Arrays

- 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.
- Arrays in bash are limited to a single dimension. We can think of them as a spreadsheet with a single column.

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=([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.
- 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
```

Assigning Array Elements with read -a

- The `read` built-in has an option (-a) to place words into an indexed array rather than a series of variables as we have done before. Here is an example:

```
[me@linuxbox ~]$ declare -a foo
[me@linuxbox ~]$ read -a foo <<< "0th 1st 2nd 3rd 4th"
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo "$i"; done
0th
1st
2nd
3rd
4th
```

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
```

Reading a File Into an Array

- Recent versions of bash include a new builtin named mapfile which can read standard input directly into an indexed array. Its syntax looks like this:

```
mapfile -options array
```

- The `mapfile` command has some interesting options some of which are shown in the table

Option	Description
`-d` *`delim`*	Use *`delim`* to terminate lines rather than a newline.
`-n` *`count`*	Only read *`count`* lines.
`-O` *`origin`*	Begin assigning array elements at index *`origin`* rathe index 0.
`-s` *`count`*	Skip *`count`* lines at the beginning of the file.
`-t`	Trim trailing delimiter from each line.

- To demonstrate mapfile in action, we’ll create a short script that produces random four-word passphrases. These are useful as alternatives to conventional passwords because they are long and yet easy to remember

```
#!/bin/bash

# array-mapfile - demonstrate mapfile builtin
```

```
DICTIONARY=/usr/share/dict/words
WORDLIST=~/wordlist.txt
declare -a words

# create filtered word list
grep -v \' < "$DICTIONARY" \
    | grep -v "[[:upper:]]" \
    | shuf > "$WORDLIST"

# read WORDLIST into array
mapfile -t -n 32767 words < "$WORDLIST"

# create four word passphrase
while [[ -z $REPLY ]]; do
    echo "${words[$RANDOM]}" \
        "${words[$RANDOM]}" \
        "${words[$RANDOM]}" \
        "${words[$RANDOM]}"
    echo
    read -r -p "Enter to continue, q to quit > "
    echo
done
```

- This script uses the /usr/share/dict/words file filtered to remove apostrophes and uppercase letters. We use the shuf command to shuffle the word list to get a nice random order.

- We next load the first 32767 words in the file into the words array. Why 32767? It's because we are going to use the RANDOM variable to choose random elements from the array and each time the RANDOM variable is referenced it returns a random integer between 0 and 32767.

```
[me@linuxbox ~]$ ./array-mapfile
conversions slumbers appendages metastasizing

Enter to continue, q to quit >

kettles rhinestones unused demagnetizes

Enter to continue, q to quit >

wear conveys characterizing extrusion

Enter to continue, q to quit > q
```

- Here we see the output. Each time we press Enter the script displays four random words.

Slicing an Array

- There is a form of parameter expansion we can use to extract a group of contiguous elements called a slice from an array. This expansion results in array elements from the desired slice of the original array as shown below.

```
[me@linuxbox ~]$ arr=(0th 1st 2nd 3rd 4th)
[me@linuxbox ~]$ echo "${arr[@]:2:3}"
2nd 3rd 4th
```

- In this example we create an array with five elements. Next, we extract the three elements from the array starting at index two. By specifying a negative index value we count from the end of the array rather than the beginning. In the example below we extract the final two elements of the array. Notice the required leading space before the minus sign.

```
[me@linuxbox ~]$ echo "${arr[@]: -2:2}"
3rd 4th
```

- We can also easily create an array containing the elements of a slice.

```
[me@linuxbox ~]$ arr2=("${arr[@]:2:3}")
[me@linuxbox ~]$ echo "${arr2[@]}"
2nd 3rd 4th
```

- Here we created an array arr2 and populated it with three elements from arr.

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 thus creating key-value pairs much alike a dictionary or hash table. 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 explicitly created with the declare command using the -A option. Associative array elements are accessed in much the same way as integer indexed arrays.

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

- We can use the key-value pair characteristic of associative arrays to perform lookup tasks.
- An associative array allows us to directly access the desired data rather than having to sequentially search for it as with an indexed array.
- In the program below, we read the names of all files in the /usr/bin directory along with their respective sizes into an array and then perform lookups based on user input.

```
#!/bin/bash

# array-lookup - demonstrate lookup using associative array

declare -A cmds
# fill array with commands and file sizes
cd /usr/bin || exit 1
echo "Loading commands..."
for i in ./*; do
    cmds["$i"]=$(stat -c "%s" "$i")
done
echo "${#cmds[@]} commands loaded"

# perform lookup
while true; do
    read -r -p "Enter command (empty to quit) -> "
    [[ -z $REPLY ]] && break
    if [[ -x $REPLY ]]; then
        echo "$REPLY" "${cmds[./$REPLY]}" "bytes"
    else
        echo "No such command '$REPLY'."
  fi
done
```

- When we run the program we get the following:

```
[me@linuxbox ~]$ ./array-lookup
Loading commands...
2329 commands loaded
Enter command (empty to quit) -> ls
ls 138216 bytes
Enter command (empty to quit) -> cp
cp 141832 bytes
Enter command (empty to quit) -> mv
mv 137752 bytes
Enter command (empty to quit) -> rm
rm 59912 bytes
Enter command (empty to quit) ->
[me@linuxbox ~]$
```

Using Associative Arrays to Simulate Multiple Dimensions

- While it's true bash only directly supports single dimension arrays, it's not hard to "fake" multi-dimensional arrays by using an associative array and creating index strings that look like multi-dimensional array addresses. Here's an example:

```
#!/bin/bash

# array-multi - simulate a multi-dimensional array

declare -A multi_array

# Load array with a sequence of numbers
counter=1
for row in {1..10}; do
    for col in {1..5}; do
        address="$row, $col"
        multi_array["$address"]=$counter
        ((counter++))
    done
done

# Output array contents
for row in {1..10}; do
    for col in {1..5}; do
        address="$row, $col"
        echo -ne "${multi_array["$address"]}" "\t"
    done
    echo
done
```

- Running this program results in the following:

```
[me@linuxbox ~]$ ./array-multi
1   2    3    4    5
6   7    8    9    10
11  12   13   14   15
16  17   18   19   20
21  22   23   24   25
26  27   28   29   30
31  32   33   34   35
36  37   38   39   40
41  42   43   44   45
46  47   48   49   50
```