Part Workbook 6. The Bash Shell
# Table of Contents

1. Introduction to Bash
   - Discussion .................................................................................................................. 5
      - The bash shell ........................................................................................................... 5
      - Interactive Shells vs. Shell Scripts ............................................................................. 5
      - Starting Shells ............................................................................................................ 5
      - The ~/.bashrc File ..................................................................................................... 6
      - Entering Commands ................................................................................................. 7
      - Command History ..................................................................................................... 7
      - History Substitution .................................................................................................. 7
      - Preserving History Between Sessions ....................................................................... 8
      - Command History Tricks ......................................................................................... 8
   - Examples ...................................................................................................................... 9
      - Example 1. Using command history to shorten the "Edit/Compile/Execute" cycle ...... 9
      - Example 2. Using ESC. ............................................................................................. 10
      - Example 3. Inhibiting Command History .................................................................. 10
   - Online Exercises ......................................................................................................... 10
      - Specification ............................................................................................................. 10
      - Deliverables ............................................................................................................... 11
   - Questions .................................................................................................................... 11
2. Command Lists and Scripts
   - Discussion ................................................................................................................... 14
      - Running Multiple Commands .................................................................................. 14
      - Running Commands in a Subshell ............................................................................. 14
      - An Introduction to Shell Scripts ............................................................................... 15
      - Return Values ............................................................................................................ 16
      - Running Multiple Commands Conditionally .......................................................... 17
   - Examples ...................................................................................................................... 18
      - Example 1. Echoing $? twice .................................................................................... 18
      - Example 2. Displaying Reminders ............................................................................. 18
   - Online Exercises ......................................................................................................... 19
      - Specification ............................................................................................................. 19
      - Deliverables ............................................................................................................... 19
      - Clean Up .................................................................................................................... 19
   - Questions .................................................................................................................... 19
3. Shell Variables
   - Discussion .................................................................................................................... 22
      - Shell Variable Basics ............................................................................................... 22
      - Shell Variable Details ............................................................................................... 23
      - Bash Variables .......................................................................................................... 24
      - Environment Variables ............................................................................................. 25
      - Listing Variables ....................................................................................................... 26
      - Examining variables with set and env ....................................................................... 26
      - Commonly Used Environment Variables .................................................................. 26
   - Examples ..................................................................................................................... 27
      - Example 1. Using Variables to Reference Commonly Used Words ....................... 27
      - Example 2. Using http_proxy to Define a HTTP Proxy Server .............................. 27
      - Example 3. Appending a Directory to Your PATH .................................................. 28
   - Online Exercises ......................................................................................................... 29
      - Specification ............................................................................................................. 29
      - Deliverables ............................................................................................................... 30
      - Cleaning Up ............................................................................................................... 30
### The Bash Shell

<table>
<thead>
<tr>
<th>Questions</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Command Line Expansion</td>
<td>34</td>
</tr>
<tr>
<td>Discussion</td>
<td>34</td>
</tr>
<tr>
<td>Command Line Expansions</td>
<td>34</td>
</tr>
<tr>
<td>Overview</td>
<td>34</td>
</tr>
<tr>
<td>History Expansion</td>
<td>34</td>
</tr>
<tr>
<td>Brace Expansion</td>
<td>34</td>
</tr>
<tr>
<td>Tilde Expansion</td>
<td>35</td>
</tr>
<tr>
<td>Variable Expansion</td>
<td>36</td>
</tr>
<tr>
<td>Arithmetic Expansion</td>
<td>36</td>
</tr>
<tr>
<td>Command Substitution</td>
<td>36</td>
</tr>
<tr>
<td>Pathname Expansion (&quot;File Globbing&quot;)</td>
<td>37</td>
</tr>
<tr>
<td>Quoting and Escaping Characters</td>
<td>37</td>
</tr>
<tr>
<td>A Note on Quotes</td>
<td>38</td>
</tr>
<tr>
<td>Command Line Expansion Subtleties</td>
<td>38</td>
</tr>
<tr>
<td>Examples</td>
<td>39</td>
</tr>
<tr>
<td>Example 1. Using Brace Expansion</td>
<td>39</td>
</tr>
<tr>
<td>Example 2. More Tab Completion</td>
<td>40</td>
</tr>
<tr>
<td>Example 3. Quoting Awkward Filenames</td>
<td>40</td>
</tr>
<tr>
<td>Online Exercises</td>
<td>41</td>
</tr>
<tr>
<td>Specification</td>
<td>41</td>
</tr>
<tr>
<td>Deliverables</td>
<td>42</td>
</tr>
<tr>
<td>Questions</td>
<td>42</td>
</tr>
<tr>
<td>5. Shell Customizations</td>
<td>46</td>
</tr>
<tr>
<td>Discussion</td>
<td>46</td>
</tr>
<tr>
<td>Shell Builtins</td>
<td>46</td>
</tr>
<tr>
<td>Aliases</td>
<td>47</td>
</tr>
<tr>
<td>The alias Command</td>
<td>47</td>
</tr>
<tr>
<td>The unalias Command</td>
<td>48</td>
</tr>
<tr>
<td>Evaluating Aliases</td>
<td>48</td>
</tr>
<tr>
<td>Running Commands</td>
<td>49</td>
</tr>
<tr>
<td>Customizing the Bash Prompt</td>
<td>49</td>
</tr>
<tr>
<td>Customizing Bash Prompts with PS1 and PS2</td>
<td>49</td>
</tr>
<tr>
<td>Bash Shell Flags and Shell Options</td>
<td>50</td>
</tr>
<tr>
<td>Shell Flags: The set builtin Command</td>
<td>50</td>
</tr>
<tr>
<td>Shell Options: The setopt builtin Command</td>
<td>51</td>
</tr>
<tr>
<td>Examples</td>
<td>52</td>
</tr>
<tr>
<td>Example 1. Red Hat Enterprise Linux Default Aliases</td>
<td>52</td>
</tr>
<tr>
<td>Example 2. Using Aliases to Create Safer rm, mv, and cp Commands</td>
<td>53</td>
</tr>
<tr>
<td>Example 3. Using Aliases to Create Timestamps</td>
<td>54</td>
</tr>
<tr>
<td>Online Exercises</td>
<td>56</td>
</tr>
<tr>
<td>Specification</td>
<td>56</td>
</tr>
<tr>
<td>Challenge Exercise</td>
<td>56</td>
</tr>
<tr>
<td>Deliverables</td>
<td>57</td>
</tr>
<tr>
<td>Questions</td>
<td>57</td>
</tr>
<tr>
<td>6. Sourcing Shell Scripts and Shell Initialization</td>
<td>60</td>
</tr>
<tr>
<td>Discussion</td>
<td>60</td>
</tr>
<tr>
<td>Sourcing Scripts</td>
<td>60</td>
</tr>
<tr>
<td>Bash Initialization: /etc/profile, ~/.profile, ~/.bashrc, and /etc/bashrc</td>
<td>61</td>
</tr>
<tr>
<td>Login, Non-login, Interactive, and Non-interactive Shells</td>
<td>61</td>
</tr>
<tr>
<td>Bash Startup Scripts</td>
<td>62</td>
</tr>
<tr>
<td>Why are there Login and Non-Login Shells?</td>
<td>63</td>
</tr>
<tr>
<td>Why are there Interactive and Non-Interactive shells?</td>
<td>64</td>
</tr>
<tr>
<td>Exiting Login Shells: ~/.bash_logout</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Examples</td>
<td>64</td>
</tr>
<tr>
<td>Example 1. The su Command, Revisited</td>
<td>64</td>
</tr>
<tr>
<td>Example 2. Explicitly sourcing /etc/profile and ~/.bash_profile</td>
<td>65</td>
</tr>
<tr>
<td>Online Exercises</td>
<td>65</td>
</tr>
<tr>
<td>Specification</td>
<td>65</td>
</tr>
<tr>
<td>Deliverables</td>
<td>65</td>
</tr>
<tr>
<td>Questions</td>
<td>66</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction to Bash

Key Concepts

• The default shell in Red Hat Enterprise Linux is the **bash** shell.

• The **bash** shell can be used interactively, or as a powerful scripting language.

• Upon startup, **bash** executes commands found in the `~/.bashrc` file, allowing users to customize their shell.

• The **bash** shell maintains a history of the command lines that it executes. Command lines can be retrieved from the history using various history expansions that begin with "!".

Discussion

The bash shell

In Linux, the shell is the most commonly used program. The shell is what you see when you log in or open a terminal, and is what you use to start most every command. Although there are a variety of different shells available, they all provide the same basic behavior: listen for commands from the user, start processes as specified in those commands, and report the results back to the user. The most commonly used shell in Linux is the **bash** shell, which is the default shell in Red Hat Enterprise Linux.

While easy to use for simple tasks, the **bash** shell also has powerful capabilities that make complicated tasks easier, or even possible. This power brings with it complexity, as a quick look at the **bash** man page (which weighs in at over 4500 lines) will convince you. This Workbook will introduce many of these powerful capabilities.

Interactive Shells vs. Shell Scripts

The **bash** shell is designed to be effective for two different types of uses. You are already acquainted with using **bash** as an interactive shell. Many of the features of **bash** allow people to enter commands more easily and efficiently, and much of this Workbook will focus on these skills.

The **bash** shell is also designed to be a powerful scripting language. Bash shell scripts are short programs written using the same syntax used on the command line. Shell scripts allow users to automate often repeated actions by combining a series of commands. Unlike interactive shells, shell scripts usually run a series of commands non-interactively, and many of the features of the **bash** shell provide programming logic (such as branches and loops) for writing sophisticated scripts. An introduction to shell scripting is found at the end of this Workbook.

As you proceed through the Workbook, try to keep in mind these two different uses of the **bash** shell. Some features of **bash**, such as the command history that we'll soon learn about, are nearly useless in shell scripts. Other features, such as arithmetic substitution, may not seem useful at the command line, but can be useful in a shell script. If the usefulness of a **bash** feature is not immediately obvious, try thinking of it in another context.

Starting Shells

In practice, users seldom need to start a shell manually. Whenever someone logs in, or opens a terminal, a shell is started automatically. Occasionally, however, users would like to run a different shell, or another
instance of the same shell. Because the shell is "just another" program, new shells can be launched from an existing shell. The new shell is referred to as a subshell of the original shell. When the subshell is exited, control is returned to the original shell. In the following example, madonna starts a bash subshell, lists processes from within it to confirm that two shells are running, and then exits the subshell.

```
[madonna@station madonna]$ bash
[madonna@station madonna]$ ps
  PID TTY          TIME CMD
 9750 pts/5    00:00:00 bash
 9786 pts/5    00:00:00 bash
 9814 pts/5    00:00:00 ps
[madonna@station madonna]$ exit
exit
[madonna@station madonna]$ 
```

When starting a bash subshell, the apparent differences between the subshell and the parent shell are minimal, and care must be taken to keep track of which shell you are in.

The ~/.bashrc File

As part of its initialization, the bash shell will look for a file titled .bashrc in a user's home directory. This file is used to customize the bash shell. As the shell starts, commands listed in this file are executed as if they were entered on the command line. Technically, the bash shell "sources" the file. The related concepts of sourcing files and shell initialization are discussed in detail in a later Lesson. Here, we quickly introduce this one file so that we can make use of it in later exercises.

In the following, madonna edits her ~/.bashrc file by adding the cal command, so that the bash shell displays a calendar of the current month upon startup.

```
[madonna@station madonna]$ nano .bashrc
... (madonna appends a single line containing the command "cal") ...
[madonna@station madonna]$ cat .bashrc
# .bashrc
# User specific aliases and functions
# Source global definitions
if [ -f /etc/bashrc ]; then
  /etc/bashrc
fi
cal ①
```

① The user madonna added this single line. The remaining lines are found in a user's default ~/.bashrc file.

Now, whenever madonna starts a bash shell (by logging into a virtual console, or opening another terminal window, for example), a calendar is displayed.

```
[madonna@station madonna]$ bash
  August 2003
  Su Mo Tu We Th Fr Sa
  1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
[madonna@station madonna]$ exit
```
# Entering Commands

Interactive shells continuously repeat a cycle of listening for a command line, evaluating the requested command and performing any requested actions, and displaying the results. The shell listens to the keyboard for input, and uses the **RETURN** key to recognize the end of input, as in the following running of the **echo** command.

```bash
[madonna@station madonna]$ echo "hello world"
hello world
```

## Command History

As a convenience to users of interactive shells, the **bash** shell keeps a history of each command entered by the user, and provides a variety of ways to make commands from this history available at the finger tips. The easiest way to view your current history is to use the **history** command.

```bash
[blondie@station blondie]$ history
1  ls -l /home/
2  ls -ln /home/
3  exit
4  exit
5  id
...
167 mv rhyme stuff/
168 ls -Rli
169 exit
170
171 exit
172 history
```

As shown, the **history** command dumps a history of previously entered commands, with each entry preceded by a "history number". The **history** command itself comes at the end of the list. From the command line, the **UP** and **DOWN** arrow keys will quickly traverse this list up and down, while the **LEFT** and **RIGHT** arrow keys will move the cursor to allow the user to edit a given command. For example, if blondie next wanted to run the command `ls -li`, she could hit the **UP** arrow 5 times and her prompt would be filled with the `ls -Rli` command. She could then hit the **LEFT** arrow twice, and **BACKSPACE** to remove the `R` from the command line, followed by the **RETURN** key. Using the arrow keys, users can quickly review, edit, and run previously typed commands.

## History Substitution

As an alternative to the arrow keys, the **bash** shell also performs "history substitution", which is triggered by the exclamation point. The following table summarizes commonly used history substitution syntax.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>!!</td>
<td>Previous command</td>
</tr>
<tr>
<td>!n</td>
<td>Command number n</td>
</tr>
<tr>
<td>!-n</td>
<td>The n-th most recent command</td>
</tr>
<tr>
<td>!cmd</td>
<td>The most recent command that began cmd</td>
</tr>
</tbody>
</table>

In order to provide examples of the above syntax, consider the following (abbreviated) output when blondie runs the **history** command.

```bash
[blondie@station blondie]$ history
```
The following table lists what blondie would enter on the command line, and the resulting command that would run.

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Resulting Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!!</code></td>
<td><code>history</code></td>
</tr>
<tr>
<td><code>!165</code></td>
<td><code>chmod 660 rhyme</code></td>
</tr>
<tr>
<td><code>!-5</code></td>
<td><code>ls -Rli</code></td>
</tr>
<tr>
<td><code>!mv</code></td>
<td><code>mv rhyme stuff/</code></td>
</tr>
</tbody>
</table>

### Preserving History Between Sessions

Not only does the bash shell maintain a command history within a session, but the shell also preserves command histories between sessions. When the bash shell exits, it dumps the current command history into a file called `.bash_history` in a user's home directory. Upon startup, the shell initializes the command history from the contents of this file.

What repercussions does this have for multiple interactive shells (owned by the same user) running at the same time? Because the history is only saved to disk as the shell exits, commands executed in one bash process are not available in the command history of a simultaneously running bash process. Also, the last shell to exit will overwrite the histories of any shells that exited previously.

If set, the following variables configure the details of how command history is saved.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Default Value</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>HISTFILE</td>
<td><code>~/.bash_history</code></td>
<td>The file to which the command history is saved on exit and from which it is initialized on startup.</td>
</tr>
<tr>
<td>HISTFILESIZE</td>
<td>1000</td>
<td>The file HISTFILE will be truncated to this size on startup.</td>
</tr>
<tr>
<td>HISTSIZE</td>
<td>1000</td>
<td>The maximum number of commands that will be written to the file HISTFILE on exit.</td>
</tr>
</tbody>
</table>

### Command History Tricks

The bash shell provides a few other techniques for accessing previously typed commands (or elements thereof).

ESC+. and ALT+. The last token of the previously typed command line can be recovered with either of the two above mentioned key sequences. Once learned, this little trick comes in
handy surprisingly often. The last token of a command often represents the object that someone is handling. For example, someone might make a directory, and then immediately `cd` to it, or edit a file, and immediately want to use `chmod` to change its permissions. If the key sequence is repeated, the `bash` shell will continue to cycle through the last tokens of earlier command lines.

**CTRL+R**

This key sequence mimics `!cmd` in spirit. Text typed after the `CTRL+R` key sequence is matched against previously typed commands, with the added advantage that matching command lines are viewed immediately as the text is typed. You also have the opportunity to edit the recalled line (using the LEFT and RIGHT arrow keys, or other command line editing keystrokes) before executing the command.

**fc**

The `fc` command allows users to "fix" the previously entered command, by opening up the user's default editor (`vi` by default) with the previously entered command as text. Upon exiting the editor (presumably after somehow editing the command), the new text will be immediately executed. For those proficient in quickly exiting an editor, the command comes in handy.

### Examples

#### Using command history to shorten the "Edit/Compile/Execute" cycle

Often, programmers of compiled languages such as C find themselves in a repetitive cycle of editing a file, compiling it, and then executing the program. In the following, madonna edits a file containing a short C program, and then compiles it with the `gcc` C compiler. After executing the program, she decides to make some changes. She makes use of command history to speed up the process.

```
[madonna@station madonna]$ nano hello.c
[madonna@station madonna]$ cat hello.c
#include "stdio.h"

int main(void)
{
    printf("hello world\n");
    return 0;
}

[madonna@station madonna]$ gcc -o hello hello.c
[madonna@station madonna]$ ./hello
hello world
[madonna@station madonna]$ !n
nano hello.c
(... madonna edits the file, replacing the string "hello world" with "hello dolly" ...)

[madonna@station madonna]$ !c
cat hello.c
#include "stdio.h"
```
int main(void)
{
    printf("hello dolly\n");
    return 0;
}

[madonna@station madonna]$ !g
 gcc -o hello hello.c
[madonna@station madonna]$ !.
 ./hello
hello dolly

Notice that the **bash** shell echos the command that was selected from madonna's history before running the command.

### Using ESC.

Now madonna would like to create a `bin` subdirectory, set its permissions so that only she may access it, and move her executable file `hello` into it. She uses the **ESC+** key sequence to speed the process.

```
[madonna@station madonna]$ mkdir bin
[madonna@station madonna]$ chmod 700 < ESC+.>
[madonna@station madonna]$ mv hello < ESC+.>
[madonna@station madonna]$ ls < ESC+.>
hello
```

Perhaps not the most exciting example, because `bin` is such a short directory to type anyway. Had the directory been `/usr/lib/perl5/vendor_perl/5.8.0/HTML/`, however, the saved keystrokes would be impressive.

### Inhibiting Command History

Being the suspicious type, madonna would like to prevent her command history from being saved to disk when she exits her shell. She removes her history file, and creates a similarly named soft link which resolves to the `/dev/null` device node.

```
[madonna@station madonna]$ rm .bash_history
[madonna@station madonna]$ ln -s /dev/null .bash_history
[madonna@station madonna]$ ls -l .bash_history
 lrwxrwxrwx    1 madonna  madonna         9 Aug 26 16:35 .bash_history -> /dev/null
[madonna@station madonna]$ cat .bash_history
[madonna@station madonna]$
```

Madonna may now use `bash`'s command history to recover commands used in the current shell, but no command history will be saved between shell instances.

### Online Exercises

**Lab Exercise**

**Objective:** Customize your `~/.bashrc` file to keep a record of when shells are started.

**Estimated Time:** 10 mins.

### Specification

1. Using a text editor, modify the `.bashrc` file from your home directory, by appending the following line to the end of the file.
date >> .bash_timestamps

2. Observe the file `.bash_timestamps`, and confirm that a new timestamp is appended every time a new `bash` shell is started.

3. Again using a text editor, add a single comment line to your `.bashrc` file that briefly describes why the `date` command was added, and includes your username as the person who made the modification.

**Deliverables**

1. In your home directory, a modified `.bashrc` that adds a timestamp to the file `.bash_timestamps` every time a `bash` shell is started.

   2. The `.bashrc` file should also contain a comment line that contains your username.

**Questions**

1. What is the default shell in Red Hat Enterprise Linux?
   a. `/bin/conch`
   b. `/bin/sh`
   c. `/usr/bin/tcsh`
   d. `/bin/bash`
   e. None of the above

2. Which of the following would the `bash` shell interpret as a comment?
   a. `/* blagh */`
   b. `% blagh`
   c. `# blagh`
   d. B and C
   e. None of the above

3. For which of the following tasks is the `bash` shell commonly used?
   a. Interactively running commands for users
   b. Numerical calculations
   c. Automated execution of commands from scripts
   d. A and C
   e. All of the above

4. Upon startup, commands from what file are automatically executed by `bash`?
   a. `~/bash_startup`
b. ~/.bash_startup

c. ~/.bash

d. ~/.bashrc

e. None of the above

Use the following output from the `history` command to answer the following 4 questions. For each question, assume that the command is typed directly after the `history` command shown below.

```
[elvis@station elvis]$ history
  4  cd ..
  5  ls
  6  cd doc/
  7  ls
...
  977  xmms &
  978  ls -al /tmp/.esd/socket
  979  rm /tmp/.esd/socket
  980  ps aux | grep karen
  981  pgrep -u karen
  982  rm /tmp/.esd/socket
  983  su
  984  ls
  985  ls -l /tmp/.esd/
  986  ls -ld /tmp/.esd/
  987  su
  988  su
  989  history
```

5. What command would be substituted if elvis next typed `!5`?

a. `xmms &`

b. `rm /tmp/.esd/socket`

c. `ls`

d. `history`

e. None of the above

6. What command would be substituted if elvis next typed `!!`?

a. `xmms &`

b. `cd ..`

c. `su`

d. `ls`

e. None of the above

7. What command would be substituted if elvis next typed `!–5`?

a. `xmms &`

b. `ls`

c. `ls -l /tmp/.esd/`
d. history

8. What command would be substituted if elvis next typed !p?
   a. ps aux | grep karen
   b. pgrep -u karen
   c. ps aux
   d. pico /etc/hosts
   e. The substitution cannot be determined from the information provided.

9. Which of the following key sequences can be used to access portions of bash's command history?
   a. ESC+. 
   b. CTRL+. 
   c. CTRL+H 
   d. CTRL+ALT+F3 
   e. None of the above

10. Which command is used to list bash's command history?
    a. history 
    b. hist 
    c. h 
    d. command 
    e. A and D
Chapter 2. Command Lists and Scripts

Key Concepts

- Multiple commands can be separated with a `;`.
- Upon exiting, every command returns an integer to its parent called a return value.
- The shell variable `?` expands to the return value of previously executed command.
- `&&` and `||` conditionally separate multiple commands.

Discussion

Running Multiple Commands

The `bash` shell allows users to join multiple commands on a single command line by separating the commands with a `;` (English is similar; independent statements may be separated by a semicolon.) The following provides an example.

```
[elvis@station elvis]$ cd /etc/X11; ls
applnk  prefdm  sysconfig  xinit  xorg.conf
fs      serverconfig  twm        Xmodmap  Xresources
[elvis@station X11]$  
```

The effect is identical to entering commands on separate lines.

```
[elvis@station elvis]$ cd /etc/X11
[elvis@station X11]$ ls
applnk  prefdm  sysconfig  xinit  xorg.conf
fs      serverconfig  twm        Xmodmap  Xresources
[elvis@station X11]$  
```

The only difference between the two approaches is that someone does not get the chance to examine the effect of the first command before the second command is executed. There is seldom an actual need to run multiple commands from a single command line, but often combining commands is convenient.

Running Commands in a Subshell

The `bash` shell allows users to easily run commands in a subshell, by wrapping the command with parenthesis. Consider the following example.

```
[elvis@station elvis]$ (cd /etc/X11; ls)
applnk  prefdm  sysconfig  xinit  xorg.conf
fs      serverconfig  twm        Xmodmap  Xresources
[elvis@station X11]$
```

At first glance, the behavior seems identical to the previous example. A closer look reveals a subtle but important difference. In the first example, when commands are merely separated by a semicolon, the commands execute in the current shell. The `bash` prompt reveals that, after the commands are executed, the shell's current working directory has been changed to `/etc/X11` as a result of the `cd` command.

In the previous example, when the commands are wrapped in parenthesis, the shell's current working directory is unchanged. When `bash` encounters parenthesis on the command line, it spawns an entirely new child `bash` process (called a subshell), and runs the commands within the subshell. After running the
commands, the subshell exits, and the user is left in the original (unchanged) shell. The effect is similar to the following sequence of commands.

```
[elvis@station elvis]$ bash
[elvis@station elvis]$ cd /etc/X11; ls
applnk  prefdm  sysconfig  xinit  xorg.conf
fs       serverconfig twm       Xmodmap  Xresources
[elvis@station /etc/X11]$ exit
exit
[elvis@station elvis]$ 
```

1. The subshell is started manually by executing the `bash` command.
2. Commands are now run in the subshell.
3. When finished, the subshell is exited.
4. Now that elvis is back in the original shell, modifications made in the subshell (such as the change in current working directory) have been left behind.

Why would someone want to run a command in a subshell? Subshells are used to avoid side effects. What happens in the subshell should not effect the original shell's environment (just as, in English, what happens in parenthesis should not change the surrounding sentence's context).

### An Introduction to Shell Scripts

The key to using Red Hat Enterprise Linux effectively is automation. A good Linux administrator should actually be extremely lazy when it comes to doing anything boring or repetitive. Previous sections illustrated how to string commands together to run consecutively or simultaneously instead of waiting for one command to finish before typing the next. They also introduced you to bash's history feature and showed how to reference previously-typed commands so that you only need to enter them once.

However, one important piece of your system administrator's toolkit is still missing: scripting. A script is, in its simplest form, just a text file with a list of commands in it. The commands are sent through a specified program, called an interpreter, which runs each command in turn. Usually this interpreter will be the bash shell (referred to as `/bin/bash` or `/bin/sh`) and each command is an ordinary Linux command. Other interpreters allow you to use more powerful programming languages like Perl, Python and Ruby.

Before you can begin writing scripts of your own, there are a few important things to remember:

- The first line of your script must specify which interpreter to send instructions to. This is done with a special string called a "shebang" (pronounced "shuh-bang"), which looks like this: `#!/`. The shebang is followed by the name of the interpreter for this script. So, for example, to use bash as your interpreter you would use `#!/bin/sh` or `#!/bin/bash`. Most scripts just use `#!/bin/sh`. Referring to the interpreter as `#!/bin/bash` enables some extra features but limits the script's compatibility with older Unix systems and is rarely necessary.

- Before you can run a script, you must enable the "executable" permission on it (otherwise, it's just a text file). The command to do this is `chmod u+x <scriptname>`. This grants you (and only you) permission to run this script just like you would any other command. The `chmod` command will be discussed in much more detail later in this class.

- If you created a script called `foo.sh` in your home directory and then just typed `foo.sh` you would get a "no such file or directory" error. This is because when you type a command, there is a fixed set of directories that Linux looks for that command in. These directories are referred to collectively as your PATH and, for security reasons, your PATH never includes the current directory. To solve this problem you have two choices:

  1. You can explicitly specify the location of the script by typing `~/foo.sh` or `/foo.sh` ("." always refers to the current directory).
2. You can place the script in a directory that is part of your PATH. Non-root users do not have permission to place files in most of these directories, but all users have a personal bin, to which they can write, in their home directory. So if foo.sh were moved to ~/bin it could be run by simply typing foo.sh at the command line. This is the preferred technique.
You will learn more about the PATH setting in subsequent chapters of this workbook.

Let's look at a simple example. Suppose you are an administrator who often needs to see which users are logged into the system. This information can be obtained by running the w command (yes, that's the whole thing) but, while this provides a nice summary of who is logged into the system, it does not print the time at which this snapshot of user activity was taken. Another command, called date prints out the current date and time, but no user information. If only you could combine those two commands into one...

Suppose you created a script called wdate.sh in your personal bin directory:

```
[student@station ~]$ cat ~/bin/wdate.sh
#!/bin/sh
date
w
```

```
[student@station ~]$ chmod u+x ~/bin/wdate.sh
[student@station ~]$ wdate.sh
Thu Jul 14 12:13:54 PDT 2005
12:13:54 up 2 days, 12:50, 8 users, load average: 0.35, 0.27, 0.18
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
student_a   tty1     -                 Mon23    ?xdm?  2:43m  3.06s /bin/bash
student     tty2    :0.0               Tue17    0.00s  2.19s  0.00s /bin/sh /home/student/bin/wdate.sh
```

Notice that the script had to be placed in ~/bin and made executable before it could be run as a normal command. When executed it runs date followed by w, giving us two commands for the price of one! Obviously, this script could then be modified to run an arbitrary number of other commands in succession. In fact, scripts can be significantly more powerful than just a list of commands and can be complex programs in their own right. The supplementary material for this lesson discusses these advanced scripting techniques and can be enabled at your instructor's discretion. For now, concentrate on mastering basic scripts as a valuable effort-saving technique. The administrator's rule of thumb is that if you have to do a task more that twice, script it!

**Return Values**

Every process in Linux has a lifespan. All processes start at the request of another process (often a shell). The requesting process is referred to as the parent, and the newly born process the child. Usually, the child process performs its duties (which might involve spawning children of its own), and then elects to die. An exiting process leaves a little piece of information behind when it dies, called the process's return value, or exit status. The parent process is responsible for collecting the return values of dead children.

Return values come in the form of integers which range from 0 to 255. Programs are free to choose what value to return upon exiting. Often, what implications are meant by different return values are part of a program's well defined interface, and are documented in that program's man page. (If you are familiar with the diff command, the "DIAGNOSTICS" section of its man page provides an example). A Linux-wide (and Unix-wide) convention is that a program returns a 0 to imply "success" at whatever it was trying to accomplish, and a non zero return value to imply some form of failure.

The bash shell stores the return value of the previously executed command in a special variable called simply ?. Unfortunately, we have not fully discussed shell variables yet (that comes next), but we will

---

1 If you have done any C programming, the integer passed as an argument to the exit() library call, or returned from the function main(), is used as the process's return value.
quickly note that the value of this variable (i.e., the previously executed program’s return value), can be examined with the `echo $?` command.

In the following example, the `ls` command is used to examine the permissions of the file `/etc/passwd`. Because the command "works", the `ls` command returns a return value of 0.

```
[elvis@station elvis]$ ls -l /etc/passwd
-rw-r--r--    1 root     root         3694 Aug 15 16:26 /etc/passwd
[elvis@station elvis]$ echo $? 0
```

In contrast, the following examples shows how the `ls` command responds to listing a nonexistent file.

```
[elvis@station elvis]$ ls -l /etc/password
ls: /etc/password: No such file or directory
[elvis@station elvis]$ echo $? 1
```

Because the command "didn't work", it returned a return value of 1. Returning a 0 on success, and a 1 when any type of error occurs, is fairly standard behavior. If a program’s man page doesn’t mention otherwise, this behavior can usually be assumed.

### Running Multiple Commands Conditionally

The bash shell uses `&&` and `||` to join two commands conditionally. When commands are conditionally joined, the first will always execute. The second command may execute or not, depending on the return value of the first command. For example, a user may want to create a directory, and then move a new file into that directory. If the creation of the directory fails, then there is no reason to move the file. The two commands can be coupled as follows.

```
[elvis@station elvis]$ echo "one two three" > numbers.txt
[elvis@station elvis]$ mkdir /tmp/boring && mv numbers.txt /tmp/boring
[elvis@station elvis]$ ls
```

By coupling two commands with `&&`, the second command will only run if the first command succeeded (i.e., had a return value of 0). This is similar to the "and" operation found in many programming languages. In the above example, the `mkdir` command succeeded, and the file was moved. What if the `mkdir` command failed?

```
[elvis@station elvis]$ echo "one two three five seven eleven" > primes.txt
[elvis@station elvis]$ mkdir /tmp/mostly/boring && mv primes.txt /tmp/mostly/boring
mkdir: cannot create directory `/tmp/mostly/boring': No such file or directory
[elvis@station elvis]$ ls primes.txt
```

Because the `mkdir` command failed (the directory `/tmp/mostly` did not exist, so the directory `/tmp/mostly/boring` couldn’t be created), bash did not try to execute the `mv` command.

Similarly, multiple commands can be combined with `||`. In this case, bash will execute the second command only if the first command "fails" (has a non zero return value). This is similar to the "or" operator found in programming languages. In the following example, elvis attempts to change the permissions on a file. If the command fails, a message to that effect is echoed to the screen.

```
[elvis@station elvis]$ chmod 600 /tmp/boring/numbers.txt || echo "chmod failed."
[elvis@station elvis]$ chmod 600 /tmp/mostly/boring/primes.txt || echo "chmod failed"
chmod: failed to get attributes of `/tmp/mostly/boring/primes.txt': No such file or directory
```

In the first case, the `chmod` command succeeded, and no message was echoed. In the second case, the `chmod` command failed (because the file didn’t exist), and the "chmod failed" message was echoed (in addition to `chmod`’s standard error message).
Examples

Echoing $? twice

The user elvis has just learned about return values, and is examining the return values of various commands. After running an (unsuccessful) `ls` command, he finds the the bash variable `?` contains 1, as expected. Examining the variable a second time, he finds that it now contains a 0. What caused the value to change?

```
[elvis@station elvis]$ ls -l /etc/password
ls: /etc/password: No such file or directory
[elvis@station elvis]$ echo $?
1
[elvis@station elvis]$ echo $?
0
```

Recall that the `bash` variable `?` contains the return value of the most recently executed command. In the first case, it contained the return value of the (unsuccessful) `ls` command. In the second case, it contained the return value of the (successful) `echo` command.

Displaying Reminders

The user elvis now wants to develop a scheme where he can leave himself reminders, and the reminders will be automatically displayed to him when he starts a shell. He creates a file in his home directory called `reminders` with the text `brush your teeth`, and adds the following line to his `~/.bashrc` file.

```
cat /home/elvis/reminders
```

He then tests his setup by manually starting a new `bash` shell.

```
[elvis@station elvis]$ echo "brush your teeth" > reminders
[elvis@station elvis]$ nano .bashrc
[elvis@station elvis]$ cat .bashrc
#.bashrc

# User specific aliases and functions
# Source global definitions
if [ -f /etc/bashrc ]; then
  . /etc/bashrc
fi

cat reminders
[elvis@station elvis]$ bash
brush your teeth
[elvis@station elvis]$ exit

[elvis@station elvis]$ bash
```

This seems to work well, until elvis follows his hygiene advice, and removes his file `reminders`. The next time he starts a shell, he is met with the following.

```
[elvis@station elvis]$ bash
cat: reminders: No such file or directory
[elvis@station elvis]$ 
```

Realizing that he would like the `cat` command to run only if the file `reminders` exists, he edits the line he added to his `.bashrc` file to the following.

```
ls reminders > /dev/null && cat reminders
```
Now, the `cat` command will only execute if the `ls` command succeeds because the file `reminders` exists. (Are there better ways to go about this? Yes, but we haven't learned enough yet.)

## Online Exercises

### Lab Exercise

**Objective:** Run commands within a subshell.

**Estimated Time:** 10 mins.

### Specification

1. Append a single line to the bottom of the `.bashrc` file in your home directory. The single line should run the commands `cd /usr` and `ls` in a *single subshell*. (When executed, the current working directory of your shell should be unaffected.)

   If implemented properly, you should see output similar to the following when starting a new shell.

   ```bash
   [elvis@station elvis]$ bash
   bin  games    kerberos  libexec  sbin   src  X11R6
   etc  include  lib       local    share  tmp
   [elvis@station elvis]$ 
   ```

### Deliverables

1. A `~/.bashrc` file whose last line runs the two commands `cd /usr/share` and `ls` in a single subshell.

### Clean Up

Once graded, restore your `~/.bashrc` file to its original state.

### Questions

After running a command line from an interactive `bash` prompt, the following messages are displayed.

`bash: timelog: Permission denied
timestamp failed`

1. Which of the following command lines could have produced the messages?
   a. `date >> timelog || echo timestamp failed`
   b. `date >> timelog && echo timestamp failed`
   c. `date >> timelog ; echo timestamp failed`
   d. A and C
   e. All of the above

2. Which expression will display the return value of the currently running shell?
a. `echo $?`
b. `echo $$`
c. `echo $RET_VAL`
d. A and C
e. The question is misconceived, because the current shell does not have a return value until it exits.

Use the following transcript to answer the next 2 questions.

```
[elvis@station elvis]$ echo "stomp" > blue_suede_shoes
[elvis@station elvis]$ chmod 600 blue_suede_shoes
[elvis@station elvis]$ cat blue_suede_shoes
stomp
```

3. Which command could elvis run to display the return value of the `cat` command?
   a. `echo $!`
   b. `echo $$`
   c. `echo $?`
   d. A and C
   e. None of the above

4. Which command could elvis run to display the return value of the `chmod` command?
   a. `echo $!`
   b. `echo $-`$`
   c. `echo $$`
   d. `echo $RET_chmod`
   e. None of the above

5. Which of the following would run the `chmod` command only if the `mkdir` command succeeds?
   a. `mkdir mail || chmod 700 mail`
   b. `mkdir mail && chmod 700 mail`
   c. `mkdir mail ; chmod 700 mail`
   d. `mkdir mail | chmod 700 mail`
   e. None of the above

6. Which of the following would run the `chmod` command only if the `mkdir` command fails?
   a. `mkdir mail | chmod 700 mail`
   b. `mkdir mail && chmod 700 mail`
c. `mkdir mail ; chmod 700 mail`

d. `mkdir mail or chmod 700 mail`

e. None of the above

7. Which of the following would run the `chmod` command regardless of the success or failure of the `mkdir` command?

a. `mkdir mail | chmod 700 mail`

b. `mkdir mail && chmod 700 mail`

c. `mkdir mail ; chmod 700 mail`

d. `mkdir mail or chmod 700 mail`

e. None of the above

8. Which of the following is the correct way to run the `cd` and `ls` commands in a single subshell?

a. `cd /etc && ls`

b. `( cd /etc ; ls)`

c. `{ cd /etc ; ls}`

d. `$( cd /etc ; ls)`

e. None of the above

9. After successfully executing the command line from the previous question, what will your current working directory be?

a. `/`

b. `~`

c. The question is moot. None of the command lines from the previous question are correct.

d. `/etc`

e. The same directory from which the command line was executed. The current shell's working directory would not change.

10. After running the command line `cd /tmp; ls`, what is your current working directory?

a. `/`

b. `~`

c. `/ls`

d. `/etc`

e. `/tmp`
Chapter 3. Shell Variables

Key Concepts

- Shell variables are assigned using an `A=apple` syntax.
- Variables are examined ("dereferenced") with the `$` character, as in `echo $A`.
- At the kernel level, every process has a collection of environment variables, which are inherited by child processes.
- The `export` command converts a shell variable into an environment variable.
- The `set` and `env` commands list shell variables and environment variables, respectively.

Discussion

Shell Variable Basics

The bash shell allows users to set and reference shell variables. A shell variable is simply a named value that the shell remembers. Shell variables can be used in commands and shell scripts and can also be referenced by programs as configuration options. For example, the mutt email client runs an external editor when composing a message. By default this editor is vi. However, before running vi it will check to see if a variable called EDITOR has been set. If it has, then the command defined by EDITOR is used instead of vi. Most programs that launch external editors work the same way.

There are two types of shell variables: local variables and environment variables. A local variable exists only within the shell in which it is created. Environment variables are inherited by child shells such as when a graphical terminal is launched after logging in. First we will see how to define a local variable, then we will talk about defining environment variables and using them to configure programs including bash itself.

Setting local variables is quite simple. In the following example, prince will set the variable `A` to the value `apple`.

```
[prince@station prince]$ A=apple
```

If you are following along, make sure that you don't place any spaces on either side of the `=` sign. Now the shell will "hang on" to this association for as long as the shell exists (or until it is explicitly unset, see below). Whenever prince would like to use the value "apple", he can use the variable `A` instead, preceding the variable with a dollar sign ($), as in the `echo` command below. This is called dereferencing the variable `A`.

```
[prince@station prince]$ echo $A
apple
```

The variable can be used anywhere on the command line (or in shell scripts). What if prince, waxing poetic, decided to write a few lines about apples, which he wanted to save in a file called `ode_to_apple.txt`. The following line could get him started.

```
[prince@station prince]$ echo "Oh, I like them squishy" >> ode_to_$A.txt
```

```
[prince@station prince]$ ls
ode_to_apple.txt
```
When the bash shell examined the command line, it replaced $A with apple. These are the basics of shell variables. Variables are established, and set, with a VAR=value syntax, and dereferenced with a $VAR syntax.

Shell Variable Details

What can be used for variable names? Variable names can be any string of alphanumeric characters (A-Z, a-z, 0-9), and the underscore (_), but cannot start with a number. Shell variables are case sensitive, as seen below.

```
[prince@station prince]$ B=banana
[prince@station prince]$ echo $B is my favorite fruit
banana is my favorite fruit
[prince@station prince]$ echo $b is my favorite fruit
is my favorite fruit
```

In the first echo, $B was replaced with the value banana. How was $b dereferenced? If the shell is asked to dereference an unset variable, it replaces the variable reference with an empty string (in other words, with nothing). Because b is considered a different variable than B, and because the variable b has never been assigned, the shell replaces the reference $b with nothing. By convention, variables are usually defined using all capital letters, but this is only a convention.

What can be a variable's value? Anything. The trick comes in the assignment. When assigning variables, the syntax is name=value, with no spaces. What if prince wanted the variable FRUIT to resolve to the phrase mushy bananas?

```
[prince@station prince]$ FRUIT=mushy bananas
-bash: bananas: command not found
```

We have stumbled into an advanced syntax for setting variables, namely name=value command, which sets the variable name only for the execution of the specified command. The bash shell dutifully set the variable FRUIT to the value mushy, and went to execute the command bananas, with expectable results. All of this is not the important bit. The important bit is that if you want to set a variable to a value which contains spaces, you must include the value in quotes.

```
[prince@station prince]$ FRUIT="mushy bananas"
[prince@station prince]$ echo $FRUIT is my favorite fruit
mushy bananas is my favorite fruit
```

With this modification, prince gets the correct behavior from the bash shell, if not correct English grammar.

When dereferencing variables, the variable name can be marked using braces {}, if need be. For example, what if above, prince had wanted to save his poem into a file called apple_ode.txt? He tries the obvious first approach, in the same directory as above.

```
[prince@station prince]$ ls
ode_to_apple.txt
[prince@station prince]$ echo $A
apple
[prince@station prince]$ echo "Oh, I like them mushy" > $A_ode.txt
[prince@station prince]$ ls
ode_to_apple.txt
```

Where is the file apple_ode.txt? A couple of things have conspired against prince. First, the bash shell dereferenced the correct variable name, but not the one that prince intended. What can a variable name be composed of? Alphanumeric characters, and the underscore. The bash shell resolved the (uninitialized) variable A_ode (to nothing), and created the resulting file .txt. Secondly, because .txt starts with a ., it is a "hidden file", as the ls -a command reveals.

```
[prince@station prince]$ ls -a
.
  .bash_profile  .gtkrc  .plan
```
Shell Variables

..             .bashrc         .kde              .txt
 bash_history .gnome-desktop ode_to_apple.txt .viminfo
 bash_logout   .gnupg          .pgpkey           .xauthizv2EF

[prince@station prince]$ cat .txt
Oh, I like them mushy

The user prince can get out of this situation by using braces to wrap the desired variable name.

[prince@station prince]$ echo "Oh, I like them mushy" > ${A}_ode.txt
[prince@station prince]$ ls
apple_ode.txt  ode_to_apple.txt

Using braces to delineate variable names is never incorrect, and in some situations, is necessary.

When finished with a variable, the variable may be unbound from its value with the `unset` command.

[prince@station prince]$ unset A
[prince@station prince]$ echo $A
$ 

Bash Variables

The following table lists some of the variables which are automatically set by the `bash` shell. These variables are read only, and may not be set by the user.

### Table 3.1. Bash Read Only Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expands To</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>The exit status of the most recently executed command</td>
</tr>
<tr>
<td>-</td>
<td>Currently enabled shell option flags</td>
</tr>
<tr>
<td>$</td>
<td>Process id (pid) of current shell</td>
</tr>
<tr>
<td>!</td>
<td>Process id (pid) of most recent background command</td>
</tr>
<tr>
<td>_</td>
<td>Last token of previous command</td>
</tr>
<tr>
<td>PPID</td>
<td>The process id (pid) of the shell's parent.</td>
</tr>
<tr>
<td>SHELLOPTS</td>
<td>Colon separated list of currently enabled shell options, as reported by the <code>set -o</code> command.</td>
</tr>
<tr>
<td>UID</td>
<td>The userid of the current user</td>
</tr>
</tbody>
</table>

These variables are set by the shell to provide information. They cannot be reassigned by the user, as prince discovers below.

[prince@station prince]$ echo $SHELLOPTS
[prince@station prince]$ SHELLOPTS=foo
-bash: SHELLOPTS: readonly variable

The following variables are initialized by the `bash` shell, but can be reassigned.

### Table 3.2. Bash Preassigned Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expands To</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASH_VERSION</td>
<td>The current <code>bash</code> version</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>The DNS hostname of the current machine</td>
</tr>
<tr>
<td>OLDPWD</td>
<td>The previous working directory</td>
</tr>
</tbody>
</table>
Shell Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expands To</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWD</td>
<td>The current working directory</td>
</tr>
<tr>
<td>RANDOM</td>
<td>A random number between 0 and 32767</td>
</tr>
<tr>
<td>SECONDS</td>
<td>The number of seconds since the shell was started</td>
</tr>
</tbody>
</table>

Environment Variables

Setting and resolving variables should be fairly straightforward (as long as you remember that bit about quoting spaces). Now we introduce a concept that’s a little more subtle, and much more useful: environment variables.

Just as the bash shell allows users to assign name-value pairs called shell variables, the Linux kernel allows any process to define name-value pairs called environment variables. These variables are a part of the process stored in the kernel, just as the process id, user id, and current working directory are part of the process. More importantly, whenever a process starts another process (such as the bash shell starting the ls command), environment variables are inherited by the child process. This allows users to use the bash shell to create or modify an environment variable, and then all commands started by the shell will inherit that variable.

How do we create environment variables within the bash shell? First, a shell variable is created, and then the shell variable is "promoted" to an environment variable using the export command. (The variable will then be exported to any future child processes). Consider the following example.

```
[prince@station prince]$ A=apple
[prince@station prince]$ B=banana
[prince@station prince]$ echo a:$A b:$B
a:apple b:banana
[prince@station prince]$ export A
[prince@station prince]$ bash
[prince@station prince]$ ps
PID TTY          TIME CMD
2251 pts/5    00:00:00 bash
2316 pts/5    00:00:00 bash
2342 pts/5    00:00:00 ps
[prince@station prince]$ echo a:$A b:$B
a:apple b:
[prince@station prince]$ exit
exit
[prince@station prince]$ echo a:$A b:$B
a:apple b:banana
[prince@station prince]$ unset A B
[prince@station prince]$ echo a:$A b:$B
a:apple b:banana
```

1. The user prince has created two shell variables, A and B.
2. The variable A is promoted to an environment variable with the export command.
3. The user prince starts a bash subshell.
4. By running the ps command, prince confirms that there are two shells running: the parent and the child (his current shell).
5. Because the variable A was promoted to be an environment variable, it was inherited by the child shell from the parent. In contrast, the child shell knows nothing about the parent's shell variable B.
6. When prince exits the child shell, he returns to the parent shell, where the variable B is still defined.
7. Lastly, prince unbinds both the environment variable A and the shell variable B with the same unset command.

Environment variables are often used to configure commands with information about local configurations, or, in other words, information about the local environment. As an example, many commands will look
for an environment variable called LANG to determine the user's language, and modify their output accordingly.

```bash
[prince@station prince]$ echo $LANG
en_US.UTF-8
[prince@station prince]$ date
Fri Aug  1 11:54:24 EDT 2002
[prince@station prince]$ LANG=de_DE
[prince@station prince]$ date
Fre Aug  1 11:54:53 EDT 2002
[prince@station prince]$ LANG=es_ES
[prince@station prince]$ date
vri ago  1 11:55:09 EDT 2002
```

By setting the LANG environment variable to de_DE, the abbreviation for the day "Friday" becomes the customary German abbreviation. By setting LANG to es_ES, the effects are even more obvious, as the day's and the month's abbreviations have changed to Spanish (as well as capitalization conventions).

An important point deserves restating. The date command did not change behavior because the bash command had an environment variable called LANG (directly). The process running the date command modified its output because it had its own environment variable called LANG. It just happened to inherit this variable from the bash shell. All processes have environment variables, not just shells.

Why didn't prince have to explicitly export the variable LANG? The variable is already an environment variable, set by startup scripts. Once a variable is an environment variable, it can be modified (and removed) using the same syntax as shell variables.

Often, users use a shorter syntax to both create and export an environment variable:

```bash
[prince@station prince]$ export EDITOR=nano
```

With this single command, prince has created, assigned, and exported the variable EDITOR.

### Listing Variables

#### Examining variables with set and env

The bash shell provides two commands for listing which variables are defined. The set command, without arguments, lists both shell variables and environment variables associated with the shell, while the env command, again without arguments, lists only variables which have been exported to the environment.

```bash
[prince@station prince]$ set
BASH=/bin/bash
BASH_VERSINFO="(\[0\]="2" \[1]="05b" \[2]="0" \[3]="1" \[4]="release" \[5]="i386-redhat-linux-gnu")
BASH_VERSION='2.05b.0(1)-release'
COLORS=/etc/DIR_COLORS.xterm
COLUMNS=80
...
[prince@station prince]$ env
HOSTNAME=localhost
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
USER=prince
MAIL=/var/spool/mail/prince
...
```

### Commonly Used Environment Variables

The following table lists some environment variables that are often used to customize a user's environment.
Table 3.3. Commonly Used Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERM</td>
<td>Specifies low level configuration of the user's terminal. This variable is more relevant when using a serial line console (&quot;dumb terminal&quot;) to access the system.</td>
</tr>
<tr>
<td>PATH</td>
<td>Specifies directories to be searched for executable files.</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>Specifies which X server clients should use in the graphical environment.</td>
</tr>
<tr>
<td>LANG</td>
<td>Specifies the preferred language for internationalized programs.</td>
</tr>
<tr>
<td>EDITOR</td>
<td>Many programs rely on an external editor for user input. Often, the default editor is vi. If the EDITOR environment variable is set, the specified editor will be used instead.</td>
</tr>
<tr>
<td>PRINTER</td>
<td>Most commands that submit or manage print jobs will examine this environment variable to determine the default printer.</td>
</tr>
</tbody>
</table>

Examples

Using Variables to Reference Commonly Used Words

The user prince likes to keep abreast of current issues relating to Open Source software, and often uses the elinks text based web browser to visit http://www.redhat.com/opensource-now/key_issues.html. Rather than repeatedly typing the long URL, prince instead modifies his ~/.bashrc file, so that the URL is saved in the variable OSNISSUES. Now prince can more easily refer to the web site.

```
[prince@station prince]$ vim .bashrc
[prince@station prince]$ cat .bashrc
# .bashrc
# User specific aliases and functions
# Source global definitions
if [ -f /etc/bashrc ]; then
  . /etc/bashrc
fi

OSNISSUES=http://www.redhat.com/opensource-now/key_issues.html
[prince@station prince]$ bash
[prince@station prince]$ elinks $OSNISSUES
```

Using http_proxy to Define a HTTP Proxy Server

Because prince is using a computer that does not have a direct connection to the Internet, he must configure his web browsers to use the proxy server found at IP address 10.1.1.1 and port 8080. While trying to figure out how to set a proxy server for the elinks text based browser, he comes across the following in the elinks(1) man page.

```
PROTOCOL_proxy       Links supports the use of proxy servers that can act as firewall gateways and caching servers. They are preferable to the older gateway servers (see WWW_access_GATEWAY, below). Each protocol used by Links, (http, ftp, gopher, etc), can be mapped separately by setting environment variables of the form PROTOCOL_proxy (literally: HTTP_proxy, FTP_proxy, HTTPS_proxy, etc), to "http://some.server.dom:port/".
```

In order to set the proxy server, he adds the following line to his ~/.bashrc file.
HTTP_proxy=http://10.1.1.1:80

He starts a new shell (so that the .bashrc file is read), and tries to access the Open Source Now website.

[prince@station prince]$ elinks http://www.redhat.com/opensourcenow/key_issues.html
Looking up www.redhat.com
www.redhat.com
Unable to locate remote host www.redhat.com
Alert!: Unable to connect to remote host.

elinks: Can't access startfile http://www.redhat.com/opensourcenow/key_issues.html

The elinks browser is apparently not trying to use the proxy server. As prince reviews his steps, he realizes that although he set the variable http_proxy, he neglected to export the variable. Because the variable is a set as a shell variable, and not an environment variable, it is not being inherited by the elinks process. He edits the line he added to his .bashrc file, prepending the word export:

[prince@station prince]$ cat .bashrc
#.bashrc
#
# Source global definitions
if [ -f /etc/bashrc ]; then
  . /etc/bashrc
fi
export HTTP_proxy=http://10.1.1.1:80

He again starts a new shell (so that the .bashrc file is read again), and tries again.

[prince@station prince]$ elinks http://www.redhat.com/opensourcenow/key_issues.html

Because the variable http_proxy is now exported as an environment variable, it is inherited by the elinks process, and elinks successfully uses the proxy server to contact the site. Because prince included the line in his ~/.bashrc file, the environment variable will be automatically set each time he starts a new shell, and prince doesn't need to worry about it again.

Appending a Directory to Your PATH

When the bash shell examines a command line, it assumes that the first word is the name of program to execute. It must next locate the file which contains the program in the filesystem. Because searching the entire filesystem for an executable file named, for example, ls, would take too long, the shell looks to the PATH environment variable for guidance.

The PATH environment variable contains a list of directories which should be searched for executable files, separated by a colon:

[prince@station prince]$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11r6/bin:/home/prince/bin

Consider running the xclock command, which starts a clock in the X graphical environment. Using the PATH variable, bash first looks for the file /bin/xclock, and not finding it, looks next for /usr/bin/xclock. The process continues until the executable file /usr/bin/X11/xclock is found.

Not all of the executable files on the system reside in directories that are listed by your PATH environment variable. Some programs are said to live "outside your path". Just because a program lives outside of your path, it does not mean that you cannot run it. It does mean, however, that you must specify the command using an absolute reference.

As an example, the lsof command lists currently open files on the system. (The name stands for List Open Files.) Because this command is usually only used by system administrators, and not "normal" users, the
command lives in the /usr/sbin directory, which falls "outside of" the default Red Hat Enterprise Linux PATH. The user prince would like to use the command to list all open files currently used by the init process.

```
[prince@station prince]$ ls -l /usr/sbin/lsof
-rwxr-xr-x    1 root     root        95640 Jan 24  2003 /usr/sbin/lsof
[prince@station prince]$ lsof -c init
-bash: lsof: command not found
```

Examining his PATH, the directory /usr/sbin is not listed, so prince next tries to run the command as an absolute reference.

```
[prince@station prince]$ /usr/sbin/lsof -c init
COMMAND PID USER  FD   TYPE DEVICE    SIZE   NODE NAME
init      1 root mem    REG    3,3   27036 245377 /sbin/init
init      1 root mem    REG    3,3  104560 244833 /lib/ld-2.3.2.so
init      1 root mem    REG    3,3 1536292 476416 /lib/tls/libc-2.3.2.so
```

Because he would rather be able to run the command directly, prince would like to add the directory /usr/sbin to his path. He uses a standard Linux (and Unix) trick to append the directory to his path.

```
[prince@station prince]$ PATH=$PATH:/usr/sbin
```

This command can be thought of as saying "set the PATH variable to be whatever it currently is, but then add :/usr/sbin. Upon examination, the PATH variable has had the directory /usr/sbin appended, and prince can now list open files the easy way.

```
[prince@station prince]$ echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/prince/bin
```

The user prince must be running on an older version of Red Hat Enterprise Linux. On a Red Hat Enterprise Linux version 6 machine, the user elvis is not having the same problems and examines his default path.

```
[elvis@station elvis]$ echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/prince/bin
```

## Online Exercises

### Lab Exercise

**Objective:** To correctly set and resolve various shell and environment variables.

**Estimated Time:** 30 mins.

### Specification

These specifications should apply to all newly started shells. Edit **bash**’s `.bashrc` startup script (found in your home directory) to include the appropriate commands.

1. A number of scripts for a new project you are about to join are being shared in the `/shared/rha_project/bin` directory. Your shell should include this directory in its search path for executable files.

2. Upon startup, your shell should create the PRINTER environment variable, which should resolve to the word `sales`. 
3. Just for kicks, upon startup, have your shell set the variable \texttt{HISTSIZE} to your shell's current process id. (What effect will this have on your shell's command history?)

4. Upon startup, your shell should create the \texttt{CLICHE} shell variable, which should resolve to the phrase \textit{that is how the cookie crumbles}. Make sure that the variable does not become an environment variable.

5. Upon startup, your shell should redirect the output of the \texttt{date} command into a file in your home directory titled \texttt{ppid_is_my_parent}, where \texttt{ppid} is replaced with your shell's parent's process id (which is stored in the shell variable \texttt{PPID}).

If you have configured your shell's \texttt{.bashrc} file correctly, you should be able to reproduce output similar to the following.

```bash
[student@station student]$ echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/prince/bin:/shared/rha_project/bin
[student@station student]$ echo $PRINTER
sales
[student@station student]$ ps
 PID TTY          TIME CMD
3914 pts/3    00:00:00 bash
3948 pts/3    00:00:00 ps
[student@station student]$ echo $HISTSIZE
3914
[student@station student]$ echo $CLICHE
that is how the cookie crumbles
[student@station student]$ echo $PPID
4293
[student@station student]$ ls
4293_is_my_parent
```

**Deliverables**

1. An appropriately configured \texttt{bash} \texttt{.bashrc} startup file, so that newly started \texttt{bash} shells have the following configuration.
   
   1. The directory \texttt{/shared/rha_project/bin} is included in the shell's search path.
   2. The environment variable \texttt{PRINTER} resolves to \texttt{sales}.
   3. The environment variable \texttt{HISTSIZE} resolves to the shell's current process id (PID).
   4. The shell variable \texttt{CLICHE} (which is not an environment variable) resolves to the phrase \textit{that is how the cookie crumbles}.
   5. Upon startup, the output of the \texttt{date} command is redirected into a file a file titled \texttt{ppid_is_my_parent} in your home directory, where \texttt{ppid} is replaced by the shell's parent's process id.

**Cleaning Up**

After your exercise has been graded, you will probably want to remove the edits you made to your \texttt{.bashrc} file. (Otherwise, you could end up with a huge history, and way too many annoying \texttt{soso_is_my_parent} files.)

**Questions**

1. Which of the following could \textit{not} be used as the name of a shell variable?
Shell Variables

1. NAME
2. PHONE_1
3. Addr2
4. ZipCode
5. All of the above could be used as the name of a shell variable.

2. Which of the following could \textit{not} be used as the name of an environment variable?

a. NAME
b. PHONE_1
c. Addr2
d. ZipCode
e. All of the above could be used as the name of an environment variable.

The user elvis reads the following paragraph from the \texttt{date(1)} man page.

\begin{verbatim}
The user elvis reads the following paragraph from the date(1) man page.

ENVIRONMENT

TZ     Specifies the timezone, unless overridden by command line parameters. If neither is specified, the setting from /etc/ localtime is used.

When he runs the date command, he discovers that the current default timezone is \texttt{EDT}.

[elvis@station elvis]$ date
Fri Sep  5 15:46:02 EDT 2003

In an effort to set his timezone to \texttt{MDT} instead, elvis appends the following line to the file \texttt{.bashrc} in his home directory.

\texttt{TZ=MDT}

3. The user elvis starts a new subshell, and runs the date command, expecting to see the date reported using the \texttt{MDT} timezone. Instead, the date command still reports the date using the \texttt{EDT} timezone. What has elvis done wrong?

a. The variable TZ is being set as a shell variable, not an environment variable.
b. In order for the changes in the \texttt{~/.bashrc} file to take effect, elvis must log off and then log back in again (instead of just starting a new subshell).
c. elvis must set executable permissions on the file \texttt{.bashrc} in his home directory.
d. elvis used the wrong syntax, and should have added the line \texttt{TZ = MDT} instead.
e. None of these explanations adequately explains his mistake.

4. What change should elvis make to correct his mistake?

a. He should Modify the line added to the \texttt{.bashrc} to read \texttt{export TZ=MDT}.

b. Run the command `chmod a+x ~/.bashrc`

c. He should modify the line added to the `.bashrc` to read `TZ=MDT; export TZ`

d. Either A or C

e. None of these steps would solve the problem.

5. Which of the following would correctly set the shell variable `ADDR` to `123 Elm St.`?

a. `ADDR= 123_Elm_St.`

b. `ADDR = 123 Elm St.`

c. `ADDR="123 Elm St."`

d. `ADDR=123_Elm_St.`

e. None of the above.

6. Which of the following is not a feature of environment variables?

a. All processes use environment variables, not just those running the `bash` shell.

b. Environment variables are inherited by child processes by default.

c. Upon startup, the `bash` shell clears all previously defined environment variables.

d. Environment variables can be examined using files found in the `/proc` filesystem.

e. Environment variables possess all these features.

7. Which command would display the current shell's process id (pid)?

a. `echo $$`

b. `echo $!`

c. `echo $PID`

d. `echo $BASH_PID`

e. None of the above.

8. Which of the following commands would create a file called `jan_reports.txt`?

a. `MONTH=jan; touch $(MONTH)_report.txt`

b. `MONTH=jan; touch $MONTH_report.txt`

c. `MONTH= jan; touch $MONTH_report.txt`

d. `MONTH= jan; touch "$MONTH"_report.txt`

e. A and B

9. Which of the following commands would append the directory `/opt/bin` to the current value of the `PATH` environment variable?
Shell Variables

a. \texttt{PATH}=$\texttt{PATH}:/opt/bin$

b. \texttt{PATH}+=/opt/bin

c. \texttt{PATH}=${\texttt{PATH}}+/opt/bin$

d. A and B

e. All of the above

The user elvis performs the following command.

\texttt{[elvis@station elvis]$ STYLE=terse}

10. Which of the following commands could be used to examine the value of \texttt{STYLE}?

a. \texttt{set}

b. \texttt{env}

c. \texttt{cat /proc/$$/environ}

d. \texttt{export}

e. A and C
Chapter 4. Command Line Expansion

Key Concepts

- The **bash** shell expands certain command line metacharacters before interpreting the command.
- Tilde expansion expands tokens that begin with a tilde (~) to users home directories.
- Brace expansion expands tokens with braces ({}) into multiple words, each of which contains a single word from the specified list.
- Command substitution expands text enclosed within backticks (``) or "dollar parenthesis" ($()) into the output produced by the enclosed command.
- Double quotes ("..."), single quotes ('...'), and the backslash character can be used to protect characters from being expanded by the shell.

Discussion

Command Line Expansions

Overview

Before executing a command, the **bash** shell performs several *expansions* on the command line. Several types of **bash** expansions, such as *pathname expansion* (file globbing) and *variable expansion* have already been introduced. The following table lists types of **bash** expansions, with a discussion of each following.

<table>
<thead>
<tr>
<th>Expansion</th>
<th>Syntax</th>
<th>Expands To</th>
</tr>
</thead>
<tbody>
<tr>
<td>History</td>
<td>!</td>
<td>A previous command line</td>
</tr>
<tr>
<td>Brace</td>
<td>{}</td>
<td>Specified text</td>
</tr>
<tr>
<td>Tilde</td>
<td>~username</td>
<td>A User's home directory</td>
</tr>
<tr>
<td>Variable</td>
<td>$, ${...}</td>
<td>Shell and environment variables</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>$((...))</td>
<td>Numeric calculation</td>
</tr>
<tr>
<td>Command Substitution</td>
<td><code>...</code>, $(...)</td>
<td>Output from command run in subshell</td>
</tr>
<tr>
<td>Pathname</td>
<td>*, ?, [...], [^...]</td>
<td>Matched filenames in filesystem</td>
</tr>
</tbody>
</table>

History Expansion

History expansion, which is invoked with an exclamation point, was covered extensively in a previous Lesson. It is included here for context.

Brace Expansion

Brace expansion expands a single word into multiple words, substituting one of the "braced" elements for each word. For example, the expression `{c, b, r}at` would expand into the three words *cat bat rat*. 
Brace expansion is used most commonly for referring to (or creating) files that have common prefixes, postfixes, or path components. (Recall that several Lab Exercises have used brace expansion to quickly create a large number of directories or files.) In the following examples, prince uses brace expansion to quickly create several directories, and then subdirectories within them.

```
[prince@station prince]$ mkdir chap{01,02,03,04}
[prince@station ~]$ tree
.
|-- chap01
|  |-- html/
|  `-- text/
|-- chap02/
|  |-- html/
|  `-- text/
|-- chap03/
|  |-- html/
|  `-- text/
|-- chap04/
|  |-- html/
|  `-- text/
4 directories, 0 files
```

The user prince now has four directories.

```
[prince@station prince]$ mkdir chap{01,02,03,04}/{html,text}
[prince@station ~]$ tree
.
|-- chap01/html
|-- chap01/text
|-- chap02/html
|-- chap02/text
|-- chap03/html
|-- chap03/text
|-- chap04/html
|  |-- music
|  `-- text
|-- chap04/text
12 directories, 0 files
```

Now more directories have been added.

In the first `mkdir` command, the braced word is expanded to the four directories `chap01`, `chap02`, `chap03`, and `chap04`. In the second `mkdir` command, the (doubly) braced word is expanded into the eight directories `chap01/html`, `chap01/text`, `chap02/html`, and so on.

Unlike file globbing, the words that result from brace expansion are not matched against files in the filesystem (the files do not have to exist first). In fact, the expanded words do not have to be filenames, though in practice they often are.

### Tilde Expansion

Perhaps the simplest expansion in concept, tilde expansion merely expands a `~` or `~username` to the user's or the `username`'s home directory, as listed in the `/etc/passwd` file (or appropriate user database). In the following, prince uses tilde expansion to refer to his own and elvis's home directories, and then a subdirectory of elvis's home directory.

```
[prince@station prince]$ ls -ld ~
elvis            drwx-----x   15 elvis    elvis        4096 Jul 21 17:41 /home/elvis
[prince@station prince]$ ls -ld ~elvis
prince          drwx-----x    9 prince   prince       4096 Aug  4 06:58 /home/prince
[prince@station prince]$ ls -l ~elvis/pub
total 4
elvis           drwxrwxr-x    2 elvis    music        4096 Jul 13 05:46 music
```

Often in this course and other texts, the tilde is used to imply that a file should exist in a user's home directory, such as the file `~/.bash_history`. We now see the reason for this convention.
Variable Expansion

Variable expansion was covered extensively in the previous Lesson. Restating, the bash shell will expand (dereference) expressions of the form $VARNAME or $(VARNAME) to the value of the shell or environment variable VARNAME.

Arithmetic Expansion

The bash shell is usually considered a poor environment for numeric calculations, and arithmetic operators such as +, -, *, and / on the command line do not have their traditional mathematical meaning. The bash shell treats text wrapped within a $(...) syntax specially, however. First, variables are treated as numeric integers where appropriate, and secondly, standard mathematical operators such as +, -, *, and / are treated as such. The bash shell will "expand" the entire expression, and replace it with the numeric result. Arithmetic operators are the same as for the C programming language, and fully documented in the bash(1) man page under "ARITHMETIC EVALUATION".

In the following example, prince will use arithmetic expansion to calculate the area of a rectangle.

```bash
[prince@station prince]$ WIDTH=16
[prince@station prince]$ HEIGHT=82
[prince@station prince]$ echo $(( $WIDTH * $HEIGHT))
1312
```

The limitations of performing numeric calculations are quickly discovered, however, when prince tries to recalculate the area using a floating point number.

```bash
[prince@station prince]$ WIDTH=16.8
[prince@station prince]$ echo $(( $WIDTH * $HEIGHT))
-bash: 16.8 * 82: syntax error in expression (error token is ".8 * 82")
```

The bash shell only supports integer arithmetic.

Command Substitution

Perhaps the most complex and useful of the expansions, command substitution allows users to run arbitrary commands in a subshell and incorporate the results into the command line. The "old school" syntax for command substitution is to encapsulate the command in "back ticks" (the left leaning single quote, found on the same key as ~, next to the 1 key on most keyboards), and command substitution is often referred to as "back tick substitution" by old-timers. The more modern syntax supported by the bash shell is similar to arithmetic expansion, but with only one set of parentheses: $(subcommand)

As an example of command substitution, prince would like to create a directory that contains the date in its name. After examining the date(1) man page, he devises a format string to generate the date in a compact format.

```bash
[prince@station prince]$ date +%d%b%Y
04May2003
```

He now runs the mkdir command, using command substitution.

```bash
[prince@station prince]$ mkdir reports.$(date +%d%b%Y)
[prince@station prince]$ ls
reports.04May2003
```

Or, he could have combined the advantages of command substitution and history substitution, as in the following.

```bash
[prince@station prince]$ mkdir reports.$(!da)
mkdir reports.$(date +%d%b%Y)
[prince@station prince]$ ls
reports.04May2003
```
The **bash** shell implements command substitution by spawning a new subshell, running the command, recording the output, and exiting the subshell. The text used to invoke the command substitution is then replaced with the recorded output from the command.

**Pathname Expansion ("File Globbing")**

Pathname expansion, or "file globbing", was discussed in a previous workbook, but was not introduced as a shell expansion. We now see that pathname expansion is one of several types of expansions implemented by the **bash** shell. To review, pathname expansion syntax is summarized in the table below.

**Table 4.2. Bash Pathname Expansion ("File Globbing")**

<table>
<thead>
<tr>
<th>Character</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more characters</td>
</tr>
<tr>
<td>?</td>
<td>exactly one characters</td>
</tr>
<tr>
<td>[...]</td>
<td>exactly one of the included characters</td>
</tr>
<tr>
<td>[^...]</td>
<td>exactly one of the excluded characters</td>
</tr>
</tbody>
</table>

**Quoting and Escaping Characters**

The **bash** shell uses various punctuation characters found on the keyboard to perform many different types of expansions, redirections, and other sorts of wizardry. While powerful, there are situations where users might want to use one of these characters, without invoking any type of special behavior. To paraphrase Sigmund Freud, "Sometimes a dollar sign is just a dollar sign."

The **bash** shell provides three mechanisms for protecting characters from being interpreted by the shell: escaping, double quoting, and single quoting.

**Table 4.3. Bash Shell Quoting and Escaping**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Protects the next character from being interpreted by the shell.</td>
</tr>
<tr>
<td>&quot;...&quot;</td>
<td>Protects enclosed characters from being interpreted by the shell, with the exceptions of the $, !, and ` (backtick) characters.</td>
</tr>
<tr>
<td>'...'</td>
<td>Protects all enclosed characters from being interpreted by the shell.</td>
</tr>
</tbody>
</table>

Consider the following few examples, where prince is trying to echo a scrap of HTML back to the screen. In the first case, prince defines the variable `CAR`, and tries echoing the line without quotes.

```
[prince@station prince]$ CAR=corvette
[prince@station prince]$ echo <pre>little red $CAR</pre>
--bash: syntax error near unexpected token `newline'
```

With no quotes, **bash** interpreted the > and < characters as requests to redirect the command's output (and input). The shell got confused when it was asked to redirect the output twice. The user prince tries again, this time with double quotes.

```
[prince@station prince]$ echo "<pre>little red $CAR</pre>"
<pre>little red corvette</pre>
```

In this case, the double quotes protected the < and > characters. The dollar sign, however, was still interpreted as a marker for a variable. The user prince tries again with single quotes.

```
[prince@station prince]$ echo '<pre>little red $CAR</pre>'
```
A Note on Quotes

As we have seen, bash make use of the variety of quote related punctuation marks, assigning each a different purpose. The three different quoting styles are illustrated in the single echo command below. In order to reinforce the differences, the three quoting styles are discussed below.

Double Quotes: "..."

Single Right Leaning Quotes (Apostrophes): '...' Single quotes are the most powerful, and are used in similar situations as double quotes when you want all punctuation, including variables and command substitution, to be treated literally.

Single Left Leaning Quotes (Back Ticks): `...`

Back ticks are fundamentally different than single quotes or double quotes; they are not used for quoting. Instead, back ticks are used to invoke command substitution on the include text.

Command Line Expansion Subtleties

We have seen that bash applies a large number of command line expansions before a command is run. This statement includes a subtlety that is not often appreciated, and can sometimes lead to confusion. Shell expansions occur before the command is run. Occasionally, some commands expect arguments that contain characters that are special to the bash shell. A common example is the find command. Unless care is taken to properly quote or escape the special characters, bash might "expand them away" before the command ever sees them. The following example of the find command in action might help.

Starting from an empty directory, prince attempts to find all files that end .conf in the /etc directory.

```
[prince@station prince]$ find /etc -name *.conf
find: /etc/sysconfig/pgsql: Permission denied
/etc/sysconfig/networking/profiles/default/resolv.conf
find: /etc/default: Permission denied
/etc/X11/gdm/factory-gdm.conf
/etc/X11/gdm/gdm.conf
/etc/modprobe.conf
...
Overlooking some complaints about inaccessible directories, the command works. Next, prince creates the files `a.conf` and `b.conf` in the local directory, and tries again.

```
[prince@station prince]$ touch a.conf b.conf
[prince@station prince]$ ls
a.conf  b.conf
[prince@station prince]$ find /etc -name *.conf
find: paths must precede expression
Usage: find [path...] [expression]
```

Why did the very same command that worked a second ago, fail now? The answer, as you might expect, has to do with command line expansion.

Let's take the second case first. The **bash** shell encountered the following command.

```
find /etc -name *.conf
```

What does **bash** do first? It applies command line expansion. After examining the local directory, and finding the files `a.conf` and `b.conf`, the shell replaces the glob `*.conf` with the matching filenames, `a.conf b.conf`. This is plain old pathname expansion. After the expansion, the command looks like this.

```
find /etc -name a.conf b.conf
```

Now **bash** executes the command, which generates an error message (because the `-name` switch cannot handle two arguments).

Returning to the first command, why did it work? When implementing pathname expansion, the **bash** shell tries to help people out. If a glob “misses” (i.e., no files match the specified expression), **bash** preserves the glob. In the first case, because no files matched `*.conf`, **bash** passed the argument to the `find` command as written.  

What is the correct way to handle the situation? Quote or escape the special characters, as in the following:

```
[prince@station prince]$ find /etc -name "*.conf"
find: /etc/sysconfig/pgsql: Permission denied
find: /etc/default: Permission denied
/etc/sysconfig/networking/profiles/default/resolve.conf
/etc/sysconfig/networking/profiles/netup/resolve.conf
/etc/X11/gdm/factory-gdm.conf
/etc/X11/gdm/gdm.conf
/etc/modprobe.conf
...
```

Because the `*` has been quoted, the **bash** shell will not attempt to perform pathname expansion, and the command works as desired. The lesson to be learned is this: If you are passing a special character into a command, you should protect the character with quotes (or a backslash escape).

## Examples

### Using Brace Expansion

The user prince is setting up a directory named `ogg` where he will store music files that he has "ripped" (copied) from his favorite cd's. To set things up, he would like to create subdirectories based on different styles of music, and in each of these subdirectories create an empty file called `playlist`. In the following transcript, he uses brace expansion to speed up his work.

---

1 While convenient to interactive users, this is actually a dangerous practice. When writing scripts, the script writer might accidentally forget to escape a `*` or `?`, test the script in his home directory, and decide that the script "works fine". Eight months later, however, another unfortunate user might be using the script where the expansions actually match a file, breaking or dramatically altering the effect of the script.
Command Line Expansion

[prince@station prince]$ mkdir ogg
[prince@station prince]$ mkdir ogg/{blues,folk,rap,pop}
[prince@station prince]$ touch ogg/{blues,folk,rap,pop}/playlist
[prince@station prince]$ tree
  .
  |-- ogg
  |   |-- blues
  |   |   |-- playlist
  |   |-- folk
  |   |   |-- playlist
  |   |-- pop
  |   |   |-- playlist
  |   |-- rap
  |   |   |-- playlist
  5 directories, 4 files

Could prince have used file globbing (pathname expansion) instead? When using the `mkdir` command, file globbing would have been useless, because the directories `blues`, `folk`, etc., did not yet exist. What about if prince had used file globbing for the `touch` command?

[prince@station prince]$ touch ogg/*/playlist
touch: creating `ogg/*/playlist': No such file or directory

Although the `blues`, `folk`, etc., directories existed, none of the `playlist` files did, so the glob missed. For situations where the file might or might not exist, brace expansion tends to work better than globbing.

More Tab Completion

We have seen that the `bash` shell saves keystrokes by completing command names or filenames when a user hits the `TAB` key. The `bash` shell will also complete usernames and variables, when words begin with the `~` or `$` characters, respectively. For example, if a user entered `~el< TAB >`, `bash` could complete the token `~elvis`. Similarly, `$PA< TAB >` might be completed `$PATH`. Similarly to command and filename expansion, if the leading characters typed so far do not uniquely specify a variable (or username), `bash` politely beeps. Hitting `TAB` twice will list the possible completions.

Quoting Awkward Filenames

In an earlier Workbook, we mentioned that filenames in Linux (and Unix) could be composed of any characters except one (do you remember which one?). In that same Workbook, however, students were advised that even though you could use special characters, they are best avoided. We are in a good position now to see why. Suppose prince wants to create a single file titled Make $$$ *Fast* !!.

[prince@station prince]$ touch Make $$$ *Fast* !!
touch Make $$$ *Fast* 1
[prince@station prince]$ ls
13986$$ *Fast* 1 Make

The `touch` command complies, creating the files that `bash` asks it to make. First, because the tokens are separated by spaces, `bash` treats them as four separate words. The `bash` shell then applies its various expansions to the words.

1. The file `Make` is created easily enough.

2. The `bash` shell applies variable substitution to $$, resulting in `13986$`. (Where did the number `13986` come from?)

3. The token `*Fast*` survives with the `*`'s preserved, but that didn't have to be the case.

2 The `/` character, used to separate directories when specifying a file's path, cannot be included in a filename.
4. Lastly, ```!!``` is expanded to `1` from the user's command history, which apparently (and a bit oddly) was the previously run command.

After the expansions have been applied, the `bash` shell invokes `touch` with four arguments, so `touch` dutifully creates four files.

How do we coax `bash` into creating a file with our Linux-given rite to include spaces and punctuation in the filename? Quotes, of course.

```
[prince@station prince]$ touch 'Make $$$ *Fast* !!'
[prince@station prince]$ ls
13986$  *Fast*  l  Make  Make $$$ *Fast* !!
```

With a simple `ls`, it is difficult to distinguish between multiple files, and a single file with spaces in the name. A `ls -l` helps clarify the situation.

```
[prince@station prince]$ ls -l
total 0
-rw-rw-r--    1 prince   prince          0 Aug 31 06:19 13986$
-rw-rw-r--    1 prince   prince          0 Aug 31 06:19 *Fast*
-rw-rw-r--    1 prince   prince          0 Aug 31 06:19 l
-rw-rw-r--    1 prince   prince          0 Aug 31 06:19 Make
-rw-rw-r--    1 prince   prince          0 Aug 31 06:40 Make $$$ *Fast* !!
```

Notice that quotes really serve two purposes.

1. Quotes inhibit interpreting punctuation characters as requests for shell expansions.

2. Quotes also inhibit *word splitting*, which is how the `bash` shell composes arguments for the programs it runs. For example, the command `touch one two three` would cause `bash` to run the `touch` command with three arguments, `one`, `two`, and `three`. In contrast, the command `touch "one two three"` would cause `bash` to pass the `touch` command the single argument `one two three` (albeit one with spaces).

## Online Exercises

### Lab Exercise

**Objective:** Use various `bash` shell substitutions effectively.

**Estimated Time:** 15 mins.

### Specification

1. Configure your `~/.bashrc` file so that, upon startup, the variable `LINUX_VERS` contains the entire first line from the file `/var/log/dmesg`. (The file `/var/log/dmesg` is regenerated each time the machine is booted, so the variable should be set dynamically. Recall that the command `head -1` will display the first line of a file.)

2. In your home directory, create files with the following file names. (The contents of the files is irrelevant.)
   a. untitled file
   b. ```'*'s and ||'s```  
   c. ```>> README!! <<```
3. In your home directory, create a subdirectory called `shirts`. Within the subdirectory, create 108 files with filenames of the form `style.size.color.ext`, where each file contains one combination of values from the following table.

<table>
<thead>
<tr>
<th>Style</th>
<th>tee, crew, turtleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>XXL, XL, L, M, S, XS</td>
</tr>
<tr>
<td>Color</td>
<td>red, yellow, blue</td>
</tr>
<tr>
<td>Extension</td>
<td>info, inv</td>
</tr>
</tbody>
</table>

For example, the directory should contain files titled `tee.XXL.red.info`, `tee.XXL.red.inv`, `tee.XL.red.info`, `tee.XL.red.inv`, and so on.

**Deliverables**

1. 1. Upon shell startup, the variable `LINUX_VERS` should be set to contain the first line of the file `/var/log/dmesg`.

2. The following files should exist in the user’s home directory.
   a. untitled file
   b. **’s and ||’s
   c. >> README!! <<

3. A directory called `~/shirts`, which contains exactly 108 files, with each file of the form `style.size.color.ext`. Each filename should contain one combination of the values tabled above.

**Questions**

1. Which of the following punctuation characters is *not* associated with bash command line expansion?
   a. `{ }`
   b. `~`
   c. `$`
   d. `*`
   e. `%`

2. When evaluated by the bash shell, how many words will the following token expand to?
   `{jan,feb,mar}. {data,report}. {txt,html}`
   a. 3
   b. 8
   c. 9
   d. 12
3. Which of the following command lines would create a file called make $ at home!
   a. touch "make $ at home"
   b. touch 'make $ at home!'
   c. touch 'make $'\ at\ home\!
   d. A and C
   e. All of the above

4. In a default Red Hat Enterprise Linux installation, the token ~aristotle will most likely expand to which directory?
   a. /home
   b. /home/aristotle
   c. /tmp/aristotle
   d. /aristotle
   e. None of the above

5. When the user plato runs the following command, what is the name of the file that is written to?
   \texttt{cal > calendar.$(id -un).txt}
   a. calendar.txt
   b. calendar.plato.txt
   c. calendar.id -un.txt
   d. calendar.512.txt
   e. None of the above

6. Assuming that prince is a member of the groups prince and music, which of these commands would produce the following output?
   \texttt{I am a member of: prince music}
   a. \texttt{echo "I am a member of: $(id -Gn)"}
   b. \texttt{echo 'I am a member of: $(id -Gn)'
   c. \texttt{echo I am a member of: "id -Gn"}
   d. \texttt{echo I am a member of: `id -Gn`}
   e. A and D

In the following transcript, prince is trying to create a file whose filename contains the current time, such as \texttt{timestamp.10.23.43}. Use the following transcript to answer the next two questions.

\texttt{[prince@station prince]$ touch timestamp.'date +%H.%M.%S'}
7. Which of the following reasons most adequately explains why prince did not create the filename he desired?
   a. Because the `date` command (and its argument) contains a space, prince should have used double quotes.
   b. The `bash` shell could not locate the `date` command, and so left the command "as is" instead of performing command substitution.
   c. Because the percent characters in the `date` were not escaped, `bash` did not perform command substitution.
   d. The user prince did not use the correct syntax for command substitution.
   e. None of the above.

8. Which of the following command lines would correctly perform prince's operation?
   a. `touch timestamp.$(date +%H.%M.%S)`
   b. `touch timestamp.$((date +%H.%M.%S))`
   c. `touch timestamp.`date +%H.%M.%S`'
   d. A and C
   e. None of the above

Use the following transcript to answer the next two questions.

```
[prince@station prince]$ COLOR=red
[prince@station prince]$ ls
```

9. If prince were to next run the following command, what output would be produced?
   ```
   echo "My favorite colors are (blue,green,$COLOR) !!"
   ```
   a. My favorite colors are (blue,green,red) ls
   b. My favorite colors are blue green red !!
   c. My favorite colors are blue green $COLOR ls
   d. My favorite colors are (blue,green,$COLOR) !!
   e. My favorite colors are (blue,green,red)

10. If prince were to next run the following command, what output would be produced?
    ```
    echo 'My favorite colors are (blue,green,$COLOR) !!'
    ```
    a. My favorite colors are (blue,green,red) ls
    b. My favorite colors are (blue,green,$COLOR) !!
    c. My favorite colors are blue green red !!
    d. My favorite colors are blue green $COLOR ls
e. None of the above
Chapter 5. Shell Customizations

Key Concepts

- The **bash** shell internally implements certain simple commands which are closely related to the shell's behavior. These are referred to as **builtin** commands.
- Shell aliases create apparent commands which expand to arbitrary text.
- Shell aliases are established and examined with the **alias** command.
- Shell aliases are removed with the **unalias** command.
- The **bash** shell prompt can be customized using the **PS1** variable.
- Shell flags can can be set with the **set -f** command, and cleared with **set +f**.
- Shell options are examined, set, and unset using the **shopt** command.

Discussion

This Lesson focuses on techniques used to customize the **bash** shell, such as creating command aliases, customizing the shell prompt, and setting shell options. The Lesson begins with a topic that is not an actual customization, but is related to shell behavior, the topic of builtin commands.

Shell Builtins

When evaluating a command line, the shell treats the first word as the command. The **bash** shell implements some commands internally, meaning that the commands do not exist in the filesystem as a loadable program, but are implemented by the shell itself. These commands are called shell **builtins**. Shell builtins are usually simple commands, and related to modifying the shell itself.

In a previous Workbook, we introduced the **which** command, which will report where in the filesystem the executable file which contains the program for a particular command resides. In the following, madonna finds that the **date** command is implemented by the program found in the executable file `/bin/date`:

```
[madonna@station madonna]$ which date
/bin/date
```

What happens when madonna uses **which** to find the executable file that contains the **cd** program?

```
[madonna@station madonna]$ which cd
/usr/bin/which: no cd in (/usr/local/j2sdk1.3.1/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/home/madonna/bin)
```

According to **which**, the command **cd** does not exist as an executable in the filesystem. The **cd** command is an example of a shell builtin. A list of shell builtins, and corresponding documentation, can be viewed using the **help** command, which is itself a shell builtin.

```
[madonna@station madonna]$ help
GNU bash, version 2.05b.0(1)-release (i386-redhat-linux-gnu)
These shell commands are defined internally. Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.
...
```
Shell Customizations

The `help` command returns versioning information about the shell, mentions a couple of other places where `bash` documentation can be found, and then dumps a list of builtin commands. Notice that the list contains the command `cd`. The `help` command can be used to see detailed documentation about a particular builtin, as well.

```
[madonna@station madonna]$ help cd
```

```
cd: cd [-L|-P] [dir]
  Change the current directory to DIR. The variable $HOME is the default DIR. The variable CDPATH defines the search path for the directory containing DIR. Alternative directory names in CDPATH are separated by a colon (:). A null directory name is the same as the current directory, i.e. `.`. If DIR begins with a slash (/), then CDPATH is not used. If the directory is not found, and the ...
```

Because the `cd` command is closely related to the shell's behavior, namely, it changes the shell's current working directory, it is a good candidate for a builtin command. Several commands that you have already been using, such as `cd`, `pwd`, and `echo`, are actually shell builtins.

**Aliases**

Aliases allow users to customize the names of commands, or bind commands with commonly used arguments or switches. Once created, aliases are used as if they were any other command.

**The alias Command**

Aliases are created (and examined) using the `alias` builtin command. When creating aliases, the `alias` command uses the following syntax.

```
alias NAME=VALUE
```

This command would create an alias named `NAME` which would resolve to the value `VALUE`. The syntax should be reminiscent of that used to assign shell variables. In particular, as in variable assignment, alias assignment does not allow spaces on either side of the equals sign. Similarly, because the syntax only expects a single token after the equals sign, phrases that contain multiple words (separated by spaces) must be quoted.

In the following example, madonna establishes the alias `h` as a shortcut for the `head` command. Because the alias resolves to a single word (`head`), madonna does not need to worry about quoting the value. She then uses the new alias to examine the first several lines of the file `/etc/services`.

```
[madonna@station madonna]$ alias h=head
[madonna@station madonna]$ h /etc/services
```

```
# /etc/services:
# $Id: 010_text.dbk,v 1.3 2004/01/07 18:41:02 bowe Exp $
#
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known port number for both TCP and UDP; hence, most entries here have two entries even if the protocol doesn't support UDP operations. Updated from RFC 1700, "Assigned Numbers" (October 1994). Not all ports are included, only the more common ones.
```
In the next example, Madonna realizes that she is often listing all of the processes running on the machine with the command `ps aux`. She decides that whenever she runs the `ps` command, she would prefer the more complete output provided by `ps aux`, so she establishes an alias for the command `ps`.

```
[madonna@station madonna]$ alias ps="ps aux"
```

In this case, because she wanted the alias to resolve to a two word phrase (`ps` and `aux`), she needed to wrap the phrase in quotes (so that, upon "word splitting", the shell treats the phrase as a single word.)

The `alias` command is also used to examine currently defined aliases. If Madonna wanted to review the aliases she had established, she could simply run the command `alias` (without arguments).

```
[madonna@station madonna]$ alias
alias h='head'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias ps='ps aux'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

The `alias` command lists the aliases that Madonna established (`h` and `ps`), as well as other aliases that were established by Madonna's `bash` startup scripts (and are part of the default Red Hat Enterprise Linux configuration).

If given arguments (that do not contain an equals sign), the `alias` command will show the current alias for the argument, if any:

```
[madonna@station madonna]$ alias ps h foo
alias ps='ps aux'
alias h='head'
-bash: alias: foo: not found
```

### The `unalias` Command

Aliases can be removed with the `unalias` shell builtin command. In order to remove an alias, merely pass the alias's name as an argument to the `unalias` command. In the following, Madonna removes the alias for `ps` which she created above.

```
[madonna@station madonna]$ unalias ps
[madonna@station madonna]$ alias ps
-bash: alias: ps: not found
```

### Evaluating Aliases

When does the `bash` shell look for aliases? Unlike variables, there is no punctuation character associated with some sort of "alias expansion". Instead, the `bash` shell looks for aliases wherever it would expect a command (namely, as the first word on a command line.) If the first word is recognized as an alias, the alias is expanded, and the expansion is in turn examined for aliases. The exception is if the alias expands to a command which is the same name as the alias, in which case the shell simply performs the expansion and moves on. (Otherwise, users could easily create aliases which would put the shell into an infinite loop.)
Often, users may not even realize that they are using aliases instead of the command itself. A case in point is the default Red Hat Enterprise Linux configuration, which aliases the `ls` command to the value `ls --color=tty`. (This instructs the `ls` command to provide special control characters which colorize different types of files, but only if the command is writing to a terminal. If it has been redirected to a file, no colorization occurs).

**Running Commands**

We have now discussed several types of words that the `bash` shell considers "commands". To summarize, and provide context, the following list outlines the steps the `bash` shell follows when evaluating the first word of a command line. (The following is not the exact algorithm, which is more complicated, but serves as a useful approximation).

1. Perform any shell expansions.
2. Is the word defined as an alias? If so, expand the alias and start over (unless the alias expands to a command which has the same name as the alias, in which case expand the alias and start over, but don't expand aliases again).
3. Is the word defined as a shell function? If so, call the shell function in the current shell. (Shell functions are beyond the scope of this course, but are included here for completeness).
4. Is the word a shell builtin command? If so, execute the builtin command.
5. Does the word contain a `/`? If so, execute the file if it exists and has executable permissions.
6. If the word does not contain a `/`, look for a file of the same name in all of the directories defined in the `PATH` environment variable, in order. If a matching file exists, and has executable permissions, execute the file.

**Customizing the Bash Prompt**

The interactive `bash` shell, while repeating its loop of "listen", "evaluate", and "perform", issues a prompt every time it returns to the step "listen". The prompt is used to tell the user that the evaluation of the previous step has finished, and that the shell is waiting to be told what to do next. In the default Red Hat Enterprise Linux configuration, the prompt also provides more information, including the current username, hostname, and working directory.

The `bash` command actually has four different prompts, which are used in different situations. The two most commonly seen are the primary prompt, used whenever `bash` is ready for a new command, and the secondary prompt, used when a user presses RETURN, but the command line has obviously unfinished syntax (such as an open quotation that has not yet been closed). In the following, `[madonna@station madonna]$` serves as the primary prompt, while `>` serves as the secondary prompt.

```
[madonna@station madonna]$ echo "Little Miss Muffet
Sat on a Tuffet"
Little Miss Muffet
Sat on a Tuffet
```

**Customizing Bash Prompts with PS1 and PS2**

Users may customize `bash` prompts using the `PS1` and `PS2` shell variables, which `bash` uses to compose the two prompts. The previous example implied that the primary prompt is `bash`'s way of saying "I'm waiting", and the secondary prompt is `bash`'s way of saying "I'm still waiting". To make the point obvious, madonna will customize her prompts to say just that.

```
[madonna@station madonna]$ PS1="I'm waiting ...
```


I'm waiting ... PS2="I'm still waiting ... "
I'm waiting ... echo "Hickory Dickory Dock"
I'm still waiting ... three mice ran up the clock
Hickory Dickory Dock
three mice ran up the clock
I'm waiting ...

Immediately upon changing the value of the variable \$PS1, \$bash began using the new value as its primary prompt.

Often users would like the prompt to display useful information as well. The \$bash shell allows users to embed escape sequences into the definition of \$PS1, which \$bash replaces with dynamic information when the prompt is generated. The following table summarizes some of the more commonly used escape sequences. For a more complete list, see the \$bash(1) man page.

### Table 5.1. Common Escape Sequences Used in bash Prompts

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>audible terminal bell</td>
</tr>
<tr>
<td>\d</td>
<td>date in &quot;Weekday Month Date&quot; format</td>
</tr>
<tr>
<td>\h</td>
<td>the hostname up to the first &quot;.&quot;</td>
</tr>
<tr>
<td>\T</td>
<td>the current time in 12-hour HH:MM:SS format</td>
</tr>
<tr>
<td>\u</td>
<td>the username of the current user</td>
</tr>
<tr>
<td>\W</td>
<td>the basename of the current working directory</td>
</tr>
<tr>
<td>!</td>
<td>the history number of this command</td>
</tr>
<tr>
<td>$</td>
<td>if the effective UID is 0, a #, otherwise a $</td>
</tr>
<tr>
<td>\nnn</td>
<td>the character corresponding to the octal number nnn</td>
</tr>
</tbody>
</table>

Parameter (variable) expansion, arithmetic expansion, and command substitution are applied to the value of \$PS1 when the prompt is generated as well.

The \$\$ escape sequence may require more explanation. The \$bash shell uses this sequence to reproduce a popular feature of the original Bourne shell (/bin/sh). The default prompt of the Bourne shell is a dollar ($) for standard users, and a pound sign (#) for the root user. With the \$\$ escape sequence, a system wide default value for \$PS1 can be used, mimicking this original behavior.

When customizing the \$bash prompt, the prompt usually looks cleaner if the \$PS1 variable is defined with a trailing space.

### Bash Shell Flags and Shell Options

Two builtin commands are used to configure the shell's behavior using shell options. One is the \$set command, which is used to modify the shell's behavior using (usually) single letter flags, and the other is \$shopt, which is used to configure shell options.

#### Shell Flags: The set Builtin Command

The \$set command performs triple duty. When used with a command line switch, as is usually the case, the command is used to set or unset shell flags. A table of more commonly used flags, and their options, is found below. When called with no arguments, the \$set command displays all shell variables and their values (as discussed in a previous Lesson). The last use of the \$set builtin command is used primarily in shell scripting, and can for now be safely ignored.
Shell Customizations

Table 5.2. Shell Flags Used by the set Builtin Command.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f</td>
<td>Disable Pathname Expansion (Globbing)</td>
</tr>
<tr>
<td>-n</td>
<td>Read commands, but do not execute them (used for performing syntax checks on scripts).</td>
</tr>
<tr>
<td>-o optionname</td>
<td>Set the specified option. Some of the more common options include the following.</td>
</tr>
<tr>
<td>emacs</td>
<td>Use emacs style command line key bindings</td>
</tr>
<tr>
<td>ignoreeof</td>
<td>Do not exit shell when EOF (CTRL+D) is read</td>
</tr>
<tr>
<td>vi</td>
<td>Use vi style command line key bindings</td>
</tr>
<tr>
<td>-v</td>
<td>Print commands as they are read (useful in debugging scripts)</td>
</tr>
<tr>
<td>-x</td>
<td>Print commands after expansions have been applied (useful in debugging scripts and examining shell expansions)</td>
</tr>
<tr>
<td>-C</td>
<td>Don’t allow the shell to clobber files on redirection.</td>
</tr>
</tbody>
</table>

The set command with normal command line switch syntax (such as set -x) enables the specified flag. To disable the flag, replace the hyphen (-) with a plus sign (+) (such as set +x). The list of currently set switches can is stored in the shell variable $-.

For example, in the following, madonna will temporarily disable file globbing (by enabling the -f shell flag), and then restore file globbing (by disabling the same).

```
[madonna@station madonna]$ set -f
ls: /etc/*.conf: No such file or directory
[madonna@station madonna]$ set +f
```

Do not be concerned if you do not yet understand the effects of all of these shell flags. Instead, make sure you know how to use the set command to enable or disable a flag if need be. If a later Lesson were to say “this action can be disabled by setting the -H shell flag”, you will know how to do it.

Shell Options: The shopt Builtin Command

The bash command also has a second collection of configuration variables, which are instead referred to as “shell options”. These are set and unset using the shopt command, where shopt -s optionname sets the option optionname, and shopt -u optionname unsets the option. The command shopt optionname displays the current option state, while just shopt displays all shell options. Some of the more understandable shell options are listed in the following table.

Table 5.3. bash Shell Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdspell</td>
<td>Attempt to correct minor misspellings of directory names when using the cd builtin command.</td>
</tr>
<tr>
<td>expand_aliases</td>
<td>Enable shell aliases</td>
</tr>
</tbody>
</table>
Shell Customizations

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>extglob</td>
<td>Enable extended globbing pattern matching syntax</td>
</tr>
<tr>
<td>nocaseglob</td>
<td>Do not consider case when applying file globbing.</td>
</tr>
</tbody>
</table>

The above are given as examples. Examine the `bash(1)` man page for a complete list. In the following, madonna examines the state of the `cdspell` shell option, enables it, and then `cd`s to the `/etc` directory, with sloppy spelling.

    [madonna@station madonna]$ shopt cdspell
    cdspell         off
    [madonna@station madonna]$ shopt -s cdspell
    [madonna@station madonna]$ cd /etc
    /etc

Once there, she discovers that the shell option only effects the `cd` builtin command. The `ls` command, for example, which is not a shell builtin, is unaffected by this option. After observing this fact, madonna disables the `cdspell` option.

    [madonna@station etc]$ ls -d /ect
    ls: /ect: No such file or directory
    [madonna@station etc]$ shopt -u cdspell
    [madonna@station etc]$ cd /ect
    -bash: cd: /ect: No such file or directory

You might be wondering how all of these options are remembered, and how you remember which are modified with `set` and which with `shopt`. The answer is that they aren't. Remember though, there are two mechanisms for setting shell flags and shell options (`set` and `shopt`, respectively), and remember where to look up information about each (the `help` command, and the `bash(1)` man page).

**Examples**

**Red Hat Enterprise Linux Default Aliases**

The default Red Hat Enterprise Linux configuration provides several illustrative uses of aliases. Upon logging in the `alias` command reveals the following.

    [madonna@station madonna]$ alias
    alias l.='ls -d .* --color=tty'
    alias ll='ls -l --color=tty'
    alias ls='ls --color=tty'
    alias vi='vim'
    alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'

The first alias, `l.` is used to list all hidden files (those beginning with a `.`) in the current directory.

    [madonna@station madonna]$ l.
    .              .bash_profile   .gconfd          .gnome-desktop  .xauthxLTmDk
    ..             .bashrc         .gnome           .gtkrc
    .bash_history  .fonts.cache-1  .gnome2          .kde
    .bash_logout   .gconf          .gnome2_private  .viminfo

Notice that the `bash` shell performs pathname expansion after expanding the alias, so that `.*` is replaced with every file in the local directory that begins with a `.`.

The second alias, `ll`, provides an easier way to invoke the `ls` command with its commonly used `-l` command line switch. The third alias, `ls`, effectively changes the default behavior of the `ls` command, so that every invocation of `ls` includes the `--color=tty` command line switch.
If one considers how bash evaluates aliases, some redundancy is discovered. When a user runs `ls`, bash expands the alias to `ls -l --color=tty`. The bash shell then examines the first word of the expansion, `ls`, which is itself an alias. Expanding this alias results in `ls --color=tty -l --color=tty`. Because the alias `ls` was expanded to a command whose first word was again `ls`, bash does not attempt further alias expansions. However, the resulting expansion `ll` specifies `--color=tty` twice.

The bash "x" shell flag is useful for discovering these types of issues. When enabled, each command is echoed back to the screen (with a "+") after all aliases and expansions have been applied.

```bash
[madonna@station madonna]$ set -x
++ echo -ne '\033[0;madonna@station:~\007'
[madonna@station madonna]$ ll
+ ls --color=ttty -l --color=ttty
total 12
```

Unfortunately, enabling the "x" shell flag generates complicated lines reflecting the generation of each prompt, but the fully expanded `ls --color=ttty -l --color=ttty` is clearly seen to be what is actually executed as a result of the `ll` command.

The fourth alias, `vi`, is used to remap a commonly used command to an alternate implementation. One of the most commonly used editors in Linux (and Unix) is the `vi` editor. Red Hat Enterprise Linux ships with an enhanced version of `vi`, known as `vim` (for "Vi IMproved"). In order to make transparent use of the enhanced editor, the more commonly used `vi` command has been remapped.

The last alias creates a wrapper for the `which` command, so that aliases are reported as well as the locations of executable files found in a user's path.

### Using Aliases to Create Safer rm, mv, and cp Commands

Examination of the usages of the `rm`, `mv`, and `cp` commands reveal that they all share a common `-i` command line switch, which is used to invoke an interactive mode. When used, these commands will prompt the user for confirmation before performing an operation that would cause a loss of information (such as the `cp` or `mv` commands clobbering an already existing destination file, or any use of the `rm` command). The use of the `-i` command line switch is illustrated below.

```bash
[madonna@station madonna]$ cp -i a b
[madonna@station madonna]$ mv -i a b
[madonna@station madonna]$ rm -i b
```

Many system administrators establish default aliases for these commands, so that every invocation of these commands is an interactive invocation.

```bash
[madonna@station madonna]$ alias cp="cp -i" mv="mv -i" rm="rm -i"
```

```bash
[madonna@station madonna]$ date > a
```

In Red Hat Enterprise Linux, the root account has such aliases established by default. Standard users, however, are left to set up such protections on their own, if they desire.

Each of these commands also comes with a \texttt{-f} command line switch, to "force" the requested behavior. Because the \texttt{-f} command line switch overrides the \texttt{-i} command line switch, it can be used to remove the potentially annoying interrogation when removing a large number of files.

### Using Aliases to Create Timestamps

The user madonna is curious to know which "hidden" files and directories in her home directory are created or modified by the applications that she runs. She would like to develop an easy way to monitor which files are created or changed before or after certain operations. She knows that the \texttt{find} command implements a \texttt{-newer} criteria, which will list all files that have been modified more recently than a specified file. She would like to create two aliases. The first alias, called \texttt{ts}, will create a "timestamp" file in her local directory. The zero length file should have the name \texttt{timestamp.08:45:32}, where the time reflects the file's creation time. The second alias, called \texttt{tsc}, will bind the appropriate arguments to the \texttt{find} command to compare all files underneath the local directory to a specified file, listing all files that have been modified more recently than the specified file. Using the two aliases, she would like to be able to monitor files that have been created or changed during certain operations.

To create the first alias, she first spends a little time reading the \texttt{date(1)} man page, and determines that the following command will generate the appropriate filename for the timestamp file.

\begin{verbatim}
[madonna@station madonna]$ date +timestamp.%H:%M:%S

timestamp.19:30:23
\end{verbatim}

To create the aliases, she adds the following two lines to her \texttt{~/.bashrc} file. Note the use of command substitution to determine the current time in the first alias.

\begin{verbatim}
alias ts="touch $(date +timestamp.%H:%M:%S)"
alias tsc="find . -newer"
\end{verbatim}

She opens a new shell (so that the modified \texttt{.bashrc} takes effect), and tries out her new aliases. She first uses \texttt{ts} to create a timestamp file.

\begin{verbatim}
[madonna@station madonna]$ ts
[madonna@station madonna]$ ls
bin  hello.c  networking  timestamp.19:39:18
\end{verbatim}

The \texttt{ts} alias seems to have worked, creating the file \texttt{timestamp.19:39:18}. Next she right clicks into the background window of her Gnome desktop, and chooses "New Folder" from the popup menu. A newly created folder named \texttt{untitled folder} appears in her desktop window. She next runs the \texttt{tsc} alias, specifying her newly created timestamp file as the file to compare against.

\begin{verbatim}
[madonna@station madonna]$ tsc timestamp.19:39:18
./.gnome-desktop
./.gnome-desktop/untitled folder
./.nautilus/metafiles
./.nautilus/metafiles/file:%2F%2F%2Fhome%2Fmadonna%2F.gnome-desktop.xml
\end{verbatim}

As expected, the alias lists all files that have been modified more recently than the specified timestamp file. Noting that a file titled \texttt{untitled folder} was created in a subdirectory called \texttt{.gnome-desktop},
madonna assumes (correctly) that the ~/.gnome-desktop directory contains all of the items that are displayed on the Gnome desktop.

She wants to probe a little further, this time exploring the behavior of the trash can. She decides to create a new timestamp file, move her new folder to the trashcan, and then again look for modified files relative to her new timestamp file.

```
[madonna@station madonna]$ ts
[madonna@station madonna]$ ls
bin  hello.c  networking  timestamp.19:39:18
```

Perplexed, madonna does not see her new file. She examines the original file's modify time by performing a `ls -l`.

```
[madonna@station madonna]$ ls -l timestamp.19:39:18
-rw-rw-r--    1 madonna madonna         0 Sep  1 19:42 timestamp.19:39:18
```

Oddly, the file's modify time implies that it has been modified about three minutes after the creation time found in the file's name. Further exploring, madonna confirms the `ts` alias.

```
[madonna@station madonna]$ alias ts
alias ts='touch timestamp.19:39:18'
```

Now madonna understands the problem. She recalls the line from her ~/.bashrc file where she defined the `ts` alias:

```
alias ts="touch $(date +timestamp.%H:%M:%S)"
```

Although she intended the `date` command substitution to occur each time the `ts` alias was invoked, that's not what is happening. Instead, the command only ran once, when the shell sourced the ~/.bashrc file as part of its startup sequence. In the above line, because double quotes do not protect against command substitution, the command substitution happens before the alias is assigned, with the result seen above: the alias always touches the same file, whose name contains the time when the bash shell started.

In order to confirm her instincts, she edits her ~/.bashrc file, renaming the `ts` alias `ts_incorrect`, and adding the following line.

```
alias ts_correct='touch $(date +timestamp.%H:%M:%S)'
```

This time, the alias is defined using single quotes, protecting the command substitution from being evaluated when the alias is created. After starting a new shell (so her edited ~/.bashrc file takes effect), madonna confirms her alias definitions.

```
[radonna@station madonna] alias ts_incorrect ts_correct
alias ts_incorrect='touch timestamp.19:47:18'
alias ts_correct='touch $(date +timestamp.%H:%M:%S)'
```

Now, when using `ts_correct`, the command substitution is performed when the alias is invoked (as opposed to when it is defined). This is the behavior that madonna desired. She now continues to explore what files are modified when the Nautilus trashcan is used.

```
[radonna@station madonna] tsc timestamp.19:48:12
./.gnome-desktop
./.nautilus/metafiles
./.nautilus/metafiles/file:%2F%2Fhome%2Fmadonna%2F.gnome-desktop.xml
```

After dragging the untitled folder icon from her desktop onto the Trash icon, she again looks for modified files.
Apparenty, the folder was moved from the directory `.gnome-desktop` to the folder `.Trash`. (The file `untitled` folder itself is not listed, because moving a file from one directory to another does not affect the file's modify time, only the modify times of the directories.)

**Online Exercises**

**Lab Exercise**

**Objective:** Customize the **bash** shell.

**Estimated Time:** 15 mins.

**Specification**

Edit your `~/.bashrc` file, so that newly started **bash** shells have the following properties.

1. The alias `dir` executes the command `ls -l`.
2. The alias `globoff` disables pathname expansion (by enabling the appropriate shell flag).
3. The complimentary alias `globon` enables pathname expansion (by disabling the appropriate shell flag).
4. When redirecting output to a file, the **bash** shell will not clobber (overwrite) an already existing file.
5. The shell has the `cdspell` shell option enabled.
6. The `rm` command is aliased to `rm -i`.

**Challenge Exercise**

The following exercise may be completed as a challenge exercise. It will not be graded by the automated grading script.

Configure the primary prompt to contain the history number and exit code of the *previously* executed command, separated by a : (with no spaces). The colon should be the only colon contained in the prompt.

If appropriately configured, the prompt should have behavior similar to the following. Note carefully the behavior of the prompt after a successful command, after an unsuccessful command, and as it relates to the history number of the previously executed command.

```bash
[madonna@station madonna] 40:0 $ ls -d /tmp
/tmp
[madonna@station madonna] 41:0 $ ls -d /TMP
ls: /TMP: No such file or directory
[madonna@station madonna] 42:1 $ history
...
  41 ls -d /tmp
  42 ls -d /TMP
  43 history
[madonna@station madonna] 43:0 $ !42
ls -d /TMP
ls: /TMP: No such file or directory
[madonna@station madonna] 44:1 $
```

*The following definition meets the requirements: `PS1=’$(<i>(\! - 1)):\$?’`*
Shell Customizations

Deliverables

1. An appropriately configured ~/.bashrc file, such that newly started bash shells have the following features.

1. The alias dir executes ls -l
2. The alias globoff disables pathname expansions.
3. The alias globon enables pathname expansions.
4. When redirecting output, the shell will not clobber (overwrite) existing files.
5. The shell has the cdspell shell option enabled.
6. An alias rm which executes the command rm -i.

Questions

The user prince is trying to find the executable for the cd command. Use the following transcript to answer the next question.

[prince@station prince]$ which cd
/usr/bin/which: no cd in (/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:
/home/prince/bin)

1. Which of the following best explains why the executable for the cd command was not found?
   a. The coreutils RPM package is not installed on the system.
   b. The command lives outside of prince's path.
   c. The command is a bash builtin command.
   d. The PATH variable is misconfigured (the listed directories should be separated by spaces, not colons).
   e. None of the above.

2. Which of the following would create an alias called tf which would expand to the phrase tail -f?
   a. alias tf=tail -f
   b. tf=tail -f
   c. alias tf="tail -f"
   d. alias tf = "tail -f"
   e. Both C and D

3. Which of the following could be used to examine the current alias for tf?
   a. alias -s tf
   b. alias tf
c. `showalias tf`

d. `alias -d tf`

e. None of the above

4. Which of the following would remove the alias `tf`?

   a. `unalias tf`

   b. `alias tf`

   c. `rmalias tf`

   d. `alias -d tf`

   e. None of the above

5. The user prince has customized his prompt to look like the following. The time is the current time, and the number 613 is the history number of the current command. Which of the following commands would configure the `PS1` variable appropriately?

   (02Sep03-11:11:53) [prince@station ~] 613 $

   a. `PS1="(d) [\u@\h \w] !\# "`

   b. `PS1="($date) [\u@\h] !\$ "`

   c. `PS1="(d) [$whoami] \! $ "`

   d. `PS1="($date --iso-8601) [\u@\h \w] \! \$ "`

   e. None of the above

The following transcript shows prince attempting to login on a virtual console. Refer to it when answering the next question.

Red Hat Enterprise Linux Server release 6.1 (Santiago)
Kernel 2.6.32-131.0.15.el6.x86_64 on an x86_64

station login: `prince`
Password:
Last login: Tue Sep  9 08:43:15 on tty3

login: `prince`
`-bash: prince: command not found`

login:

6. Which of the following most adequately explains prince's problem?

   a. There is no account for the user prince.

   b. There is a syntax error in the file `~/.bashrc`, so the shell is exiting on startup.

   c. The user prince does not have a home directory on the local machine.

   d. The user prince has forgotten his password.
e. The file ~/.bashrc contains the line `PS1=\nlogin: "`

Use the following transcript to answer the next 3 questions.

```
[prince@station prince]$ ls /
+ ls --color=tty -w80 /
bin  dev  home  lost+found  misc  net  proc  sbin  srv  tmp  var
boot  etc  lib  media  mnt  opt  root  selinux  sys  usr
++ echo -ne '\033[0;prince@station:-\007'

[prince@station prince]$
```

7. What would prince expect the command **alias ls** to return?
   a. alias ls='ls --color=tty'
   b. alias ls='ls --color=tty -w80'
   c. alias ls='ls --color=tty -w80 /'
   d. alias ls='ls -w80 '
   e. -bash: alias: ls: not found

8. What command did prince apparently run to enable commands being echoed back to the screen?
   a. shopt -x
   b. set +x
   c. shopt x
   d. set -x
   e. None of the above

9. If prince were tired of his commands being echoed back to the screen, which command could he run?
   a. set -x
   b. set +x
   c. shopt x
   d. shopt -s x
   e. None of the above

10. Which of the following commands would enable the **extglob** bash shell option?
    a. shopt -s extglob
    b. shopt extglob=yes
    c. shopt extglob
    d. shopt +extglob
    e. None of the above
Chapter 6. Sourcing Shell Scripts and Shell Initialization

Key Concepts

• Shell scripts are sourced with the **source** or `.` commands.

• Shell scripts are all executed in the same shell that sources the script.

• Bash shells can be either **login** or **non-login** shells.

• Bash shells can be either **interactive** or **non-interactive** shells.

• `/etc/profile`, files within the `/etc/profile.d` directory, and `~/.bash_profile` are sourced on startup of every login shell.

• `/etc/bashrc` and `~/.bashrc` are sourced on startup of every shell.

• `~/.bash_logout` is sourced by every exiting login shell.

Discussion

So far in this Workbook, we have been using the `~/.bashrc` file as if it were the only file that could be used to customize the **bash** shell. This has allowed us to focus on the topics at hand, such as aliases and shell expansions, without complicating matters with the details of shell scripts and shell initialization, at the possible cost of leaving the student with the impression that the `~/.bashrc` file is the only file which can be used to customize the **bash** shell. If that impression has been given, the purpose of this Lesson is to take it away.

In this Lesson, we will discuss how shell scripts can be sourced, and which scripts are sourced on startup by default in Red Hat Enterprise Linux. Knowing which scripts are sourced at startup allows users to automatically apply many of the customizations learned in previous Lessons.

Sourcing Scripts

The **bash** shell allows users to collect multiple commands into a single file, and then execute the commands as if they were typed directly at that prompt. This is called **sourcing** the file. Appropriately, the command used to source a file is the **source** shell builtin command.

As an example, blondie would like to explore different styles of shell prompts. In order to help make comparisons, she uses an editor to create a text file called `prompts.script`. She adds several lines, each providing a different definition for the variable `PS1`. Lastly, she comments out all but the first line.

```
[blondie@station blondie]$ cat prompts.script
PS1="whadda you want? "
#PS1="\e\H \$(date --iso-8601) \t \[\!] \$ "
#PS1="[\\u\H \W]\$ "
```

In order to try out the first prompt, she sources the file `prompts.script`.

```
[blondie@station blondie]$ source prompts.script
```

60
whadda you want?

Upon sourcing the file, the variable PS1 is modified, as if the contents of the script were typed at the command line. This important point deserves restating: when sourcing shell scripts, the contents of the script executes within (and potentially modifies) the current shell.

Next, blondie edits prompts.script, commenting out the first line and uncommenting the second. She again sources the script.

whadda you want? cat prompts.script
#PS1="whadda you want? 
PS1="\a\u\H \$(date --iso-8601) \t \[\!] \$ 
#PS1="[\u\h \W]\$ 
whadda you want? source prompts.script
blondie@station.example.com 2003-09-02 05:55:35 [15] $

She again edits the file, commenting out the second line and uncommenting the last (which restores her prompt to the Red Hat Enterprise Linux default).

blondie@station.example.com 2003-09-02 05:57:07 [16] $ cat prompts.script
#PS1="whadda you want? 
#PS1="\a\u\H \$(date --iso-8601) \t \[\!] \$ 
PS1="[\u\h \W]\$ 

As a shortcut, the bash shell also has a builtin command called simply ., which is a synonym for the source command. This time, when sourcing the file, blondie makes use of this shortcut.

blondie@station.example.com 2003-09-02 05:57:09 [17] $ . prompts.script
[blondie@station blondie]$

(Confusingly, the word . takes on different meanings in different contexts. When . appears as the first word on a line, it is a synonym for the source command. When . appears anywhere else, it is the current directory.)

Bash Initialization: /etc/profile, ~/.profile, ~/.bashrc, and /etc/bashrc

Login, Non-login, Interactive, and Non-interactive Shells

Having learned how to source arbitrary shell scripts, we now turn our attention to bash initialization, which tends to be an overly complicated topic. After we discuss the startup sequence in detail, the process will be summarized in a more succinct form. In order to begin, it is important to distinguish between subtly different types of shells: login and non-login shells, and interactive and non-interactive shells.

As the name implies, login shells are the first shell that you see when you login to a machine. When you login in using a virtual console, or by shelling into a remote machine, your first interactive shell is a login shell. When logging in using the X graphical environment, the unseen shell which is used to kick off your X initialization is a login shell.

Every other shell is unimaginatively named a non-login shell. How often do you see non-login shells? All over the place. Obviously, if you were to manually start a subshell, the subshell would be a non-login shell. Every time you open a new terminal in an X graphical environment, the shell is a non-login shell. Every time you use command substitution, or group commands with parenthesis, or execute shell scripts (as discussed in the following Lesson), a new non-login shell is created.

Additionally, shells may be interactive shells, or non-interactive shells. Usually, manually started subshells, or the shells in newly opened X terminals, are interactive shells. Shells invoked as a result of
command substitution and the like are non-interactive shells. The following table will help distinguish different types of shells.

Table 6.1. Different Types of Bash Shells

<table>
<thead>
<tr>
<th>Context</th>
<th>Login Shell?</th>
<th>Interactive Shell?</th>
</tr>
</thead>
<tbody>
<tr>
<td>login shells from virtual consoles</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>login shells when shelling into a remote machine over the network</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>X initialization shell</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>X terminal shell</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>subshell started by manually running bash</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>subshell used in command No substitution</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>subshell used in for commands No grouped in ()’s</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>subshell used when executing No scripts</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

The above table lists the defaults for Red Hat Enterprise Linux. Different behaviors can be invoked by explicitly specifying certain command line switches. See the `bash(1)` man page for details.

### Bash Startup Scripts

What is the difference between a login and non-login shell, or an interactive and non-interactive shell? Primarily, the shells differ in which initialization files they source upon startup. The following will trace out `bash` shell initialization as it occurs in Red Hat Enterprise Linux.

**Login Shells**

Upon startup, login shells first source the file `/etc/profile`. In Red Hat Enterprise Linux, this file initializes various environment variables, such as `PATH`, `USER`, `HOSTNAME`, and `HISTSIZE`. The `/etc/profile` script then sources any files that match the pattern `/etc/profile.d/*.*.sh`.  

Next, the `bash` shell looks for a series of files in the user's home directory, and sources the first one that exists. In Red Hat Enterprise Linux, the default file is `~/.bash_profile`. The default version of this file merely appends `$HOME/bin` to a user's `PATH`, and looks for and sources the `~/.bashrc` file described below.

**Non-Login Shells (Interactive)**

As far as the `bash` program is concerned, the only file used to customize non-login shells is the `~/.bashrc` file found in a user's home directory. In the default Red Hat Enterprise Linux configuration, the only customization
this file performs is to look for and source the file /etc/bashrc.

The global /etc/bashrc file performs several operations, including setting the user's default umask and defining the PS1 variable (which defines the shell's prompt).

Non-Login Shells (Non-Interactive)

Non-interactive, non-login shells do not source any startup files by default. (Although this behavior may be changed by setting the BASH_ENV environment variable. See the bash(1) man page for more details.)

Figure 6.1. Bash Startup Configuration Files

Why are there Login and Non-Login Shells?

What is the reason for all of this complexity? When starting up, the various startup scripts sourced by login shells tend to spend a lot of time customizing environment variables (such as the user's PATH, HISTSIZE, etc.). When subshells are started, however, there is no need to repeat all of this work. Whenever a child process is spawned by the login shell, be it a subshell or otherwise, environment variables are copied to the child automatically (by the Linux kernel). If the subshell then sources the same startup scripts as its parent login shell, the effort of initializing environment variables would be needlessly duplicated. Therefore, in order to speed up startup times, non-login shells attempt to shortcut the process. Other shell customizations, notably aliases, are bash specific concepts, not kernel level concepts, and must be reinitialized each time a new shell is started.

When examining the startup scripts listed above, notice that the /etc/profile and ~/.bash_profile files are sourced by login shells only, while the ~/.bashrc and /etc/bashrc files are sourced by all shells (login and non-login). Conventionally, the former two files are used to configure environment variables, as they only have to be initialized once. The latter two files are conventionally used for shell specific customizations, such as aliases and shell options.

Notice the symmetry. For environment variable configuration, there is a global startup file (/etc/profile) and a local startup file (~/.bash_profile). The former can be used by the system
Sourcing Shell Scripts
and Shell Initialization

The administrator to configure system wide configuration. The latter can be used by individual users to customize their own environment. A similar situation exists with /etc/bashrc and ~/.bashrc, as well.

### Table 6.2. Bash Startup Configuration Files

<table>
<thead>
<tr>
<th>File</th>
<th>Shell Type</th>
<th>Intended Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/profile</td>
<td>Login Shells</td>
<td>Globally Useful Environment Variables</td>
</tr>
<tr>
<td>/etc/profile.d/* .sh</td>
<td>Login Shells</td>
<td>Package Specific Environment Variables</td>
</tr>
<tr>
<td>~/.bash_profile</td>
<td>Login Shells</td>
<td>User Specific Environment Variables</td>
</tr>
<tr>
<td>~/.bashrc</td>
<td>All Interactive Shells</td>
<td>User specific aliases, shell functions, and shell options</td>
</tr>
<tr>
<td>/etc/bashrc</td>
<td>All Interactive Shells</td>
<td>Global aliases, shell functions, and shell options</td>
</tr>
</tbody>
</table>

**Why are there Interactive and Non-Interactive shells?**

Non-interactive shells are shells that are started as a result of executing a script (see next Lesson), using parenthesis to group commands, or using command substitution. In order to avoid potential startup script side effects, non-interactive shells do not source startup scripts upon startup.

**Exiting Login Shells: ~/.bash_logout**

Upon exiting, login shells will source the file ~/.bash_logout if it exists. In Red Hat Enterprise Linux, the file merely executes the `clear` command, so that the screen is cleared of possibly sensitive information upon exiting.

**Examples**

**The su Command, Revisited**

In an earlier Workbook, we introduced the `su` command as a way for a user to switch userid. At that point, we introduced two techniques of using the `su` command. The first was to simply `su username`, which starts a new non-login shell as the new user. The second is to `su - username`, which starts a login shell as the new user.

In order to explore the implications of these two forms, we will have blondie add the following line to her ~/.bash_profile file.

```
echo "~/.bash_profile sourcing"
```

Likewise, she adds the following line to her ~/.bashrc file.

```
echo "~/.bashrc sourcing"
```

Now we will have the user elvis `su` to blondie, using both forms.

```
[elvis@station elvis]$ su blondie
Password:
~/.bashrc sourcing
```
In the first case, just the file `~/.bashrc` file was sourced. In the second case, both `~/.bash_profile` and `~/.bashrc` are sourced.

**Explicitly sourceing /etc/profile and ~/.bash_profile**

Often in this Workbook, and in Linux (and Unix) directions in general, users are asked to "log out and log back in again" for configuration changes to take effect. Often, this is because installing new software or some other change has altered the user's (or the system's) bash initialization files, and the directions would like the user to restart the shell so that the startup files will be sourced anew.

Often, users can get away with resourcing the key configuration files instead. For example, if blondie suspects that a change has occurred in the system wide bash startup files, she could manually source the file `/etc/profile`.

```
[blondie@station blondie]$ . /etc/profile
```

Likewise, if blondie suspects that her local bash startup files have changed, then she can manually source her local bash configuration file.

```
[blondie@station blondie]$ . ~/.bash_profile
```

(Note that this will not work for all configuration changes, but only those that relate to the bash shell. For example, if a user's group memberships have changed, then the user does have to logout and login again for the changes to take effect.)

**Online Exercises**

**Lab Exercise**

**Objective:** Configure bash startup scripts

**Estimated Time:** 10 mins.

**Specification**

1. Append the following line to your `~/.bashrc` startup file.
   ```
   echo "sourcing ~/.bashrc"
   ```

2. Append the following line to your `~/.bashrc` startup file.
   ```
   echo "sourcing ~/.bash_profile"
   ```

3. Append the following line to your `~/.bash_logout` file.
   ```
   rm -fr ~/.local/share/Trash/files/*
   ```

**Deliverables**

1.
1. A bash shell that, upon startup of a non-login shell, emits the message `sourcing ~/.bashrc`

2. A bash shell that, upon startup of a login shell, emits the messages `sourcing ~/.bashrc` and `sourcing ~/.bash_profile`.

3. A bash shell that, upon exiting a login shell, removes the (non-hidden) contents of Nautilus's trashcan.

**Questions**

1. Which of the following command lines could be used to source the file `config.script`?
   a. `.. config.script`
   b. `source config.script`
   c. `< config.script`
   d. `bash config.script`
   e. None of the above

2. Which of the following commands would apply changes made to the file `~/.bashrc` to the current shell?
   a. `./~/.bashrc`
   b. `./~/.bash_profile`
   c. A and B
   d. `./etc/profile`
   e. All of the above

3. Which of the following files are sourced when a login shell is exited?
   a. `~/.bash_logout`
   b. `~/.bash_profile`
   c. `/etc/bash_logout`
   d. A and C
   e. None of the above

4. Which files are sourced by shells started to implement command substitution?
   a. `~/.bashrc`
   b. No files are sourced.
   c. `/etc/profile`
   d. `/etc/bashrc`
5. Which of the following files are sourced when a user logs in from a virtual console?
   a. /etc/profile
   b. /etc/profile.d/less.sh
   c. ~/.bashrc
   d. All of the above
   e. None of the above

6. After installing a new RPM package, a new file is added to the /etc/profile.d directory. Which of the following commands would apply the file's configuration to the current shell?
   a. source /etc/profile
   b. . ~/.bashrc
   c. bash /etc/bashrc
   d. All of the above
   e. None of the above

The user blondie adds the following lines to the default Red Hat Enterprise Linux versions of the files ~/.bashrc and ~/.bash_profile.

```
[blondie@station blondie]$ echo 'echo "sourcing ~/.bashrc"' >> ~/.bashrc
[blondie@station blondie]$ echo 'echo "sourcing ~/.bash_profile"' >> ~/.bash_profile
```

She then starts a new bash subshell.

```
[blondie@station blondie]$ bash
sourcing ~/.bashrc
[blondie@station blondie]$
```

7. Why did blondie not see the line sourcing ~/.bash_profile?
   a. She neglected to add execute permissions to the file ~/.bash_profile.
   b. Because the subshell is a non-login shell, the file ~/.bash_profile does not get sourced.
   c. For her changes to take effect, blondie must log out and log back in again.
   d. When applying quotes to echo the command line, blondie mistakingly used command substitution.
   e. None of the above reasons apply.

When logging in from a virtual console, blondie sees the following.

```
Red Hat Enterprise Linux Server release 6.1 (Santiago)
Kernel 2.6.32-131.0.15.el6.x86_64 on an x86_64

station login: blondie
Password:
Last login: Wed Sep 3 09:54:31 on tty2
```
8. Why does the file ~/.bashrc get sourced before the file ~/.bash_profile?
   a. That is the behavior of the bash executable for login shells.
   b. The two files are sourced in order of modify time, and the file ~/.bashrc had the least recent modify time.
   c. The file ~/.bashrc is not sourced first. For login shells, the bash shell only sources the file ~/.bash_profile directly. In the default Red Hat Linux configuration, this file sources the file ~/.bashrc. The line sourcing ~/.bashrc comes before the line containing the echo command blondie added above.
   d. Neither of the files are sourced by bash directly. Instead, they are both sourced by the file /etc/login, in the order implied above.
   e. None of the above apply.

9. What permissions are needed on a file in order for it to be sourced?
   a. execute permissions
   b. read permissions
   c. write permissions
   d. A and B
   e. All of the above

10. When sourcing a file with the source builtin command, what type of subshell is invoked?
    a. A non-interactive login shell
    b. An interactive non-login shell
    c. A non-interactive non-login shell
    d. The question is misguided, because no subshell is invoked when sourcing a file (the commands in the file execute in the current shell).
    e. None of the above