

Overview

In this tutorial, learn to customize your Linux bash shell environment to meet user needs. Learn to:

- Modify global and user profiles
- Set environment variables when you log in or spawn a new shell
- Create bash functions for frequently used sequences of commands
- Maintain skeleton directories for new user accounts
- Set your command search path

Linux shells

You use a Linux shell program to interact with your system by typing commands (the *input stream*) at a terminal and seeing output (the *output stream*) and error messages (the *error stream*) on the same terminal. Sometimes you need to run commands before the system has booted far enough to allow terminal connections, and sometimes you need to run commands periodically, whether you are logged in. A shell can do these tasks for you, too. The standard input and output streams do not have to come from, or be directed to, a real user at a terminal. In this tutorial, you learn more about shells and customizing the user's environment. In particular, you learn about the bash (Bourne again) shell — an enhancement of the original Bourne shell — and about changes to bash that make it more compliant with the Portable Operating System Interface (POSIX). I point out features of some other shells along the way.

This tutorial helps you prepare for Objective 105.1 in Topic 105 of the Linux Server Professional (LPIC-1) exam 102. The objective has a weight of 4.

Prerequisites

To get the most from the tutorials in this series, you need a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial. You should be familiar with GNU and UNIX commands. This tutorial builds on material covered in the tutorials for Topic 103 of exam 101. Sometimes different versions of a program format output

About this series

This series of tutorials helps you learn Linux system administration tasks. You can also use the material in these tutorials to prepare for the [Linux Professional Institute's LPIC-1: Linux Server Professional Certification exams](#). See "[Learn Linux, 101: A roadmap for LPIC-1](#)" for a

differently, so your results might not always look exactly like the listings and figures shown here.

The examples in this tutorial are largely distribution independent. Unless otherwise noted, the examples use Fedora 22, with a 4.2.3 kernel.

description of and link to each tutorial in this series. The roadmap is in progress and reflects the version 4.0 objectives of the LPIC-1 exams as updated April 15th, 2015. As tutorials are completed, they will be added to the roadmap.

Shells and environments

A shell provides a layer between you and the intricacies of an operating system. With Linux (and UNIX) shells, you can build complex operations by combining basic functions. Using programming constructs, you can then build functions for direct execution in the shell or save functions as *shell scripts*. A shell script is a sequence of commands that is stored in a file that the shell can run as needed.

POSIX is a series of IEEE standards collectively referred to as IEEE 1003. (The first POSIX standard was IEEE Standard 1003.1-1988, released in 1988.) Bash implements many POSIX features and can run in a mode that complies even more closely with POSIX. Other well-known shells include the Korn shell (ksh), the C shell (csh) and its derivative tcsh, and the Almquist shell (ash) and its Debian derivative (dash). I recommend that you know at least enough about these other shells to recognize when a script uses features from any of them.

Recall from the tutorial “[Learn Linux, 101: The Linux command line](#)” that when you are running in a bash shell, you have a *shell environment*. The environment is a set of name-value pairs that define the form of your prompt, your home directory, your working directory, the name of your shell, files that you have opened, functions that you have defined, and more. The environment is available to every shell process. When a shell starts, it assigns values from the environment to *shell variables*. You can use shells, including bash, to create and modify additional shell variables. You can then export such variables to be part of the environment that is inherited by child processes that you spawn from your current shell.

A long time ago

Early interaction with computers used a batch-processing model with input provided on a deck of punched cards and output printed on a line printer. Later came the ASCII terminal, which programmers used to type characters into a typewriter-like display terminal. Thus, the shell was born. Early display terminals usually had about 25 lines of 80 characters, a far cry from the capabilities of today’s GUIs, but still a big improvement over punched cards.

Shell variables have a name (or identifier). Table 1 shows a few common bash environment variables that are usually set for you automatically.

Table 1. Common bash environment variables

Name	Purpose
USER	The name of the logged-in user
UID	The numeric user ID of the logged-in user
HOME	The user’s home directory
PWD	The current working directory
SHELL	The name of the shell
PPID	The parent process PID — the process ID of the process that started this process

In addition to variables, the shell has some special parameters that are set but cannot be modified. Table 2 shows some examples.

Table 2. Common bash parameters

Name	Purpose
\$	The process ID (or PID of the running bash shell [or other] process)
?	The exit code of the last command
0	The name of the shell or shell script

Using variables

You use the value of a variable by prefixing its name with a \$, as shown in Listing 1.

Listing 1. Using variable values

```
[ian@atticf22 ~]$ echo $UID
1000
[ian@atticf22 ~]$ echo $HOME
/home/ian
```



Show less ▾

Setting variable values and making them available

You create or set a shell variable in the bash shell by typing a name followed immediately by an equals sign (=) with no intervening space. Variable names are words consisting only of alphanumeric characters and underscores. The names begin with an alphabetic character or an underscore. Variables names are case sensitive, so `var1` and `VAR1` are different variables. Variable names, particularly for exported variables, are often uppercase, like the examples in Table 1, but this is a convention and not a requirement. Some variables, such as `$$` and `$?`, are really shell parameters rather than variables — they can only be referenced; you cannot assign a value to them.

Shell variables are visible only to the process in which you create them unless you *export* them so that child processes can see and use them. A child process cannot export a variable to a parent process. Use the `export` command to export variables. In the bash shell, you can assign and export in one step, but not all shells support this capability.

A good way to explore these concepts is to create child processes by using another shell. You can use the `ps` command to help you keep track of where you are and what's running. Listing 2 shows some examples with inline comments to help you.

Listing 2. Setting shell variables and exporting them to the environment

```
[ian@atticf22 ~]$ #Use the ps command to list current PID, parent PID and running c
[ian@atticf22 ~]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID CMD
12761  9169 bash
[ian@atticf22 ~]$ #Start a child bash process
[ian@atticf22 ~]$ bash
[ian@atticf22 ~]$ #Assign two variables
[ian@atticf22 ~]$ VAR1=v1
[ian@atticf22 ~]$ VAR2="Variable 2"
[ian@atticf22 ~]$ #Export examples
[ian@atticf22 ~]$ export VAR2
[ian@atticf22 ~]$ export VAR3="Assigned and exported in one step"
```



```

[ian@atticf22 ~]$ #Use the $ character to reference the variables
[ian@atticf22 ~]$ echo $VAR1 '/' $VAR2 '/' $VAR3
v1 / Variable 2 / Assigned and exported in one step
[ian@atticf22 ~]$ #What is the value of the SHELL variable?
[ian@atticf22 ~]$ echo $SHELL
/bin/bash
[ian@atticf22 ~]$ #Now start ksh child and export VAR4
[ian@atticf22 ~]$ ksh
$ ps -p $$ -o "pid ppid cmd"
    PID  PPID  CMD
26212 22923 ksh
$ export VAR4=var4
$ #See what is visible
$ echo $VAR1 '/' $VAR2 '/' $VAR3 '/' $VAR4 '/' $SHELL
/ Variable 2 / Assigned and exported in one step / var4 / /bin/bash
$ #No VAR1 and shell is /bin/bash - is that right?
$ exit
[ian@atticf22 ~]$ ps -p $$ -o "pid ppid cmd"
    PID  PPID  CMD
22923 12761 bash
[ian@atticf22 ~]$ #See what is visible
[ian@atticf22 ~]$ echo $VAR1 '/' $VAR2 '/' $VAR3 '/' $VAR4 '/' $SHELL
v1 / Variable 2 / Assigned and exported in one step / / /bin/bash
[ian@atticf22 ~]$ #No VAR4 - our child cannot export back to us
[ian@atticf22 ~]$ exit
exit
[ian@atticf22 ~]$ ps -p $$ -o "pid ppid cmd"
    PID  PPID  CMD
12761  9169 bash
[ian@atticf22 ~]$ #See what is visible
[ian@atticf22 ~]$ echo $VAR1 '/' $VAR2 '/' $VAR3 '/' $VAR4 '/' $SHELL
/ / / / /bin/bash
[ian@atticf22 ~]$ #None of VAR1 through VAR4 is exported back to parent

```



Show less ▾

Did you notice in Listing 2 that ksh did not set the SHELL variable? This variable is usually set when you log in or when you use the su command to switch to another user with options that create a *login shell*. You learn more about login shells later in this tutorial.

Use the echo command to see what's in some of the common bash variables and parameters from Table 1 and Table 2, as illustrated in Listing 3.

Listing 3. Common environment and shell variables



```

[ian@atticf22 ~]$ echo $USER $UID
ian 1000
[ian@atticf22 ~]$ echo $SHELL $HOME $PWD
/bin/bash /home/ian /home/ian
[ian@atticf22 ~]$ (exit 0);echo $?;(exit 4);echo $?
0
4

[ian@atticf22 ~]$ echo $0
bash


```

```
[ian@atticf22 ~]$ echo $$ $PPID
12761 9169
[ian@atticf22 ~]$ #see what my process and its parent are running
[ian@atticf22 ~]$ ns -n $$.$PPID -o "pid pid cmd"
```

Show more ▾

In the bash shell, you can also set environment values for the duration of a single command by prefixing the command with name=value pairs, as shown in Listing 4.

Listing 4. Setting bash environment values for a single command

```
[ian@atticf22 ~]$ echo "$VAR5 / $VAR6" / [ian@atticf22 ~]$ VAR5=5 VAR6="some val" 
```

Show less ▾

readonly and other variable attributes

I mentioned that some shell parameters cannot be modified. You can also constrain variables to be readonly, integer, or string, among other possibilities. You can use the declare command to set variable attributes. Use the -p option to display variables with various attributes. To find out more about the declare command, try running:

```
info bash "Shell Builtin Commands" "Bash Builtins" --index-search declare
```

or

```
help declare
```

Listing 5 shows a few examples.

Listing 5. Variable attributes

```
[ian@atticf22 ~]$ declare -r rov1="this is readonly"
[ian@atticf22 ~]$ rov="Who says it's read only?"
[ian@atticf22 ~]$ readonly rov2="another constant value"
[ian@atticf22 ~]$ rov2=3
bash: rov2: readonly variable
[ian@atticf22 ~]$ UID=99
bash: UID: readonly variable
[ian@atticf22 ~]$ declare -pr
declare -r BASHOPTS="checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob
force_ignore:histappend:interactive_comments:progcomp:promptvars:sourcepath"
```



```
declare -ir BASHPID
declare -r BASH_COMPLETION_COMPAT_DIR="/etc/bash_completion.d"
declare -ar BASH_VERSINFO='([0]="4" [1]="3" [2]="42" [3]="1" [4]="release" [5]=
"x86_64-redhat-linux-gnu")'
declare -ir EUID="1000"
declare -ir PPID="12761"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:interactive-comm
declare -ir UID="1000"
declare -r rovl="this is readonly"
declare -r rovr="another constant value"
[ian@atticf22 ~]$ help declare
declare: declare [-aAfFgilnrtux] [-p] [name[=value] ...]
    Set variable values and attributes.
```

Declare variables and give them attributes. If no NAMES are given, display the attributes and values of all variables.

Options:

- f restrict action or display to function names and definitions
- F restrict display to function names only (plus line number and source file when debugging)
- g create global variables when used in a shell function; otherwise ignored
- p display the attributes and value of each NAME

Options which set attributes:

- a to make NAMES indexed arrays (if supported)
- A to make NAMES associative arrays (if supported)
- i to make NAMES have the 'integer' attribute
- l to convert NAMES to lower case on assignment
- n make NAME a reference to the variable named by its value
- r to make NAMES readonly
- t to make NAMES have the 'trace' attribute
- u to convert NAMES to upper case on assignment
- x to make NAMES export

Using '+' instead of '-' turns off the given attribute.

Variables with the integer attribute have arithmetic evaluation (see the `let` command) performed when the variable is assigned a value.

When used in a function, `declare` makes NAMES local, as with the `local` command. The `-g` option suppresses this behavior.

Exit Status:

Returns success unless an invalid option is supplied or a variable assignment error occurs.




Show less

Unsetting variables

Use the `unset` command to remove a variable from the bash shell. Use the `-v` option to ensure that you are removing a variable definition. Functions (which I cover later in this tutorial) can have the same names as variables, so use `-f` if you want to remove a function definition.

Without either `-f` or `-v`, the `bash unset` command removes a variable definition if it exists; otherwise, it removes a function definition if one exists. Listing 6 shows some examples.

Listing 6. Unsetting bash variables



```
[ian@atticf22 ~]$ bash
[ian@atticf22 ~]$ VAR1=v1
[ian@atticf22 ~]$ declare -i VAR2
[ian@atticf22 ~]$ VAR2=3+4
[ian@atticf22 ~]$ echo $VAR1 $VAR2
v1 7
[ian@atticf22 ~]$ unset VAR1
[ian@atticf22 ~]$ echo $VAR1 $VAR2
7
[ian@atticf22 ~]$ unset -v VAR2
[ian@atticf22 ~]$ echo $VAR1 $VAR2


[ian@atticf22 ~]$ exit
exit
```

Show less▼

Note that if a variable is defined as integer, an assignment to it is evaluated as an arithmetic expression.

By default, `bash` treats unset variables as if they had an empty value. So why unset a variable rather than assign it an empty value? With `bash` and many other shells, you can generate an error if you reference an undefined variable. Use the `set -u` command to generate an error for undefined variables, and `set +u` to disable the warning. Listing 7 illustrates these points.

Listing 7. Errors and unset variables



```
[ian@atticf22 ~]$ bash [ian@atticf22 ~]$ set -u [ian@atticf22 ~]$ VAR1=var1 [iar
```


Show more▼

As you see, it is not an error to unset a variable that does not exist, even when `set -u` has been specified.

Environments, variables, and the C shell

The `cs`h and `tc`sh shells handle environments and variables slightly differently from `bash`. You use the `set` command to set variables in your shell, and the `setenv` command to set and export variables. The syntax differs slightly from that of the `bash` `export` command, as illustrated in Listing 8. Note the equals (=) sign when `set` is used.

Listing 8. Setting variables and environment values in the `C` and `tc`sh shells




```
[ian@atticf22 ~]$ tcsh
[ian@atticf22 ~]$ setenv E1 "Env variable 1"
[ian@atticf22 ~]$ set V2="variable 2"
[ian@atticf22 ~]$ echo "$E1 / $V2"
Env variable 1 / variable 2
[ian@atticf22 ~]$ tcsh
[ian@atticf22 ~]$ echo $E1
Env variable 1
[ian@atticf22 ~]$ echo $V2
V2: Undefined variable.
[ian@atticf22 ~]$ echo "$?E1 / $?V2"
1 / 0
[ian@atticf22 ~]$ exit
```

Show more ▾

Note that the second `tc`sh shell did not inherit the `V2` variable. Trying to reference `V2` produces an error, as happens in `bash` when `set -u` is set. In `cs`h and `tc`sh, you can check whether `NAME` is set by using the `$?NAME` construct, which returns 1 if `NAME` is set and 0 otherwise.

Another difference in environment handling is that `cs`h and `tc`sh maintain separate namespaces for variables and environment values. If the same name appears in both places, the variable definition takes precedence. As with `bash`, the `unset` command unsets a variable. Use the `unsetenv` command to unset an environment value. Listing 9 shows these commands.

Listing 9. Unsetting `cs`h and `tc`sh variables and environment values



```
[ian@atticf22 ~]$ echo "$E1 / $V2"
Env variable 1 / variable 2
[ian@atticf22 ~]$ set E1="I'm now a regular variable"
[ian@atticf22 ~]$ echo $E1
I'm now a regular variable
[ian@atticf22 ~]$ unset E1
[ian@atticf22 ~]$ echo $E1
Env variable 1
[ian@atticf22 ~]$ unsetenv E1
[ian@atticf22 ~]$ echo $E1
E1: Undefined variable.
```

Show more ▾

