Bash by example, Part 2

More bash programming fundamentals

Level: Intermediate

Daniel Robbins (drobbins@gentoo.org), President and CEO, Gentoo Technologies, Inc.

01 Apr 2000

In his introductory article on bash, Daniel Robbins walked you through some of the scripting language's basic elements and reasons for using bash. In this, the second installment, Daniel picks up where he left off and looks at bash's basic constructs like conditional (if-then) statements, looping, and more.

Let's start with a brief tip on handling command-line arguments, and then look at bash's basic programming constructs.

Accepting arguments

In the sample program in the introductory article, we used the environment variable "$1", which referred to the first command-line argument. Similarly, you can use "$2", "$3", etc. to refer to the second and third arguments passed to your script. Here's an example:

```
#!/usr/bin/env bash

echo name of script is $0
echo first argument is $1
echo second argument is $2
echo seventeenth argument is $17
echo number of arguments is $#
```

The example is self explanatory except for two small details. First, "$0" will expand to the name of the script, as called from the command line, and "$#" will expand to the number of arguments passed to the script. Play around with the above script, passing different kinds of command-line arguments to get the hang of how it works.

Sometimes, it's helpful to refer to *all* command-line arguments at once. For this purpose, bash features the "$@" variable, which expands to all command-line parameters separated by spaces. We'll see an example of its use when we take a look at "for" loops, a bit later in this article.

Bash programming constructs

If you've programmed in a procedural language like C, Pascal, Python, or Perl, then you're familiar with standard programming constructs like "if" statements, "for" loops, and the like. Bash has its own versions of most of these standard constructs. In the next several sections, I will introduce several bash constructs and demonstrate the differences between these constructs and others you are already familiar with from other programming languages. If you haven't programmed much before, don't worry. I include enough information and examples so that you can follow the text.

Conditional love

If you've ever programmed any file-related code in C, you know that it requires a significant amount of effort to see if a particular file is newer than another. That's because C doesn't have any built-in syntax for performing such a comparison; instead, two stat() calls and two stat structures must be used to perform the comparison by hand. In contrast, bash has standard file comparison operators built in, so determining if "/tmp/myfile is readable" is as easy as checking to see if "$myvar is greater than 4".

The following table lists the most frequently used bash comparison operators. You'll also find an example of how to use every option correctly. The example is meant to be placed immediately after the "if". For example:

```
if [ -z "$myvar" ]
then
        echo "myvar is not defined"
fi
```

Sometimes, there are several different ways that a particular comparison can be made. For example, the following two snippets of code function identically:

```
if [ "$myvar" -eq 3 ]
then
    echo "myvar equals 3"
fi


if [ "$myvar" = "3" ]
then
    echo "myvar equals 3"
fi
```

In the above two comparisons do exactly the same thing, but the first uses arithmetic comparison operators, while the second uses string comparison operators.

String comparison caveats

Most of the time, while you can omit the use of double quotes surrounding strings and string variables, it's not a good idea. Why? Because your code will work perfectly, unless an environment variable happens to have a space or a tab in it, in which case bash will get confused. Here's an example of a fouled-up comparison:

```
if [ $myvar = "foo bar oni" ]
then
    echo "yes"
```

```
fi
```

In the above example, if myvar equals "foo", the code will work as expected and not print anything. However, if myvar equals "foo bar oni", the code will fail with the following error:

```
[: too many arguments
```

In this case, the spaces in "$myvar" (which equals "foo bar oni") end up confusing bash. After bash expands "$myvar", it ends up with the following comparison:

```
[ foo bar oni = "foo bar oni" ]
```

Because the environment variable wasn't placed inside double quotes, bash thinks that you stuffed too many arguments in-between the square brackets. You can easily eliminate this problem by surrounding the string arguments with double-quotes. Remember, if you get into the habit of surrounding all string arguments and environment variables with double-quotes, you'll eliminate many similar programming errors. Here's how the "foo bar oni" comparison *should* have been written:

```
if [ "$myvar" = "foo bar oni" ]
then
    echo "yes"
fi
```

The above code will work as expected and will not create any unpleasant surprises.

More quoting specifics
If you want your environment variables to be expanded, you must enclose them in *double quotes*, rather than single quotes. Single quotes *disable* variable (as well as history) expansion.

Looping constructs: "for"

OK, we've covered conditionals, now it's time to explore bash looping constructs. We'll start with the standard "for" loop. Here's a basic example:

```
#!/usr/bin/env bash

for x in one two three four
do
    echo number $x
done

output:
number one
number two
number three
number four
```

What exactly happened? The "for x" part of our "for" loop defined a new environment variable (also called a loop control variable) called "$x", which was successively set to the values "one", "two", "three", and "four". After each assignment, the body of the loop (the code between the "do" ... "done") was executed once. In the body, we referred to the loop control variable "$x" using standard variable expansion syntax, like any other environment variable. Also notice that "for" loops always accept some kind of word list after the "in" statement. In this case we specified four English words, but the word list can also refer to file(s) on disk or even file wildcards. Look at the following example, which demonstrates how to use standard shell wildcards:

```
#!/usr/bin/env bash

for myfile in /etc/r*
do
    if [ -d "$myfile" ]
    then
      echo "$myfile (dir)"
    else
      echo "$myfile"
    fi
done

output:

/etc/rc.d (dir)
/etc/resolv.conf
/etc/resolv.conf~
/etc/rpc
```

The above code looped over each file in /etc that began with an "r". To do this, bash first took our wildcard /etc/r* and expanded it, replacing it with the string /etc/rc.d /etc/resolv.conf /etc/resolv.conf~ /etc/rpc before executing the loop. Once inside the loop, the "-d" conditional operator was used to perform two different actions, depending on whether myfile was a directory or not. If it was, a " (dir)" was appended to the output line.

We can also use multiple wildcards and even environment variables in the word list:

```
for x in /etc/r??? /var/lo* /home/drobbins/mystuff/* /tmp/${MYPATH}/*
do
    cp $x /mnt/mydir
done
```

Bash will perform wildcard and variable expansion in all the right places, and potentially create a very long word list.

While all of our wildcard expansion examples have used *absolute* paths, you can also use relative paths, as follows:

```
for x in ../* mystuff/*
do
    echo $x is a silly file
done
```

In the above example, bash performs wildcard expansion relative to the current working directory, just like when you use relative paths on the command line. Play around with wildcard expansion a bit. You'll notice that if you use absolute paths in your wildcard, bash will expand the wildcard to a list of absolute paths. Otherwise, bash will use relative paths in the subsequent word list. If you simply refer to files in the current working directory (for example, if you type "for x in *"), the resultant list of files will not be prefixed with any path information. Remember that preceding path information can be stripped using the "basename" executable, as follows:

```
for x in /var/log/*
```

```
do
    echo `basename $x` is a file living in /var/log
done
```

Of course, it's often handy to perform loops that operate on a script's command-line arguments. Here's an example of how to use the "$@" variable, introduced at the beginning of this article:

```
#!/usr/bin/env bash

for thing in "$@"
do
    echo you typed ${thing}.
done

output:

$ allargs hello there you silly
you typed hello.
you typed there.
you typed you.
you typed silly.
```

Shell arithmetic

Before looking at a second type of looping construct, it's a good idea to become familiar with performing shell arithmetic. Yes, it's true: You can perform simple integer math using shell constructs. Simply enclose the particular arithmetic expression between a "$((" and a "))", and bash will evaluate the expression. Here are some examples:

```
$ echo $(( 100 / 3 ))
33
$ myvar="56"
$ echo $(( $myvar + 12 ))
68
$ echo $(( $myvar - $myvar ))
0
$ myvar=$(( $myvar + 1 ))
$ echo $myvar
57
```

Now that you're familiar performing mathematical operations, it's time to introduce two other bash looping constructs, "while" and "until".

More looping constructs: "while" and "until"

A "while" statement will execute as long as a particular condition is true, and has the following format:

```
while [ condition ]
do
    statements
done
```

"While" statements are typically used to loop a certain number of times, as in the following example, which will loop exactly 10 times:

```
myvar=0
while [ $myvar -ne 10 ]
do
    echo $myvar
    myvar=$(( $myvar + 1 ))
done
```

You can see the use of arithmetic expansion to eventually cause the condition to be false, and the loop to terminate.

"Until" statements provide the inverse functionality of "while" statements: They repeat as long as a particular condition is *false*. Here's an "until" loop that functions identically to the previous "while" loop:

```
myvar=0
until [ $myvar -eq 10 ]
do
    echo $myvar
    myvar=$(( $myvar + 1 ))
done
```

Case statements

Case statements are another conditional construct that comes in handy. Here's an example snippet:

```
case "${x##*.}" in
     gz)
           gzunpack ${SROOT}/${x}
           ;;
     bz2)
           bz2unpack ${SROOT}/${x}
           ;;
     *)
           echo "Archive format not recognized."
           exit
           ;;
```

```
esac
```

Above, bash first expands "${x##*.}". In the code, "$x" is the name of a file, and "${x##.*}" has the effect of stripping all text except that following the last period in the filename. Then, bash compares the resultant string against the values listed to the left of the ")"s. In this case, "${x##.*}" gets compared against "gz", then "bz2" and finally "*". If "${x##.*}" matches any of these strings or patterns, the lines immediately following the ")" are executed, up until the ";;", at which point bash continues executing lines after the terminating "esac". If no patterns or strings are matched, no lines of code are executed; however, in this particular code snippet, at least one block of code will execute, because the "*" pattern will catch everything that didn't match "gz" or "bz2".

Functions and namespaces

In bash, you can even define functions, similar to those in other procedural languages like Pascal and C. In bash, functions can even accept arguments, using a system very similar to the way scripts accept command-line arguments. Let's take a look at a sample function definition and then proceed from there:

```
tarview() {
    echo -n "Displaying contents of $1 "
    if [ ${1##*.} = tar ]
    then
        echo "(uncompressed tar)"
        tar tvf $1
    elif [ ${1##*.} = gz ]
    then
        echo "(gzip-compressed tar)"
        tar tzvf $1
    elif [ ${1##*.} = bz2 ]
    then
        echo "(bzip2-compressed tar)"
        cat $1 | bzip2 -d | tar tvf -
    fi
}
```

Above, we define a function called "tarview" that accepts one argument, a tarball of some kind. When the function is executed, it identifies what type of tarball the argument is (either uncompressed, gzip-compressed, or bzip2-compressed), prints out a one-line informative message, and then displays the contents of the tarball. This is how the above function should be called (whether from a script or from the command line, after it has been typed in, pasted in, or sourced):

Another case
The above code could have been written using a "case" statement. Can you figure out how?

```
$ tarview shorten.tar.gz
Displaying contents of shorten.tar.gz (gzip-compressed tar)
drwxr-xr-x ajr/abbot         0 1999-02-27 16:17 shorten-2.3a/
-rw-r--r-- ajr/abbot      1143 1997-09-04 04:06 shorten-2.3a/Makefile
-rw-r--r-- ajr/abbot      1199 1996-02-04 12:24 shorten-2.3a/INSTALL
-rw-r--r-- ajr/abbot       839 1996-05-29 00:19 shorten-2.3a/LICENSE
....
```

As you can see, arguments can be referenced inside the function definition by using the same mechanism used to reference command-line arguments. In addition, the "$#" macro will be expanded to contain the number of arguments. The only thing that may not work completely as expected is the variable "$0", which will either expand to the string "bash" (if you run the function from the shell, interactively) or to the name of the script the function is called from.

Use 'em interactively
Don't forget that functions, like the one above, can be placed in your ~/.bashrc or ~/.bash_profile so that they are available for use whenever you are in bash.

Namespace

Often, you'll need to create environment variables inside a function. While possible, there's a technicality you should know about. In most compiled languages (such as C), when you create a variable inside a function, it's placed in a separate local namespace. So, if you define a function in C called myfunction, and in it define a variable called "x", any global (outside the function) variable called "x" will not be affected by it, eliminating side effects.

While true in C, this isn't true in bash. In bash, whenever you create an environment variable inside a function, it's added to the *global* namespace. This means that it will overwrite any global variable outside the function, and will continue to exist even after the function exits:

```
#!/usr/bin/env bash

myvar="hello"

myfunc() {

    myvar="one two three"
    for x in $myvar
    do
        echo $x
    done
}

myfunc

echo $myvar $x
```

When this script is run, it produces the output "one two three three", showing how "$myvar" defined in the function clobbered the global variable "$myvar", and how the loop control variable "$x" continued to exist even after the function exited (and also would have clobbered any global "$x", if one were defined).

In this simple example, the bug is easy to spot and to compensate for by using alternate variable names. However, this isn't the right approach; the best way to solve this problem is to prevent the possibility of clobbering global variables in the first place, by using the "local" command. When we use "local" to create variables inside a function, they will be kept in the *local* namespace and not clobber any global variables. Here's how to implement the above code so that no global variables are overwritten:

```
#!/usr/bin/env bash

myvar="hello"

myfunc() {
    local x
    local myvar="one two three"
```

```
    for x in $myvar
    do
        echo $x
    done
}

myfunc

echo $myvar $x
```

This function will produce the output "hello" -- the global "$myvar" doesn't get overwritten, and "$x" doesn't continue to exist outside of myfunc. In the first line of the function, we create x, a local variable that is used later, while in the second example (local myvar="one two three"") we create a local myvar *and* assign it a value. The first form is handy for keeping loop control variables local, since we're not allowed to say "for local x in $myvar". This function doesn't clobber any global variables, and you are encouraged to design all your functions this way. The only time you should *not* use "local" is when you explicitly want to modify a global variable.

Wrapping it up

Now that we've covered the most essential bash functionality, it's time to look at how to develop an entire application based in bash. In my next installment, we'll do just that. See you then!

Resources

- Read "Bash by example: Part 1" on *developerWorks*.
- Read "Bash by example: Part 3" on *developerWorks*.
- Visit GNU's bash home page.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the Chief Architect of the Gentoo Project, CEO of Gentoo Technologies, Inc., the mentor for the Linux Advanced Multimedia Project (LAMP), and 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, who is expecting a child this spring. You can contact Daniel at drobbins@gentoo.org.