

Introduction to Shell and Script Programming

Objectives

- Understand the program development cycle
- Compare UNIX/Linux shells for creating scripts
- Use shell variables, operators, and wildcard characters
- Use shell logic structures
- Employ shell scripting to create a menu

Objectives (continued)

- Use commands to help debug shell scripts
- Explain ways to customize your personal environment
- Use the *trap* command
- Develop a menu-based application

Previewing the Application

- **Shell variables** temporarily store values in memory for use by a shell script
- **Symbolic name:** consists of letters, numbers, or characters
 - References the contents of a variable
 - Often reflects a variable's purpose or contents
- Shell scripts support many **shell script operators**
- Shell scripts support **logic structures**
 - Sequential, decision, looping, and case logic
 - Also called **control structures**

Using High-Level Languages

- **High-level language:** computer language that uses English-like expressions
 - Examples:

```
ADD 1 TO COUNTER (COBOL)
```

```
counter = counter + 1; (C++)
```
- Statements are stored in a **source file**
 - Created with an editor such as vi or Emacs
- **Compiler** reads source file and converts it to machine-language instructions
 - If source file contains **syntax errors**, it cannot be converted into an executable file

Using UNIX/Linux Shell Scripts

- Shell scripts contain sequences of commands
 - Interpreted instead of compiled
 - Interpreted by UNIX/Linux shell
 - If a syntax error is encountered, execution halts
 - Must be identified as an executable file
 - Example: `$ chmod ugo+x filename <Enter>`
 - Can be run in several ways:
 - Enter name at prompt (**PATH** must be set)
 - Precede name with `./`
 - Provide an absolute path to file
 - Run less quickly than compiled programs

Prototyping an Application

- **Prototype:** running model of your application
 - Allows review of final results before committing to design
- Shell scripts can be used to create prototypes
 - Quickest and most efficient method
- After prototype is approved, script can be rewritten to run faster using a compiled language (e.g., C++)
 - If script performs well, no need to convert it to a compiled program

Using Comments

- Comment lines begin with a pound (#) symbol

```
# =====  
# Script Name: pact  
# By: Your initials  
# Date: November 2009  
# Purpose: Create temporary file, pnum, to hold the  
# count of the number of projects each  
# programmer is working on. The pnum file  
# consists of:  
# prog_num and count fields  
# =====  
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:  
%s\n",$2,$1}' > pnum  
# cut prog_num, pipe output to sort to remove duplicates  
# and get count for prog/projects.  
# output file with prog_number followed by count
```


Using Comments (continued)

- Some examples of what you might comment:
 - Script name, author(s), creation date, and purpose
 - Modification date(s) and purpose of each of them
 - Purpose and types of variables used
 - Files that are accessed, created, or modified
 - How logic structures work
 - Purpose of shell functions
 - How complex lines of code work
 - The reasons for including specific commands

The Programming Shell

- Three shells that come with most Linux distributions

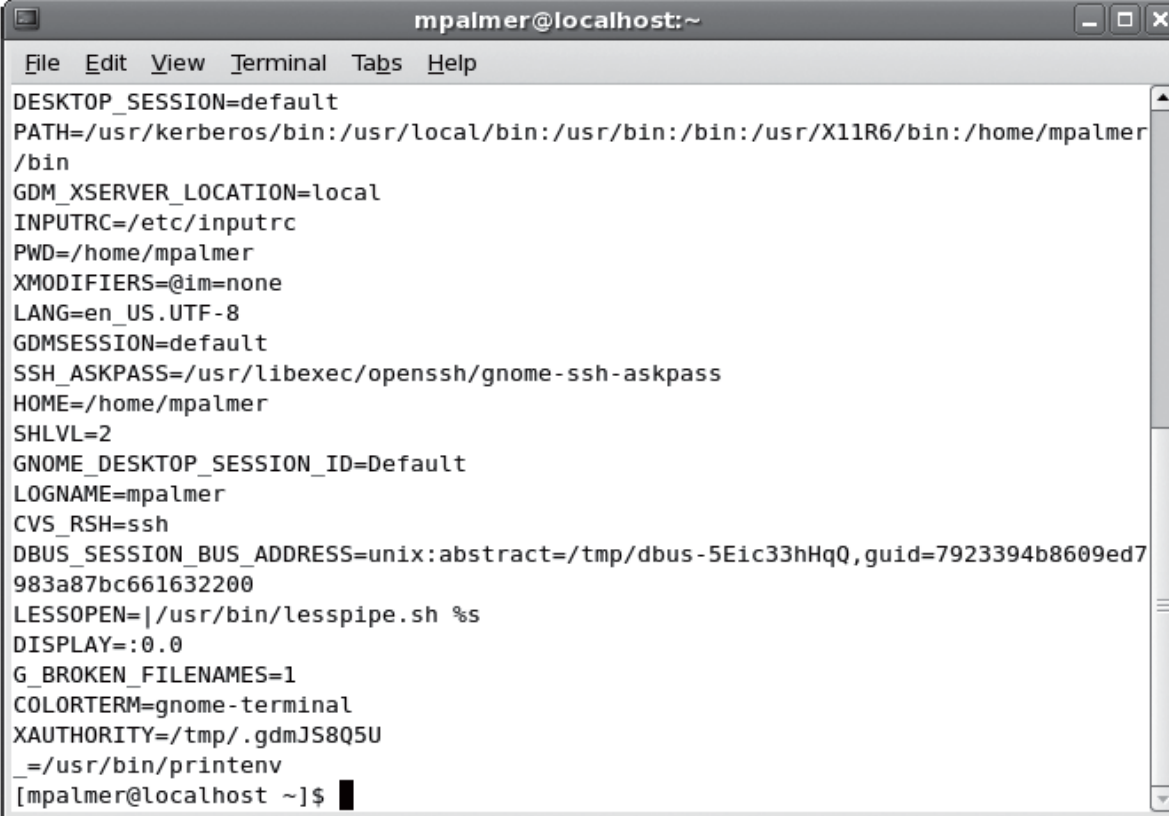
Table 6-1 Linux shells

Shell Name	Original Shell from Which Derived	Description in Terms of Shell Programming
Bash	Bourne and Korn shells	Offers strong scripting and programming language features, such as shell variables, logic structures, and math/logic expressions; combines the best features of the Bourne and Korn shells
csh/tcsh	C shell	Conforms to a scripting and programming language format; shell expressions use operators similar to those found in the C programming language
ksh/zsh	Korn shell	Is similar to the Bash shell in many respects, but also has syntax similar to that of C programming; useful if you are familiar with older Korn shell scripts

Variables

- **Configuration variables**
 - Store information about the setup of OS
 - Not typically modified after they are set up
- **Environment variables**
 - Initial values can be changed as needed
- Shell variables are created at command line or in a shell script
 - Useful for temporarily storing information

Environment and Configuration Variables

A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The window displays a list of environment variables and their values. The variables listed are: DESKTOP_SESSION=default, PATH=/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer/bin, GDM_XSERVER_LOCATION=local, INPUTRC=/etc/inputrc, PWD=/home/mpalmer, XMODIFIERS=@im=none, LANG=en_US.UTF-8, GDMSESSION=default, SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass, HOME=/home/mpalmer, SHLV=2, GNOME_DESKTOP_SESSION_ID=Default, LOGNAME=mpalmer, CVS_RSH=ssh, DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-5Eic33hHqQ,guid=7923394b8609ed7983a87bc661632200, LESSOPEN=|/usr/bin/lesspipe.sh %s, DISPLAY=:0.0, G_BROKEN_FILENAMES=1, COLORTERM=gnome-terminal, XAUTHORITY=/tmp/.gdmJS8Q5U, and _=/usr/bin/printenv. The prompt '[mpalmer@localhost ~]\$' is visible at the bottom.

```
mpalmer@localhost:~  
File Edit View Terminal Tabs Help  
DESKTOP_SESSION=default  
PATH=/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/mpalmer  
/bin  
GDM_XSERVER_LOCATION=local  
INPUTRC=/etc/inputrc  
PWD=/home/mpalmer  
XMODIFIERS=@im=none  
LANG=en_US.UTF-8  
GDMSESSION=default  
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass  
HOME=/home/mpalmer  
SHLV=2  
GNOME_DESKTOP_SESSION_ID=Default  
LOGNAME=mpalmer  
CVS_RSH=ssh  
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-5Eic33hHqQ,guid=7923394b8609ed7  
983a87bc661632200  
LESSOPEN=|/usr/bin/lesspipe.sh %s  
DISPLAY=:0.0  
G_BROKEN_FILENAMES=1  
COLORTERM=gnome-terminal  
XAUTHORITY=/tmp/.gdmJS8Q5U  
_=/usr/bin/printenv  
[mpalmer@localhost ~]$
```

Figure 6-2 Viewing the environment variable listing

Environment and Configuration Variables (continued)

- Use *printenv* to view list of your current environment and configuration variables

Syntax **printenv** [-options] [*variable name*]

Dissection

- Prints a listing of environment and configuration variables
 - Specifies one or more variables as arguments to view information only about those variables
-

Table 6-2 Standard Bash shell environment and configuration variables

Name	Variable Contents	Determined by
HOME	Identifies the path name for user's home directory	System
LOGNAME	Holds the account name of the user currently logged in	System
PPID	Refers to the parent ID of the shell	System
TZ	Holds the time zone set for use by the system	System
IFS	Enables the user to specify a default delimiter for use in working with files	Redefinable
LINEND	Holds the current line number of a function or script	Redefinable
MAIL	Identifies the name of the mail file checked by the mail utility for received messages	Redefinable
MAILCHECK	Identifies the interval for checking and received mail (example: 60)	Redefinable
PATH	Holds the list of path names for directories searched for executable commands	Redefinable
PS1	Holds the primary shell prompt	Redefinable
PS2	Contains the secondary shell prompt	Redefinable
PS3 and PS4	Holds prompts used by the <i>set</i> and <i>select</i> commands	Redefinable
SHELL	Holds the path name of the program for the type of shell you are using	Redefinable
BASH	Contains the absolute path to the Bash shell, such as <i>/bin/bash</i>	User defined
BASH_VERSION	Holds the version number of Bash	User defined
CDPATH	Identifies the path names for directories searched by the <i>cd</i> command for subdirectories	User defined
ENV	Contains the file name containing commands to initialize the shell, as in <i>.bashrc</i> or <i>.tcshrc</i>	User defined
EUID	Holds the user identification number (UID) of the currently logged in user	User defined
EXINIT	Contains the initialization commands for the <i>vi</i> editor	User defined
FCEDIT	Enables you to access a range of commands in the command history file; FCEDIT is a Bash shell utility and is the variable used to specify which editor (<i>vi</i> by default) is used when you invoke the <i>FC</i> command	User defined
FIGORE	Specifies file name suffixes to ignore when working with certain files	User defined

Table 6-2 Standard Bash shell environment and configuration variables (continued)

Name	Variable Contents	Determined by
FUNCNAME	Contains the name of the function that is running, or is empty if there is no shell function running	User defined
GROUPS	Identifies the current user's group memberships	User defined
HISTCMD	Contains the sequence number that the currently active command is assigned in the history index of commands that already have been used	User defined
HISTFILE	Identifies the file in which the history of the previously executed commands is stored	User defined
HISTFILESIZE	Sets the upward limit of command lines that can be stored in the file specified by the HISTFILE variable	User defined
HISTSIZE	Establishes the upward limit of commands that the Bash shell can recall	User defined
HOSTFILE	Holds the name of the file that provides the Bash shell with information about its network host name (such as <i>localhost.localdomain</i>) and IP address (such as <i>129.0.0.24</i>); if the HOSTFILE variable is empty, the system uses the file <i>/etc/hosts</i> by default	User defined
HOSTTYPE	Contains information about the type of computer that is hosting the Bash shell, such as <i>i386</i> for an Intel-based processor	User defined
INPUTRC	Identifies the file name for the Readline start-up file overriding the default of <i>/etc/inputrc</i>	User defined
MACHTYPE	Identifies the type of system, including CPU, operating system, and desktop	User defined
MAILPATH	Contains a list of mail files to be checked by mail for received messages	User defined
MAILWARNING	Enables (when set) the user to determine if she has already read the mail currently in the mail file	User defined
OLDPWD	Identifies the directory accessed just before the current directory	User defined
OPTIND	Shows the index number of the argument to be processed next, when a command is run using one or more option arguments	User defined
OPTARG	Contains the last option specified when a command is run using one or more option arguments	User defined

Table 6-2 Standard Bash shell environment and configuration variables (continued)

Name	Variable Contents	Determined by
OPTERR	Enables Bash to display error messages associated with command-option arguments, if set to 1 (which is the default established each time the Bash shell is invoked)	User defined
OSTYPE	Identifies the type of operating system on which Bash is running, such as linux-gnu	User defined
PROMPT_COMMAND	Holds the command to be executed prior to displaying a primary prompt	User defined
PWD	Holds the name of the directory that is currently accessed	User defined
RANDOM	Yields a random integer each time it is called, but you must first assign a value to the RANDOM variable to properly initialize random number generation	User defined
REPLY	Specifies the line to read as input, when there is no input argument passed to the built-in shell command, which is read	User defined
SHLVL	Contains the number of times Bash is invoked plus one, such as the value 3 when there are two Bash (terminal) sessions currently running	User defined
TERM	Contains the name of the terminal type in use by the Bash shell	User defined
TIMEFORMAT	Contains the timing for pipelines	User defined
TMOU	Enables Bash to stop or close due to inactivity at the command prompt, after waiting the number of seconds specified in the TMOU variable (TMOU is empty by default so that Bash does not automatically stop due to inactivity.)	User defined
UID	Holds the user identification number of the currently logged in user	User defined

Shell Variables

- Basic guidelines for handling shell variables:
 - Omit spaces when you assign a variable without using single/double quotation marks around value
 - To assign a variable that must contain spaces, enclose value in “” or ‘’
 - To reference a variable, use a \$ in front of it or enclose it in { }
 - Use [] to refer to a specific value in an array
 - Export a shell variable to make the variable available to other scripts
 - To make a shell variable read-only: *readonly fname*

Shell Variables (continued)

- Sample guidelines for naming shell variables:
 - Avoid using dollar sign in variable names
 - Use descriptive names
 - Use capitalization appropriately and consistently
 - If a variable name is to consist of two or more words, use underscores between the words

Shell Operators

- Bash shell operators are divided into four groups:
 - Defining operators
 - Evaluating operators
 - Arithmetic and relational operators
 - Redirection operators

Defining Operators

- **Defining operators:** assigns a value to a variable

- Examples:

```
NAME=Becky
```

```
NAME="Becky J. Zubrow"
```

```
LIST= `ls`
```

Evaluating Operators

- Display contents of a variable via an **evaluating operator**

- Examples:

- `echo $NAME`

- `echo "$NAME"`

- `echo '$NAME'`

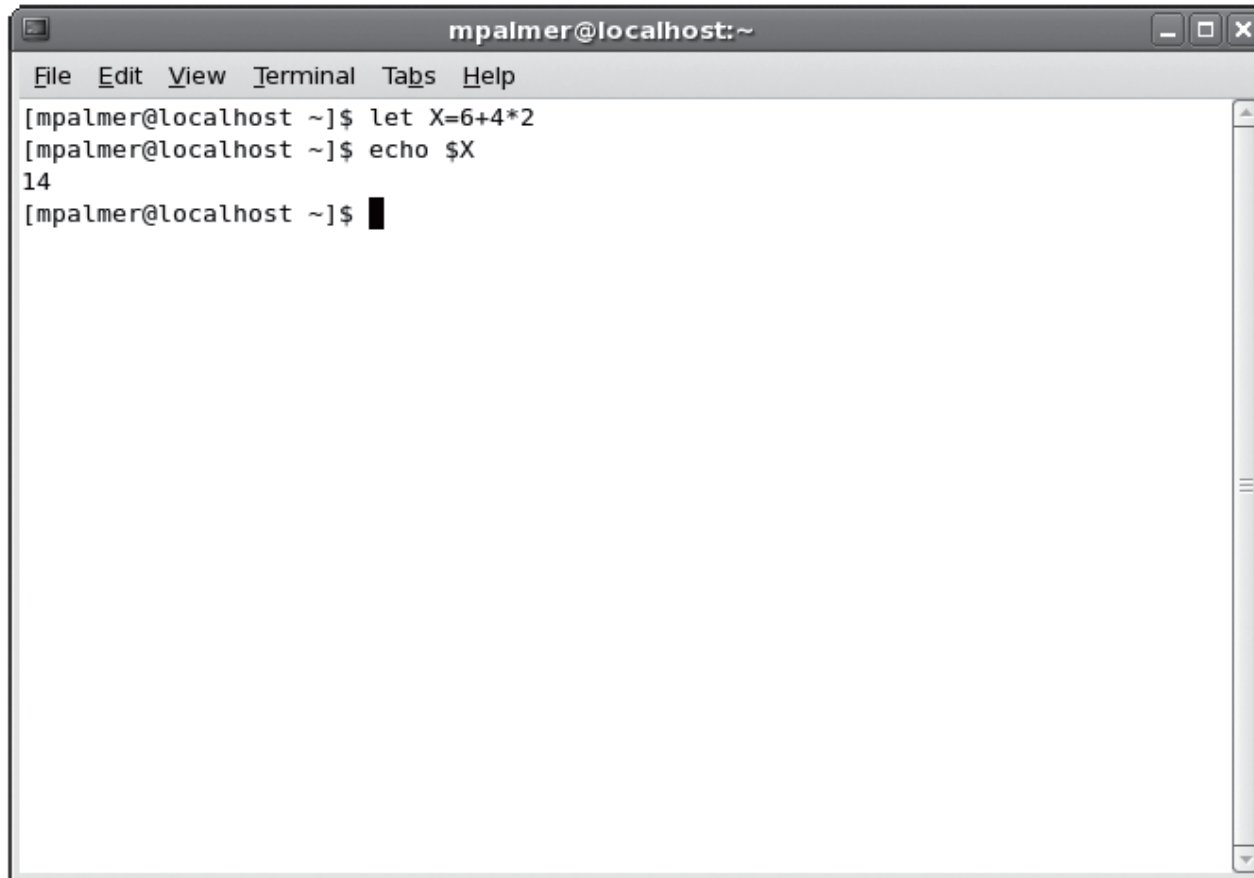
Arithmetic and Relational Operators

Table 6-3 Examples of the shell's arithmetic and relational operators

Operator	Description	Example
<code>-</code> , <code>+</code>	Unary minus and plus	<code>+R</code> (denotes positive R) <code>-R</code> (denotes negative R)
<code>!</code> , <code>~</code>	Logical and bitwise negation	<code>!Y</code> (returns 0 if Y is nonzero, returns 1 if Y is zero) <code>~X</code> (reverses the bits in X)
<code>*</code> , <code>/</code> , <code>%</code>	Multiplication, division, and remainder	<code>A * B</code> (returns A times B) <code>A / B</code> (returns A divided by B) <code>A % B</code> (returns the remainder of A divided by B)
<code>+</code> , <code>-</code>	Addition, subtraction	<code>X + Y</code> (returns X plus Y) <code>X - Y</code> (returns X minus Y)
<code>></code> , <code><</code>	Greater than and less than	<code>M > N</code> (Is M greater than N?) <code>M < N</code> (Is M less than N?)
<code>=</code> , <code>!=</code>	Equality and inequality	<code>Q = R</code> (Is Q equal to R?) <code>Q != R</code> (Is Q not equal to R?)

- The usual mathematical precedence rules apply:
 - Multiplication and division are performed before addition and subtraction

Arithmetic and Relational Operators (continued)



```
mpalmer@localhost:~  
File Edit View Terminal Tabs Help  
[mpalmer@localhost ~]$ let X=6+4*2  
[mpalmer@localhost ~]$ echo $X  
14  
[mpalmer@localhost ~]$
```

Figure 6-3 Using *let* to set the contents of a shell variable

Arithmetic and Relational Operators (continued)

Syntax **let** *expression with operators*

Dissection

- Performs a given action on numbers that is specified by operators and stores the result in a shell variable
 - Parentheses are used around specific expressions if you want to alter the mathematical precedence rules or to simply ensure the result is what you intend.
-

```
let Y=X+4
```

```
Y=$((X+4))
```

- *let* is a built-in command for the Bash shell

Redirection Operators

Syntax `set [-options] [arguments]`

Dissection

- With no options, displays the current listing of Bash environment and shell script variables
 - Useful options include:
 - a exports all variables after they are defined
 - n takes commands without executing them, so you can debug errors without affecting data (Also see the `sh -n` command later in this chapter.)
 - o sets a particular shell mode—when used with `noclobber` as the argument, it prevents files from being overwritten by use of the `>` operator
 - u shows an error when there is an attempt to use an undefined variable
 - v displays command lines as they are executed
-

- Examples:

```
$ set -o noclobber <Enter>
$ cat new_file > old_file <Enter>
bash: old_file: cannot overwrite existing
file
$ cat new_file >| old_file <Enter>
```

Exporting Shell Variables to the Environment

- Scripts cannot automatically access variables created/assigned on command line or by other scripts
 - You must use *export* first

Syntax **export** [-options] [*variable names*]

Dissection

- Makes a shell variable global so that it can be accessed by other shell scripts or programs, such as shell scripts or programs called within a shell script
 - Useful options include:
 - n undoes the export, so the variable is no longer global
 - p lists exported variables
-

Modifying the PATH Variable

- The shell looks for programs in the PATH
 - `./filename` runs script
 - `./` needed if current directory is not in PATH
- To see the directories in your path:
 - `echo $PATH`
 - Sample output:
`/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin`
- To add the current working directory to the PATH:
`PATH=$PATH:.`

More About Wildcard Characters

- Wildcard characters are known as **glob** characters
- Shell processes glob characters in unquoted arguments for file name generation
- Glob characters are part of **glob patterns**
 - Intended to match file names and words
 - Special constructions that appear in glob patterns:
 - ?
 - *
 - [*chars*]
- **Example:** `more chap[1-3] <Enter>`

Shell Logic Structures

- The four basic logic structures needed for program development:
 - Sequential logic
 - Decision logic
 - Looping logic
 - Case logic

Sequential Logic

- **Sequential logic:** commands are executed in the order in which they appear in the script or program
 - Only break in sequence logic comes when a **branch instruction** changes the flow of execution
 - Example: Programming Activity Status Report (shown on next slide)

Sequential Logic (continued)

```
#=====
# Script Name: practivity
# By: MP
# Date: November 2009
# Purpose: Generate Programmer Activity Status Report
#=====
cut -d: -f4 project | sort | uniq -c | awk '{printf "%s:
%s \n",$2,$1}' > pnum
cut -d: -f1-4 programmer | sort -t: +0 -1 | uniq > pnn
join -t: -a1 -j1 1 -j2 1 pnn pnum > pactrep
# Print the report
awk '
BEGIN {
    { FS = ":" }
    { print "\tProgrammer Activity Status Report\n" }
    { "date" | getline d }
    { printf "\t %s\n",d }
    { print "Prog# \t*--Name--* Projects\n" }
    { print "=====\n"}
}
    { printf "%-s\t%-12.12s %-12.12s %s\t%d\n",
        $1, $2, $3, $4, $5 } ' pactrep
# remove all the temporary files
rm pnum pnn pactrep
```

Decision Logic

- **Decision logic** enables your script to execute a **statement** (or series of) if a certain condition exists
 - *if* statement: primary decision-making logic structure
- Example:

```
echo -n "What is your favorite vegetable? "  
read veg_name  
if [ "$veg_name" = "broccoli" ]  
then  
    echo "Broccoli is a healthy choice."  
else  
    if [ "$veg_name" = "carrots" ]  
    then  
        echo "Carrots are great for you."  
    else  
        echo "Don't forget to eat your broccoli  also."  
    fi  
fi
```


Looping Logic

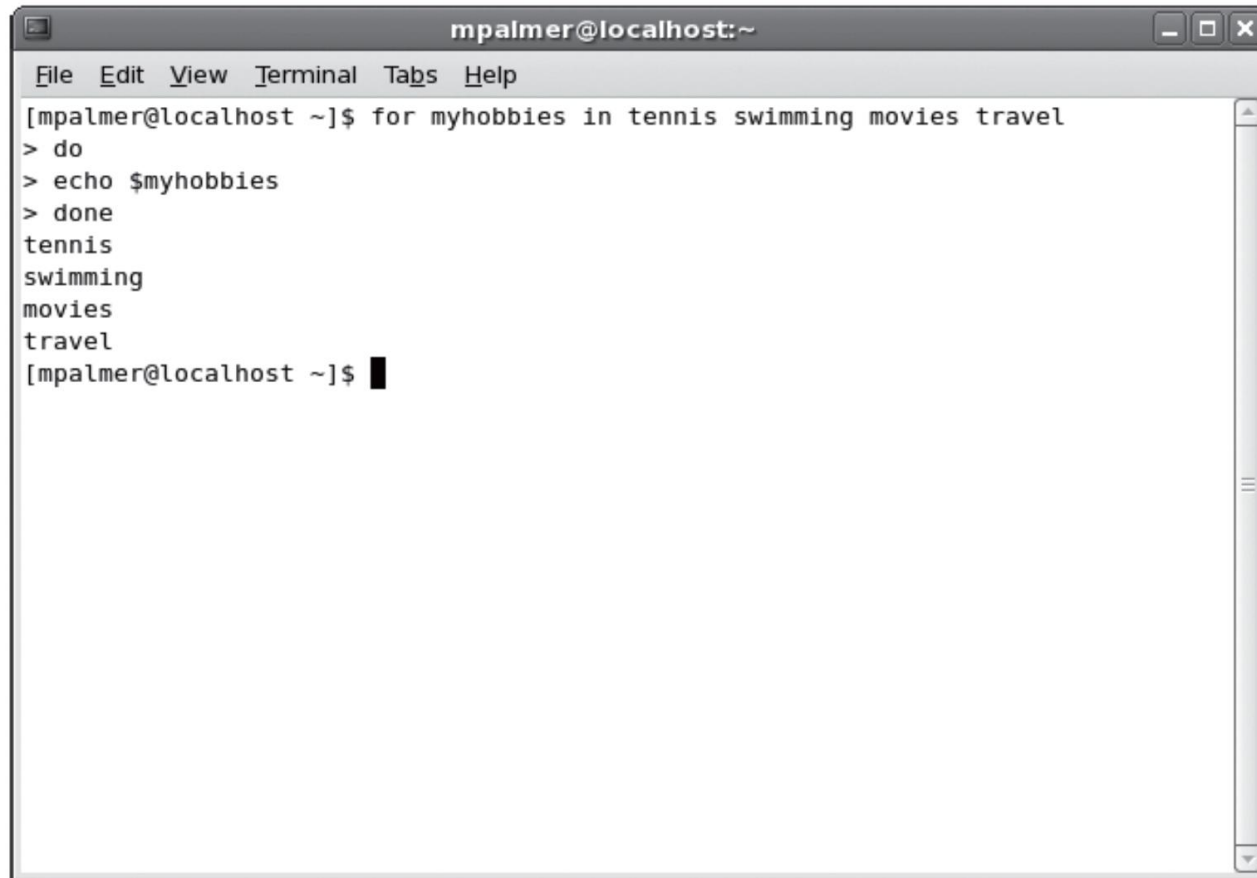
- In **looping logic**, a control structure repeats until a specific condition exists or some action occurs
- Examples:
 - *for* loop
 - *while* loop

The For Loop

- Use *for* to loop through a range of values
- In the example, the loop repeats six times

```
for USERS in john ellen tom becky eli jill  
do  
    echo $USERS  
done
```

Executing Control Structures at the Command Line

A terminal window titled 'mpalmer@localhost:~' with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows a 'for' loop iterating over the words 'tennis', 'swimming', 'movies', and 'travel'. Inside the loop, the command 'echo \$myhobbies' is executed, printing each word on a new line. The prompt '[mpalmer@localhost ~]\$' is shown at the end of the output.

```
mpalmer@localhost:~  
File Edit View Terminal Tabs Help  
[mpalmer@localhost ~]$ for myhobbies in tennis swimming movies travel  
> do  
> echo $myhobbies  
> done  
tennis  
swimming  
movies  
travel  
[mpalmer@localhost ~]$
```

Figure 6-4 Using the *for* loop from the command line

Using Wildcard Characters in a Loop

- The [] wildcard characters can be useful in loops
- Example:

```
for file in chap[1234]; do  
    more $file  
done
```

The While Loop

- *while* continues to loop and execute statements as long as condition is true
- Example:

```
echo -n "Try to guess my favorite color: "  
read guess  
while [ "$guess" != "red" ]; do  
    echo "No, not that one. Try again. "; read guess  
done
```

Case Logic

- **Case logic** structure simplifies selection of a match when you have a list of choices
 - Useful for user menus
- Example:

```
echo -n "Enter your favorite color: "; read color
case "$color" in
    "blue") echo "As in My Blue Heaven.>";;
    "yellow") echo "As in the Yellow Sunset.>";;
    "red") echo "As in Red Rover, Red Rover.>";;
    "orange") echo "As in Autumn has shades of Orange.>";;
    *) echo "Sorry, I do not know that color.>";;
esac
```

Using Shell Scripting to Create a Menu

Syntax **tput** [-options] *arguments*

Dissection

- Can be used to initialize the terminal or terminal window display, position text, and position the cursor
 - Useful options include:
 - bold*=`tput smso` *offbold*=`tput rmso` enables/disables boldfaced type
 - clear* clears the screen
 - cols* prints the number of columns
 - cup* positions the cursor and text on the screen
-

- Examples:
 - tput cup 0 0*
 - tput clear*
 - tput cols*

Debugging a Shell Script

- *sh* includes several options for debugging

Syntax **sh** [-options] [*shell script*]

Dissection

- In UNIX and Linux, it calls the command interpreter for shell scripts; and in Linux, it uses the Bash shell with the command interpreter
 - Useful options include:
 - n checks the syntax of a shell script, but does not execute command lines
 - v displays the lines of code while executing a shell script
 - x displays the command and arguments as a shell script is run
-

Customizing Your Personal Environment

- **Login script:** runs after you log in to your account
 - **.bashrc file** can be used to establish customizations that take effect for each login session
- **Alias:** name that represents another command

```
alias rm="rm -i"
```

Syntax **alias** [-options] [*name* = "*command*"]

Dissection

- Creates an alternate name for a command
 - Useful options include:
 - p prints a list of all aliases
-

Customizing Your Personal Environment (continued)

- Example:

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc # if any global definitions are defined
                  # run them first
alias rm='rm -i'  # make sure user is prompted before
                  # removing files
alias mv='mv -i'  # make sure user is prompted before
                  # overlaying files
set -o ignoreeof  # Do not allow Ctrl-d to log out
set -o noclobber  # Force user to enter >| to write
                  # over existing files
PS1="\w \ $"      # Set prompt to show working directory
```

The trap Command

Syntax `trap [command] [signal number]`

Dissection

- When a signal is received from the operating system, the argument included with *trap* is executed.
- Common signals used with *trap* include:
 - 0 The completion of a shell script has occurred
 - 1 A hang up or logout signal has been issued
 - 2 An interrupt has been received, such as Ctrl+c
 - 3 A quit signal has been issued
 - 4 An illegal instruction has been received
 - 9 A termination signal has been issued
 - 15 A program has been ended, such as through a *kill* command
 - 19 A process has been stopped
 - 20 A process has been suspended
- Useful options include:
 - l displays a listing of signal numbers and their associated signal designations

- Example: `trap "rm ~/tmp/* 2> /dev/null; exit" 0`

Putting it All Together in an Application

- Create a multifunctional application:
 - Assign and manage variables
 - Use shell operators
 - Employ shell logic structures
 - Use additional wildcard characters
 - Use *tput* for screen initialization and text placement
 - Use *trap* to clean up temporary files
- In Hands-On Projects 6-15 through 6-20, you build a multipurpose application

Summary

- A shell interprets UNIX/Linux shell scripts
 - Scripts are not interpreted, not compiled
- Shell scripts can be created with any text editor
- Linux shells are derived from the UNIX Bourne, Korn, and C shells
- UNIX/Linux employ three types of variables:
 - Configuration, environment, and shell
- The shell supports many operators
- Use wildcard characters in shell scripts

CENTENNIAL COLLEGE TWINNEN

Summary (continued)

- Logic structures supported by the shell:
 - Sequential, decision, looping, and case
- Use *tput* to manage cursor placement
- Customize your `.bashrc` file
 - Create aliases to simplify commonly used commands
- Use *trap* inside a script file to remove temporary files after the script file has been run (exited)

Command Summary

Command	Purpose	Options Covered in This Chapter
alias	Establishes an alias	-p prints all aliases.
case. . .in. . .esac	Allows one action from a set of possible actions to be performed, depending on the value of a variable	
export	Makes a shell variable an environment variable	-n can be used to undo the export. -p lists the exported variables.
for: do. . .done	Causes a variable to take on each value in a set of values; an action is performed for each value	
if. . .then. . . else. . .fi	Causes one of two actions to be performed, depending on the condition	
let	Stores arithmetic values in a variable	
printenv	Prints a list of environment variables	

Command	Purpose	Options Covered in This Chapter
set	Displays currently set shell variables; when options are used, sets the shell environment	<ul style="list-style-type: none"> -a exports all shell variables after they are assigned. -n takes commands without executing them, so you can debug errors. -o sets a particular shell mode—when used with noclobber as the argument, it prevents files from being overwritten by use of the > operator. -u yields an error message when there is an attempt to use an undefined variable. -v displays command lines as they are executed.
sh	Calls the command interpreter for shell scripts	<ul style="list-style-type: none"> -n checks the syntax of a shell script, but does not execute command lines. -v displays the lines of code while executing a shell script. -x displays the command and arguments as a shell script is run.
tput cup	Moves the screen cursor to a specified row and column	
tput clear	Clears the screen	
tput cols	Prints the number of columns on the current terminal	
tput smso	Enables boldfaced output	
tput rmso	Disables boldfaced output	
trap	Executes a command when a specified signal is received from the operating system	-l displays a listing of signal numbers and their signal designations.
while: do. . .done	Repeats an action while a condition exists	