Common threads: Awk by example, Part 2
Records, loops, and arrays

Daniel Robbins
President/CEO, Gentoo Technologies, Inc.
January 2001

In this sequel to his previous intro to awk, Daniel Robbins continues to explore awk, a great language with a strange name. Daniel will show you how to handle multi-line records, use looping constructs, and create and use awk arrays. By the end of this article, you'll be well versed in a wide range of awk features, and you'll be ready to write your own powerful awk scripts.

Contents:

Multi-line records
OFS and ORS
Multi-line to tabbed
Looping constructs
Break and continue
Arrays
Array index stringiness
Array tools
Resources
About the author

Multi-line records

Awk is an excellent tool for reading in and processing structured data, such as the system's /etc/passwd file. /etc/passwd is the UNIX user database, and is a colon-delimited text file, containing a lot of important information, including all existing user accounts and user IDs, among other things. In my previous article, I showed you how awk could easily parse this file. All we had to do was to set the FS (field separator) variable to ":".

By setting the FS variable correctly, awk can be configured to parse almost any kind of structured data, as long as there is one record per line. However, just setting FS won't do us any good if we want to parse a record that exists over multiple lines. In these situations, we also need to modify the RS record separator variable. The RS variable tells awk when the current record ends and a new record begins.

As an example, let's look at how we'd handle the task of processing an address list of Federal Witness Protection Program participants:

```
Jimmy the Weasel
100 Pleasant Drive
San Francisco, CA 12345

Big Tony
200 Incognito Ave.
Suburbia, WA 67890
```

Ideally, we'd like awk to recognize each 3-line address as an individual record, rather than as three separate records. It would make our code a lot simpler if awk would recognize the first line of the address as the first field ($1), the street address as the second field ($2), and the city, state, and zip code as field $3. The following code will do just what we want:

```
BEGIN {
        FS="\n"
        RS=""
}
```

Above, setting FS to "\n" tells awk that each field appears on its own line. By setting RS to "", we also tell awk that each address record is separated by a blank line. Once awk knows how the input is formatted, it can do all the parsing work for us, and the rest of the script is simple. Let's look at a complete script that will parse this address list and print out each address record on a single line, separating each field with a comma.

address.awk

```
BEGIN {
        FS="\n"
        RS=""
}

{
        print $1 ", " $2 ", " $3

}
```

If this script is saved as address.awk, and the address data is stored in a file called address.txt, you can execute this script by typing "awk -f address.awk address.txt". This code produces the following output:

```
Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345
Big Tony, 200 Incognito Ave., Suburbia, WA 67890
```

OFS and ORS

In address.awk's print statement, you can see that awk concatenates (joins) strings that are placed next to each other on a line. We used this feature to insert a comma and a space (", ") between the three address fields that appeared on the line. While this method works, it's a bit ugly looking. Rather than inserting literal ", " strings between our fields, we can have awk do it for us by setting a special awk variable called OFS. Take a look at this code snippet.

```
print "Hello", "there", "Jim!"
```

The commas on this line are not part of the actual literal strings. Instead, they tell awk that "Hello", "there", and "Jim!" are separate fields, and that the OFS variable should be printed between each string. By default, awk produces the following output:

```
Hello there Jim!
```

This shows us that by default, OFS is set to " ", a single space. However, we can easily redefine OFS so that

awk will insert our favorite field separator. Here's a revised version of our original address.awk program that uses OFS to output those intermediate ", " strings:

A revised version of address.awk

```
BEGIN {
        FS="\n"
        RS=""
        OFS=", "
}

{
        print $1, $2, $3

}
```

Awk also has a special variable called ORS, called the "output record separator". By setting ORS, which defaults to a newline ("\n"), we can control the character that's automatically printed at the end of a print statement. The default ORS value causes awk to output each new print statement on a new line. If we wanted to make the output double-spaced, we would set ORS to "\n\n". Or, if we wanted records to be separated by a single space (and no newline), we would set ORS to " ".

Multi-line to tabbed

Let's say that we wrote a script that converted our address list to a single-line per record, tab-delimited format for import into a spreadsheet. After using a slightly modified version of address.awk, it would become clear that our program only works for three-line addresses. If awk encountered the following address, the fourth line would be thrown away and not printed:

```
Cousin Vinnie
Vinnie's Auto Shop
300 City Alley
Sosueme, OR 76543
```

To handle situations like this, it would be good if our code took the number of records per field into account, printing each one in order. Right now, the code only prints the first three fields of the address. Here's some code that does what we want:

A version of address.awk that works for addresses with any number of fields

```
BEGIN {
    FS="\n"
    RS=""
    ORS=""
}

{
        x=1
        while ( x<NF ) {
                print $x "\t"
                x++
        }
        print $NF "\n"
}
```

First, we set the field separator FS to "\n" and the record separator RS to "" so that awk parses the multi-line addresses correctly, as before. Then, we set the output record separator ORS to "", which will cause the print statement to *not* output a newline at the end of each call. This means that if we want any text to start on a new line, we need to explicitly write `print "\n"`.

In the main code block, we create a variable called x that holds the number of current field that we're processing. Initially, it's set to 1. Then, we use a while loop (an awk looping construct identical to that found in the C language) to iterate through all but the last record, printing the record and a tab character. Finally, we print the last record and a literal newline; again, since ORS is set to "", print won't output newlines for us. Program output looks like this, which is exactly what we wanted:

Our intended output. Not pretty, but tab delimited for easy import into a spreadsheet

```
Jimmy the Weasel        100 Pleasant Drive      San Francisco, CA 12345
Big Tony        200 Incognito Ave.      Suburbia, WA 67890
Cousin Vinnie   Vinnie's Auto Shop      300 City Alley  Sosueme, OR 76543
```

Looping constructs

We've already seen awk's while loop construct, which is identical to its C counterpart. Awk also has a "do...while" loop that evaluates the condition at the end of the code block, rather than at the beginning like a standard while loop. It's similar to "repeat...until" loops that can be found in other languages. Here's an example:

do...while example

```
{
        count=1
        do {
                print "I get printed at least once no matter what"
        } while ( count != 1 )
}
```

Because the condition is evaluated after the code block, a "do...while" loop, unlike a normal while loop, will always execute at least once. On the other hand, a normal while loop will never execute if its condition is false when the loop is first encountered.

for loops
Awk allows you to create for loops, which like while loops are identical to their C counterpart:

```
for ( initial assignment; comparison; increment ) {
        code block
}
```

Here's a quick example:

```
for ( x = 1; x <= 4; x++ ) {
        print "iteration",x
}
```

This snippet will print:

```
iteration 1
iteration 2
iteration 3
iteration 4
```

Break and continue
Again, just like C, awk provides break and continue statements. These statements provide better control over awk's various looping constructs. Here's a code snippet that desperately needs a break statement:

An infinite while loop

```
while (1) {
        print "forever and ever..."
}
```

Because `1` is always true, this while loop runs forever. Here's a loop that only executes ten times:

An example of the break statement

```
x=1
while(1) {
        print "iteration",x
        if ( x == 10 ) {
                break
        }
        x++
}
```

Here, the break statement is used to "break out" of the innermost loop. "break" causes the loop to immediately terminate and execution to continue at the line after the loop's code block.

The continue statement complements break, and works like this:

```
x=1
while (1) {
        if ( x == 4 ) {
                x++
                continue
        }
        print "iteration",x
        if ( x > 20 ) {
                break
        }
        x++
}
```

This code will print "iteration 1" through "iteration 21", except for "iteration 4". If iteration equals 4, x is incremented and the continue statement is called, which immediately causes awk to start to the next loop iteration without executing the rest of the code block. The continue statement works for every kind of awk iterative loop, just as break does. When used in the body of a for loop, continue will cause the loop control variable to be automatically incremented. Here's an equivalent for loop:

```
for ( x=1; x<=21; x++ ) {
        if ( x == 4 ) {
                continue
        }
        print "iteration",x
}
```

It wasn't necessary to increment x just before calling continue as it was in our while loop, since the for loop increments x automatically.

Arrays

You'll be pleased to know that awk has arrays. However, under awk, it's customary to start array indices at 1, rather than 0:

```
myarray[1]="jim"
myarray[2]=456
```

When awk encounters the first assignment, `myarray` is created and the element `myarray[1]` is set to "jim". After the second assignment is evaluated, the array has two elements.

Iterating over arrays
Once defined, awk has a handy mechanism to iterate over the elements of an array, as follows:

```
for ( x in myarray ) {
        print myarray[x]
}
```

This code will print out every element in the array `myarray`. When you use this special "in" form of a for loop, awk will assign every existing index of `myarray` to `x` (the loop control variable) in turn, executing the loop's code block once after each assignment. While this is a very handy awk feature, it does have one drawback -- when awk cycles through the array indices, it doesn't follow any particular order. That means that there's no way for us to know whether the output of above code will be:

```
jim
456
```

or

```
456
jim
```

To loosely paraphrase Forrest Gump, iterating over the contents of an array is like a box of chocolates -- you never know what you're going to get. This has something to do with the "stringiness" of awk arrays, which we'll now take a look at.

Array index stringiness
In my previous article, I showed you that awk actually stores numeric values in a string format. While awk performs the necessary conversions to make this work, it does open the door for some odd-looking code:

```
a="1"
b="2"
c=a+b+3
```

After this code executes, `c` is equal to `6`. Since awk is "stringy", adding strings "1" and "2" is functionally no different than adding the numbers 1 and 2. In both cases, awk will successfully perform the math. Awk's "stringy" nature is pretty intriguing -- you may wonder what happens if we use string indexes for arrays. For instance, take the following code:

```
myarr["1"]="Mr. Whipple"
print myarr["1"]
```

As you might expect, this code will print "Mr. Whipple". But how about if we drop the quotes around the second "1" index?

```
myarr["1"]="Mr. Whipple"
print myarr[1]
```

Guessing the result of this code snippet is a bit more difficult. Does awk consider `myarr["1"]` and `myarr[1]` to be two separate elements of the array, or do they refer to the same element? The answer is that they refer to the same element, and awk will print "Mr. Whipple", just as in the first code snippet. Although it may seem strange, behind the scenes awk has been using string indexes for its arrays all this time!

After learning this strange fact, some of us may be tempted to execute some wacky code that looks like this:

```
myarr["name"]="Mr. Whipple"
print myarr["name"]
```

Not only does this code not raise an error, but it's functionally identical to our previous examples, and will print "Mr. Whipple" just as before! As you can see, awk doesn't limit us to using pure integer indexes; we can use string indexes if we want to, without creating any problems. Whenever we use non-integer array indices like `myarr["name"]`, we're using *associative arrays*. Technically, awk isn't doing anything different behind the scenes than when we use a string index (since even if you use an "integer" index, awk still treats it as a string). However, you should still call 'em *associative arrays* -- it sounds cool and will impress your boss. The stringy index thing will be our little secret. ;)

Array tools

When it comes to arrays, awk gives us a lot of flexibility. We can use string indexes, and we aren't required to have a continuous numeric sequence of indices (for example, we can define `myarr[1]` and `myarr[1000]`, but leave all other elements undefined). While all this can be very helpful, in some circumstances it can create confusion. Fortunately, awk offers a couple of handy features to help make arrays more manageable.

First, we can delete array elements. If you want to delete element `1` of your array `fooarray`, type:

```
delete fooarray[1]
```

And, if you want to see if a particular array element exists, you can use the special "in" boolean operator as follows:

```
if ( 1 in fooarray ) {
        print "Ayep!  It's there."
} else {
        print "Nope!  Can't find it."
}
```

Next time

We've covered a lot of ground in this article. Next time, I'll round out your awk knowledge by showing you how to use awk's math and string functions and how to create your own functions. I'll also walk you through the creation of a checkbook balancing program. Until then, I encourage you to write some of your own awk programs, and to check out the following resources.

Resources

- Read Awk by example, Part 1 on *developerWorks*.
- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition is a wonderful choice.
- Be sure to check out the comp.lang.awk FAQ. It also contains lots of additional awk links.
- Patrick Hartigan's awk tutorial is packed with handy awk scripts.
- Thompson's TAWK Compiler compiles awk scripts into fast binary executables. Versions are available for Windows, OS/2, DOS, and UNIX.
- The GNU Awk User's Guide is available for online reference.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can reach Daniel at drobbins@gentoo.org.