
LPI exam 102 prep, Topic 109: Shells, scripting, programming, and compiling

Junior Level Administration (LPIC-1) topic 109

Skill Level: Intermediate

[Ian Shields \(ishields@us.ibm.com\)](mailto:ishields@us.ibm.com)
Senior Programmer
IBM

30 Jan 2007

In this tutorial, Ian Shields continues preparing you to take the Linux Professional Institute® Junior Level Administration (LPIC-1) Exam 102. In this fifth in a [series of nine tutorials](#), Ian introduces you to the Bash shell, and scripts and programming in the Bash shell. By the end of this tutorial, you will know how to customize your shell environment, use shell programming structures to create functions and scripts, set and unset environment variables, and use the various login scripts.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The [Linux Professional Institute](#) (LPI) certifies Linux system administrators at two levels: *junior level* (also called "certification level 1") and *intermediate level* (also called "certification level 2"). To attain certification level 1, you must pass exams 101 and 102; to attain certification level 2, you must pass exams 201 and 202.

developerWorks offers tutorials to help you prepare for each of the four exams. Each exam covers several topics, and each topic has a corresponding self-study tutorial on developerWorks. For LPI exam 102, the nine topics and corresponding developerWorks tutorials are:

Table 1. LPI exam 102: Tutorials and topics

LPI exam 102 topic	developerWorks tutorial	Tutorial summary
Topic 105	LPI exam 102 prep: Kernel	Learn how to install and maintain Linux kernels and kernel modules.
Topic 106	LPI exam 102 prep: Boot, initialization, shutdown, and runlevels	Learn how to boot a system, set kernel parameters, and shut down or reboot a system.
Topic 107	LPI exam 102 prep: Printing	Learn how to manage printers, print queues and user print jobs on a Linux system.
Topic 108	LPI exam 102 prep: Documentation	Learn how to use and manage local documentation, find documentation on the Internet and use automated logon messages to notify users of system events.
Topic 109	LPI exam 102 prep: Shells, scripting, programming, and compiling	(This tutorial.) Learn how to customize shell environments to meet user needs, write Bash functions for frequently used sequences of commands, write simple new scripts, using shell syntax for looping and testing, and customize existing scripts. See the detailed objectives below.
Topic 111	LPI exam 102 prep: Administrative tasks	Coming soon.
Topic 112	LPI exam 102 prep: Networking fundamentals	Coming soon.
Topic 113	LPI exam 102 prep: Networking services	Coming soon.
Topic 114	LPI exam 102 prep: Security	Coming soon.

To pass exams 101 and 102 (and attain certification level 1), you should be able to:

- Work at the Linux command line
- Perform easy maintenance tasks: help out users, add users to a larger system, back up and restore, and shut down and reboot
- Install and configure a workstation (including X) and connect it to a LAN, or connect a stand-alone PC via modem to the Internet

To continue preparing for certification level 1, see the [developerWorks tutorials for LPI exams 101 and 102](#), as well as the [entire set of developerWorks LPI tutorials](#).

The Linux Professional Institute does not endorse any third-party exam preparation

material or techniques in particular. For details, please contact info@lpi.org.

About this tutorial

Welcome to "Shells, scripting, programming, and compiling," the fifth of nine tutorials designed to prepare you for LPI exam 102. In this tutorial, you learn how to use the Bash shell, how to use shell programming structures to create functions and scripts, how to customize your shell environment, how to set and unset environment variables, and how to use the various login scripts.

The title for this tutorial duplicates the corresponding topic in the LPI 102 exam, and therefore includes "programming and compiling," but the LPI objectives limit "programming" to that required for writing shell functions and scripts. And no objectives for compiling programs are included in the topic.

This tutorial is organized according to the LPI objectives for this topic. Very roughly, expect more questions on the exam for objectives with higher weight.

Table 2. Shells, scripting, programming, and compiling: Exam objectives covered in this tutorial

LPI exam objective	Objective weight	Objective summary
1.109.1 Customize and use the shell environment	Weight 5	Customize shell environments to meet user needs. Set environment variables (at login or when spawning a new shell). Write Bash functions for frequently used sequences of commands.
1.109.2 Customize or write simple scripts	Weight 3	Write simple Bash scripts and customize existing ones.

Prerequisites

To get the most from this tutorial, you should have a basic knowledge of Linux and a working Linux system on which to practice the commands covered in this tutorial.

This tutorial builds on content covered in previous tutorials in this LPI series, so you may want to first review the [tutorials for exam 101](#). In particular, you should be thoroughly familiar with the material from the "[LPI exam 101 prep \(topic 103\): GNU and UNIX commands](#)" tutorial, as many of the building blocks for this tutorial were covered in that tutorial, especially the section "Using the command line."

Different versions of a program may format output differently, so your results may not look exactly like the listings and figures in this tutorial.

Section 2. Shell customization

This section covers material for topic 1.109.1 for the Junior Level Administration (LPIC-1) exam 102. The topic has a weight of 5.

In this section, learn how to:

- Set and unset environment variables
- Use profiles to set environment variables at login or when spawning a new shell
- Write shell functions for frequently used sequences of commands
- Use command lists

Shells and environments

Before the advent of graphical interfaces, programmers used a typewriter terminal or an ASCII display terminal to connect to a UNIX® system. A typewriter terminal allowed them to type commands, and the output was usually printed on continuous paper. Most ASCII display terminals had 80 characters per line and about 25 lines on the screen, although both larger and smaller terminals existed. Programmers typed a command and pressed **Enter**, and the system interpreted and then executed the command.

While this may seem somewhat primitive today in an era of drag-and-drop graphical interfaces, it was a huge step forward from writing a program, punching cards, compiling the card deck, and running the program. With the advent of editors, programmers could even create programs as card images and compile them in a terminal session.

The stream of characters typed at a terminal provided a *standard input* stream to the shell, and the stream of characters that the shell returned on either paper or display represented the *standard output*.

The program that accepts the commands and executes them is called a *shell*. It provides a layer between you and the intricacies of an operating system. UNIX and Linux shells are extremely powerful in that you can build quite 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* so that you can reuse them over and over.

Sometimes you need to execute commands before the system has booted far enough to allow terminal connections, and sometimes you need to execute commands periodically, whether or not you are logged on. A shell can do this for you, too. The standard input and output do not have to come from or be directed to a

real user at a terminal.

In this section, you learn more about shells. In particular, you learn about the *bash* or *Bourne again* shell, which is an enhancement of the original Bourne shell, along with some features from other shells and some changes from the Bourne shell to make it more POSIX compliant.

POSIX is the ***P*ortable *O*perating *S*ystem *I*nterface for *u*ni*X***, which is a series of IEEE standards collectively referred to as IEEE 1003. The first of these was IEEE Standard 1003.1-1988, released in 1988. Other well known shells include the *Korn* shell (ksh), the *C* shell (csh) and its derivative tcsh, the *Almquist* shell (ash) and its Debian derivative (dash). You need to know something about many of these shells, if only to recognize when a particular script requires features from one of them.

Many aspects of your interaction with a computer will be the same from one session to another. Recall from the tutorial ["LPI exam 101 prep \(topic 103\): GNU and UNIX commands"](#) that when you are running in a Bash shell, you have a shell *environment*, which defines such things as 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 so on. The environment is made available to every shell process. Shells, including bash, allow you to create and modify *shell variables*, which you may *export* to your environment for use by other processes running in the shell or by other shells that you may spawn from the current shell.

Both environment variables and shell variables have a *name*. You reference the value of a variable by prefixing its name with '\$'. Some of the common bash environment variables that are set for you are shown in Table 3.

Table 3. Common bash environment variables	
Name	Function
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
\$	The process id (or <i>PID</i> of the running Bash shell (or other) process)
PPID	The process id of the process that started this process (that is, the id of the parent process)
?	The exit code of the last command

Setting variables

In the Bash shell, you create or *set* a shell variable by typing a name followed immediately by an equal sign (=). Variable names (or *identifiers*) are words consisting only of alphanumeric characters and underscores, that begin with an alphabetic character or an underscore. Variables are case sensitive, so var1 and VAR1 are different variables. By convention, variables, particularly exported variables, are upper case, but this is not a requirement. Technically, \$\$ and \$? are shell *parameters* rather than variables. They may only be referenced; you cannot assign a value to them.

When you create a shell variable, you will often want to *export* it to the environment so it will be available to other processes that you start from this shell. Variables that you export are **not** available to a parent shell. You use the `export` command to export a variable name. As a shortcut in bash, you can assign and export in one step.

To illustrate assignment and exporting, let's run the bash command while in the Bash shell and then run the Korn shell (ksh) from the new Bash shell. We will use the `ps` command to display information about the command that is running.

Listing 1. Setting and exporting shell variables

```
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID  CMD
30576  30575  -bash
[ian@echidna ian]$ bash
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID  CMD
16353  30576  bash
[ian@echidna ian]$ VAR1=var1
[ian@echidna ian]$ VAR2=var2
[ian@echidna ian]$ export VAR2
[ian@echidna ian]$ export VAR3=var3
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3
var1 var2 var3
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ian]$ ksh
$ ps -p $$ -o "pid ppid cmd"
  PID  PPID  CMD
16448  16353  ksh
$ export VAR4=var4
$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var2 var3 var4 /bin/bash
$ exit
$ [ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
var1 var2 var3 /bin/bash
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID  CMD
16353  30576  bash
[ian@echidna ian]$ exit
[ian@echidna ian]$ ps -p $$ -o "pid ppid cmd"
  PID  PPID  CMD
30576  30575  -bash
[ian@echidna ian]$ echo $VAR1 $VAR2 $VAR3 $VAR4 $SHELL
/bin/bash
```

Notes:

1. At the start of this sequence, the Bash shell had PID 30576.

2. The second Bash shell has PID 16353, and its parent is PID 30576, the original Bash shell.
3. We created VAR1, VAR2, and VAR3 in the second Bash shell, but only exported VAR2 and VAR3.
4. In the Korn shell, we created VAR4. The `echo` command displayed values only for VAR2, VAR3, and VAR4, confirming that VAR1 was not exported. Were you surprised to see that the value of the SHELL variable had not changed, even though the prompt had changed? You cannot always rely on SHELL to tell you what shell you are running under, but the `ps` command does tell you the actual command. Note that `ps` puts a hyphen (-) in front of the first Bash shell to indicate that this is the *login shell*.
5. Back in the second Bash shell, we can see VAR1, VAR2, and VAR3.
6. And finally, when we return to the original shell, none of our new variables still exist.

Listing 2 shows what you might see in some of these common bash variables.

Listing 2. Environment and shell variables

```
[ian@echidna ian]$ echo $USER $UID
ian 500
[ian@echidna ian]$ echo $SHELL $HOME $PWD
/bin/bash /home/ian /home/ian
[ian@echidna ian]$ (exit 0);echo $?;(exit 4);echo $?
0
4
[ian@echidna ian]$ echo $$ $PPID
30576 30575
```

Environments and the C shell

In shells such as the C and tcsh shells, 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 `export` command as illustrated in Listing 3. Note the equals (=) sign when using `set`.

Listing 3. Setting environment variables in the C shell

```
ian@attic4:~$ echo $VAR1 $VAR2

ian@attic4:~$ csh
% set VAR1=var1
% setenv VAR2 var2
% echo $VAR1 $VAR2
var1 var2
% bash
ian@attic4:~$ echo $VAR1 $VAR2
```

```
var2
```

Unsetting variables

You remove a variable from the Bash shell using the `unset` command. You can use the `-v` option to be sure that you are removing a variable definition. Functions can have the same name as variables, so use the `-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. (Functions are covered in more detail later in the [Shell functions](#) section.)

Listing 4. The bash `unset` command

```
ian@attic4:~$ VAR1=var1
ian@attic4:~$ VAR2=var2
ian@attic4:~$ echo $VAR1 $VAR2
var1 var2
ian@attic4:~$ unset VAR1
ian@attic4:~$ echo $VAR1 $VAR2
var2
ian@attic4:~$ unset -v VAR2
ian@attic4:~$ echo $VAR1 $VAR2
```

The bash default is to treat unset variables as if they had an empty value, so you might wonder why you would unset a variable rather than just assign it an empty value. Bash and many other shells allow you to generate an error if an undefined variable is referenced. Use the command `set -u` to generate an error for undefined variables, and `set +u` to disable the warning. Listing 5 illustrates these points.

Listing 5. Generating errors with unset variables

```
ian@attic4:~$ set -u
ian@attic4:~$ VAR1=var1
ian@attic4:~$ echo $VAR1
var1
ian@attic4:~$ unset VAR1
ian@attic4:~$ echo $VAR1
-bash: VAR1: unbound variable
ian@attic4:~$ VAR1=
ian@attic4:~$ echo $VAR1

ian@attic4:~$ unset VAR1
ian@attic4:~$ echo $VAR1
-bash: VAR1: unbound variable
ian@attic4:~$ unset -v VAR1
ian@attic4:~$ set +u
ian@attic4:~$ echo $VAR1

ian@attic4:~$
```

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

Profiles

When you log in to a Linux system, your id has a default shell, which is your *login* shell. If this shell is bash, then it executes several profile scripts before you get control. If `/etc/profile` exists, it is executed first. Depending on your distribution, other scripts in the `/etc` tree may also be executed, for example, `/etc/bash.bashrc` or `/etc/bashrc`. Once the system scripts have run, a script in your home directory is run if it exists. Bash looks for the files `~/.bash_profile`, `~/.bash_login`, and `~/.profile` in that order. The first one found is executed.

When you log off, bash executes the `~/.bash_logout` script from your home directory if it exists.

Once you have logged in and are already using bash, you may start another shell, called an *interactive* shell to run a command, for example to run a command in the background. In this case, bash executes only the `~/.bashrc` script, assuming one exists. It is common to check for this script in your `~/.bash_profile`, so that you can execute it at login as well as when starting an interactive shell, using commands such as those shown in Listing 6.

Listing 6. Checking for `~/.bashrc`

```
# include .bashrc if it exists
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

You may force bash to read profiles as if it were a login shell using the `--login` option. If you do not want to execute the profiles for a login shell, specify the `--noprofile` option. Similarly, if you want to disable execution of the `~/.bashrc` file for an interactive shell, start bash with the `--norc` option. You can also force bash to use a file other than `~/.bashrc` by specifying the `--rcfile` option with the name of the file you want to use. Listing 8 illustrates creation of a simple file called `testrc` and its use with the `--rcfile` option. Note that the `VAR1` variable is **not** set in the outer shell, but has been set for the inner shell by the `testrc` file.

Listing 7. Using the `--rcfile` option

```
ian@attic4:~$ echo VAR1=var1>testrc
ian@attic4:~$ echo $VAR1

ian@attic4:~$ bash --rcfile testrc
ian@attic4:~$ echo $VAR1
var1
```

Starting bash in other ways

In addition to the standard ways of running bash from a terminal as outlined above, bash may also be used in other ways.

Unless you *source* a script to run in the current shell, it will run in its own *non-interactive* shell, and the above profiles are not read. However, if the `BASH_ENV` variable is set, bash expands the value and assumes it is the name of a file. If the file exists, then bash executes the file before whatever script or command it is executing in the non-interactive shell. Listing 8 uses two simple files to illustrate this.

Listing 8. Using `BASH_ENV`

```
ian@attic4:~$ cat testenv.sh
#!/bin/bash
echo "Testing the environment"
ian@attic4:~$ cat somescript.sh
#!/bin/bash
echo "Doing nothing"
ian@attic4:~$ export BASH_ENV=~/.testenv.sh
ian@attic4:~$ ./somescript.sh
Testing the environment
Doing nothing
```

Non-interactive shells may also be started with the `--login` option to force execution of the profile files.

Bash may also be started in *POSIX* mode using the `--posix` option. This mode is similar to the non-interactive shell, except that the file to execute is determined from the `ENV` environment variable.

It is common in Linux systems to run bash as `/bin/sh` using a symbolic link. When bash detects that it is being run under the name `sh`, it attempts to follow the startup behavior of the older Bourne shell while still conforming to POSIX standards. When run as a login shell, bash attempts to read and execute `/etc/profile` and `~/.profile`. When run as an interactive shell using the `sh` command, bash attempts to execute the file specified by the `ENV` variable as it does when invoked in POSIX mode. When run interactively as `sh`, it **only** uses a file specified by the `ENV` variable; the `--rcfile` option will always be ignored.

If bash is invoked by the remote shell daemon, then it behaves as an interactive shell, using the `~/.bashrc` file if it exists.

Shell aliases

The Bash shell allows you to define *aliases* for commands. The most common reasons for aliases are to provide an alternate name for a command, or to provide some default parameters for the command. The `vi` editor has been a staple of UNIX and Linux systems for many years. The `vim` (Vi IMproved) editor is like `vi`, but with many improvements. So if you are used to typing "`vi`" when you want an editor, but you would really prefer to use `vim`, then an alias is for you. Listing 9 shows how to use the `alias` command to accomplish this.

Listing 9. Using `vi` as an alias for `vim`

```
[ian@pinguino ~]$ alias vi='vim'
[ian@pinguino ~]$ which vi
alias vi='vim'
      /usr/bin/vim
[ian@pinguino ~]$ /usr/bin/which vi
/bin/vi
```

Notice in this example that if you use the `which` command to see where the `vi` program lives, you get two lines of output: the first shows the alias, and the second the location of `vim` (`/usr/bin/vim`). However, if you use the `which` command with its full path (`/usr/bin/which`), you get the location of the `vi` command. If you guessed that this might mean that the `which` command itself is aliased on this system you would be right.

You can also use the `alias` command to display all the aliases if you use it with no options or with just the `-p` option, and you can display the aliases for one or more names by giving the names as arguments without assignments. Listing 10 shows the aliases for `which` and `vi`.

Listing 10. Aliases for `which` and `vi`

```
[ian@pinguino ~]$ alias which vi
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
alias vi='vim'
```

The alias for the `which` command is rather curious. Why pipe the output of the `alias` command (with no arguments) to `/usr/bin/which`? If you check the man pages for the `which` command, you will find that the `--read-alias` option instructs `which` to read a list of aliases from stdin and report matches on stdout. This allows the `which` command to report aliases as well as commands from your `PATH`, and is so common that your distribution may have set it up as a default for you. This is a good thing to do since the shell will execute an alias before a command of the same name. So now that you know this, you can check it using `alias which`. You can also learn whether this type of alias has been set for `which` by running `which which`.

Another common use for aliases is to add parameters automatically to commands, as you saw above for the `--read-alias` and several other parameters on the `which` command. This technique is often done for the root user with the `cp`, `mv`, and `rm` commands so that a prompt is issued before files are deleted or overwritten. This is illustrated in Listing 11.

Listing 11. Adding parameters for safety

```
[root@pinguino ~]# alias cp mv rm
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
```

Command lists

In the earlier tutorial "[LPI exam 101 prep \(topic 103\): GNU and UNIX commands](#)," you learned about command *sequences* or *lists*. You have just seen the pipe (|) operator used with an alias, and you can use command lists as well. Suppose, for a simple example, that you want a command to list the contents of the current directory and also the amount of space used by it and all its subdirectories. Let's call it the `lsdu` command. So you simply assign a sequence of the `ls` and `du` commands to the alias `lsdu`. Listing 12 shows the wrong way to do this and also the right way. Look carefully at it before you read, and think about why the first attempt did not work.

Listing 12. Aliases for command sequences

```
[ian@pinguino developerworks]$ alias lsdu=ls;du -sh # Wrong way
2.9M .
[ian@pinguino developerworks]$ lsdu
a tutorial  new-article.sh  new-tutorial.sh  readme  tools  xsl
my-article new-article.vbs  new-tutorial.vbs  schema  web
[ian@pinguino developerworks]$ alias 'lsdu=ls;du -sh' # Right way way
[ian@pinguino developerworks]$ lsdu
a tutorial  new-article.sh  new-tutorial.sh  readme  tools  xsl
my-article new-article.vbs  new-tutorial.vbs  schema  web
2.9M .
```

You need to be very careful to quote the full sequence that will make up the alias. You also need to be very careful about whether you use double or single quotes if you have shell variables as part of the alias. Do you want the shell to expand the variables when the alias is defined or when it is executed? Listing 13 shows the wrong way to create a custom command called `mywd` intended to print your working directory name

Listing 13. Custom pwd - attempt 1

```
[ian@pinguino developerworks]$ alias mywd="echo \"My working directory is $PWD\""
[ian@pinguino developerworks]$ mywd
My working directory is /home/ian/developerworks
[ian@pinguino developerworks]$ cd ..
[ian@pinguino ~]$ mywd
My working directory is /home/ian/developerworks
```

Remember that the double quotes cause bash to expand variables before executing a command. Listing 14 uses the `alias` command to show what the resulting alias actually is, from which our error is evident. Listing 14 also shows a correct way to define this alias.

Listing 14. Custom pwd - attempt 2

```
[ian@pinguino developerworks]$ alias mywd
alias mywd='echo \"My working directory is $PWD\"'
[ian@pinguino developerworks]$ mywd
"My working directory is /home/ian/developerworks"
[ian@pinguino developerworks]$ cd ..
```

```
[ian@pinguino ~]$ mywd
"My working directory is /home/ian"
```

Success at last.

Shell functions

Aliases allow you to use an abbreviation or alternate name for a command or command list. You may have noticed that you can add additional things, such as the program name you are seeking with the `which` command. When your input is executed, the alias is expanded, and anything else you type after that is added to the expansion before the final command or list is executed. This means that you can only add parameters to the end of the command or list, and you can use them only with the final command. Functions provide additional capability, including the ability to process parameters. Functions are part of the POSIX shell definition. They are available in shells such as `bash`, `dash`, and `ksh`, but are not available in `csh` or `tcsh`.

In the next few paragraphs, you'll build a complex command piece-by-piece from smaller building blocks, refining it each step of the way and turning it into a function that you will further refine.

A hypothetical problem

You can use the `ls` command to list a variety of information about directories and files in your file system. Suppose you would like a command, let's call it `ldirs`, that will list directory names with output like that in Listing 15.

Listing 15. The `ldirs` command output

```
[ian@pinguino developerworks]$ ldirs *[st]* tools/*a*
my dw article
schema
tools
tools/java
xsl
```

To keep things relatively simple, the examples in this section use the directories and files from the developerWorks author package (see [Resources](#)), which you can use if you'd like to write articles or tutorials for developerWorks. In these examples, we used the `new-article.sh` script from the package to create a template for a new article that we've called "my dw article".

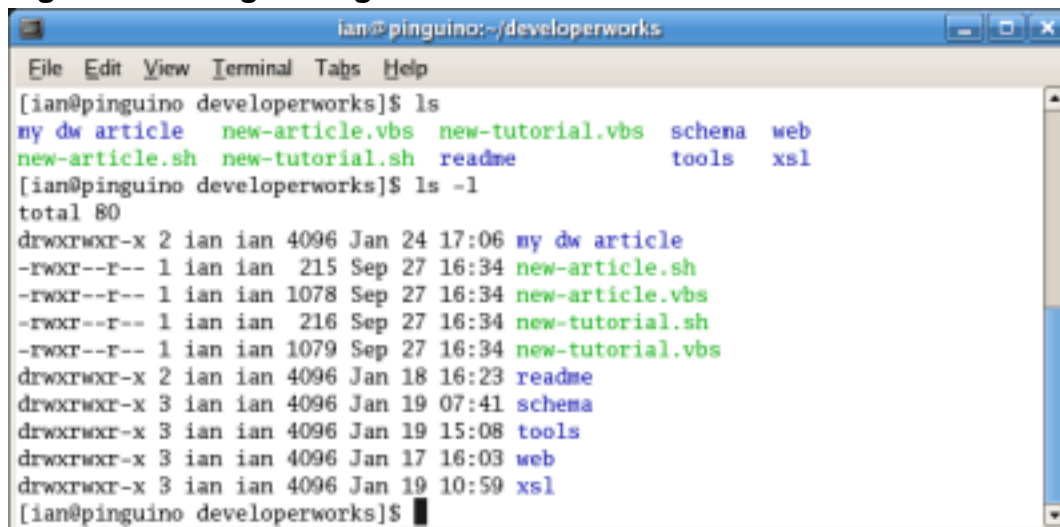
At the time of writing, the version of the developerWorks author package is 5.6, so you may see differences if you use a later version. Or just use your own files and directories. The `ldirs` command will handle those too. You'll find additional bash function examples in the tools that come with the developerWorks author package.

Finding directory entries

Ignoring the `*[st]* tools/*a*` for the moment, if you use the `ls` command with

the color options as shown in the aliases above, you will see output similar to that shown in Figure 1.

Figure 1. Distinguishing files and directories with the ls command



```

ian@pinguino:~/developerworks
File Edit View Terminal Tabs Help
[ian@pinguino developerworks]$ ls
my dw article  new-article.vbs  new-tutorial.vbs  schema  web
new-article.sh  new-tutorial.sh  readme            tools    xsl
[ian@pinguino developerworks]$ ls -l
total 80
drwxrwxr-x 2 ian ian 4096 Jan 24 17:06 my dw article
-rwxr--r-- 1 ian ian  215 Sep 27 16:34 new-article.sh
-rwxr--r-- 1 ian ian 1078 Sep 27 16:34 new-article.vbs
-rwxr--r-- 1 ian ian  216 Sep 27 16:34 new-tutorial.sh
-rwxr--r-- 1 ian ian 1079 Sep 27 16:34 new-tutorial.vbs
drwxrwxr-x 2 ian ian 4096 Jan 18 16:23 readme
drwxrwxr-x 3 ian ian 4096 Jan 19 07:41 schema
drwxrwxr-x 3 ian ian 4096 Jan 19 15:08 tools
drwxrwxr-x 3 ian ian 4096 Jan 17 16:03 web
drwxrwxr-x 3 ian ian 4096 Jan 19 10:59 xsl
[ian@pinguino developerworks]$

```

The directories are shown in dark blue in this example, but that's a bit hard to decode with the skills you have developed in this series of tutorials. Using the `-l` option, though, gives a clue on how to proceed: directory listings have a 'd' in the first position. So your first step might be to simply filter these from the long listing using `grep` as shown in Listing 16.

Listing 16. Using grep to find just directory entries

```

[ian@pinguino developerworks]$ ls -l | grep "^d"
drwxrwxr-x 2 ian ian 4096 Jan 24 17:06 my dw article
drwxrwxr-x 2 ian ian 4096 Jan 18 16:23 readme
drwxrwxr-x 3 ian ian 4096 Jan 19 07:41 schema
drwxrwxr-x 3 ian ian 4096 Jan 19 15:08 tools
drwxrwxr-x 3 ian ian 4096 Jan 17 16:03 web
drwxrwxr-x 3 ian ian 4096 Jan 19 10:59 xsl

```

Trimming the directory entries

You might consider using `awk` instead of `grep` so that in one pass you can filter the list and strip off the last part of each line, which is the directory name, as shown in Listing 17.

Listing 17. Using awk instead

```

[ian@pinguino developerworks]$ ls -l | awk '/^d/ { print $NF } '
article
readme
schema
tools
web
xsl

```

The problem with the approach in Listing 17 is that it doesn't handle the directory with spaces in the name, such as "my dw article". As with most things in Linux and

life, there are often several ways to solve a problem, but the objective here is to learn about functions, so let's return to using `grep`. Another tool you learned about earlier in this series is `cut`, which cuts fields out of a file, including stdin. Looking back at Listing 16 again, you see eight blank-delimited fields before the filename. Adding `cut` to the previous command gives you output as shown in Listing 18. Note that the `-f9-` option tells `cut` to print fields 9 and above.

Listing 18. Using cut to trim names

```
[ian@pinguino developerworks]$ ls -l | grep "^d" | cut -d" " -f9-
my dw article
readme
schema
tools
web
xsl
```

A small problem with our approach is made obvious if we try our command on the `tools` directory instead of on the current directory as shown in Listing 19.

Listing 19. A problem with cut

```
[ian@pinguino developerworks]$ ls -l tools | grep "^d" | cut -d" " -f9-
11:25 java
[ian@pinguino developerworks]$ ls -ld tools/[fjt]*
-rw-rw-r-- 1 ian ian 4798 Jan 8 14:38 tools/figure1.gif
drwxrwxr-x 2 ian ian 4096 Oct 31 11:25 tools/java
-rw-rw-r-- 1 ian ian 39431 Jan 18 23:31 tools/template-dw-article-5.6.xml
-rw-rw-r-- 1 ian ian 39407 Jan 18 23:32 tools/template-dw-tutorial-5.6.xml
```

How did the timestamp get in there? The two template files have 5-digit sizes, while the `java` directory has only a 4-digit size, so `cut` interpreted the extra space as another field separator.

Use seq to find a cut point

The `cut` command can also cut using character positions instead of fields. Rather than counting characters, the Bash shell has lots of utilities that you can use, so you might try using the `seq` and `printf` commands to print a ruler above your long directory listing so you can easily figure where to cut the lines of output. The `seq` command takes up to three arguments, which allow you to print all the numbers up to a given value, all the numbers from one value to another, or all the numbers from one value, stepping by a given value, up to a third value. See the man pages for all the other fancy things you can do with `seq`, including printing octal or hexadecimal numbers. For now let's use `seq` and `printf` to print a ruler with positions marked every 10 characters as shown in Listing 20.

Listing 20. Printing a ruler with seq and printf

```
[ian@pinguino developerworks]$ printf "....+...%2.d" `seq 10 10 60`;printf "\n";ls -l
....+...10....+...20....+...30....+...40....+...50....+...60
total 88
drwxrwxr-x 2 ian ian 4096 Jan 24 17:06 my dw article
```



```
-rwxr--r-- 1 ian ian 215 Sep 27 16:34 new-article.sh
-rwxr--r-- 1 ian ian 1078 Sep 27 16:34 new-article.vbs
-rwxr--r-- 1 ian ian 216 Sep 27 16:34 new-tutorial.sh
-rwxr--r-- 1 ian ian 1079 Sep 27 16:34 new-tutorial.vbs
drwxrwxr-x 2 ian ian 4096 Jan 18 16:23 readme
drwxrwxr-x 3 ian ian 4096 Jan 19 07:41 schema
drwxrwxr-x 3 ian ian 4096 Jan 19 15:08 tools
drwxrwxr-x 3 ian ian 4096 Jan 17 16:03 web
drwxrwxr-x 3 ian ian 4096 Jan 19 10:59 xsl
```

Aha! Now you can use the command `ls -l | grep "^d" | cut -c40-` to cut lines starting at position 40. A moment's reflection reveals that this doesn't really solve the problem either, because larger files will move the correct cut position to the right. Try it for yourself.

Sed to the rescue

Sometimes called the "Swiss army knife" of the UNIX and Linux toolbox, `sed` is an extremely powerful editing filter that uses regular expressions. You now understand that the challenge is to strip off the first 8 words and the blanks that follow them from every line of output that begins with 'd'. You can do it all with `sed`: select only those lines you are interested in using the pattern-matching expression `/^d/`, substituting a null string for the first eight words using the substitute command `s/^d\([^]* *\)\{8\}/`. Use the `-n` option to print only lines that you specify with the `p` command as shown in Listing 21.

Listing 21. Trimming directory names with sed

```
[ian@pinguino developerworks]$ ls -l | sed -ne 's/^d\([^ ]* *\)\{8\}/p'
my dw article
readme
schema
tools
web
xsl
[ian@pinguino developerworks]$ ls -l tools | sed -ne 's/^d\([^ ]* *\)\{8\}/p'
java
```

To learn more about `sed`, see the [Resources](#) section.

A function at last

Now that you have the complex command that you want for your `ldirs` function, it's time to learn about making it a function. A function consists of a name followed by `()` and then a compound command. For now, a compound command will be any command or command list, terminated by a semicolon and surrounded by braces (which must be separated from other tokens by white space). You will learn about other compound commands in the [Shell scripts](#) section.

Note: In the Bash shell, a function name may be preceded by the word 'function', but this is not part of the POSIX specification and is not supported by more minimalist shells such as `dash`. In the [Shell scripts](#) section, you will learn how to make sure that a script is interpreted by a particular shell, even if you normally use a different shell.

Inside the function, you can refer to the parameters using the bash special variables in Table 4. You prefix these with a \$ symbol to reference them as with other shell variables.

Table 4. Shell parameters for functions	
Parameter	Purpose
0, 1, 2, ...	The positional parameters starting from parameter 0. Parameter 0 refers to the name of the program that started bash, or the name of the shell script if the function is running within a shell script. See the bash man pages for information on other possibilities, such as when bash is started with the <code>-c</code> parameter. A string enclosed in single or double quotes will be passed as a single parameter, and the quotes will be stripped. In the case of double quotes, any shell variables such as <code>\$HOME</code> will be expanded before the function is called. You will need to use single or double quotes to pass parameters that contain embedded blanks or other characters that might have special meaning to the shell.
*	The positional parameters starting from parameter 1. If the expansion is done within double quotes, then the expansion is a single word with the first character of the interfield separator (IFS) special variable separating the parameters or no intervening space if IFS is null. The default IFS value is a blank, tab, and newline. If IFS is unset, then the separator used is a blank, just as for the default IFS.
@	The positional parameters starting from parameter 1. If the expansion is done within double quotes, then each parameter becomes a single word, so that <code>"\$@"</code> is equivalent to <code>"\$1" "\$2"</code> . If your parameters are likely to contain embedded blanks, you will want to use this form.
#	The number of parameters, not including parameter 0.

Note: If you have more than 9 parameters, you cannot use `$10` to refer to the tenth one. You must first either process or save the first parameter (`$1`), then use the `shift` command to drop parameter 1 and move all remaining parameters down 1, so that `$10` becomes `$9` and so on. The value of `$#` will be updated to reflect the remaining number of parameters.

Now you can define a simple function to do nothing more than tell you how many parameters it has and display them as shown in Listing 22.

Listing 22. Function parameters

```
[ian@pinguino developerworks]$ testfunc () { echo "$# parameters"; echo "$@"; }
[ian@pinguino developerworks]$ testfunc
0 parameters

[ian@pinguino developerworks]$ testfunc a b c
3 parameters
a b c
[ian@pinguino developerworks]$ testfunc a "b c"
2 parameters
a b c
```

Whether you use `$*`, `"$*"`, `$@`, or `"$@"`, you won't see much difference in the output of the above function, but rest assured that when things become more complex, the distinctions will matter very much.

Now take the complex command that we tested up to this point and create a `ldirs` function with it, using `"$@"` to represent the parameters. You can enter all of the function on a single line as you did in the previous example, or `bash` lets you enter commands on multiple lines, in which case a semicolon will be added automatically as shown in Listing 23. Listing 23 also shows the use of the `type` command to display the function definition. Note from the output of `type` that the `ls` command has been replaced by the expanded value of its alias. You could use `/bin/ls` instead of plain `ls` if you needed to avoid this.

Listing 23. Your first `ldirs` function

```
[ian@pinguino developerworks]$ # Enter the function on a single line
[ian@pinguino developerworks]$ ldirs () { ls -l "$@"|sed -ne 's/^d\([^ ]* *)\{8\} //p'; }
[ian@pinguino developerworks]$ # Enter the function on multiple lines
[ian@pinguino developerworks]$ ldirs ()
> {
> ls -l "$@"|sed -ne 's/^d\([^ ]* *)\{8\} //p'
> }
[ian@pinguino developerworks]$ type ldirs
ldirs is a function
ldirs ()
{
    ls --color=tty -l "$@" | sed -ne 's/^d\([^ ]* *)\{8\} //p'
}
[ian@pinguino developerworks]$ ldirs
my dw article
readme
schema
tools
web
xsl
[ian@pinguino developerworks]$ ldirs tools
java
```

So now your function appears to be working. But what happens if you run `ldirs *` as shown in Listing 24?

Listing 24. Running `ldirs *`

```
[ian@pinguino developerworks]$ ldirs *
5.6
java
www.ibm.com
5.6
```

Are you surprised? You didn't find directories in the current directory, but rather second-level subdirectories. Review the man page for the `ls` command or our earlier tutorials in this series to understand why. Or run the `find` command as shown in Listing 25 to print the names of second-level subdirectories.

Listing 25. Finding second-level subdirectories

```
[ian@pinguino developerworks]$ find . -mindepth 2 -maxdepth 2 -type d
./tools/java
./web/www.ibm.com
./xsl/5.6
./schema/5.6
```

Adding some tests

Using wildcards has exposed a problem with the logic in this approach. We blithely ignored the fact that `ldirs` without any parameters displayed the subdirectories in the current directory, while `ldirs tools` displayed the `java` subdirectory of the `tools` directory rather than the `tools` directory itself as you would expect using `ls` with files rather than directories. Ideally, you should use `ls -l` if no parameters are given and `ls -ld` if some parameters are given. You can use the `test` command to test the number of parameters and then use `&&` and `||` to build a command list that executes the appropriate command. Using the `[test expression]` form of `test`, your expression might look like `{ [$# -gt 0] && /bin/ls -ld "$@" || /bin/ls -l } | sed -ne`

There is a small issue with this code, though, in that if the `ls -ld` command doesn't find any matching files or directories, it will issue an error message and return with a non-zero exit code, thus causing the `ls -l` command to be executed as well. Perhaps not what you wanted. One answer is to construct a compound command for the first `ls` command so that the number of parameters is tested again if the command fails. Expand the function to include this, and your function should now appear as in Listing 26. Try using it with some of the parameters in Listing 26, or experiment with your own parameters to see how it behaves.

Listing 26. Handling wildcards with `ldirs`

```
[ian@pinguino ~]$ type ldirs
ldirs is a function
ldirs ()
{
    {
        [ $# -gt 0 ] && {
            /bin/ls -ld "$@" || [ $# -gt 0 ]
        } || /bin/ls -l
    } | sed -ne 's/^d\([^ ]* *\)\{8\} //p'
}
```

```
[ian@pinguino developerworks]$ ldirs *
my dw article
readme
schema
tools
web
xsl
[ian@pinguino developerworks]$ ldirs tools/*
tools/java
[ian@pinguino developerworks]$ ldirs *xxx*
/bin/ls: *xxx*: No such file or directory
[ian@pinguino developerworks]$ ldirs *a* *s*
my dw article
readme
schema
schema
tools
xsl
```

Final touchup

At this point you might get a directory listed twice as in the last example of Listing 26. You could extend the pipeline by piping the `sed` output through `sort | uniq` if you wish.

Starting from some small building blocks, you have now built quite a complex shell function.

Customizing keystrokes

The keystrokes you type at a terminal session, and also those used in programs such as FTP, are processed by the readline library and can be configured. By default, the customization file is `.inputrc` in your home directory, which will be read during bash startup if it exists. You can configure a different file by setting the `INPUTRC` variable. If it is not set, `.inputrc` in your home directory will be used. Many systems have a default key mapping in `/etc/inputrc`, so you will normally want to include these using the `$include` directive.

Listing 27 illustrates how you might bind your `ldirs` function to the Ctrl-t key combination (press and hold Ctrl, then press t). If you want the command to be executed with no parameters, add `\n` to the end of the configuration line.

Listing 27. Sample `.inputrc` file

```
# My custom key mappings
$include /etc/inputrc
```

You can force the `INPUTRC` file to be read again by pressing Ctrl-x then Ctrl-r. Note that some distributions will set `INPUTRC=/etc/inputrc` if you do not have your own `.inputrc`, so if you create one on such a system, you will need to log out and log back in to pick up your new definitions. Just resetting `INPUTRC` to null or to point to your new file will reread the original file, not the new specification.

The `INPUTRC` file can include conditional specifications. For example, the behavior

of your keyboard should be different according to whether you are using emacs editing mode (the bash default) or vi mode. See the man pages for bash for more details on how to customize your keyboard.

Saving aliases and functions

You will probably add your aliases and functions to your `~/.bashrc` file, although you may save them in any file you like. Whichever you do, remember to source the file or files using the `source` or `.` command so that the contents of your file will be read and executed in the current environment. If you create a script and just execute it, it will be executed in a subshell and all your valuable customization will be lost when the subshell exits and returns control to you.

In the next section, you learn how to go beyond simple functions. You learn how to add programming constructs such as conditional tests and looping constructs and combine these with multiple functions to create or modify Bash shell scripts.

Section 3. Shell scripts

This section covers material for topic 1.109.2 for the Junior Level Administration (LPIC-1) exam 102. The topic has a weight of 3.

In this section, learn how to:

- Use standard shell syntax, such as loops and tests
- Use command substitution
- Test return values for success or failure or other information provided by a command
- Perform conditional mailing to the superuser
- Select the correct script interpreter through the shebang (`#!`) line
- Manage the location, ownership, execution, and `suid`-rights of scripts

This section builds on what you learned about simple functions in the last section and demonstrates some of the techniques and tools that add programming capability to the shell. You have already see simple logic using the `&&` and `||` operators, which allow you to execute one command based on whether the previous command exits normally or with an error. In the `ldirs` function, you used this to alter the call to `ls` according to whether or not parameters were passed to your `ldirs` function. Now you will learn how to extend these basic techniques to more complex shell programming.

Tests

The first thing you need to do in any kind of programming language after learning how to assign values to variables and pass parameters is to test those values and parameters. In shells the tests you do set the return status, which is the same thing that other commands do. In fact, `test` is a builtin *command*!

`test` and `[`

The `test` builtin command returns 0 (True) or 1 (False) depending on the evaluation of an expression *expr*. You can also use square brackets so that `test expr` and `[expr]` are equivalent. You can examine the return value by displaying `$?`; you can use the return value as you have before with `&&` and `||`; or you can test it using the various conditional constructs that are covered later in this section.

Listing 28. Some simple tests

```
[ian@pinguino ~]$ test 3 -gt 4 && echo True || echo false
false
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ test -d "$HOME" ;echo $?
0
```

In the first of these examples, the `-gt` operator was used to perform an arithmetic comparison between two literal values. In the second, the alternate `[]` form was used to compare two strings for inequality. In the final example, the value of the `HOME` variable is tested to see if it is a directory using the `-d` unary operator.

Arithmetic values may be compared using one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, or `-ge`, meaning equal, not equal, less than, less than or equal, greater than, and greater than or equal, respectively.

Strings may be compared for equality, inequality, or whether the first string sorts before or after the second one using the operators `=`, `!=`, `<` and `>`, respectively. The unary operator `-z` tests for a null string, while `-n` or no operator at all returns True if a string is not empty.

Note: the `<` and `>` operators are also used by the shell for redirection, so you must escape them using `\<` or `\>`. Listing 29 shows some more examples of string tests. Check that they are as you expect.

Listing 29. Some string tests

```
[ian@pinguino ~]$ test "abc" = "def" ;echo $?
1
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \< "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \> "def" ];echo $?
1
```

```
[ian@pinguino ~]$ [ "abc" \<"abc" ];echo $?
1
[ian@pinguino ~]$ [ "abc" \> "abc" ];echo $?
1
```

Some of the more common file tests are shown in Table 5. The result is True if the file tested is a file that exists and that has the specified characteristic.

Table 5. Some file tests	
Operator	Characteristic
-d	Directory
-e	Exists (also -a)
-f	Regular file
-h	Symbolic link (also -L)
-p	Named pipe
-r	Readable by you
-s	Not empty
-S	Socket
-w	Writable by you
-N	Has been modified since last being read

In addition to the unary tests above, two files can be compared with the binary operators shown in Table 6.

Table 6. Testing pairs of files	
Operator	True if
-nt	Test if file1 is newer than file 2. The modification date is used for this and the next comparison.
-ot	Test if file1 is older than file 2.
-ef	Test if file1 is a hard link to file2.

Several other tests allow you to check things such as the permissions of the file. See the man pages for bash for more details or use `help test` to see brief information on the test builtin. You can use the `help` command for other builtins too.

The `-o` operator allows you to test various shell options that may be set using `set -o option`, returning True (0) if the option is set and False (1) otherwise, as shown in Listing 30.

Listing 30. Testing shell options

```
[ian@pinguino ~]$ set +o nounset
[ian@pinguino ~]$ [ -o nounset ];echo $?
1
[ian@pinguino ~]$ set -u
[ian@pinguino ~]$ test -o nounset; echo $?
0
```

Finally, the `-a` and `-o` options allow you to combine expressions with logical AND and OR, respectively, while the unary `!` operator inverts the sense of the test. You may use parentheses to group expressions and override the default precedence. Remember that the shell will normally run an expression between parentheses in a subshell, so you will need to escape the parentheses using `\(` and `\)` or enclosing these operators in single or double quotes. Listing 31 illustrates the application of de Morgan's laws to an expression.

Listing 31. Combining and grouping tests

```
[ian@pinguino ~]$ test "a" != "$HOME" -a 3 -ge 4 ; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o 3 -lt 4 \) ]; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o '(' 3 -lt 4 ')' ')' ]; echo $?
1
```

((and [[

The `test` command is very powerful, but somewhat unwieldy with its requirement for escaping and the difference between string and arithmetic comparisons. Fortunately `bash` has two other ways of testing that are somewhat more natural for people who are familiar with C, C++, or Java syntax. The `(())` *compound command* evaluates an arithmetic expression and sets the exit status to 1 if the expression evaluates to 0, or to 0 if the expression evaluates to a non-zero value. You do not need to escape operators between `((` and `))`. Arithmetic is done on integers. Division by 0 causes an error, but overflow does not. You may perform the usual C language arithmetic, logical, and bitwise operations. The `let` command can also execute one or more arithmetic expressions. It is usually used to assign values to arithmetic variables.

Listing 32. Assigning and testing arithmetic expressions

```
[ian@pinguino ~]$ let x=2 y=2**3 z=y*3;echo $? $x $y $z
0 2 8 24
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 3 8 16
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 4 8 13
```

As with `(())`, the `[[]]` compound command allows you to use more natural syntax for filename and string tests. You can combine tests that are allowed for the `test` command using parentheses and logical operators.

Listing 33. Using the `[[` compound

```
[ian@pinguino ~]$ [[ ( -d "$HOME" ) && ( -w "$HOME" ) ]] &&
> echo "home is a writable directory"
home is a writable directory
```

The `[[` compound can also do pattern matching on strings when the `=` or `!=` operators are used. The match behaves as for wildcard globbing as illustrated in Listing 34.

Listing 34. Wildcard tests with `[[`

```
[ian@pinguino ~]$ [[ "abc def .d,x--" == a[abc]*\ ?d* ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
```

You can even do arithmetic tests within `[[` compounds, but be careful. Unless within a `((` compound, the `<` and `>` operators will compare the operands as strings and test their order in the current collating sequence. Listing 35 illustrates this with some examples.

Listing 35. Including arithmetic tests with `[[`

```
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || (( 3 > 2 )) ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 -gt 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 > 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a > 2 ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a -gt 2 ]]; echo $?
-bash: a: unbound variable
```

Conditionals

While you could accomplish a huge amount of programming with the above tests and the `&&` and `||` control operators, bash includes the more familiar "if, then, else" and case constructs. After you learn about these, you will learn about looping constructs and your toolbox will really expand.

If, then, else statements

The bash `if` command is a compound command that tests the return value of a test or command (`$?` and branches based on whether it is True (0) or False (not 0). Although the tests above returned only 0 or 1 values, commands may return other values. You will learn more about testing those a little later in this tutorial. The `if` command in bash has a `then` clause containing a list of commands to be executed if the test or command returns 0. The command also has one or more optional `elif`

clauses. Each of these optional `elif` clauses has an additional test and then clause with an associated list of commands, an optional final `else` clause, and list of commands to be executed if neither the original test, nor any of the tests used in the `elif` clauses was true. A terminal `fi` marks the end of the construct.

Using what you have learned so far, you could now build a simple calculator to evaluate arithmetic expressions as shown in Listing 36.

Listing 36. Evaluating expressions with if, then, else

```
[ian@pinguino ~]$ function mycalc ( )
> {
>   local x
>   if [ $# -lt 1 ]; then
>     echo "This function evaluates arithmetic for you if you give it some"
>   elif (( $* )); then
>     let x="$*"
>     echo "$* = $x"
>   else
>     echo "$* = 0 or is not an arithmetic expression"
>   fi
> }
[ian@pinguino ~]$ mycalc 3 + 4
3 + 4 = 7
[ian@pinguino ~]$ mycalc 3 + 4**3
3 + 4**3 = 67
[ian@pinguino ~]$ mycalc 3 + (4**3 /2)
-bash: syntax error near unexpected token `('
[ian@pinguino ~]$ mycalc 3 + "(4**3 /2)"
3 + (4**3 /2) = 35
[ian@pinguino ~]$ mycalc xyz
xyz = 0 or is not an arithmetic expression
[ian@pinguino ~]$ mycalc xyz + 3 + "(4**3 /2)" + abc
xyz + 3 + (4**3 /2) + abc = 35
```

The calculator makes use of the `local` statement to declare `x` as a local variable that is available only within the scope of the `mycalc` function. The `let` function has several possible options, as does the `declare` function to which it is closely related. Check the man pages for `bash`, or use `help let` for more information.

As you see in Listing 36, you need to be careful making sure that your expressions are properly escaped if they use shell metacharacters such as `(,), *, >, and <`. Nevertheless, you have quite a handy little calculator for evaluating arithmetic as the shell does it.

You may have noticed the `else` clause and the last two examples. As you see, it is not an error to pass `xyz` to `mycalc`, but it evaluates to 0. This function is not smart enough to identify the character values in the final example of use and thus be able to warn the user. You could use a string pattern matching test such as

```
[[ ! ("${*}" == *[a-zA-Z]* ) ]]
```

(or the appropriate form for your locale) to eliminate any expression containing alphabetic characters, but that would prevent using hexadecimal notation in your input, since you might use `0x0f` to represent 15 using hexadecimal notation. In fact, the shell allows bases up to 64 (using `base#value` notation), so you could legitimately use any alphabetic character, plus `_` and `@` in your input. Octal and hexadecimal use the usual notation of a leading 0 for octal and leading `0x` or `0X` for

hexadecimal. Listing 37 shows some examples.

Listing 37. Calculating with different bases

```
[ian@pinguino ~]$ mycalc 015
015 = 13
[ian@pinguino ~]$ mycalc 0xff
0xff = 255
[ian@pinguino ~]$ mycalc 29#37
29#37 = 94
[ian@pinguino ~]$ mycalc 64#1az
64#1az = 4771
[ian@pinguino ~]$ mycalc 64#1azA
64#1azA = 305380
[ian@pinguino ~]$ mycalc 64#1azA_@
64#1azA_@ = 1250840574
[ian@pinguino ~]$ mycalc 64#1az*64**3 + 64#A_@
64#1az*64**3 + 64#A_@ = 1250840574
```

Additional laundering of the input is beyond the scope of this tutorial, so use your calculator with appropriate care.

The `elif` statement is really a convenience. It will help you in writing scripts by allowing you to simplify the indenting. You may be surprised to see the output of the `type` command for the `mycalc` function as shown in Listing 38.

Listing 38. Type mycalc

```
[ian@pinguino ~]$ type mycalc
mycalc is a function
mycalc ()
{
    local x;
    if [ $# -lt 1 ]; then
        echo "This function evaluates arithmetic for you if you give it some";
    else
        if (( $* )); then
            let x="$*";
            echo "$* = $x";
        else
            echo "$* = 0 or is not an arithmetic expression";
        fi;
    fi
}
```

Case statements

The `case` compound command simplifies testing when you have a list of possibilities and you want to take action based on whether a value matches a particular possibility. The `case` compound is introduced by `case WORD in` and terminated by `esac` ("case" spelled backwards). Each case consists of a single pattern, or multiple patterns separated by `|`, followed by `)`, a list of statements, and finally a pair of semicolons `::`.

To illustrate, imagine a store that serves coffee, decaffeinated coffee (decaf), tea, or soda. The function in Listing 39 might be used to determine the response to an order.

Listing 39. Using case commands

```
[ian@pinguino ~]$ type myorder
myorder is a function
myorder ()
{
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
            ;;
        "tea")
            echo "Hot tea on its way"
            ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
            ;;
        *)
            echo "Sorry, we don't serve that here"
            ;;
    esac
}
[ian@pinguino ~]$ myorder decaf
Hot coffee coming right up
[ian@pinguino ~]$ myorder tea
Hot tea on its way
[ian@pinguino ~]$ myorder milk
Sorry, we don't serve that here
```

Note the use of '*' to match anything that had not already been matched.

Bash has another construct similar to `case` that can be used for printing output to a terminal and having a user select items. It is the `select` statement, which will not be covered here. See the `bash` man pages, or type `help select` to learn more about it.

Of course, there are many problems with such a simple approach to the problem; you can't order two drinks at once, and the function doesn't handle anything but lower-case input. Can you do a case-insensitive match? The answer is "yes", so let's see how.

Return values

The Bash shell has a `shopt` builtin that allows you to set or unset many shell options. One is `nocasematch`, which, if set, instructs the shell to ignore case in string matching. Your first thought might be to use the `-o` operand that you learned about with the `test` command. Unfortunately, `nocasematch` is not one of the options you can test with `-o`, so you'll have to resort to something else.

The `shopt` command, like most UNIX and Linux commands sets a return value that you can examine using `$?`. The tests that you learned earlier are not the only things with return values. If you think about the tests that you do in an `if` statement, they really test the return value of the underlying test command for being True (0) or False (1 or anything other than 0). This works even if you don't use a test, but use some other command. Success is indicated by a return value of 0, and failure by a non-zero return value.

Armed with this knowledge, you can now test the `nocasematch` option, set it if it is not already set, and then return it to the user's preference when your function terminates. The `shopt` command has four convenient options, `-pqsu` to print the current value, don't print anything, set the option, or unset the option. The `-p` and `-q` options set a return value of 0 to indicate that the shell option is set, and 1 to indicate it is unset. The `-p` options prints out the command required to set the option to its current value, while the `-q` option simply sets a return value of 0 or 1.

Your modified function will use the return value from `shopt` to set a local variable representing the current state of the `nocasematch` option, set the option, run the case command, then reset the `nocasematch` option to its original value. One way to do this is shown in Listing 40.

Listing 40. Testing return values from commands

```
[ian@pinguino ~]$ type myorder
myorder is a function
myorder ()
{
    local restorecase;
    if shopt -q nocasematch; then
        restorecase="-s";
    else
        restorecase="-u";
        shopt -s nocasematch;
    fi;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
            ;;
        "tea")
            echo "Hot tea on its way"
            ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
            ;;
        *)
            echo "Sorry, we don't serve that here"
            ;;
    esac;
    shopt $restorecase nocasematch
}
[ian@pinguino ~]$ shopt -p nocasematch
shopt -u nocasematch
[ian@pinguino ~]$ # nocasematch is currently unset
[ian@pinguino ~]$ myorder DECAF
Hot coffee coming right up
[ian@pinguino ~]$ myorder Soda
Your ice-cold soda will be ready in a moment
[ian@pinguino ~]$ shopt -p nocasematch
shopt -u nocasematch
[ian@pinguino ~]$ # nocasematch is unset again after running the myorder function
```

If you want your function (or script) to return a value that other functions or commands can test, use the `return` statement in your function. Listing 41 shows how to return 0 for a drink that you can serve and 1 if the customer requests something else.

Listing 41. Setting your own return values from functions

```
[ian@pinguino ~]$ type myorder
```

```

myorder is a function
myorder ()
{
    local restorecase=$(shopt -p nocasematch) rc=0;
    shopt -s nocasematch;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
            ;;
        "tea")
            echo "Hot tea on its way"
            ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
            ;;
        *)
            echo "Sorry, we don't serve that here";
            rc=1
            ;;
    esac;
    $restorecase;
    return $rc
}
[ian@pinguino ~]$ myorder coffee;echo $?
Hot coffee coming right up
0
[ian@pinguino ~]$ myorder milk;echo $?
Sorry, we don't serve that here
1

```

If you don't specify your own return value, the return value will be that of the last command executed. Functions have a habit of being reused in situations that you never anticipated, so it is good practice to set your own value.

Commands may return values other than 0 and 1, and sometimes you will want to distinguish between them. For example, the `grep` command returns 0 if the pattern is matched and 1 if it is not, but it also returns 2 if the pattern is invalid or if the file specification doesn't match any files. If you need to distinguish more return values besides just success (0) or failure (non-zero), then you will probably use a `case` command or perhaps an `if` command with several `elif` parts.

Command substitution

You met command substitution in the ["LPI exam 101 prep \(topic 103\): GNU and UNIX commands"](#) tutorial, but let's do a quick review.

Command substitution allows you to use the output of a command as input to another command by simply surrounding the command with `$()` or with a pair of backticks - ```. You will find the `$()` form advantageous if you want to nest output from one command as part of the command that will generate the final output, and it can be easier to figure out what's really going on as the parentheses have a left and right form as opposed to two identical backticks. However, the choice is yours, and backticks are still very common,

You will often use command substitution with loops (covered later under [Loops](#)). However, you can also use it to simplify the `myorder` function that you just created. Since `shopt -p nocasematch` actually prints the command that you need to set the `nocasematch` option to its current value, you only need to save that output and

then execute it at the end of the `case` statement. This will restore the `nocasematch` option regardless of whether you actually changed it or not. Your revised function might now look like Listing 42. Try it for yourself.

Listing 42. Command substitution instead of return value tests

```
[ian@pinguino ~]$ type myorder
myorder is a function
myorder ()
{
    local restorecase=$(shopt -p nocasematch) rc=0;
    shopt -s nocasematch;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
            ;;
        "tea")
            echo "Hot tea on its way"
            ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
            ;;
        *)
            echo "Sorry, we don't serve that here"
            rc=1
            ;;
    esac;
    $restorecase
    return $rc
}
[ian@pinguino ~]$ shopt -p nocasematch
shopt -u nocasematch
[ian@pinguino ~]$ myorder DECAF
Hot coffee coming right up
[ian@pinguino ~]$ myorder TeA
Hot tea on its way
[ian@pinguino ~]$ shopt -p nocasematch
shopt -u nocasematch
```

Debugging

If you have typed functions yourself and made typing errors that left you wondering what was wrong, you might also be wondering how to debug functions. Fortunately the shell lets you set the `-x` option to trace commands and their arguments as the shell executes them. Listing 43 shows how this works for the `myorder` function of Listing 42.

Listing 43. Tracing execution

```
[ian@pinguino ~]$ set -x
++ echo -ne '\033]0;ian@pinguino:~'

[ian@pinguino ~]$ myorder tea
+ myorder tea
++ shopt -p nocasematch
+ local 'restorecase=shopt -u nocasematch' rc=0
+ shopt -s nocasematch
+ case "$*" in
+ echo 'Hot tea on its way'
Hot tea on its way
+ shopt -u nocasematch
+ return 0
```

```
++ echo -ne '\033]0;ian@pinguino:~'  
[ian@pinguino ~]$ set +x  
+ set +x
```

You can use this technique for your aliases, functions, or scripts. If you need more information, add the `-v` option for verbose output.

Loops

Bash and other shells have three looping constructs that are somewhat similar to those in the C language. Each will execute a list of commands zero or more times. The list of commands is surrounded by the words `do` and `done`, each preceded by a semicolon.

for

loops come in two flavors. The most common form in shell scripting iterates over a set of values, executing the command list once for each value. The set may be empty, in which case the command list is not executed. The other form is more like a traditional C for loop, using three arithmetic expressions to control a starting condition, step function, and end condition.

while

loops evaluate a condition each time the loop starts and execute the command list if the condition is true. If the condition is not initially true, the commands are never executed.

until

loops execute the command list and evaluate a condition each time the loop ends. If the condition is true the loop is executed again. Even if the condition is not initially true, the commands are executed at least once.

If the conditions that are tested are a list of commands, then the return value of the last one executed is the one used. Listing 44 illustrates the loop commands.

Listing 44. For, while, and until loops

```
[ian@pinguino ~]$ for x in abd 2 "my stuff"; do echo $x; done  
abd  
2  
my stuff  
[ian@pinguino ~]$ for (( x=2; x<5; x++ )); do echo $x; done  
2  
3  
4  
[ian@pinguino ~]$ let x=3; while [ $x -ge 0 ] ; do echo $x ;let x--;done  
3  
2  
1  
0  
[ian@pinguino ~]$ let x=3; until echo -e "x=\c"; (( x-- == 0 )) ; do echo $x ; done  
x=2  
x=1  
x=0
```


These examples are somewhat artificial, but they illustrate the concepts. You will most often want to iterate over the parameters to a function or shell script, or a list created by command substitution. Earlier you discovered that the shell may refer to the list of passed parameters as `$*` or `$@` and that whether you quoted these expressions or not affected how they were interpreted. Listing 45 shows a function that prints out the number of parameters and then prints the parameters according to the four alternatives. Listing 46 shows the function in action, with an additional character added to the front of the IFS variable for the function execution.

Listing 45. A function to print parameter information

```
[ian@pinguino ~]$ type testfunc
testfunc is a function
testfunc ()
{
    echo "$# parameters";
    echo Using '$*';
    for p in $*;
    do
        echo "[$p]";
    done;
    echo Using '"$*"';
    for p in "$*";
    do
        echo "[$p]";
    done;
    echo Using '$@';
    for p in $@;
    do
        echo "[$p]";
    done;
    echo Using '"$@"';
    for p in "$@";
    do
        echo "[$p]";
    done
}
```

Listing 46. Printing parameter information with testfunc

```
[ian@pinguino ~]$ IFS="|${IFS}" testfunc abc "a bc" "1 2"
> 3
3 parameters
Using $*
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$*"
[abc|a bc|1 2
3]
Using $@
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$@"
[abc]
[a bc]
[1 2
3]
```

Study the differences carefully, particularly for the quoted forms and the parameters that include white space such as blanks or newline characters.

Break and continue

The `break` command allows you to exit from a loop immediately. You can optionally specify a number of levels to break out of if you have nested loops. So if you had an `until` loop inside a `for` loop inside another `for` loop and all inside a `while` loop, then `break 3` would immediately terminate the `until` loop and the two `for` loops, and return control to the code in the `while` loop.

The `continue` statement allows you to bypass remaining statements in the command list and go immediately to the next iteration of the loop.

Listing 47. Using break and continue

```
[ian@pinguino ~]$ for word in red blue green yellow violet; do
> if [ "$word" = blue ]; then continue; fi
> if [ "$word" = yellow ]; then break; fi
> echo "$word"
> done
red
green
```

Revisiting ldirs

Remember how much work you did to get the `ldirs` function to extract the file name from a long listing and also figure out if it was a directory or not? The final function that you developed was not too bad, but suppose you had all the information you now have. Would you have created the same function? Perhaps not. You know how to test whether a name is a directory or not using `[-d $name]`, and you know about the `for` compound. Listing 48 shows another way you might have coded the `ldirs` function.

Listing 48. Another approach to ldirs

```
[ian@pinguino developerworks]$ type ldirs
ldirs is a function
ldirs ()
{
    if [ $# -gt 0 ]; then
        for file in "$@";
        do
            [ -d "$file" ] && echo "$file";
        done;
    else
        for file in *;
        do
            [ -d "$file" ] && echo "$file";
        done;
    fi;
    return 0
}
[ian@pinguino developerworks]$ ldirs
my dw article
my-tutorial
```

```

readme
schema
tools
web
xsl
[ian@pinguino developerworks]$ ldirs *s* tools/*
schema
tools
xsl
tools/java
[ian@pinguino developerworks]$ ldirs *www*
[ian@pinguino developerworks]$

```

You will note that the function quietly returns if there are no directories matching your criteria. This may or may not be what you want, but if it is, this form of the function is perhaps easier to understand than the version that used `sed` to parse output from `ls`. At least you now have another tool in your toolbox.

Creating scripts

Recall that `myorder` could handle only one drink at a time? You could now combine that single drink function with a `for` compound to iterate through the parameters and handle multiple drinks. This is as simple as placing your function in a file and adding the `for` instruction. Listing 49 illustrates the new `myorder.sh` script.

Listing 49. Ordering multiple drinks

```

[ian@pinguino ~]$ cat myorder.sh
function myorder ()
{
    local restorecase=$(shopt -p nocasematch) rc=0;
    shopt -s nocasematch;
    case "$*" in
        "coffee" | "decaf")
            echo "Hot coffee coming right up"
            ;;
        "tea")
            echo "Hot tea on its way"
            ;;
        "soda")
            echo "Your ice-cold soda will be ready in a moment"
            ;;
        *)
            echo "Sorry, we don't serve that here";
            rc=1
            ;;
    esac;
    $restorecase;
    return $rc
}

for file in "$@"; do myorder "$file"; done

[ian@pinguino ~]$ . myorder.sh coffee tea "milk shake"
Hot coffee coming right up
Hot tea on its way
Sorry, we don't serve that here

```

Note that the script was *sourced* to run in the current shell environment rather than its own shell using the `.` command. To be able to execute a script, either you have to source it, or the script file must be marked executable using the `chmod -x`

command as illustrated in Listing 50.

Listing 50. Making the script executable

```
[ian@pinguino ~]$ chmod +x myorder.sh
[ian@pinguino ~]$ ./myorder.sh coffee tea "milk shake"
Hot coffee coming right up
Hot tea on its way
Sorry, we don't serve that here
```

Specify a shell

Now that you have a brand-new shell script to play with, you might ask whether it works in all shells. Listing 51 shows what happens if you run the exact same shell script on a Ubuntu system using first the Bash shell, then the dash shell.

Listing 51. Shell differences

```
ian@attic4:~$ ./myorder tea soda
-bash: ./myorder: No such file or directory
ian@attic4:~$ ./myorder.sh tea soda
Hot tea on its way
Your ice-cold soda will be ready in a moment
ian@attic4:~$ dash
$ ./myorder.sh tea soda
./myorder.sh: 1: Syntax error: "(" unexpected
```

That's not too good.

Remember earlier when we mentioned that the word 'function' was optional in a bash function definition, but that it wasn't part of the POSIX shell specification? Well, dash is a smaller and lighter shell than bash and it doesn't support that optional feature. Since you can't guarantee what shell your potential users might prefer, you should always ensure that your script is portable to all shell environments, which can be quite difficult, or use the so-called *shebang* (`#!`) to instruct the shell to run your script in a particular shell. The shebang line must be the first line of your script, and the rest of the line contains the path to the shell that your program must run under, so it would be `#!/bin/bash` the `myorder.sh` script.

Listing 52. Using shebang

```
$ head -n3 myorder.sh
#!/bin/bash
function myorder ()
{
$ ./myorder.sh Tea Coffee
Hot tea on its way
Hot coffee coming right up
```

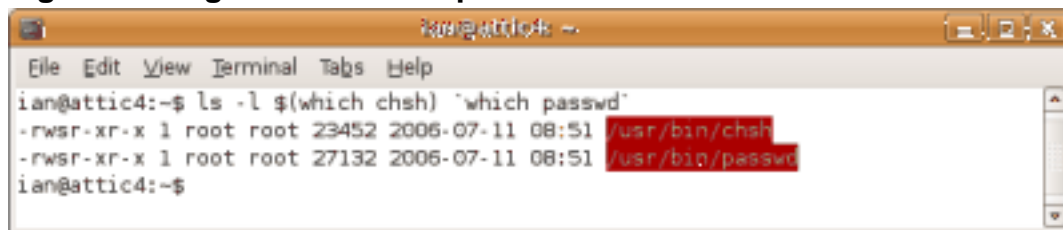
You can use the `cat` command to display `/etc/shells`, which is the list of shells on your system. Some systems do list shells that are not installed, and some listed shells (possibly `/dev/null`) may be there to ensure that FTP users cannot accidentally

escape from their limited environment. If you need to change your default shell, you can do so with the `chsh` command, which updates the entry for your userid in `/etc/passwd`.

Suid rights and script locations

In the earlier tutorial [LPI exam 101 prep: Devices, Linux filesystems, and the Filesystem Hierarchy Standard](#) you learned how to change a file's owner and group and how to set the `suid` and `sgid` permissions. An executable file with either of these permissions set will run in a shell with *effective* permissions of the file's owner (for `suid`) or group (for `sgid`). Thus, the program will be able to do anything that the owner or group could do, according to which permission bit is set. There are good reasons why some programs need to do this. For example, the `passwd` program needs to update `/etc/shadow`, and the `chsh` command, which you use to change your default shell, needs to update `/etc/passwd`. If you use an alias for `ls`, listing these programs is likely to result in a red, highlighted listing to warn you, as shown in Figure 2. Note that both of these programs have the `suid` bit (`s`) set and thus operate as if root were running them.

Figure 2. Programs with suid permission



Listing 53 shows that an ordinary user can run these and update files owned by root.

Listing 53. Using suid programs

```

ian@attic4:~$ passwd
Changing password for ian
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
ian@attic4:~$ chsh
Password:
Changing the login shell for ian
Enter the new value, or press ENTER for the default
Login Shell [/bin/bash]: /bin/dash
ian@attic4:~$ find /etc -mmin -2 -ls
308865      4 drwxr-xr-x 108 root      root          4096 Jan 29 22:52 /etc
find: /etc/cups/ssl: Permission denied
find: /etc/lvm/archive: Permission denied
find: /etc/lvm/backup: Permission denied
find: /etc/ssl/private: Permission denied
311170      4 -rw-r--r--   1 root      root          1215 Jan 29 22:52 /etc/passwd
309744      4 -rw-r-----   1 root      shadow         782 Jan 29 22:52 /etc/shadow
ian@attic4:~$ grep ian /etc/passwd
ian:x:1000:1000:Ian Shields,,,:/home/ian:/bin/dash

```

You can set `suid` and `sgid` permissions for shell scripts, but most modern shells ignore these bits for scripts. As you have seen, the shell has a powerful scripting

language, and there are even more features that are not covered in this tutorial, such as the ability to interpret and execute arbitrary expressions. These features make it a very unsafe environment to allow such wide permission. So, if you set `suid` or `sgid` permission for a shell script, don't expect it to be honored when the script is executed.

Earlier, you changed the permissions of `myorder.sh` to mark it *executable* (`x`). Despite that, you still had to qualify the name by prefixing `./` to actually run it, unless you sourced it in the current shell. To execute a shell by name only, it needs to be on your path, as represented by the `PATH` variable. Normally, you do **not** want the current directory on your path, as it is a potential security exposure. Once you have tested your script and found it satisfactory, you should place it in `~/nom` if it is a personal script, or `/usr/local/bin` if it is to be available for others on the system. If you simply used `chmod -x` to mark it executable, it is executable by everyone (owner, group and world). This is generally what you want, but refer back to the earlier tutorial, [LPI exam 101 prep: Devices, Linux filesystems, and the Filesystem Hierarchy Standard](#), if you need to restrict the script so that only members of a certain group can execute it.

You may have noticed that shells are usually located in `/bin` rather than in `/usr/bin`. According to the Filesystem Hierarchy Standard, `/usr/bin` may be on a filesystem shared among systems, and so it may not be available at initialization time. Therefore, certain functions, such as shells, should be in `/bin` so they are available even if `/usr/bin` is not yet mounted. User-created scripts do not usually need to be in `/bin` (or `/sbin`), as the programs in these directories should give you enough tools to get your system up and running to the point where you can mount the `/usr` filesystem.

Mail to root

If your script is running some administrative task on your system in the dead of night while you're sound asleep, what happens when something goes wrong? Fortunately, it's easy to mail error information or log files to yourself or to another administrator or to root. Simply pipe the message to the `mail` command, and use the `-s` option to add a subject line as shown in Listing 54.

Listing 54. Mailing an error message to a user

```
ian@attic4:~$ echo "Midnight error message" | mail -s "Admin error" ian
ian@attic4:~$ mail
Mail version 8.1.2 01/15/2001.  Type ? for help.
"/var/mail/ian": 1 message 1 new
>N 1 ian@localhost      Mon Jan 29 23:58   14/420   Admin error
&
Message 1:
From ian@localhost  Mon Jan 29 23:58:27 2007
X-Original-To: ian
To: ian@localhost
Subject: Admin error
Date: Mon, 29 Jan 2007 23:58:27 -0500 (EST)
From: ian@localhost (Ian Shields)

Midnight error message
```

```
& d  
& q
```

If you need to mail a log file, use the `<` redirection function to redirect it as input to the mail command. If you need to send several files, you can use `cat` to combine them and pipe the output to mail. In Listing 54, mail was sent to user `ian` who happened to also be the one running the command, but admin scripts are more likely to direct mail to root or another administrator. As usual, consult the man pages for mail to learn about other options that you can specify.

This brings us to the end of this tutorial. We have covered a lot of material on shells and scripting. Don't forget to rate this tutorial and give us your feedback.

Resources

Learn

- Review the entire [LPI exam prep tutorial series](#) on developerWorks to learn Linux fundamentals and prepare for system administrator certification.
- At the [LPIC Program](#), find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- "[Shell Command Language](#)" defines the shell command language as specified by The Open Group and IEEE.
- In "[Basic tasks for new Linux developers](#)" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- Read these developerWorks articles for other ways to work with Bash:
 - [Bash by example, Part 1](#)
 - [Bash by example, Part 2](#)
 - [Bash by example, Part 3](#)
 - [System Administration Toolkit: Get the most out of bash](#)
 - [Working in the bash shell](#)
- Sed and awk each deserve a tutorial on their own, but read these developerWorks articles for a more complete background.
Sed:
 - [Common threads: Sed by example, Part 1](#)
 - [Common threads: Sed by example, Part 2](#)
 - [Common threads: Sed by example, Part 3](#)**Awk:**
 - [Common threads: Awk by example, Part 1](#)
 - [Common threads: Awk by example, Part 2](#)
 - [Common threads: Awk by example, Part 3](#)
 - [Get started with GAWK: AWK language fundamentals](#)
- The [Linux Documentation Project](#) has a variety of useful documents, especially its HOWTOs.
- [LPI Linux Certification in a Nutshell, Second Edition](#) (O'Reilly, 2006) and [LPIC I Exam Cram 2: Linux Professional Institute Certification Exams 101 and 102 \(Exam Cram 2\)](#) (Que, 2004) are LPI references for readers who prefer book format.

- Find more [tutorials for Linux developers](#) in the [developerWorks Linux zone](#).
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Download the developerWorks author package from "[Authoring with the developerWorks XML templates](#)" (developerWorks, January 2007).
- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Download [IBM trial software](#) directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content](#).
- Read [developerWorks blogs](#), and get involved in the developerWorks community.

About the author

Ian Shields

Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University.

Trademarks

DB2, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.