In this chapter:

7

Arrays in awk

An *array* is a table of values called *elements*. The elements of an array are distinguished by their indices. *Indices* may be either numbers or strings.

This chapter describes how arrays work in *awk*, how to use array elements, how to scan through every element in an array, and how to remove array elements. It also describes how *awk* simulates multidimensional arrays, as well as some of the less obvious points about array usage. The chapter finishes with a discussion of *gawk*'s facility for sorting an array based on its indices.

awk maintains a single set of names that may be used for naming variables, arrays, and functions (see the section "User-Defined Functions" in Chapter 8, *Functions*). Thus, you cannot have a variable and an array with the same name in the same *awk* program.

Introduction to Arrays

The *awk* language provides one-dimensional arrays for storing groups of related strings or numbers. Every *awk* array must have a name. Array names have the same syntax as variable names; any valid variable name would also be a valid array name. But one name cannot be used in both ways (as an array and as a variable) in the same *awk* program.

Arrays in *awk* superficially resemble arrays in other programming languages, but there are fundamental differences. In *awk*, it isn't necessary to specify the size of an array before starting to use it. Additionally, any number or string in *awk*, not just consecutive integers, may be used as an array index.

In most other languages, arrays must be *declared* before use, including a specification of how many elements or components they contain. In such languages, the declaration causes a contiguous block of memory to be allocated for that many elements. Usually, an index in the array must be a positive integer. For example, the index zero specifies the first element in the array, which is actually stored at the beginning of the block of memory. Index one specifies the second element, which is stored in memory right after the first element, and so on. It is impossible to add more elements to the array, because it has room only for as many elements as given in the declaration. (Some languages allow arbitrary starting and ending indices—e.g., `15 .. 27`—but the size of the array is still fixed when the array is declared.)

A contiguous array of four elements might look like Figure 7-1 conceptually, if the element values are 8, `"foo"`, `""`, and 30.

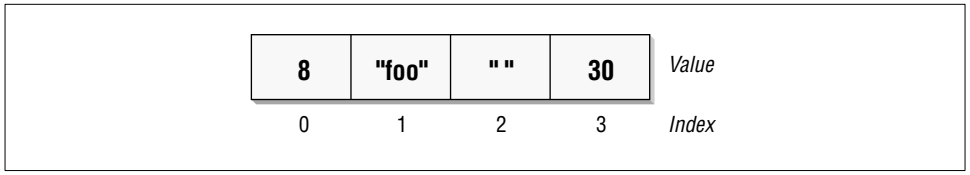

Figure 7-1. Array indexing

Only the values are stored; the indices are implicit from the order of the values. Here, 8 is the value at index zero, because 8 appears in the position with zero elements before it.

Arrays in *awk* are different—they are *associative*. This means that each array is a collection of pairs: an index and its corresponding array element value:

Index	Value
3	30
2	"foo"
0	8
2	""

The pairs are shown in jumbled order because their order is irrelevant.

One advantage of associative arrays is that new pairs can be added at any time. For example, suppose a tenth element is added to the array whose value is "number ten". The result is:

Index	Value
10	"number ten"
3	30
1	"foo"
0	8
2	""

Now the array is *sparse*, which just means some indices are missing. It has elements 0–3 and 10, but doesn't have elements 4, 5, 6, 7, 8, or 9.

Another consequence of associative arrays is that the indices don't have to be positive integers. Any number, or even a string, can be an index. For example, the following is an array that translates words from English to French:

Index	Value
"dog"	"chien"
"cat"	"chat"
"one"	"un"
1	"un"

Here we decided to translate the number one in both spelled-out and numeric form—thus illustrating that a single array can have both numbers and strings as indices. In fact, array subscripts are always strings; this is discussed in more detail in the section "Using Numbers to Subscript Arrays" later in this chapter. Here, the number 1 isn't double-quoted, since *awk* automatically converts it to a string.

The value of IGNORECASE has no effect upon array subscripting. The identical string value used to store an array element must be used to retrieve it. When *awk* creates an array (e.g., with the split built-in function), that array's indices are consecutive integers starting at one. (See the section "String-Manipulation Functions" in Chapter 8.)

awk's arrays are efficient—the time to access an element is independent of the number of elements in the array.

Referring to an Array Element

The principal way to use an array is to refer to one of its elements. An array reference is an expression as follows:

```
array[index]
```

Here, *array* is the name of an array. The expression *index* is the index of the desired element of the array.

The value of the array reference is the current value of that array element. For example, `foo[4.3]` is an expression for the element of array `foo` at index `4.3`.

A reference to an array element that has no recorded value yields a value of `""`, the null string. This includes elements that have not been assigned any value as well as elements that have been deleted (see the section "The delete Statement" later in this chapter). Such a reference automatically creates that array element, with the null string as its value. (In some cases, this is unfortunate, because it might waste memory inside *awk*.)

To determine whether an element exists in an array at a certain index, use the following expression:

```
index in array
```

This expression tests whether the particular index exists, without the side effect of creating that element if it is not present. The expression has the value one (true) if *array*[*index*] exists and zero (false) if it does not exist. For example, this statement tests whether the array `frequencies` contains the index `2`:

```
if (2 in frequencies)
    print "Subscript 2 is present."
```

Note that this is *not* a test of whether the array `frequencies` contains an element whose *value* is two. There is no way to do that except to scan all the elements. Also, this *does not* create `frequencies[2]`, while the following (incorrect) alternative does:

```
if (frequencies[2] != "")
    print "Subscript 2 is present."
```

Assigning Array Elements

Array elements can be assigned values just like *awk* variables:

```
array[subscript] = value
```

array is the name of an array. The expression *subscript* is the index of the element of the array that is assigned a value. The expression *value* is the value to assign to that element of the array.

Basic Array Example

The following program takes a list of lines, each beginning with a line number, and prints them out in order of line number. The line numbers are not in order when they are first read—instead they are scrambled. This program sorts the lines by making an array using the line numbers as subscripts. The program then prints out the lines in sorted order of their numbers. It is a very simple program and gets confused upon encountering repeated numbers, gaps, or lines that don't begin with a number:

```
{
    if ($1 > max)
        max = $1
    arr[$1] = $0
}

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

The first rule keeps track of the largest line number seen so far; it also stores each line into the array `arr`, at an index that is the line's number. The second rule runs after all the input has been read, to print out all the lines. When this program is run with the following input:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

Its output is:

```
1  Who is number one?
2  Who are you?  The new number two!
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

If a line number is repeated, the last line with a given number overrides the others. Gaps in the line numbers can be handled with an easy improvement to the program's `END` rule, as follows:

```
END {
    for (x = 1; x <= max; x++)
        if (x in arr)
            print arr[x]
}
```

Scanning All Elements of an Array

In programs that use arrays, it is often necessary to use a loop that executes once for each element of an array. In other languages, where arrays are contiguous and indices are limited to positive integers, this is easy: all the valid indices can be found by counting from the lowest index up to the highest. This technique won't do the job in *awk*, because any number or string can be an array index. So *awk* has a special kind of `for` statement for scanning an array:

```
for (var in array)
    body
```

This loop executes *body* once for each index in *array* that the program has previously used, with the variable *var* set to that index.

The following program uses this form of the `for` statement. The first rule scans the input records and notes which words appear (at least once) in the input, by storing a one into the array `used` with the word as index. The second rule scans the elements of `used` to find all the distinct words that appear in the input. It prints each word that is more than 10 characters long and also prints the number of such words. See the section "String-Manipulation Functions" in Chapter 8 for more information on the built-in function `length`:

```
# Record a 1 for each word that is used at least once
{
    for (i = 1; i <= NF; i++)
        used[$i] = 1
}

# Find number of distinct words more than 10 characters long
END {
    for (x in used)
        if (length(x) > 10) {
            ++num_long_words
            print x
        }
    print num_long_words, "words longer than 10 characters"
}
```

See the section "Generating Word-Usage Counts" in Chapter 13, *Practical awk Programs*, for a more detailed example of this type.

The order in which elements of the array are accessed by this statement is determined by the internal arrangement of the array elements within *awk* and cannot be controlled or changed. This can lead to problems if new elements are added to *array* by statements in the loop body; it is not predictable whether the `for` loop will reach them. Similarly, changing *var* inside the loop may produce strange results. It is best to avoid such things.

The delete Statement

To remove an individual element of an array, use the `delete` statement:

```
delete array[index]
```

Once an array element has been deleted, any value the element once had is no longer available. It is as if the element had never been referred to or had been given a value. The following is an example of deleting elements in an array:

```
for (i in frequencies)
    delete frequencies[i]
```

This example removes all the elements from the array `frequencies`. Once an element is deleted, a subsequent `for` statement to scan the array does not report that element and the `in` operator to check for the presence of that element returns zero (i.e., false):

```
delete foo[4]
if (4 in foo)
    print "This will never be printed"
```

It is important to note that deleting an element is *not* the same as assigning it a null value (the empty string, `""`). For example:

```
foo[4] = ""
if (4 in foo)
    print "This is printed, even though foo[4] is empty"
```

It is not an error to delete an element that does not exist. If *--lint* is provided on the command line (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*), *gawk* issues a warning message when an element that is not in the array is deleted.

All the elements of an array may be deleted with a single statement by leaving off the subscript in the `delete` statement, as follows:

```
delete array
```

This ability is a *gawk* extension; it is not available in compatibility mode (see the section "Command-Line Options" in Chapter 11).

Using this version of the `delete` statement is about three times more efficient than the equivalent loop that deletes each element one at a time.

The following statement provides a portable but nonobvious way to clear out an array:*

```
split("", array)
```

The `split` function (see the section "String-Manipulation Functions" in Chapter 8) clears out the target array first. This call asks it to split apart the null string. Because there is no data to split out, the function simply clears the array and then returns.

Deleting an array does not change its type; you cannot delete an array and then use the array's name as a scalar (i.e., a regular variable). For example, the following does not work:

```
a[1] = 3; delete a; a = 3
```

Using Numbers to Subscript Arrays

An important aspect about arrays to remember is that *array subscripts are always strings*. When a numeric value is used as a subscript, it is converted to a string value before being used for subscripting (see the section "Conversion of Strings and Numbers" in Chapter 5, *Expressions*). This means that the value of the built-in variable `CONVFMT` can affect how your program accesses elements of an array. For example:

```
xyz = 12.153
data[xyz] = 1
CONVFMT = "%2.2f"
if (xyz in data)
    printf "%s is in data\n", xyz
else
    printf "%s is not in data\n", xyz
```

* Thanks to Michael Brennan for pointing this out.

This prints `12.15 is not in data`. The first statement gives `xyz` a numeric value. Assigning to `data[xyz]` subscripts `data` with the string value `"12.153"` (using the default conversion value of `CONVFMT`, `"%.6g"`). Thus, the array element `data["12.153"]` is assigned the value one. The program then changes the value of `CONVFMT`. The test `(xyz in data)` generates a new string value from `xyz`—this time `"12.15"`—because the value of `CONVFMT` only allows two significant digits. This test fails, since `"12.15"` is a different string from `"12.153"`.

According to the rules for conversions (see the section "Conversion of Strings and Numbers" in Chapter 5), integer values are always converted to strings as integers, no matter what the value of `CONVFMT` may happen to be. So the usual case of the following works:

```
for (i = 1; i <= maxsub; i++)
    do something with array[i]
```

The "integer values always convert to strings as integers" rule has an additional consequence for array indexing. Octal and hexadecimal constants (see the section "Octal and Hexadecimal Numbers" in Chapter 5) are converted internally into numbers, and their original form is forgotten. This means, for example, that `array[17]`, `array[021]`, and `array[0x11]` all refer to the same element!

As with many things in *awk*, the majority of the time things work as one would expect them to. But it is useful to have a precise knowledge of the actual rules which sometimes can have a subtle effect on your programs.

Using Uninitialized Variables as Subscripts

Suppose it's necessary to write a program to print the input data in reverse order. A reasonable attempt to do so (with some test data) might look like this:

```
$ echo 'line 1
> line 2
> line 3' | awk '{ l[lines] = $0; ++lines }
> END {
>     for (i = lines-1; i >= 0; --i)
>        print l[i]
> }'
line 3
line 2
```

Unfortunately, the very first line of input data did not come out in the output!

At first glance, this program should have worked. The variable `lines` is uninitialized, and uninitialized variables have the numeric value zero. So, *awk* should have printed the value of `l[0]`.

The issue here is that subscripts for *awk* arrays are *always* strings. Uninitialized variables, when used as strings, have the value `""`, not zero. Thus, `line 1` ends up stored in `l[""]`. The following version of the program works correctly:

```
{ l[lines++] = $0 }
END {
    for (i = lines - 1; i >= 0; --i)
        print l[i]
}
```

Here, the `++` forces `lines` to be numeric, thus making the "old value" numeric zero. This is then converted to `"0"` as the array subscript.

Even though it is somewhat unusual, the null string (`""`) is a valid array subscript. (d.c.) *gawk* warns about the use of the null string as a subscript if *––lint* is provided on the command line (see the section "Command-Line Options" in Chapter 11).

Multidimensional Arrays

A multidimensional array is an array in which an element is identified by a sequence of indices instead of a single index. For example, a two-dimensional array requires two indices. The usual way (in most languages, including *awk*) to refer to an element of a two-dimensional array named `grid` is with `grid[x,y]`.

Multidimensional arrays are supported in *awk* through concatenation of indices into one string. *awk* converts the indices into strings (see the section "Conversion of Strings and Numbers" in Chapter 5) and concatenates them together, with a separator between them. This creates a single string that describes the values of the separate indices. The combined string is used as a single index into an ordinary, one-dimensional array. The separator used is the value of the built-in variable `SUBSEP`.

For example, suppose we evaluate the expression `foo[5,12] = "value"` when the value of `SUBSEP` is `"@"`. The numbers 5 and 12 are converted to strings and concatenated with an `@` between them, yielding `"5@12"`; thus, the array element `foo["5@12"]` is set to `"value"`.

Once the element's value is stored, *awk* has no record of whether it was stored with a single index or a sequence of indices. The two expressions `foo[5,12]` and `foo[5 SUBSEP 12]` are always equivalent.

The default value of `SUBSEP` is the string `"\034"`, which contains a nonprinting character that is unlikely to appear in an *awk* program or in most input data. The usefulness of choosing an unlikely character comes from the fact that index values that contain a string matching `SUBSEP` can lead to combined strings that are ambiguous. Suppose that `SUBSEP` is `"@"`; then `foo["a@b", "c"]` and `foo["a",`

`"b@c"]` are indistinguishable because both are actually stored as `foo["a@b@c"]`.

To test whether a particular index sequence exists in a multidimensional array, use the same operator (`in`) that is used for single dimensional arrays. Write the whole sequence of indices in parentheses, separated by commas, as the left operand:

```
(subscript1, subscript2, ...) in array
```

The following example treats its input as a two-dimensional array of fields; it rotates this array 90 degrees clockwise and prints the result. It assumes that all lines have the same number of elements:

```
{
    if (max_nf < NF)
        max_nf = NF
    max_nr = NR
    for (x = 1; x <= NF; x++)
        vector[x, NR] = $x
}

END {
    for (x = 1; x <= max_nf; x++) {
        for (y = max_nr; y >= 1; --y)
            printf("%s ", vector[x, y])
        printf("\n")
    }
}
```

When given the input:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

the program produces the following output:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

Scanning Multidimensional Arrays

There is no special `for` statement for scanning a "multidimensional" array. There cannot be one because, in truth, there are no multidimensional arrays or elements—there is only a multidimensional *way of accessing* an array.

However, if your program has an array that is always accessed as multidimensional, you can get the effect of scanning it by combining the scanning `for`

statement (see the section "Scanning All Elements of an Array" earlier in this chapter) with the built-in `split` function (see the section "String-Manipulation Functions" in Chapter 8). It works in the following manner:

```
for (combined in array) {
    split(combined, separate, SUBSEP)
    ...
}
```

This sets the variable `combined` to each concatenated combined index in the array, and splits it into the individual indices by breaking it apart where the value of `SUBSEP` appears. The individual indices then become the elements of the array `separate`.

Thus, if a value is previously stored in `array[1, "foo"]`; then an element with index `"1\034foo"` exists in `array`. (Recall that the default value of `SUBSEP` is the character with code 034.) Sooner or later, the `for` statement finds that index and does an iteration with the variable `combined` set to `"1\034foo"`. Then the `split` function is called as follows:

```
split("1\034foo", separate, "\034")
```

The result is to set `separate[1]` to `"1"` and `separate[2]` to `"foo"`. Presto! The original sequence of separate indices is recovered.

Sorting Array Values and Indices with gawk

The order in which an array is scanned with a `for (i in array)` loop is essentially arbitrary. In most *awk* implementations, sorting an array requires writing a `sort` function. While this can be educational for exploring different sorting algorithms, usually that's not the point of the program. *gawk* provides the built-in `asort` function (see the section "String-Manipulation Functions" in Chapter 8) that sorts an array. For example:

```
populate the array data
n = asort(data)
for (i = 1; i <= n; i++)
    do something with data[i]
```

After the call to `asort`, the array `data` is indexed from 1 to some number *n*, the total number of elements in `data`. (This count is `asort`'s return value.) `data[1]` ≤ `data[2]` ≤ `data[3]`, and so on. The comparison of array elements is done using *gawk*'s usual comparison rules (see the section "Variable Typing and Comparison Expressions" in Chapter 5).

An important side effect of calling `asort` is that *the array's original indices are irrevocably lost.* As this isn't always desirable, `asort` accepts a second argument:

```
populate the array source
n = asort(source, dest)
for (i = 1; i <= n; i++)
     do something with dest[i]
```

In this case, *gawk* copies the `source` array into the `dest` array and then sorts `dest`, destroying its indices. However, the `source` array is not affected.

Often, what's needed is to sort on the values of the *indices* instead of the values of the elements. To do this, use a helper array to hold the sorted index values, and then access the original array's elements. It works in the following way:

```
populate the array data
# copy indices
j = 1
for (i in data) {
    ind[j] = i    # index value becomes element value
    j++
}
n = asort(ind)    # index values are now sorted
for (i = 1; i <= n; i++)
     do something with data[ind[i]]
```

Sorting the array by replacing the indices provides maximal flexibility. To traverse the elements in decreasing order, use a loop that goes from *n* down to 1, either over the elements or over the indices.

Copying array indices and elements isn't expensive in terms of memory. Internally, *gawk* maintains *reference counts* to data. For example, when `asort` copies the first array to the second one, there is only one copy of the original array elements' data, even though both arrays use the values. Similarly, when copying the indices from `data` to `ind`, there is only one copy of the actual index strings.

As with array subscripts, the value of `IGNORECASE` does not affect array sorting.