (i = 1; i <= cnt; i++)
    print x[i]
}
```

Suppose you want to specify both the index and the value of every element in your main array, and STILL access them in the order you create them. Use an auxiliary array, which we will call "aux", which is used like a data-base index into the main array: the indices of array "aux" are the sequential integers, and the values are the indices of the main array.

```
{
  main[$3] = $0
  aux[++cnt] = $3
}

END {
  for (i = 1; i <= cnt; i++)
    print "index is" aux[i],
      "and value is" main[aux[i]]
}
```

### Arrays as Arbitrary Structures

In TAWK, an array can be viewed as a repository of information, much like a "structure" in other languages. The elements of the structure are created dynamically at runtime, rather than being declared in a structure definition. Furthermore, a pointer to such a "structure" can be trivially derived: the variable containing the array is, in fact, a pointer to the array. Using the variable name without any brackets gives you the pointer to the array. New arrays can be allocated with the table() function. Deallocating arrays is equally trivial: when the array is no longer accessible, TAWK deletes it for you. These features allow TAWK to be used to build up arbitrarily complex data structures that are interconnected with pointers. For example, consider the following C and TAWK code, which are equivalent:

C Language Code

```

struct foo {
    int a;
    char *b;
    struct foo *ptr;
} *x;
x = malloc(sizeof(struct foo));
x->a = 1;
x->b = "this"
x->ptr = x

```

TAWK Code

```

x = table()
x["a"] = 1
x["b"] = "this"
x["ptr"] = x

```

When using arrays as structures, you must be very careful not to misspell the array indices. If you misspell a structure element name in a structured language, like C, the compiler will complain vociferously. If you misspell an array index in TAWK, you will simply create a new array index, without any warning.

As an illustration of an arbitrary data structure, here is the complete code to create and manipulate doubly linked lists. A doubly linked list consists of a head element, called the sentinel, and elements in the list, which we will call items. Each doubly linked list must have one sentinel. The sentinel is allocated before you start using the doubly linked list, and is never deallocated while the list is in use. The sentinel, and each item, contain two special pointers, which we will name "flink" and "blink". The flink pointer points to the next item in the list, and the blink pointer points to the previous item in the list. If the list is empty, such as when it is first allocated, the flink and blink pointers of the sentinel point to the sentinel itself. Using two pointers, rather than a single pointer, makes it very easy to remove items from the list, which is why doubly linked lists are often preferred to singly linked lists. In the code below, the dlinit function creates the sentinel for a new linked list, the dllink function adds an item to a linked list, and the dlunlink function removes an item from a linked list. The dllink function adds the new item into the list immediately after the item specified by the "list" argument. For example, if head is the sentinel of a list, then dllink(head,item) adds the item to the front of the list, and dllink(head["blink"],item) adds the item to the end of the list, because head["blink"] always points to the last item in the list. The dlprint function illustrates how you can traverse every element in a linked list; this function prints the contents of the list.

```

# Add an existing item to a doubly linked list.
# The new item is added after the item: list
function dllink(list,item) {
    item["flink"] = list["flink"]
    item["blink"] = list
    list["flink"]["blink"] = item
    list["flink"] = item
}

# Delete item from a doubly linked list.
function dlunlink(item) {
    item["flink"]["blink"] = item["blink"]
    item["blink"]["flink"] = item["flink"]
}

# Allocate and return a new sentinel
# for a doubly linked list.
function dlinit() {
    local head = table() # Allocate a new table
    head["flink"] = head["blink"] = head
    return head
}

```

```

# Print the contents of a doubly linked list.
# Do not print the "flink" and "blink" fields.
function dlprint(list) {
    local i,item,cnt=1
    for (item = list["flink"];
        item != list;
        item = item["flink"]) {
        print "ITEM NUMBER",cnt++
        for (i in item) {
            if (i != "flink" && i != "blink") {
                print i,"=",item[i]
            }
        }
    }
}

# Demonstrate the functions: allocate a list,
# add two items, and print the list.
BEGIN {
    local head
    head = dlinit()
    head["label"] = "sentinel"
    dllink(head,table("label","first item"))
    dllink(head,table("label","second item"))
    dlprint(head)
}

```

## Common Mistakes

If an uninitialized variable is used as an array index, the resulting index is "", not 0. This is because array indices are strings, and the string value of an uninitialized variable is "", not "0".

This sometimes surprises users, as illustrated in the following example. The author here attempted to read in an array with indices 0, 1, 2, ... and then print the array back out. But the first line printed will be blank because the first element in the array is x[""], not x["0"]. (Before we started printing this manual, this error was the number one generator of technical support calls on TAWK. Now, very few people call with this problem. So, thanks to all our users for reading this!)

```

{
    x[i++] = $0
}

END {
    for (j = 0; j < NF; j++) {
        print x[j]
    }
}

```

To avoid this problem, you can initialize your index variable to 0, as in the following example, or you could change i++ in the above example to ++i. The following initializes i to 0, so it works ok:

```

BEGIN { i = 0 }

{
    x[i++] = $0
}

```

## Old Style Arrays

TAWK still supports an obsolete method of simulating multi-dimensional arrays using one dimensional arrays. You may see this used in older awk programs that you obtain from UNIX. It works like this: an array subscript may contain multiple subscripts separated by commas. The "real" subscript is created from these multiple subscripts by concatenating them together with a special character between them. The special character must be one that is not likely to appear in an array subscript, and defaults to "\x1c".

(This is the TAWK notation for a single character represented by hex 1c.) This special character is contained in the built-in variable SUBSEP. For example, the array reference:

```
x[1,2,3]
```

is translated by TAWK into the expression:

```
x[ 1 SUBSEP 2 SUBSEP 3 ]
```

When testing for an element in such an array, you may use a construct like this:

```
if ((1,2,3) in x) ...
```

which is translated by TAWK into the expression:

```
if ((1 SUBSEP 2 SUBSEP 3) in x) ...
```

There is no easy way to create a **for** loop to run through a single dimension of this type of pseudo-multi-dimensional array, which is why they were not often used.

## Chapter 9: Fields

### Summary

Field variables are denoted by a dollar sign (\$) followed by an integer, or an expression that evaluates to an integer. \$0 denotes the current input record. The fields in the current record are denoted by \$1, \$2, etc. The NF variable specifies the number of fields in the current record. The FS and FPAT variables specify the method TAWK uses to determine the fields in the current record. As a convenience, TAWK determines the type of fields by examining their contents: a field is considered a number if it contains only a number, otherwise it is a string.

Assignment to \$0 causes \$1, \$2, etc. and NF to be recalculated. Assignment to \$n where n > 0 causes \$0 to be reconstructed by concatenating the values of the current fields separated by the contents of the OFS variable. If a new field beyond \$NF is created this way, then NF is also changed, and any intervening fields are given an uninitialized value (meaning that they act like "" or 0, whichever is appropriate.) Assignment to NF will truncate existing fields or create new empty fields as necessary, and also automatically reconstruct \$0 as above.

### Discussion

The built-in variables: FS (Field Separator) and FPAT (Field Pattern) are used to control the algorithm that TAWK uses to find the fields in the current record. By default, the fields are separated by white-space, so the fields are just the words in the current input record.

For example, suppose \$0 (the current input record) contains:

```
Pat Thompson      Hiking      1/1/86  165
```

It then consists of five fields as follows:

```
$1 = "Pat"
$2 = "Thompson"
$3 = "Hiking"
$4 = "1/1/86"
$5 = "165"
```

Fields are implemented efficiently: TAWK does not actually do the work of finding the fields in the record until you use them.

The \$ can be followed by an expression instead of an integer. For example: \$1 refers to field 1, \$(1+4) refers to field 5, and if x == 2 then \$x refers to field 2. The parentheses are needed in \$(1+4), because \$1+4 means add four to the value of the first field.

To make programs to process data-files more readable, we can use variables to hold the field numbers. This also makes the program easier to change if we rearrange the order of the fields in the future:

```
BEGIN {
    Fname = 1
    Lname = 2
    Hobby = 3
    Hire_date = 4
    Salary = 5
}

$Salary > 120 {
    print "Employee", $Fname, $Lname,
        "is being overpaid!"
}
```

### Assignment to fields.

You can make an assignment to any field at any time. Assignment to \$0 sets the current input record, and therefore changes all the other fields simultaneously (\$1, \$2, etc.) An assignment to any other field (like \$1) changes that field, and also causes \$0 to be reconstructed from the new values of the fields. The new value of \$0 is created by concatenating the values of all the fields together, separated by the value of the OFS (Output Field Separator) variable.



For example:

```
BEGIN {
    OFS = "."
    $0 = "    this    and    that    "
    $2 = "or"      # This causes $0 to be recalculated
    print $0      # prints "this.or.that"
}
```

Note that the OFS variable does double duty: It is used both to reconstruct \$0, and also as the output field separator for the print statement.

## Other methods to obtain fields:

### Using the split or splitp functions

The field notation using \$n is merely a convenience. You can also create fields using the TAWK **split** or **splitp** functions. These functions find the fields and place them in an array. Additionally, while the number of fields available using the \$n notation is limited in some versions of TAWK, there is no practical limit on the number of fields created using the **split** or **splitp** functions. For example, if you want to get the first field out of variable x you could do it using either of these methods:

```
BEGIN {
    x = "hello world"
    # Method 1: using built-in field notation:
    $0 = x
    print $1      # Prints "hello"

    # Method 2: using built-in split function:
    split(x,fields)
    print fields[1]  # Prints "hello"
}
```

### Using the match function

Sometimes a record contains fields that are too complicated to specify using the FS or FPAT variables. For example, the first few fields may use one field separator, and the other fields may use another. The match function allows you to write a regular expression that matches a string, and to use parentheses in the regular expression to specify parts of the string that you are interested in. The match() function optionally returns the location in the string that was matched by each parenthesized part of the regular expression. See the match() function in the Built-In Functions Chapter.

### Using the pack and unpack functions

The built-in fields in TAWK are always variable length fields. To manipulate fixed-length fields like data-base records use the unpack() and pack() functions. See the Built-In Functions Chapter.

## Examples

Here is a slightly more complicated program block that can be added to any program to process the data file in chapter 3. The first line in the data file specifies the names of the fields as follows:

```
Fname  Lname  Hobby  Hire-Date  Salary
```

The following program block reads in the field names from the first record in the data file, and places them in an array called fieldname[]. A program that includes this program block can then refer to field \$1 as: \$fieldname["Fname"], to field \$2 as: \$fieldname["Lname"], etc. If your program uses the names of the fields (like "Fname" and "Lname") instead of the field numbers, the order of the fields in the data file becomes unimportant. This allows you to rearrange the fields in the datafile, without changing your TAWK programs, as long as you always keep the same field names in the first record associated with the same fields.

```
NR == 1 {  
  for (i = 1; i <= NF; i++) {  
    fieldname[i] = i  
  }  
  next    # Go on to the next record  
}
```

## Chapter 10: Regular Expression Patterns

TAWK uses a special notation to specify patterns that can match characters in a string. This notation is called “regular expression” notation in the technical literature on text processing, and we use this term in TAWK.

### Regular Expression Matching Patterns

A regular expression is made up of normal characters, which simply match the character specified, and special characters from the following table. The table is arranged in order of precedence:

.	A period matches any single character.								
[ <i>abc</i> ]	A list of characters enclosed in square brackets matches any single character from the specified set of characters. For example, [ <i>abc</i> ] matches a, b, or c. Two characters separated by a hyphen denotes a range: [ <i>a-z</i> ] matches characters a through z. A circumflex at the start of a range indicates inversion of a range: [ <i>^abc</i> ] matches any characters except a, b, or c. If a right-bracket appears in a character set, it must be the first character: [ <i>]]</i> matches ]. If a hyphen appears in a character set, it must be the first or last character.								
( <i>regexp</i> )	Parentheses are used for grouping. This simply matches whatever is matched by the regular expression <i>regexp</i> .								
<i>regexp</i> *	A regular expression <i>regexp</i> followed by “*” matches zero or more occurrences of whatever <i>regexp</i> matches. For example, <i>A*</i> matches "" or "A" or "AA", etc.								
<i>regexp</i> +	A regular expression <i>regexp</i> followed by “+” matches one or more occurrences of whatever <i>regexp</i> matches. For example, <i>A+</i> matches "A" or "AA", etc.								
<i>regexp</i> ?	A regular expression <i>regexp</i> followed by “?” matches zero or one occurrences of whatever <i>regexp</i> matches.								
<i>regexp</i> { <i>m</i> }	Where <i>m</i> is an integer: matches exactly <i>m</i> occurrences of the regular expression <i>regexp</i> .								
<i>regexp</i> { <i>m,n</i> }	Where <i>m</i> and <i>n</i> are integers: matches <i>m</i> to <i>n</i> (inclusive) occurrences of the regular expression <i>regexp</i> .								
<i>regexp</i> { <i>m</i> ,}	Where <i>m</i> is an integer: matches <i>m</i> or more occurrences of the regular expression <i>regexp</i> .								
<i>regexp1 regexp2</i>	Two adjacent regular expressions simply matches whatever matches <i>regexp1</i> followed by whatever matches <i>regexp2</i> . This is called concatenation.								
<i>^regexp</i>	Specifies that whatever is matched by the regular expression <i>regexp</i> must begin at the beginning of the string or record. This is called an “anchored” match.								
<i>regexp</i> \$	Specifies that whatever is matched by the regular expression <i>regexp</i> must end at the end of the string or record.								
<i>regexp1 regexp2</i>	Matches either regular expression <i>regexp1</i> or <i>regexp2</i> . This is called alternation because one of the two alternates must match.								
<i>\$n</i>	Where <i>n</i> is a digit: matches whatever the string was that was matched by the regular expression enclosed by the <i>n</i> th set of parentheses in this regular expression. This will match only the exact string that was previously matched. The parentheses are numbered by counting ‘(’ characters in the regular expression, starting from the left: <i>\$1</i> matches whatever was matched by the first set of parentheses, <i>\$2</i> by the second set of parentheses, etc. For example, / ( <i>A+</i> ) B <i>\$1</i> / matches "ABA" or "AABAA" or "AAABAAA", etc.								
\	A backslash is the “escape character”, and is used to turn off the special meaning of the other regular expression pattern characters. A backslash followed by any of the special characters in this table simply matches that character. In addition, the following escape sequences are recognized: <table> <tr> <td>\t</td><td>matches a tab character</td></tr> <tr> <td>\b</td><td>matches a backspace character</td></tr> <tr> <td>\f</td><td>matches a form-feed character</td></tr> <tr> <td>\n</td><td>matches a new-line character</td></tr> </table>	\t	matches a tab character	\b	matches a backspace character	\f	matches a form-feed character	\n	matches a new-line character
\t	matches a tab character								
\b	matches a backspace character								
\f	matches a form-feed character								
\n	matches a new-line character								

<code>\r</code>	matches a carriage-return character
<code>\a</code>	matches an audible alert (bell) character (char 7)
<code>\v</code>	matches a vertical tab character (char 11)
<code>\\</code>	matches a backslash character
<code>\/</code>	matches a slash character
<code>\xdd</code>	where <i>dd</i> stands for two hexadecimal digits: matches the character whose value is <i>dd</i> in hex notation
<code>\ddd</code>	where <i>ddd</i> stands for three octal digits: matches the character whose value is <i>ddd</i> in octal notation

### Notes:

When you specify a regular expression in a literal string, you must double any backslashes, because backslash is the escape character for both strings and regular expressions. To match a single backslash you can use `/\\/` or `"\\\\"`.

The `$n` construct does not do backtracking, that is, if there is more than one way that the regular expression enclosed in parentheses can match the string being tested, only one of the possible ways is considered.

When matching regular expressions that contain parentheses, TAWK attempts to maximize the length of the string matched by the parenthesized sub-expressions. This is true even if a shortest-match is requested: In this case TAWK will find the shortest over-all match, but within this constraint, attempt to maximize the length of parenthesized sub-expressions. This feature can be used when matching complex patterns to help control what a wild-card will match. In the following example, the parenthesized sub-expression matches "aaab", and the initial `a*` matches nothing:

```
match("aaab", /a*(.*)/, 1, pstart, plength)
```

Now if we add another set of parentheses, the first set of parentheses matches "aaa" and the second set matches "b":

```
match("aaab", /(a*)(.*)/, 1, pstart, plength)
```

### Compatibility Notes:

For Thompson Toolkit and UNIX Users: There are two common forms of regular expressions used in UNIX commands. The `grep`, `sed`, `vi` and `expr` commands use a simpler form of regular expressions than the `awk` and `egrep` commands. For this reason, the regular expressions used by `awk` and `egrep` are sometimes called "extended regular expressions". The `{ }` constructs are part of the POSIX standard, but not yet implemented uniformly in all versions of UNIX. The `$n` construct is a TAWK extension.

### Regular Expression Flags

Regular Expression flags modify the way the regular expression is matched. If the regular expression is specified using forward-slashes, the regular expression flags may be appended to the final slash, with no intervening space. Regular expression flags can also be specified to the `regex()` function. The flags are:

- i** Ignore case when matching letters. For example, "a" will be the same as "A".
- s** Shortest match. Normally a regular expression matches the longest possible string. This flag makes it match the shortest possible string. Note that the `$n` construct in a regular expression may not consider all the possible ways that the part of the regular expression in parentheses could be matched, so regular expressions containing these may not find the absolutely longest or shortest possible match.

For example:

```
/this/      # matches "this"
/this/i     # matches "this", "This", "THis", etc.
```

Shortest match is often used to find matching symbols. For example, C-style comments enclosed in `/*` and `*/` can be matched by the following regular expression. This regular expression looks confusing because each `"` and `/` character that represents a literal character is preceded by a backslash, because these characters are special in regular expressions:

```
/\/*.*\*/s
```

## Regular Expression Tutorial

Regular expressions are utilized in the pattern part of pattern-action statements, by the tilde operator, and as arguments to pattern matching functions like `match()`, `sub()` and `gsub()`. A regular expression can be entered into a program by surrounding it with slashes, or it can be entered as a string that is to be interpreted as a regular expression. For example, the following are equivalent:

```
match($0,/a pattern/)
match($0,"a pattern")
```

Note that a regular expression entered as a literal string undergoes backslash processing twice: once for the string processing and again for the regular expression. For example, to match a single backslash, the following are equivalent:

```
match($0,/\/)
match($0,"\\")
```

The `match` function searches for a regular expression pattern in a string. At its simplest, the regular expression is no different from a string:

```
match("automation", "tom") # returns 3
match("thompson", "tom")   # returns 0
```

But the regular expression recognizes characters that have special meanings, which we will now show. Suppose you want to search for a number. A regular expression can search for one or more occurrences of a class of characters:

```
match("agent 007", "[0-9]+") # returns 7
```

The `"[0-9]"` is a character class. It specifies a set of characters that will match in that position. You can list them individually or in groups. The following two lines have the same effect in a regular expression:

```
[0123456789]
[0-9]
```

The `"+"` means one or more of the preceding character or character class or other pattern. So `"[0-9]+"` means one or more digits between 0 and 9, inclusive. At this point you might be wondering why you would specify how many there are, if `match` just returns the first position. That is, the following two lines return the same thing:

```
match("agent 007", "[0-9]+") # returns 7
match("agent 007", "[0-9]")  # returns 7
```

But `match` also sets the built-in variable `RLENGTH`, which is the length of the matched string.

```
match("agent 007", "[0-9]+") # RLENGTH set to 3
match("agent 007", "[0-9]")  # RLENGTH set to 1
```

Or you might be searching for a number surrounded by `"#"`, as in:

```
match("agent #007#", "#[0-9]#") # returns 0
match("agent #007#", "#[0-9]+#") # returns 7
```

The first example is looking for exactly one digit between the `"#"`, so that search fails. The second is looking for one or more digits between the `"#"`, so it succeeds.

Now suppose you want to allow an optional minus sign in front of your number. The regular expression is

```
"-?[0-9]+"
```

The question mark says that the preceding character or character class (or pattern, described below) is optional.

```
match("agent 007", "-?[0-9]+") # returns 7
match("agent -007", "-?[0-9]+") # returns 7
```

Now suppose you want to search for a variable name. The first character must be lower-case and alphabetic, and the rest can be that or numeric. The regular expression would be:

```
"[a-z][a-z0-9]*"
```

The asterisk is like the plus except that it means 0 or more. The only difference then between ?, +, and \* is that ? is 0 or 1, + is 1 or more, and \* is 0 or more. This example also shows that a character class can have more than one group of characters specified in it.

Now suppose you want to find either a number or a variable name. The regular expression would be:

```
"[0-9]+|[a-z][a-z0-9]*"
```

The "|" specifies two alternatives, either the "[0-9]+" before or the "[a-z][a-z0-9]\*" after. Why is the second alternative "[a-z][a-z0-9]\*" rather than, say, "[a-z]", with either alternative followed by "[a-z0-9]\*"? There is a hierarchy of precedence for the regular expression operators, expressed in the following table, with the highest precedence operators at the top:

A?, A*, A+	0 or 1, 0 or more, 1 or more
A B	concatenation
^A, A\$	anchoring
A B	alternation

This says that ?, \*, and + have precedence over concatenation, so the following two regular expressions are equivalent:

```
"ab*"
"a(b*)"
```

Similarly, concatenation has precedence over alternatives, so the following two regular expressions are equivalent:

```
"ab|cd"
"(ab)|(cd)"
```

And that's why these two are equivalent:

```
"[0-9]+|[a-z][a-z0-9]*"
"([0-9]+)|([a-z]([a-z0-9]*))"
```

You'll have noticed that we're using parentheses to group parts of the regular expression, just like grouping the parts of an arithmetic expression. So you can use parentheses to change the grouping when necessary:

```
"one|two" matches "one" or "two"
"on(e|t)wo" matches "onewo" or "ontwo"
```

Now suppose it's not sufficient just to match a number anywhere in the string. You may want to match the number only if it's at the beginning, or only if it's at the end, or only if there is nothing except the number in the string.

```
"[0-9]+" matches number anywhere
"^ [0-9]+" matches number only at beginning
"[0-9]+$" matches number only at end
"^ [0-9]+$" matches string with nothing but number
```

"^" and "\$" have the same precedence as concatenation, so "^a?b" is the same as "^a?)b" and matches "b" or "ab" at the beginning of a string. If it were "(^a)?b", it would match "ab" at the beginning, or "b" anywhere. Also, "^one|two\$" is the same as "(^one)|(two\$)", and matches "one" at the beginning of a string, or "two" at the end. If it were "^one|two)\$", it would match only the strings: "one" or "two", that is, there can be nothing else in the string.

Besides using regular expressions in the match function, you can also use them in the sub and gsub functions, and with the ~ and !~ operators. The ~ operator takes a string or expression that yields a string on the left and a regular expression on the right, and returns true if any part of the string matches the regular expression.

```
"one" ~ "one|two" # returns true
```

The `!~` operator is the same as `~` except that it returns the complement.

```
"one" !~ "one|two" # returns false
```

A regular expression doesn't have to be a constant string. It can be a string that you compute:

```
"f3" ~ "f" sqrt(9) # returns true
```

If you have a constant regular expression, you can put the regular expression inside slashes instead of quotes, and TAWK will do more of the compilation of the regular expression at program compilation or startup, speeding up program execution. The following are equivalent in terms of execution results, but the second may be faster:

```
a ~ "f3"
a ~ /f3/
```

## Efficiency Issues

Regular expression matching occurs in two phases: First the regular expression is compiled into an internal state machine. Then that state machine is used to determine if a string matches the compiled pattern. TAWK uses an algorithm that allows a string to be matched in a single pass through the string. (Most regular expression matching techniques use expensive recursive back-tracking techniques that can take exponential time to match complicated patterns.)

Regular expressions entered in slashes are compiled into state machines just once, while regular expressions that are specified as strings or computed at run-time may be compiled into state machines each time they are used. Thus, regular expressions entered `/like this/` are faster. As an optimization, TAWK also keeps the most recently used state machine in memory, just in case the same regular expression is used again.

You can also create a compiled regular expression from a string using the `regex` function. This function takes a string and pattern flags as arguments, and returns a pre-compiled regular expression that can be used anywhere a regular expression can be used. If your program makes repeated use of a regular expression that is not known until run-time, you may be able to speed up the program by using `regex` to compile the regular expression just once. For example, the following program takes two or more arguments: The first is a pattern to match and the rest are filenames. Lines that match the pattern are printed.

```
BEGIN {
    # First program argument is the pattern to match
    x = regex(ARGV[1])
    # Do not process ARGV[1] as a filename:
    ARGV++
}

$0 ~ x { print "this matched:", $0 }
```

For some patterns the tilde operator can be faster than the `match()` function because the tilde operator does not need to report the length of the match. Thus it can stop processing as soon as it finds a match.

## Chapter 11: TAWK Flow Control Statements

### The “if” statement

The if statement provides a way to execute a statement only if a condition is TRUE. There are two forms:

```
if (expression) statement1

if (expression) statement1 else statement2
```

If the *expression* evaluates to a TRUE (non-zero number or non-empty string) value, then *statement1* is executed. If the *expression* evaluates to a FALSE (zero or empty string) value, then in the first form nothing happens; while in the second form *statement2* is executed. For example:

```
if (a < 4) print "true"

if (a < 4) print "true"
else print "false"
```

The *expression* must always be in parentheses:

```
if a < 4 print "true"      # WRONG
if (a < 4) print "true"   # RIGHT
```

Each of the statements (*statement1* and *statement2*) may be either a single statement, an empty statement (consisting of just a semi-colon) or a list of statements enclosed in curly braces: { }.

The optional **else** can cause confusion in combined **if-else** statements. This code is clear:

```
if (a)
    if (b) print "a && b"
    else print "a and !b"
else
    if (b) print "!a && b"
    else print "!a && !b"
```

This code is misleading:

```
if (a)
    if (b) print "a && b"
else print "!a"                                # WRONG
```

From the indentation and from the print string, you can tell the author meant for the **else** to go with the first **if**, but it doesn't. An **else** will go with the most recent **if** that doesn't already have an **else**, and isn't inside curly braces. You can rewrite the above code correctly in two different ways:

1) You can add curly braces:

```
if (a) {
    if (b) print "a && b"
}
else print "!a"
```

2) You can add in the missing **else**. Since an **else** must be followed by a statement, we used a semi-colon to indicate an empty-statement:

```
if (a)
    if (b) print "a && b"
    else ;
else print "!a"
```



## TRUE and FALSE

Variables in TAWK can hold numbers, strings, arrays, regular expressions, or other types of data. Consider what happens when a variable appears in a boolean context (a context where a TRUE or FALSE result is required), for example, the variable *x* in the following statement:

```
if (x) print "TRUE"; else print "FALSE"
```

TAWK must decide if *x* evaluates to TRUE or FALSE. The decision depends on both the type and the value of *x* according to the following table:

Meaning of TRUE and FALSE by Expression Type

<u>Value of x</u>	<u>Meaning</u>
<i>x</i> is uninitialized	Always FALSE.
<i>x</i> is a number	TRUE if <i>x</i> is non-zero.
<i>x</i> is a string	TRUE if <i>x</i> is not ""
<i>x</i> is a regular expression, for example, returned by the <code>regex()</code> function	TRUE if ( $\$0 \sim x$ ). In other words, a regular expression is considered TRUE if the current record matches the regular expression.
<i>x</i> is an array	TRUE if the array is non-empty. This may indicate a mistake in the program, so it also generates a warning message about array used in unexpected context. To suppress the warning message use: <code>if (length(x))</code> , which does the same thing, instead of just: <code>if (x)</code> .

For example, if *x* has a numeric value, then it is considered FALSE if it is zero and TRUE if it has any non-zero value. An expression with a string value is considered FALSE if it is an empty string ("") and TRUE if it contains any characters.

Note that the following are NOT equivalent:

```
if (x) ...
if (x != 0) ...
```

If *x* is a number, these statements are identical. However, if *x* is a string or other type then the second statement uses the rules for comparing strings and numbers, which may yield a different result. Consider what happens if *x* contains "0". The first statement will return TRUE, because a string that is not "" is TRUE. But the second statement will convert the string "0" to a number (0) and the expression will evaluate to FALSE.

## The "while" loop

The simplest looping statement is the **while** loop:

```
while (expression) statement
```

The *statement* is executed while the *expression* is TRUE. As usual, the test *expression* must be in parentheses, and the *statement* may be either a single statement, an empty-statement (consisting of just a semi-colon) or a list of statements enclosed in curly braces: { }. For example:

```
BEGIN {
  i = 1;
  while (i <= 4)
    print i++
}
```

Notice the use of the increment operator ++ in this example to increment i on each pass through the loop. The output of the above is:

```
1
2
3
4
```

To make a loop that runs forever, use this:

```
while (1) ...
```

A non-zero expression (1 in this case) is always TRUE. You had better include a **break** statement (see below) or other way to get out of the loop if you do this.

## The “do” loop

Another type of looping is the **do** loop, which is basically just a **while** loop in reverse:

```
do statement while (expression)
```

The above can be rewritten as a **while** loop as follows:

```
statement
while (expression) statement
```

That is, one *statement* is always executed, then the *expression* is tested before executing each *statement* after that. The *statement* must either be terminated by a semicolon or newline, or put in curly braces:

```
do print i--; while (i > 0)      # RIGHT
do { print i-- } while (i > 0)  # RIGHT
do print i--                    # RIGHT
while (i > 0)
do print i-- while (i > 0)      # WRONG
```

## The “for” loop

The most general looping statement in TAWK is the **for** loop. Its general form is:

```
for (initialization_clause; expression; iteration_clause) statement
```

The parentheses and the two (and only two) semi-colons within the parentheses are required. As usual the *statement* may be either a single statement, an empty-statement (consisting of just a semi-colon) or a list of statements enclosed in curly braces: { }.

The easiest way to explain how a **for** loop works is to show how it would look if written using a **while** loop. Note that *initialization\_clause* and *iteration\_clause* are used as statements, and the *expression* in the middle is the one that is tested.

```
initialization_clause
while (expression) {
    statement
    iteration_clause
}
```

The **for** loop is included in TAWK because it is ideally suited to loops that use an index variable, such as the variable i in the following example:

```
for (i = 1; i <= 4; i++) {
    print i
}
```

The above could be rewritten using a **while** loop as follows:

```
i = 1
while (i < 4) {
    print i
    i = i + 1
}
```

In the above example, the **for** loop is easier to read (once you understand how it works) because all the statements that modify or test variable *i* are contained in the initial **for** statement. This is especially true when the statement is not a single statement, as above, but may be a list of statements several pages long.

### Additional Aspects of For Loops

Any of the *initialization\_clause*, the *expression*, or the *iteration\_clause* in the **for** loop expression may be omitted. The semicolons are still required. For example, the following **for** loop acts more like a **while** loop:

```
for ( ; i < 5; ) {
    print i
    i = i + 1
}
```

If the test expression is omitted in a **for** loop, then TRUE is assumed and the loop runs forever.

As a special case, the *initialization\_clause* and the *iteration\_clause* may include multiple statements separated by commas. For example, to initialize and increment two variables simultaneously, you could use:

```
for (i = 1, j = 10; i <= 5; i++, j++) print i, j
```

The above example will produce the following output:

```
1 10
2 11
3 12
4 13
5 14
```

A special type of **for** loop using the **in** keyword can be used to go through all the elements of an array. This type of **for** loop is used only with arrays, and is discussed in the chapter: Arrays.

### The “break” and “continue” statements

The **break** and **continue** statements control execution flow in the **for**, **while** and **do** loops. Inside any loop, the **continue** statement causes execution to continue immediately with the next loop iteration, and the **break** statement immediately terminates the loop.

For example, assume you have an array of values to process. You could use a loop like this:

```
for (i in my_array) {
    # process my_array[i] ...
}
```

Now assume you do not want to process the array element “jim”. You could use the same loop but skip this one element like this:

```
for (i in my_array) {
    if (my_array[i] == "jim") continue
    # process my_array[i] ...
}
```

Now suppose you want to stop processing when you find an array element containing the word “stop”. Try this:

```
for (i in my_array) {
    if (my_array[i] == "jim") continue;
    if (my_array[i] == "stop") break;
    # process my_array[i] ...
}
```

The following demonstrates that the **continue** statement causes the iteration\_clause of a **for** loop to be executed before continuing the next iteration. Here we use a **continue** statement to skip over the case  $i == 3$ .

```
for (i = 1; i <= 5; i++) {  
    if (i == 3) continue;  
    print i  
}
```

This prints:

```
1  
2  
4  
5
```

## Empty statement

A semi-colon by itself is an empty statement. An empty statement can go anywhere, but is used primarily with the **if**, **for**, **while** or **do** loops to indicate that a statement is missing. Consider the following code:

```
BEGIN {  
    x[0] = 1  
    x[1] = 1  
    x[2] = 1  
    for (i = 0; i in x; i++) ;  
    print i, "is the value of i"  
}
```

Which prints:

```
3 is the value of i
```

Now consider what happens if the semi-colon is omitted. The **print** statement following the incomplete **for** statement gets turned into the statement part of the **for** loop. Here's what happens:

```
BEGIN {  
    x[0] = 1  
    x[1] = 1  
    x[2] = 1  
    for (i = 0; i in x; i++)  
        print i-1, "is the value of i"  
}
```

Which prints:

```
-1 is the value of i  
0 is the value of i  
1 is the value of i
```

Some people make their code easier to read by using a **continue** statement instead of an empty statement in a loop, or by using curly braces, or both, like this:

```
for (i = 0; i in x; i++) { continue; }
```

## Chapter 12: Variable Declarations

TAWK allows variable declarations to aid the development of large programs. Variable declarations are optional. There are three classes of variables: global, module-local and function-local.

### Summary

Variables are declared using the **global** or **local** keywords.

Variable declarations have the following form. Square brackets: [ ] indicate optional components:

```
keyword var_name [ "=" expression ] [ , ... ] [ ; ]
```

The *keyword* is either **global** or **local**. The *var\_name* is the name of the variable. The *expression* is an optional initialization expression that specifies the initial value for the variable. Multiple variable names can be declared in a single statement if they are separated with commas. Like other TAWK program statements, multiple declarations can appear on separate lines, or can appear on the same line, if separated by semi-colons.

### Global Variables

Global variables have two purposes: 1) they are the variables that are shared between all program source files; 2) they are the variables that can be changed on the command line of the TAWK program, for example, using the -v option.

If a variable is declared **global** in any source file, then that variable is shared by all source files. Including a global declaration for a variable in every source file where it is used is good programming practice, but is not required. If a global variable has multiple declarations, only one of them may include an initialization.

Global variable declarations use the **global** keyword and typically appear at the top of the source file, although they may appear anywhere except inside functions or program blocks. Variable declarations can include an optional initialization to any valid TAWK expression.

It is a good idea to give global variables long and descriptive names. Names like x and y for global variables are unwise because it is too easy to accidentally use these same names in another source file without even knowing it.

### Module Local Variables:

A “module” is the program code contained in a single program file. Module local variables can be used only in the module (source file) in which they are declared. The same name can be used for a module-local variable in more than one source file: each is a different variable. If a module local variable name is the same as a global variable declared in another source file, the module-local variable takes precedence in the source file in which it is declared. Module local variables are declared with the **local** keyword and typically appear near the top of the source file, although they may appear anywhere except inside functions or program blocks.

### Function Local Variables

Function-local variables also use the **local** keyword and appear at the beginning of a function or program block before any other programming statements. Function-local variables can only be used in the function or program block in which they are declared, and over-ride any other variables with the same name inside of that function or program block. Function-local variables are “dynamic” rather than “static”, which means that their contents are not saved when the function or program block in which they appear is exited. If a function-local variable includes an initialization, it is performed every time the function is called. If you need to save the contents of a variable between function calls use a global or module-local variable instead.

### Examples

```
global start, finish = 3, linecount;

local x,y = "sss";
local z
```

```
function print_ten(output)
{
    local i, j = 10
    for (i = 1; i <= j; i++) print output
}

BEGIN {
    local msg = "hello"
    print_ten(msg)
}
```

This declares the variables: start, finish, and linecount as global variables. They may be used freely in any source file. The variables: x, y and z are declared local to the current module (source file). Variable: finish is initialized to the value 3, and y is initialized to the value "sss", before your program starts. The semi-colons are optional.

The function print\_ten() has two local variables: i and j, as well as a function argument: output. Variable j is initialized to 10 each time the function is called. The BEGIN block has a local variable: msg that is initialized to "hello" when the BEGIN block starts.

## Undeclared Variables

Variables that are not declared are treated like module-local variables, that is, they are accessible only in the source file in which they appear. This helps prevent accidental naming conflicts in very large TAWK programs consisting of many program files.

Notice that if an undeclared variable is later declared as **global** in another program source file it then refers to that global variable. Thus you can introduce bugs into your TAWK program when an undeclared variable switches from referring to an implicit module-local variable to a global variable declared in another file.

It is a good idea to declare all variables in large TAWK programs. The -s option to the TAWK compiler prints warnings for undeclared variables, to help you find them. Many users turn this option on permanently by placing it in the awkc.cfg file.

See also the TAWK Compiler -g option, which makes undeclared variables global so they are shared among all source files.

## Uninitialized Variables

A variable that has not yet been assigned a value in your program is called an uninitialized variable. In most programming languages, an "uninitialized" variable has an unknown value, and may cause unpredictable results when used in the program. In TAWK, all variables that are not explicitly initialized by the program are given a special value, called the "uninitialized" value. A variable with the "uninitialized" value appears to have the string value "", or the numeric value 0, whichever is appropriate.

## Restrictions on Command Line Variable Assignments

Variable declarations may appear on the command line of your TAWK program using the -v option to supply a new initial value for the variable or by placing an argument of the form *name=value* in the arguments processed by the Automatic Input Loop. Normally only global variables may be initialized this way, however, if your program consists of only one source file, then undeclared variables can be initialized this way too.

The initialization on the command line over-rides the initialization, if any, specified in the variable declaration.

Command Line Assignment to variables is covered in the chapter on Program Arguments.

## Chapter 13: Using the "awk" Program.

The "awk" program allows you to compile and execute a TAWK program in a single step. For small programs, this is sometimes easier to use than the TAWK compiler. The "awk" program is named `awk.exe` under DOS, or `awkw.exe` under Win32 (Windows NT or Windows 95), or `awkp.exe` under OS2. Our "awk" program is not really an awk interpreter; it is actually a front end to our TAWK Compiler. It quickly compiles the program you specify and then, if no errors were encountered, immediately executes the program with the program arguments that you specify.

The "awk" command line syntax is compatible with the "awk" programs available on UNIX systems.

### Specifying the program.

There are three ways to specify your program:

- 1) The program is in a file.

If your program is the file: `myfile.awk`, you can run it using the `-f` option using the command.

```
awk -f myfile.awk
```

If no filename extension is specified, ".awk" is assumed, so you can just type:

```
awk -f myfile
```

- 2) The program is specified on the command line.

If no `-f` option is specified, then "awk" assumes that its first argument contains the entire program to be executed, like this:

```
awk 'program ...' other arguments
```

This method can be useful when a very small awk program is to be included in a shell script. The program must be contained in quotes so that "awk" knows that it is a single argument. Either single-quotes (apostrophes) or double-quotes may be used. It is a good idea to use single-quotes because many awk programs contain double-quotes. For example:

```
awk 'BEGIN { print "hello world" }'
```

#### NOTE:

We do not recommend this method under DOS or OS2 because the default command interpreters for these systems do not know about quoting. Any `>` or `<` symbols in your program will be interpreted as file redirections and cause havoc. You will also quickly run into the command line length limit under DOS, which is less than 128 characters. Alternatively, you can use the Thompson Toolkit Shell as a replacement command interpreter for DOS or OS2 and everything will work fine.

- 3) The program is entered interactively.

If you enter `awk` (or `awkw` or `awkp`) with no arguments:

```
awk
```

Then the "awk" program will prompt you for both the program to run and then the program arguments. The program is entered interactively at the terminal and can be multiple lines. The "awk" program will then compile and run your program on the specified files.

### AWK Program Options

The remaining arguments to the awk program can consist of the following options and/or arguments that will be passed to your AWK program. Options are processed from left to right. When an unrecognized option is encountered, it and all remaining arguments are passed to your AWK program. The recognized options are:

`-F "x"` Makes `x` the initial field separator. The quotes are optional if `x` is not a space or other special character. This option is identical to saying: `-vFS="x"`.

**-v *name*="value"**

Sets the AWK program variable *name* to the specified *value* before the AWK program starts. No spaces are allowed around the equal sign, but the quoted *value* may contain spaces. The quotes are optional if the *value* does not contain any spaces or other special characters. Multiple -v options may be given to change multiple variables. Note: Not all variables can be changed with the -v option. Normally only variables that are declared global can be changed. If there is only one source file, or if the -g option is specified, then undeclared variables can also be changed.

**-f *filename***

Reads the program from the file: *filename.awk*. If no extension is specified, ".awk" is assumed. If the filename does not include a path specifier and is not in the current directory, then the filename is searched for in the directories specified by the AWKPATH environment variable, if any, and then in the directory where the "awk" program resides. Multiple -f options may be specified to compile multiple program files together.

The following options are not part of the AWK standard:

- eb** This two letter option enables an alternate backslash processing method: see the full description under AWK Compiler options below.
- g** Makes all undeclared variables global. This is useful only when two or more AWK program files are compiled simultaneously. This option is useful when a program that contains undeclared variables is broken up into multiple files, so the undeclared variables in the original source file are shared globally among all the new smaller source files.
- w** Suppresses both compile time and runtime warning messages from the AWK program.
- This option causes all option processing to stop, and is itself removed from the command line. It is useful to allow an AWK option to be passed as an argument to an AWK program without being interpreted by AWK as an AWK option. For example, to pass "-v" as an argument to an AWK program so that ARGV[1] will be "-v", you could do this:

```
awk -f myprog.awk -- -v
```

## Additional Command Line Processing by AWK

The command line of the "awk" program is processed using UNIX-like rules for command line processing, even under DOS, Windows or OS2. Using these rules, you can place quoted strings, the contents of environment variables, or the output of other commands into the command line. The awk command line can contain:

- 'string'** All characters in the *string* are quoted.
- "string"** The characters in the *string* are quoted, except that \$ and `` processing still occurs.
- \** A backslash quotes the characters: \$, `, ", or '. This allows these special characters to be placed in the command line.
- \$name** This is replaced with the value of the *named* environment variable. There can be as many of these on the AWK command line as desired.
- `command`** The *command* is executed and its output is placed in the command line. If the command output consists of multiple lines, they are compressed into a single line by replacing newlines with spaces. This is most useful to process a list of arguments. For example, you can place the list of filenames that you want awk to process in a file called "filename". Then to process the list of filenames, use (under DOS):

```
awk -f foo.awk `type filename`
```

### Notes:

- 1) Using the standard DOS or OS/2 command interpreter, the characters: < > & or | can be placed into the command when surrounded by double quotes only. Single quotes or backslashes do not quote these characters.
- 2) The DOS command line length is limited to approximately 125 characters. The results of \$name or 'command' substitution are not subject to this limitation. This is one way to get around the command line length limit in DOS. In the following example, the arguments to awk are arg1 through arg8:



```
set args1=arg1 arg2 arg3 arg4
set args2=arg5 arg6 arg7 arg8
awk -f foo.awk $args1 $args2
```

The Thompson Toolkit Shell, available from Thompson Automation, is a better command interpreter that allows you to overcome these limitations.

## Chapter 14: Using the TAWK Compiler

The TAWK Compiler program is named `awkc.exe` (DOS), `awkcw.exe` (Win32), `awkc.exe` (OS2), or `awkc` (UNIX). Its syntax is as follows:

```
awkc [ -options ] filename ...
```

The specified *filenames* are all compiled and linked together into a single program. The *filenames* must have the suffix “.awk”. The suffix need not be given on the command line: “.awk” is assumed.

The TAWK Compiler can produce several different types of executable files, controlled by the `-x` option:

- **Minimized Executable File:**

This is the default option. The TAWK Compiler produces an executable file that uses the TAWK runtime module as an overlay. This makes the smallest possible executable file, because most of the code remains in the TAWK runtime module. The compiled program **MUST** be able to find the TAWK runtime module file when it runs. This file is named “awkr50.exe”, or something similar, and resides in the TAWK installation directory by default. Your TAWK program will look for this file in the directory from which it was invoked, then along the `AWKPATH` environment variable, and finally along the `PATH` environment variable.

- **Stand-alone Executable File:**

This is selected by the `-xe` option. The TAWK Compiler creates a stand-alone executable file. It will be much larger than the file produced by the Minimized Executable File option, but it will run anywhere, anytime.

- **Statically Linked Executable or Object Files [DOS Version only]:**

The DOS version of the TAWK Compiler can create an object file, and optionally call a linker, to combine the TAWK program with object files or libraries written in C or C++ to create a combined TAWK/C executable file. This option is selected by the `-xo` and `-xl` options, which are discussed in the chapter on combined TAWK/C programs. This option is not needed in other operating systems, because TAWK can call Dynamic Link Libraries (DLLs) in those operating systems.

In any case, your compiled TAWK program will process the following options on its command line: `-F`, `-w`, `-v` and `--`. The affect of these options on the compiled program is the same as that documented for the “awk” program. If you do not want your compiled program to process these options, you can specify the `-eo` option when you compile it. Unrecognized options or options following `--` are passed to your compiled TAWK program in the `ARGV` array.

## TAWK Compiler Options in Alphabetic Order

### `-c compiler_option`

[ DOS Version Only ]

This option selects the host compiler and linker to be used for static linking of combined TAWK/C programs for the DOS operating system. The possible *compiler\_options* are:

<u>Option</u>	<u>Compiler Selected</u>
<code>-cm</code>	Microsoft Visual C or C++, or Microsoft C or Quick-C version 7 or higher. (This is the default)
<code>-cm6</code>	Microsoft C version 6;
<code>-cb</code>	Borland or Turbo C or C++ version 3 or higher.

Note: `-cm7` and `-cm8` are synonyms for `-cm`; `-cb3` and `-cb4` are synonyms for `-cb`.

**-eb** (This option is two letters) Compatible backslash option: Unfortunately, some implementations of awk process backslashes inside quoted strings using an alternate and partially incompatible method. The `-eb` option causes the TAWK Compiler to use this alternate method. If the `-eb` option is specified, then a back-slash in a quoted string is left alone, unless it occurs before one of the characters that causes it to have a special meaning; for example, “\t” is a tab character. Note that this option affects only literal strings in the TAWK source file(s); it has no affect on strings read into the TAWK program at runtime, because these do not undergo backslash processing. For example:

<u>String</u>	<u>Normally Produces:</u>	<u>With -eb Produces:</u>
---------------	---------------------------	---------------------------

"\z"	"z"	"\z"
"\\z"	"\z"	"\z"
"\\\z"	"\z"	"\z"
"\\z"	"\z"	"\z"
"\t"	"<tab>"	"<tab>"
"\\t"	"\t"	"\t"
"\\\t"	"\ <tab>"	"\ <tab>"

Note that without the `-eb` option, the string `"\z"` would produce a warning message from the TAWK Compiler, because the backslash is unnecessary and is ignored.

- `-ee` (This option is two letters) Disables filename expansion of filenames in the ARGV array of the compiled program. The filename patterns are left in the ARGV array, allowing you to do your own filename expansion, if you so desire. Note that a compiled TAWK program still processes filename patterns in each element of the ARGV array when it comes time to read and process that filename, so this option does not actually disable the use of filename patterns in the compiled program, it just makes the original filename patterns show up in the ARGV array. As usual, if your program changes any element of the ARGV array before TAWK processes that element as a filename, TAWK processes the changed ARGV element rather than the original.
- `-eo` (This option is two letters) Suppresses default command line processing of options by the compiled TAWK program. Specifically, the compiled TAWK program will ignore any `-F`, `-v`, `-w` or `--` options in its command line. The `-eo` and `-ee` options can be combined, for example: `-eoe`
- `-g` Makes all undeclared variables global. Normally, undeclared variables are made local to the file in which they appear, so that variables that are accidentally left undeclared do not affect other TAWK source files. This option is useful when a program that contains undeclared variables is broken up into multiple files, so the undeclared variables in the original source file are shared globally among all the new smaller source files.

#### `-linkopts "options"`

[ DOS Version Only ]

The specified options are passed to the linker. The default link *"options"* vary depending on the C compiler. For Microsoft C the default options are `/SE:200 /NOE /EXEP` (under DOS.) If the specified options begin with an equal-sign, these options over-ride the previous link options. Otherwise the specified options add to the existing link options. Multiple `-linkopts` can be given. The quotes around the options are optional if the options contain no spaces or other special characters. This option is ignored if no linking occurs. (See your C linker documentation for explanations of linker options.)

#### `-linkspec linkpath`

[ DOS Version Only ]

For a statically linked combined AWK/C program: this option specifies the full executable name of the linker to use, for example: `-linkspec "D:\c700\bin\link.exe"` This option is ignored if no linking occurs.

#### `-linkstack number`

[ DOS Version Only ]

For a statically linked combined AWK/C program: the specified *number* is the stack size passed to the linker. This option is valid ONLY for Microsoft C. Borland and others specify the stack size using a variable in the C program.

#### `-o filename`

Names the output file. Do not include the extension in the specified filename.

#### `-Ot` and `-Os`

Selects optimization for execution time (`-Ot`) or size of executable file (`-Os`). Default is `-Os`, which creates smaller executables. This option currently has no effect except under DOS with `-xe` flag, in which case the `-Os` option creates an executable that was compressed using the `pklite` program from PKWare.

**-p path**

This option specifies a *path* where TAWK will search for files in addition to any path specified by the AWKPATH environment variable.

**-s** Strict option: Generate a warning message for undeclared variables.

**-t letter**

Where *letter* is a single letter that specifies the target operating system. The TAWK Compiler creates an executable for the specified operating system. The possible options are:

<u>Option</u>	<u>Target Operating System</u>	<u>Address Space</u>
-td	Regular 16-bit DOS (Default for DOS version.)	640K + XMS/EMS
-te	32-bit extended DOS	virtual
-tw	Win32 (Default for Windows NT and Windows 95 version)	virtual
-tp	OS/2 protected mode. The generated TAWK program will execute on any version of OS/2.	32-bit segmented
-tb	OS/2 bound dual-mode executable that runs under both DOS and OS2. (Default for OS/2 version.) Note that bound dual-mode executables are larger than DOS or OS2 protected mode executables.	32-bit segmented
-to	OS/2 32-bit program for OS/2 version 2.1 or higher, or OS/2 WARP.	virtual
-tu	Sun Solaris.	virtual

This option can be used when you want to cross-compile, that is, you want to generate an executable that runs on a different operating system than the one you are using. Note that the TAWK Compiler is supplied in separate versions for the various operating systems. An error message will be generated if you attempt to specify a target executable for which the appropriate version of the TAWK Compiler is not installed.

**[ UNIX Version: ]**

You can not cross compile between UNIX and Intel platforms, due to differences in memory addressing arrangements. In other words, you can not compile a TAWK program for UNIX on an Intel PC, or vice versa. You have to take your TAWK program to the other machine, and compile it there.

**-v** Verbose option: Print each filename as it is compiled. If the file was previously compiled and does not need to be recompiled, that fact is also noted.

**-w** Warnings option: This option suppresses warning messages. Note that the -w option to the TAWK compiler suppresses compilation warning messages, and the -w option to the compiled program suppresses runtime warning messages. The WARNINGS variable can also be used to suppress runtime warning messages.

**-W** Causes the TAWK compiler to abort the compilation and return a non-zero exit status if any warning or error occurs. Normally, TAWK does not abort for warnings, only for errors. This option might be useful when the TAWK compiler is invoked from a makefile.

**-x letter**

This option selects the type of output file according to the following table:

-xm	Generates a minimum size executable file. This is the default if no -x option is specified and only TAWK (.awk) files are specified on the command line.
-xe	Generates a stand-alone executable file.

- xl Static linking [DOS version only]: uses your linker to generate a combined TAWK/C executable. This is the default if no -x option is specified and object (.obj) or library (.lib) files are specified on the command line. See the chapter on combined AWK/C programs.
- xo Output is an object file [DOS version only]. See the chapter on combined AWK/C programs.
- xa Generate just an ".ae" file. This file contains only the TAWK executable code and can not be executed by itself. The .ae file can be run using the awkr program directly like this:

```
awkr??? filename.ae program_arguments
```

where ??? is a number unique to each version of TAWK, filename.ae is the .ae file, and the *program\_arguments* are any remaining arguments to the TAWK program.

- z Disables incremental compilation, causing the TAWK compiler to recompile all source files completely. This option is equivalent to deleting any previously compiled file before running the compiler. After many compilations, the resulting compiled file sometimes gets larger due to memory fragmentation, as files that are recompiled are first deleted from, and then restored into, new locations in the resulting compiled file. This option sometimes makes the resulting compiled file smaller, but of course, it makes compilation slower if there is more than one TAWK source file.

## Config File

The TAWK compiler reads options from a configuration file named "awkc.cfg" before reading them from the command line. The configuration file may also contain comment lines that begin with # as the first character. The configuration file is searched for in the current directory, then in the directory from which awkc.exe was invoked, then along the path specified by the AWKPATH environment variable, if any, then along any path specified with the -p option. It is possible and common to have different config files for the DOS, OS2 and Win32 versions of the TAWK Compiler, by placing a separate awkc.cfg in the bin, binp and binw directories.

## Long Command Lines And Response Files

The TAWK compiler is designed to compile many TAWK source files simultaneously. The TAWK compiler provides two ways to avoid long command lines:

- 1) Any part of a command line can be placed in an environment variable. To use the environment variable in the TAWK Compiler command line, precede it with a dollar (\$) character. For example:

```
set TAWKSTUFF=-xo -eoe -omyfile file1.awk
awkc -w $TAWKSTUFF file2.awk file3.awk
```

- 2) Any part of a command line can be placed in a response file. Precede the response file name with @ (at character) in the TAWK Compiler command line. For example:

```
echo file1.awk file2.awk > respfile
awkc -eb @respfile file3.awk
```

To prevent a fatal error if the response file does not exist, precede it with @? instead of @. The response file may contain comment lines that begin with # as the first character.

## Libraries

TAWK allows you to place code into a library. This feature is most useful for commonly used functions, so you do not need to include the function in every program that needs it. The global functions and global variables defined in the library are automatically used by any program that references them. Note that if any part of any particular library file is needed, the entire

file will be linked into your program. Therefore, the library files can contain things like INIT or TERM blocks that will be included in your program only if a global function or global variable is used from that particular library file.

To add a new function to the TAWK library, put the function into a ".awk" file, and put a copy of the ".awk" file into the "awklib" sub-directory of the TAWK installation directory. Next, cd (change directory) to the TAWK installation directory, and run the "mkawklib" program. That's it. After this, whenever your program uses any of the global functions or variables defined in the ".awk" file, that file will automatically be compiled in with your program.

How it works: when the TAWK compiler finds an unresolved reference, such as a call to a function that does not exist, or a use of variable that was never defined, it checks in a file called "awkindex", which it looks for in the directories specified in the AWKPATH environment variable, the current directory, and the directory from which it was invoked. The awkindex file contains a list of symbol names and the corresponding filenames of ".awk" files to be linked in, to resolve that symbol name. If the unresolved reference is listed in the awkindex file, the TAWK compiler links in the corresponding filename. This process is repeated as many times as necessary, so library functions can use other library functions. The awkindex file is created by the mkawklib program. This program is a simple tawk program, whose source code is provided. The mkawklib program searches each of the filenames specified on its command line, or, if no files are specified, each of the files in the "awklib" sub-directory, and creates a new awkindex file in the current directory, that lists all the global functions and global variables in those files, in the format required by the TAWK compiler.

#### [ DOS Version ]

If your computer does not have enough free memory, the TAWK Compiler may not be able to run. You can decrease the memory requirement of the TAWK Compiler by reducing the size of the awkindex file. You can edit the awkindex file, and remove any lines corresponding to function names that you expect never to use. Do not remove the first line in the file, which identifies the file.

## AWKPATH Environment Variable

If a TAWK source filename is specified without a path and is not found in the current directory, then TAWK will search each of the directories specified by the AWKPATH environment variable for the source file. The AWKPATH environment variable is set up like the PATH variable: if more than one directory is specified, they must be separated by semi-colons (DOS or OS/2) or colons (UNIX).

The TAWK Compiler also uses AWKPATH to search for the other files required for compilation including: awkcr31.exe, the overlay files, and the awk.lib file.

## Temporary Files

During the compilation phase, the awk compiler and the awk programs create temporary files in the directory specified by the TMP environment variable, or in the current directory if there is no TMP variable. The name of the temp files are "awktmp.NNN", where NNN is a number. When the user's awk program is run, it may create temporary files named "awktmp.NNN" in the directory specified by the TMPDIR variable, which is initialized to the value of the TMP environment variable, but can be changed inside the user's awk program.

#### [ DOS and OS/2 bound versions (-td and -tb options) ]

If a TAWK program requires more memory than is available, it will use disk space by creating a virtual memory paging file in the TMP directory. Temporary files will also be created if your program uses pipes, or the **system** or **spawn** functions. If you need to maximize the amount of memory your TAWK program will be able to use, you should set the TMP variable to a directory on a disk that has plenty of free disk space. Currently the DOS version will use up to 16 MB (Mega Bytes) of paging space and the OS/2 version will use up to 32 MB. Note that TAWK uses regular memory and EMS and XMS memory before using disk space, so if you have more than 16MB of XMS memory available under DOS, then TAWK will not need a paging file.

#### [ DOS 32-bit version (-te option) ]

In addition to the normal TAWK temp files, the DOS extender has the option to create a virtual memory swap file. At the time of printing, this file was named dos4vgm.swp and was created in the root directory. This file may remain after the TAWK program completes, with the ostensible purpose of making the program start faster next time. This can be changed with options set in an environment variable. See the documentation in the "awk32.txt" file if you are using this version of the TAWK compiler.

#### [ Other Versions ]

The compiled TAWK program may create temporary files in order to process pipes used in your TAWK program. Future versions of TAWK may use true pipes, in which case no temporary files will be needed.

## Incremental Compilation

This discussion applies only if your program consists of more than one source file. To decrease compilation time, the TAWK compiler recompiles only those files that have changed since the last compilation. For this purpose it records the timestamps of the source files into the executable file. Therefore you must keep the time and date on your computer correct. Alternatively, you can use the `-z` option to disable incremental compilation, or just delete the output file before each compilation to force all source files to be recompiled.

Some compiler warning messages are only printed once when the source file is first compiled. Because incremental compilation normally allows the compiler to re-compile only the most recently modified source files, these warning messages may not be printed again until you modify the source file that caused the warning, delete the output file, or specify the `-z` option to force full recompilation.

Some compiler options affect all source files and force a full recompile when you start or stop using the option. These options are `-g` and `-eb`.

For interest's sake, you can specify the `-v` option to make the compiler display the names of files that it compiles, and also the names of those files that were previously compiled and left unchanged.