Bash by example, Part 1
Fundamental programming in the Bourne again shell (bash)
Daniel Robbins
President and CEO, Gentoo Technologies, Inc.
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.

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, Ping 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 yourself? 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 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 can run 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:

```bash
$ myvar="This is my environment variable!"
```

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 written the above 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, 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 examination 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:

```bash
$ 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:

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

We wanted this to echo "fooThis is my environment variable!bar", but it didn't work. What went wrong? In a nutshell, bash's terminology, this is called "variable expansion". But, what if we try the following:

```bash
$ echo "foo$myvarbar"
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" the environment variable using $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:

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

int main( void )
{
    char *myenvvar = getenv("EDITOR");
    printf( "%s\n", myenvvar );
}
```

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

```
$ gcc myenv.c
```

The program should output the name of your default text editor. To change the value of the environment variable, we can do:

```bash
$ export EDITOR=xemacs
```

And our environment variable will now point to xemacs. Note that "export" is a typical environment variable setting, and if you use other shells, this may not work as expected. 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.
What we did above is called "command substitution". Several things are worth noticing in this example. On the first line, we:

```bash
$ echo $MYDIR/usr/local/share/doc/foo
```

If you guessed that we use the character immediately

"#" 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

This may seem extremely cryptic, so I'll show you an easy way to remember this functionality. When searching for the longest

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".

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:

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

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:

```
fo              MATCHES *fo
foo              MATCHES *fo
food              MATCHES *fo
foodfor              MATCHES *fo
foodforth              MATCHES *fo
foodforthought              MATCHES *fo
foodforthought.jpg              MATCHES *fo
```

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
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
    ...
  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,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

- Visit GNU's bash home page
- Check out the bash online reference manual

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the Chief Architect of the Genoto Project, CEO of Genoto 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. He can be reached at drobbins@gentoo.org.

What do you think of this article?

- Killer!
- Good stuff
- So-so; not bad
- Needs work
- Lame!

Comments?

Submit feedback
