35 – Arrays

In the previous chapter, we looked at how the shell can manipulate strings and numbers. The data types we have looked at so far are known in computer science circles as *scalar variables*; that is, they are variables that contain a single value.

In this chapter, we will look at another kind of data structure called an *array*, which holds multiple values. Arrays are a feature of virtually every programming language. The shell supports them, too, though in a rather limited fashion. Even so, they can be very useful for solving some types of programming problems.

What Are Arrays?

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let's consider a spreadsheet as an example. A spreadsheet acts like a *two-dimensional array*. It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called *elements*, and each element contains data. An individual array element is accessed using an address called an *index* or *subscript*.

Most programming languages support *multidimensional arrays*. A spreadsheet is an example of a multidimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two- and three-dimensional arrays are probably the most commonly used.

Arrays in `bash` are limited to a single dimension. We can think of them as a spreadsheet with a single column. Even with this limitation, there are many applications for them. Array support first appeared in `bash` version 2. The original Unix shell program, `sh`, did not support arrays at all.

Creating an Array

Array variables are named just like other `bash` variables, and are created automatically when they are accessed. Here is an example:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Here we see an example of both the assignment and access of an array element. With the first command, element 1 of array `a` is assigned the value "foo". The second command displays the stored value of element 1. The use of braces in the second command is required to prevent the shell from attempting pathname expansion on the name of the array element.

An array can also be created with the `declare` command.

```
[me@linuxbox ~]$ declare -a a
```

Using the `-a` option, this example of `declare` creates the array `a`.

Assigning Values to an Array

Values may be assigned in one of two ways. Single values may be assigned using the following syntax:

name[*subscript*]=*value*

where *name* is the name of the array and *subscript* is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. *value* is a string or integer assigned to the array element.

Multiple values may be assigned using the following syntax:

name=(*value1 value2 ...*)

where *name* is the name of the array and *value* placeholders are values assigned sequentially to elements of the array, starting with element zero. For example, if we wanted to assign abbreviated days of the week to the array `days`, we could do this:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

It is also possible to assign values to a specific element by specifying a subscript for each value.

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu
[5]=Fri [6]=Sat)
```

Accessing Array Elements

So what are arrays good for? Just as many data-management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays.

Let's consider a simple data-gathering and presentation example. We will construct a script that examines the modification times of the files in a specified directory. From this data, our script will output a table showing at what hour of the day the files were last modified. Such a script could be used to determine when a system is most active. This script, called `hours`, produces this result:

```
[me@linuxbox ~]$ hours .
Hour  Files  Hour  Files
----  -----  ----  -----
00    0      12    11
01    1      13    7
02    0      14    1
03    0      15    7
04    1      16    6
05    1      17    5
06    6      18    4
07    3      19    4
08    1      20    1
09    14     21    0
10    2      22    0
11    5      23    0

Total files = 80
```

We execute the `hours` program, specifying the current directory as the target. It produces a table showing, for each hour of the day (0-23), how many files were last modified. The code to produce this is as follows:

```
#!/bin/bash

# hours: script to count files by modification time

usage () {
    echo "usage: ${0##*/} directory" >&2
}

# Check that argument is a directory
```

```
if [[ ! -d "$1" ]]; then
    usage
    exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
    j="${i#0}"
    ((++hours[j]))
    ((++count))
done

# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" \
         "$i" \
         "${hours[i]}" \
         "$j" \
         "${hours[j]}"
done
printf "\nTotal files = %d\n" $count
```

The script consists of one function (`usage`) and a main body with four sections. In the first section, we check that there is a command line argument and that it is a directory. If it is not, we display the usage message and exit.

The second section initializes the array `hours`. It does this by assigning each element a value of zero. There is no special requirement to prepare arrays prior to use, but our script needs to ensure that no element is empty. Note the interesting way the loop is constructed. By employing brace expansion (`{0..23}`), we are able to easily generate a sequence of words for the `for` command.

The next section gathers the data by running the `stat` program on each file in the directory. We use `cut` to extract the two-digit hour from the result. Inside the loop, we need to remove leading zeros from the hour field, since the shell will try (and ultimately fail) to interpret values `00` through `09` as octal numbers (see Table 34-2). Next, we increment the value of the array element corresponding with the hour of the day. Finally, we increment

a counter (`count`) to track the total number of files in the directory.

The last section of the script displays the contents of the array. We first output a couple of header lines and then enter a loop that produces four columns of output. Lastly, we output the final tally of files.

Array Operations

There are many common array operations. Such things as deleting arrays, determining their size, sorting, and so on, have many applications in scripting.

Outputting the Entire Contents of an Array

The subscripts `*` and `@` can be used to access every element in an array. As with positional parameters, the `@` notation is the more useful of the two. Here is a demonstration:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

We create the array `animals` and assign it three two-word strings. We then execute four loops to see the effect of word splitting on the array contents. The behavior of notations `${animals[*]}` and `${animals[@]}` is identical until they are quoted. The `*` notation results in a single word containing the array's contents, while the `@` notation results

in three two-word strings, which matches the array's "real" contents.

Determining the Number of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]}  # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]}  # length of element 100
3
```

We create array `a` and assign the string `foo` to element 100. Next, we use parameter expansion to examine the length of the array, using the `@` notation. Finally, we look at the length of element 100, which contains the string `foo`. It is interesting to note that while we assigned our string to element 100, bash reports only one element in the array. This differs from the behavior of some other languages in which the unused elements of the array (elements 0-99) would be initialized with empty values and counted. In `bash`, array elements exist only if they have been assigned a value regardless of their subscript.

Finding the Subscripts Used by an Array

As `bash` allows arrays to contain "gaps" in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

`${!`*`array`*`[*]}`

`${!`*`array`*`[@]}`

where *array* is the name of an array variable. Like the other expansions that use `*` and `@`, the `@` form enclosed in quotes is the most useful, as it expands into separate words.

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
b
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
2
4
```

```
6
```

Adding Elements to the End of an Array

Knowing the number of elements in an array is no help if we need to append values to the end of an array since the values returned by the * and @ notations do not tell us the maximum array index in use. Fortunately, the shell provides us with a solution. By using the += assignment operator, we can automatically append values to the end of an array. Here, we assign three values to the array foo and then append three more.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

Sorting an Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding.

```
#!/bin/bash

# array-sort: Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array:   ${a_sorted[@]}"
```

When executed, the script produces this:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:   a b c d e f
```

The script operates by copying the contents of the original array (`a`) into a second array (`a_sorted`) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

Deleting an Array

To delete an array, use the `unset` command.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

`unset` may also be used to delete single array elements.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

In this example, we delete the third element of the array, subscript 2. Remember, arrays start with subscript zero, not one! Notice also that the array element must be quoted to prevent the shell from performing pathname expansion.

Interestingly, the assignment of an empty value to an array does not empty its contents.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

Any reference to an array variable without a subscript refers to element zero of the array.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]}
A b c d e f
```

Associative Arrays

bash versions 4.0 and greater support *associative arrays*. Associative arrays use strings rather than integers as array indexes. This capability allows interesting new approaches to managing data. For example, we can create an array called colors and use color names as indexes.

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

Unlike integer indexed arrays, which are created by merely referencing them, associative arrays must be created with the declare command using the new -A option. Associative array elements are accessed in much the same way as integer indexed arrays.

```
echo ${colors["blue"]}
```

In the next chapter, we will look at a script that makes good use of associative arrays to produce an interesting report.

Summing Up

If we search the bash man page for the word *array*, we find many instances of where bash makes use of array variables. Most of these are rather obscure, but they may provide occasional utility in some special circumstances. In fact, the entire topic of arrays is rather under-utilized in shell programming owing largely to the fact that the traditional Unix shell programs (such as sh) lacked any support for arrays. This lack of popularity is unfortunate because arrays are widely used in other programming languages and provide a powerful tool for solving many kinds of programming problems.

Arrays and loops have a natural affinity and are often used together. The following form

of loop is particularly well-suited to calculating array subscripts:

```
for ((expr; expr; expr))
```

Further Reading

- A couple of Wikipedia articles about the data structures found in this chapter: http://en.wikipedia.org/wiki/Scalar_(computing)

  http://en.wikipedia.org/wiki/Associative_array