Bash by example, Part 1

Fundamental programming in the Bourne again shell (bash)

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

01 March 2000

By learning how to program in the bash scripting language, your day-to-day interaction with Linux will become more fun and productive, and you'll be able to build upon those standard UNIX constructs (like pipelines and redirection) that you already know and love. In this three-part series, Daniel Robbins will teach you how to program in bash by example. He'll cover the absolute basics (making this an excellent series for beginners) and bring in more advanced features as the series proceeds.

You might wonder why you ought to learn Bash programming. Well, here are a couple of compelling reasons:

You're already running it

If you check, you'll probably find that you are running bash right now. Even if you changed your default shell, bash is probably *still* running somewhere on your system, because it's the standard Linux shell and is used for a variety of purposes. Because bash is already running, any additional bash scripts that you run are inherently memory-efficient because they share memory with any already-running bash processes. Why load a 500K interpreter if you already are running something that will do the job, and do it well?

You're already using it

Not only are you already running bash, but you're actually interacting with bash on a daily basis. It's always there, so it makes sense to learn how to use it to its fullest potential. Doing so will make your bash experience more fun and productive. But why should you learn bash *programming*? Easy, because you already think in terms of running commands, CPing files, and piping and redirecting output. Shouldn't you learn a language that allows you to use and build upon these powerful time-saving constructs you already know how to use? Command shells unlock the potential of a UNIX system, and bash is *the* Linux shell. It's the high-level glue between you and the machine. Grow in your knowledge of bash, and you'll automatically increase your productivity under Linux and UNIX -- it's that simple.

Bash confusion

Learning bash the wrong way can be a very confusing process. Many newbies type "man bash" to view the bash man page, only to be confronted with a very terse and technical description of shell functionality. Others type "info bash" (to view the GNU info documentation), causing either the man page to be redisplayed, or (if they are lucky) only slightly more friendly info documentation to appear.

While this may be somewhat disappointing to novices, the standard bash documentation can't be all things to all people, and caters towards those already familiar with shell programming in general. There's definitely a lot of excellent technical information in the man page, but its helpfulness to beginners is limited.

That's where this series comes in. In it, I'll show you how to actually use bash programming constructs, so that you will be able to write your own scripts. Instead of technical descriptions, I'll provide you with explanations in plain English, so that you will know not only what something does, but when you should actually use it. By the end of this three-part series, you'll be able to write your own intricate bash scripts, and be at the level where you can comfortably use bash and supplement your knowledge by reading (and understanding!) the standard bash documentation. Let's begin.

Environment variables

Under bash and almost all other shells, the user can define environment variables, which are stored internally as ASCII strings. One of the handiest things about environment variables is that they are a standard part of the UNIX process model. This means that environment variables not only are exclusive to shell scripts, but can be used by standard compiled programs as well. When we "export" an environment variable under bash, any subsequent program that we run can read our setting, whether it is a shell script or not. A good example is the vipw command, which normally allows root to edit the system password file. By setting the EDITOR environment variable to the name of your favorite text editor, you can configure vipw to use it instead of vi, a handy thing if you are used to xemacs and really dislike vi.

The standard way to define an environment variable under bash is:

```
$ myvar='This is my environment variable!'
```

Quoting specifics

For extremely detailed information on how quotes should be used in bash, you may want to look at the "QUOTING" section in the bash man page. The existence of special character sequences that get "expanded" (replaced) with other values does complicate how strings are handled in bash. We will just cover the most often-used quoting functionality in this series.

The above command defined an environment variable called "myvar" and contains the string "This is my environment variable!". There are several things to notice above: first, there is no space on either side of the "=" sign; any space will result in an error (try it and see). The second thing to notice is that while we could have done away with the quotes if we were defining a single

word, they are necessary when the value of the environment variable is more than a single word (contains spaces or tabs).

Thirdly, while we can normally use double quotes instead of single quotes, doing so in the above example would have caused an error. Why? Because using single quotes disables a bash feature called expansion, where special characters and sequences of characters are replaced with values. For example, the "!" character is the history expansion character, which bash normally replaces with a previously-typed command. (We won't be covering history expansion in this series of articles, because it is not frequently used in bash programming. For more information on it, see the "HISTORY EXPANSION" section in the bash man page.) While this macro-like functionality can come in handy, right now we want a literal exclamation point at the end of our environment variable, rather than a macro.

Now, let's take a look at how one actually uses environment variables. Here's an example:

```
$ echo $myvar
This is my environment variable!
```

By preceding the name of our environment variable with a $, we can cause bash to replace it with the value of myvar. In bash terminology, this is called "variable expansion". But, what if we try the following:

```
$ echo foo$myvarbar
foo
```

We wanted this to echo "fooThis is my environment variable!bar", but it didn't work. What went wrong? In a nutshell, bash's variable expansion facility in got confused. It couldn't tell whether we wanted to expand the variable $m, $my, $myvar, $myvarbar, etc. How can we be more explicit and clearly tell bash what variable we are referring to? Try this:

```
$ echo foo${myvar}bar
fooThis is my environment variable!bar
```

As you can see, we can enclose the environment variable name in curly braces when it is not clearly separated from the surrounding text. While $myvar is faster to type and will work most of the time, ${myvar} can be parsed correctly in almost any situation. Other than that, they both do the same thing, and you will see both forms of variable expansion in the rest of this series. You'll want to remember to use the more explicit curly-brace form when your environment variable is not isolated from the surrounding text by whitespace (spaces or tabs).

Recall that we also mentioned that we can "export" variables. When we export an environment variable, it's automatically available in the environment of any subsequently-run script or executable. Shell scripts can "get to" the environment variable using that shell's built-in environment-variable support, while C programs can use the getenv() function call. Here's some example C code that you should type in and compile -- it'll allow us to understand environment variables from the perspective of C:

myvar.c -- a sample environment variable C program

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char *myenvvar=getenv("EDITOR");
  printf("The editor environment variable is set to %s\n",myenvvar);
}
```

Save the above source into a file called myenv.c, and then compile it by issuing the command:

```
$ gcc myenv.c -o myenv
```

Now, there will be an executable program in your directory that, when run, will print the value of the EDITOR environment variable, if any. This is what happens when I run it on my machine:

```
$ ./myenv
The editor environment variable is set to (null)
```

Hmmm... because the EDITOR environment variable was not set to anything, the C program gets a null string. Let's try setting it to a specific value:

```
$ EDITOR=xemacs
$ ./myenv
The editor environment variable is set to (null)
```

While you might have expected myenv to print the value "xemacs", it didn't quite work, because we didn't export the EDITOR environment variable. This time, we'll get it working:

```
$ export EDITOR
$ ./myenv
The editor environment variable is set to xemacs
```

So, you have seen with your very own eyes that another process (in this case our example C program) cannot see the environment variable until it is exported. Incidentally, if you want, you can define and export an environment variable using one line, as follows:

```
$ export EDITOR=xemacs
```

It works identically to the two-line version. This would be a good time to show how to erase an environment variable by using unset:

```
$ unset EDITOR
$ ./myenv
The editor environment variable is set to (null)
```

dirname and basename

Both dirname and basename do not look at any files or directories on disk; they are purely string manipulation commands.

Chopping strings overview

Chopping strings -- that is, splitting an original string into smaller, separate chunk(s) -- is one of those tasks that is performed daily by your average shell script. Many times, shell scripts need to take a fully-qualified path, and find the terminating file or directory. While it's possible (and fun!) to code this in bash, the standard basename UNIX executable performs this extremely well:

```
$ basename /usr/local/share/doc/foo/foo.txt
foo.txt
$ basename /usr/home/drobbins
drobbins
```

Basename is quite a handy tool for chopping up strings. It's companion, called dirname, returns the "other" part of the path that basename throws away:

```
$ dirname /usr/local/share/doc/foo/foo.txt
/usr/local/share/doc/foo
$ dirname /usr/home/drobbins/
/usr/home
```

Command substitution

One very handy thing to know is how to create an environment variable that contains the result of an executable command. This is very easy to do:

```
$ MYDIR=`dirname /usr/local/share/doc/foo/foo.txt`
$ echo $MYDIR
/usr/local/share/doc/foo
```

What we did above is called "command substitution". Several things are worth noticing in this example. On the first line, we simply enclosed the command we wanted to execute in *back quotes*. Those are not standard single quotes, but instead come from the keyboard key that normally sits above the Tab key. We can do exactly the same thing with bash's alternate command substitution syntax:

```
$ MYDIR=$(dirname /usr/local/share/doc/foo/foo.txt)
$ echo $MYDIR
/usr/local/share/doc/foo
```

As you can see, bash provides multiple ways to perform exactly the same thing. Using command substitution, we can place any command or pipeline of commands in between ` ` or $( ) and assign it to an environment variable. Handy stuff! Here's an example of how to use a pipeline with command substitution:

```
MYFILES=$(ls /etc | grep pa)
bash-2.03$ echo $MYFILES
pam.d passwd
```

Chopping strings like a pro

While basename and dirname are great tools, there are times where we may need to perform more advanced string "chopping" operations than just standard pathname manipulations. When

we need more punch, we can take advantage of bash's advanced built-in variable expansion functionality. We've already used the standard kind of variable expansion, which looks like this: ${MYVAR}. But bash can also perform some handy string chopping on its own. Take a look at these examples:

```
$ MYVAR=foodforthought.jpg
$ echo ${MYVAR##*fo}
rthought.jpg
$ echo ${MYVAR#*fo}
odforthought.jpg
```

In the first example, we typed ${MYVAR##*fo}. What exactly does this mean? Basically, inside the ${ }, we typed the name of the environment variable, two ##s, and a wildcard ("*fo"). Then, bash took MYVAR, found the *longest* substring from the beginning of the string "foodforthought.jpg" that matched the wildcard "*fo", and chopped it off the beginning of the string. That's a bit hard to grasp at first, so to get a feel for how this special "##" option works, let's step through how bash completed this expansion. First, it began searching for substrings at the beginning of "foodforthought.jpg" that matched the "*fo" wildcard. Here are the substrings that it checked:

```
f
fo              MATCHES *fo
foo
food
foodf
foodfo          MATCHES *fo
foodfor
foodfort
foodforth
foodfortho
foodforthou
foodforthoug
foodforthought
foodforthought.j
foodforthought.jp
foodforthought.jpg
```

After searching the string for matches, you can see that bash found two. It selects the longest match, removes it from the beginning of the original string, and returns the result.

The second form of variable expansion shown above appears identical to the first, except it uses only one "#" -- and bash performs an *almost* identical process. It checks the same set of substrings as our first example did, except that bash removes the *shortest* match from our original string, and returns the result. So, as soon as it checks the "fo" substring, it removes "fo" from our string and returns "odforthought.jpg".

This may seem extremely cryptic, so I'll show you an easy way to remember this functionality. When searching for the longest match, use ## (because ## is longer than #). When searching for the shortest match, use #. See, not that hard to remember at all! Wait, how do you remember that we are supposed to use the '#' character to remove from the *beginning* of a string? Simple! You will notice that on a US keyboard, shift-4 is "$", which is the bash variable expansion character. On the keyboard, immediately to the left of "$" is "#". So, you can see that "#" is "at the beginning" of "$", and thus (according to our mnemonic), "#" removes characters from the beginning of the

string. You may wonder how we remove characters from the end of the string. If you guessed that we use the character immediately *to the right* of "$" on the US keyboard ("%"), you're right! Here are some quick examples of how to chop off trailing portions of strings:

```
$ MYFOO="chickensoup.tar.gz"
$ echo ${MYFOO%%.*}
chickensoup
$ echo ${MYFOO%.*}
chickensoup.tar
```

As you can see, the % and %% variable expansion options work identically to # and ##, except they remove the matching wildcard from the end of the string. Note that you don't have to use the "*" character if you wish to remove a specific substring from the end:

```
MYFOOD="chickensoup"
$ echo ${MYFOOD%%soup}
chicken
```

In this example, it doesn't matter whether we use "%%" or "%", since only one match is possible. And remember, if you forget whether to use "#" or "%", look at the 3, 4, and 5 keys on your keyboard and figure it out.

We can use another form of variable expansion to select a specific substring, based on a specific character offset and length. Try typing in the following lines under bash:

```
$ EXCLAIM=cowabunga
$ echo ${EXCLAIM:0:3}
cow
$ echo ${EXCLAIM:3:7}
abunga
```

This form of string chopping can come in quite handy; simply specify the character to start from and the length of the substring, all separated by colons.

Applying string chopping

Now that we've learned all about chopping strings, let's write a simple little shell script. Our script will accept a single file as an argument, and will print out whether it appears to be a tarball. To determine if it is a tarball, it will look for the pattern ".tar" at the end of the file. Here it is:

mytar.sh -- a sample script

```
#!/bin/bash

if [ "${1##*.}" = "tar" ]
then
        echo This appears to be a tarball.
else
        echo At first glance, this does not appear to be a tarball.
fi
```

To run this script, enter it into a file called mytar.sh, and type "chmod 755 mytar.sh" to make it executable. Then, give it a try on a tarball, as follows:

```
$ ./mytar.sh thisfile.tar
This appears to be a tarball.
$ ./mytar.sh thatfile.gz
At first glance, this does not appear to be a tarball.
```

OK, it works, but it's not very functional. Before we make it more useful, let's take a look at the "if" statement used above. In it, we have a boolean expression. In bash, the "=" comparison operator checks for string equality. In bash, all boolean expressions are enclosed in square brackets. But what does the boolean expression actually test for? Let's take a look at the left side. According to what we've learned about string chopping, "${1##*.}" will remove the longest match of "*." from the beginning of the string contained in the environment variable "1", returning the result. This will cause everything after the last "." in the file to be returned. Obviously, if the file ends in ".tar", we will get "tar" as a result, and the condition will be true.

You may be wondering what the "1" environment variable is in the first place. Very simple -- $1 is the first command-line argument to the script, $2 is the second, etc. OK, now that we've reviewed the function, we can take our first look at "if" statements.

If statements

Like most languages, bash has its own form of conditional. When using them, stick to the format above; that is, keep the "if" and the "then" on separate lines, and keep the "else" and the terminating and required "fi" in horizontal alignment with them. This makes the code easier to read and debug. In addition to the "if,else" form, there are several other forms of "if" statements:

```
if [ condition ]
then
        action
fi
```

This one performs an action only if `condition` is true, otherwise it performs no action and continues executing any lines following the "fi".

```
if [ condition ]
then
        action
elif [ condition2 ]
then
        action2
.
.
.
elif [ condition3 ]
then

else
        actionx
fi
```

The above "elif" form will consecutively test each condition and execute the action corresponding to the first *true* condition. If none of the conditions are true, it will execute the "else" action, if one is present, and then continue executing lines following the entire "if,elif,else" statement.

Next time

Now that we've covered the most basic bash functionality, it's time to pick up the pace and get ready to write some real scripts. In the next article, I'll cover looping constructs, functions, namespace, and other essential topics. Then, we'll be ready to write some more complicated scripts. In the third article, we'll focus almost exclusively on very complex scripts and functions, as well as several bash script design options. See you then!

Resources

- Read "Bash by example: Part 2" on *developerWorks*.
- Read "Bash by example: Part 3" on *developerWorks*.
- Visit GNU's bash home page.

About the author

Daniel Robbins

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.