UNIX Shell Scripting

by Mark Virtue

and The Virtual Training Company

Chapter 1

Introduction

About this Course

- This course describes how to create programs using the syntax and features of the UNIX shell – in other words, *shell scripts*
- Other useful UNIX programs will be discussed, such as:
  - `grep`
  - `test`
  - `expr`
- The techniques taught in this course will be compatible with *all* brands of UNIX (including Linux)
- There are several UNIX shells available – this course will focus on the most popular of these, the *Bourne shell*

Audience

- There are three main uses for UNIX shell scripts:
  1. *Automation of common tasks*. Most users of the UNIX shell have tasks that they wish to perform on a regular basis. Shell scripts can cut down on the time required to perform such tasks
  2. *UNIX system administration*. Much of the administration of UNIX systems is done via the use of shell scripts. Scripting expertise is a prerequisite for UNIX system administrators
  3. *CGI programming*. Interactive and data-driven web sites employ various server-side scripts for processing data and constructing web pages. These scripts are often written in the language of the Bourne shell
- Persons with needs in one or more of the above three areas will make up 90% of the audience of this course

Prerequisites

- A working knowledge of basic UNIX shell operation is essential – preferably *Bourne* shell or compatible. Specifically:
  - Logging in and out
  - Running commands and specifying command-line parameters
  - Common UNIX commands, such as `ls`, `ps`, `who`, etc
  - An understanding of file and directories
- Familiarity with some UNIX text editor (such as `vi`) is necessary
- A familiarity with programming concepts (such as *variables*, *loops*, *if-statements*, *subroutines*, etc) will be helpful, but is not necessary

Course Structure

- This course will be presented in a series of chapters. Each chapter will contain several movies
- Movies can be paused at any time. Movies should be repeated where necessary
- At the end of each chapter, some exercises will be given to help reinforce the learnings in the chapter. A worked solution to each exercise will also be provided and fully explained
- The bulk of the exercises will focus on a "course project" – a complex and useful script that we will develop in stages throughout the course

What is the UNIX Shell?

- What is a *shell*?
- A shell is simply *a program that is used to start other programs*
- All operating systems have shells

Operating System	**Shell**
DOS	`command.com`
Windows 3.1 / Windows NT 3.x	Program Manager
Windows 95 / 98 / ME / NT4 / 2000 / XP	Windows Explorer

What is the UNIX Shell? (cont.)

- A command-line shell requires that commands be entered with a particular syntax
- Command-line shells also provide certain built-in features to the user (eg. `cd` and `ls > file`)
- If the shell offers the facility to read its commands from a text file, then the aforementioned syntax and features may be thought of as a *programming language*, and such files may be thought of as *scripts*
- The MS-DOS shell `command.com` can also be thought of as such a programming language, via *batch files* (however, the syntax and built-in commands offered are far more rudimentary)

Which Shell?

- There are a variety of UNIX shells to choose from
- The original and most widely supported shell is called the *Bourne shell* (after S.R. Bourne). Its program filename is sh
- 95% of the world's shell scripts are created for use with the Bourne Shell. It is the subject of this course
- There are a number of Bourne shell derivatives, each offering a variety of extra features, including:
  - The Korn shell (after David Korn) (***ksh***) (not open/free)
  - The Bourne-again shell (***bash***) (open/free)
  - zsh
- Such shells are supposed to be Bourne-compatible

Which Shell? (cont.)

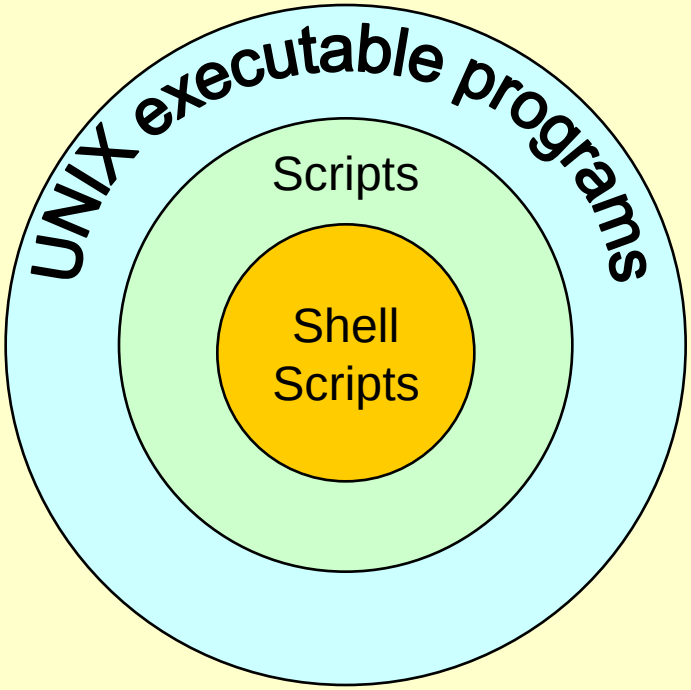
- In an effort to offer more power to shell programmers, another shell was developed in which the programming language was more closely related to the powerful “C” programming language. This shell was called the *C-shell* (filename ***csh***)
- This in turn spawned its own family of compatible shells, most notably ***tcsh***
- The Bourne and C-shell families are almost completely incompatible from a programming standpoint

What is a Shell Script?

- Some of the files on a UNIX system are deemed *executable*. These can be thought of as *programs*
- Some of these programs are compiled from source code (such as C). These are sometimes known as *binary executables*
- Others contain only text. These are *scripts*
- There are many different interpreters for scripts, such as `awk`, `sed`, `perl` and others. The *Bourne shell* is simply one such interpreter (albeit the most common)
- This is summarised on the next page

What is a Shell Script? (cont.)

- The diagram below shows the relationship between shell scripts and other UNIX executables:



Chapter 2

Your First Shell Script

A Basic Script

- The simplest of all scripts is just a text file that contains the names of other UNIX programs that you wish to run, in sequence, such as:

```
pwd
ls -C
date
```

- If this text is put into a file, and the file is made executable, the file can be executed as a script
- To make a file executable, use the `chmod` program
- For example, if the file was named `myscript`:

```
chmod +x myscript
```

A Basic Script (cont)

- To run the file as a program, simply type:

  ```
  ./myscript
  ```

- If the directory that contains the script is in your PATH, this can be abbreviated further to:

  ```
  myscript
  ```

  (we will talk more about the PATH shortly)

- *Any* UNIX command may be added to a script
- Most of this course describes internal shell commands and features that have little use outside scripts (we will also look at some UNIX programs that were written specifically to be used in conjunction with shell scripting)

The echo Command

- echo is a shell built-in command
- Its function is simple: to write its command-line parameters to Standard Output. If no parameters are given, a blank line (carriage-return) is output
- It is primarily used to display messages to the users of the script
- For example:

```
$ echo My name is Mark
My Name is Mark
$
```

The `read` Command

- `read` is a shell built-in command for reading from Standard Input (usually the keyboard) and storing the information in shell *variables*
- It is mostly used to receive the answers to questions and prompts issued by the script
- For example:

```
$ read name
Mark Virtue
$
```

- The shell variable `name` now contains the value `Mark Virtue` and can be examined by typing

```
$ echo $name
```

The `read` Command (cont.)

- `read` can break the line of input into several variables, as follows:

```
$ read firstname surname
  Mark Virtue
  $
```

- The shell variable `firstname` contains the value `Mark` and `surname` contains the value `Virtue`
- Input is separated by spaces and tabs
- If more words are provided than there are variables, the extra words are added to the last variable
- If not enough words are provided, the extra variables will contain nothing

Shell Basics Revisited

- Standard Output of any program (including scripts) can be redirected to a file using >. For example:

```
who > userlist
```

- Standard Output of any program can be *appended* to the end of a file using >>. For example:

```
cat /etc/passwd >> userlist
```

- Standard Output of any program can be "piped" to the input of any other program using |. For example:

```
who | wc –l
```

- Standard Input may be read from any file using <.

```
script < answer_file
```

Shell Basics Revisited (cont.)

- Standard Error is a separate output stream from Standard Output, and may be redirected to a file using 2>. For example:

```
find . –name fred.txt > results 2> errors
```

- Standard Output and Standard Error can be directed to the same file as follows:

```
backup > results 2>&1
```

- Any unneeded output may be redirected to the special file `/dev/null`, for example:

```
find . –name mark.doc 2> /dev/null
```

- Commands may be run *asynchronously* ("in the background") by using &. For example:

```
nohup backup &
```

Special Characters

- The following characters have a special meaning to the shell:

```
&   *   ?   [   ]   <   >   |
    (   )   `   #   $   ^   =
      '   "   {   }   ;   \
```

- You are expected to know the meaning of the characters on the top line
- We shall be learning the meanings of the rest during this course

Special Characters (cont.)

- These special characters should be avoided when naming files
  - Note that it is *never* possible to give a file a name that includes the / character (although this character is not special to the shell)
- If it ever becomes necessary to pass one of these characters as a parameter to another program, one of three actions is required:
  - Prefix the character with a \ (for example, \$)
  - Surround the character with a pair of “ characters (for example "#") Note, this doesn’t work for all characters
  - Surround the character with a pair of ‘ characters (for example '$') This works for all characters except '

Comments

- A *comment* is a piece of human-readable text added to a script to make the code more understandable
- A comment is any part of a line of a script that follows the # character
- For example:

```
#  Count the number of users on the system
who | wc -l   # wc means "word count"
```

- Comments are an important part of software development – their use dramatically cuts down on maintenance time and costs
- You are strongly encouraged to comment *all* your code

Chapter Exercises

- Create a (fully commented) script that prompts the user for their name, address and phone number. All details entered should be stored in a file that looks like this:

```
Name: Mark Virtue
Address: 123 Smith St, Smithville
Phone: 123-4567
```

  Also display a message like the following:

```
The details have been stored in "details.out"
```

- Create a (fully commented) script that produces output similar to the following:

```
Number of possible users on the system:  103
Number of users logged onto the system:   43
Total number of processes running:       167
```

Exercise Solutions

Chapter 3

Running a Shell Script

Running a Script on the Command-line

- There are four ways to run a shell script on the command-line
- For example, suppose we have a shell script called `myscript`, and it has been made executable. The four methods are:
  1. `myscript`
     This is the method described earlier. The script must be executable. A new shell is loaded into memory, which reads the commands from the file
  2. `sh myscript`
     Technically the same as method 1., but the file does not need to be made executable

Different Ways to Run a Shell Script

3. `. myscript`
   The commands in the script are executed by the *current* shell
4. `exec myscript`
   The current shell terminates, spawning a new shell in its place to execute the script. As soon as the script has terminated, the user is logged off

- Keep these methods in mind for the remainder of this course

Running a Script from within `vi`

- The following technique can dramatically speed up the script's development and testing phase if you are editing in `vi`
- While using `vi`, typing `:!` allows you to run any single UNIX command (including the script being edited)
- For example:

  `:!ls -l`<*ENTER*>
- This exact command can be repeated at any time by typing:

  `:!!`<*ENTER*>

Running a Script from within `vi`

- If, for example, the file being edited was called `myscript`, the script could thus be tested by typing:

  `:!myscript<`*ENTER*`>`

- A further shortcut offered by `vi` is that the name of the file currently being edited can be inserted into the command-line by typing the % character
- For example:

  `:!chmod +x %<`*ENTER*`>`

  or

  `:!%<`*ENTER*`>`

Your PATH and bin

- Many UNIX users like to write simple, personal shell scripts to help automate their daily routines
- These scripts are usually collected together and housed in a single directory, usually called `bin` and usually a subdirectory of the user's home directory
- Once that directory has been added to your PATH, these scripts can be run from anywhere while you are logged in
- *Recap:* Your PATH is a shell variable containing a list of directories that the shell should look in to locate executable files (separated by ":"). It is typically set in your `.profile`

Script Interpreters

- The Bourne shell is not the only program that can be used to run scripts. Others include:
  - The C shell (`csh`)
  - `sed`
  - `awk`
  - `perl`
  - `tcl`
  - And many others
- Of course, each *interpreter* has its own syntax, language and features

Script Interpreters (cont.)

- When you type in the name of a script on the command-line, the following actions are taken by your shell:
  1. The file is located and opened
  2. The first few characters are examined to determine what type of executable it is (binary or script)
  3. If it is a script (*and no interpreter is specified*), the shell attempts to interpret and run the script
- It may be that your login shell is the C shell, and the script you are trying to run was written for the Bourne shell (or vice-versa).
- It is also possible the script could be a perl script

Script Interpreters (cont.)

- To enable the shell to know what program should be run to interpret the script, the script interpreter may be specified on the first line of the script, in the following manner:

```
#!/bin/csh
```

or

```
#!/usr/local/bin/perl
```

- As we are writing Bourne shell scripts, it is strongly recommended that you add the following line to the top of every script you write:

```
#!/bin/sh
```

- This is absolutely essential for CGI programming

CGI Scripts

- A CGI program is a program that is used to provide interactivity, automation or database connectivity to a web site
- A CGI program can be any type of program, including Bourne shell scripts
- For a Bourne shell script to act as a CGI script, it must have the following attributes:
  - It must be executable
  - It must begin with the line `#!/bin/sh`
  - Any output produced must be recognised as a valid HTTP stream. This requires that Standard Output begin with a line such as the following:

```
Content-type: text/html
```

Chapter 4

Shell Programming Features

Shell Variables

- Variables are small, named pieces of memory that can be assigned values. For example:

  `month=August`

- These values can change
- Variables can be thought of as “boxes” that can contain “contents”
- Variables names:
  - Must be unique
  - Can only contain the characters:

    a-z, A-Z, 0-9 and “_”

  - Must not begin with a digit
  - Are case sensitive (`month` is a different variable from `Month`)

Shell Variables (cont.)

- (For experienced programmers: All shell variables are *strings*)
- Values may be assigned to a variable by use of the "=" sign, for example:

```
sport=basketball
```

- There must be no spaces on either side of the "="
- If you need to assign a value that contains spaces to a variable, use the " character. For example:

```
street="Smith Avenue"
```

- To retrieve the contents of a variable, use the "$" sign before the variable name:

```
echo You live on $street
```

Environment Variables

- Many shell variables are "inherited" from the login shell *environment*. In other words, they are preset variables
- For example, when running a script the following variables will be available (amongst others):
  - `HOME`
  - `PATH`
  - `LOGNAME`
  - `TERM`
- Such variables may be changed by the script, but the changes will not be seen by the login shell unless the script was run using the "`.`" operator.

Environment Variables (cont.)

- When you create a new variable, the variable is not "visible" to other programs (including other scripts) unless the variable has been added to the *environment*
- A variable is added to the environment by using the `export` command:

```
month=January
export month
```

The Trouble with Quotes

- UNIX Shell Scripting makes use of three different types of quotes:
  1. *Single* quotes (apostrophes) – the ' character
  2. *Double* quotes (quotation marks) – the " character
  3. *Back* quotes – the ` character
- As a shell programmer, sorting out the various nuances of using these different quote types will bring you hours of frustration and bug-chasing
- In this module we will learn about the different behaviours obtained by quoting text with each of these three types of quote

Single Quotes

- If you recall from Chapter 2, there are many special characters used by the shell, such as $, [, *, and #
- If text in a shell script is surrounded by single quotes, then *every character* in that text is considered to be a normal, non-special character (except another quote)
  - This includes spaces and even newline characters
- For example, the shell command:

```
$ echo 'The total is
nearly $750'
```

will cause the following output to appear on the screen:

```
The total is
nearly $750
```

Double Quotes

- Single quotes remove *all* of the shell's special-character features. Sometimes this is excessive – we may prefer *some* of the special characters to work, specifically:
  - `$` (for variable substitution, e.g. `$PATH`)
  - `` ` `` (see the next section)
  - Also, we may want the use of certain constructs, like `\"` or `\$`
- In these situations we can surround the text with *double* quotes. Other characters are still treated as special
- For example:

```
echo "$LOGNAME made \$1000 in `date +%B`"
```

produces

```
peter made $1000 in November
```

Back Quotes

- Unlike single and double quotes, the back quotes have nothing to do with special characters
- Any text enclosed in back quotes is treated as a UNIX command, and is executed in its own shell. Any output from the command is substituted into the script line, replacing the quoted text
- For example

```
list=`who | sort`
echo $list
```

produces

```
fred tty02 Aug 21 11:01 peter tty01 Aug 22
09:58 tony tty05 Aug 22 10:32
```

Grouping Commands

- Consider the various actions that may be applied to a command:
  - Run the command in the background (via the & character)
  - Redirect the output to a file or another program (> and |)
  - etc
- If you wish to apply the same action(s) to several consecutive programs, it is possible to *group* the programs together and apply the action to the group, as follows:

```
$ ( echo the date is
date ) > output_file
```

Line Control

- It is possible to run two or more UNIX commands on the same line in a shell script, by separating the commands with the `;` (semicolon) character
- For example:

```
echo Please enter your Name:; read name
```

- For aesthetic reasons, you may wish to split a command line over more than one line of text. This is achieved by quoting the newline character, using either single quotes, double quotes or the backslash character
- For example:

```
echo This command is split \
over several lines
```

Chapter Exercises

- Which of the following are valid variable names?
  A. `month`
  B. `echo`
  C. `$year`
  D. `24_hours`
  E. `hours-24`
  F. `fifty%`
  G. `First Name`
  H. `a`
  I. `_First_Name`
  J. `winner!`

Chapter Exercises (cont.)

- Create variables that contain the following values:
  A. `Fred`
  B. `Fred Smith`
  C. `That's Life`

  Ensure that you `echo` the values out to the screen to prove that you got it right
- Set a variable called `age` to `45` on the shell's command line, then write a script to echo that value to the screen
- What's wrong with the following code?

```
myname=Joe
echo 'Hello $myname, today's date
is "date +%d/%m/%y"'
```

Chapter Exercises (cont.)

- Cause all of the following output to be sent to one instance of the `more` program:
  - Today's date
  - The current directory's name
  - The contents of the current directory
  - The list of users currently logged in

  Try to make it one single line of code

Exercise Solutions

- Which of the following are valid variable names?
  A. `month` — Valid
  B. `echo` — Valid
  C. `$year` — The `$` causes the *contents* of the variable to be displayed
  D. `24_hours` — Variable names cannot start with a digit
  E. `hours-24` — Variable names cannot contain a `-`
  F. `fifty%` — Variable names cannot contain a `%`
  G. `First Name` — Variable names cannot contain a space
  H. `a` — Valid
  I. `_First_Name` — Valid
  J. `winner!` — Variable names cannot contain a `!`

Introducing the Course Project

- For the remainder of this course, we will be developing a *course project* - an application for tracking people's contact details, such as their name and address
  - This will be a menu-based program, with a main screen offering the user options to create, view, search for and delete contact details
  - All the details will be stored in a text file

Introducing the Course Project (cont.)

- Here are the specifications for the initial phase of the project:
  - The program will prompt the user for six items of data:
    - First name, Surname, Address, City, State, Zip
  - This information will be appended to the end of a text file, all on one line, separated by the `":"` character
  - There will be a variable called `fname` that will store the name of the file into which all this information will be written. The value of this variable can be something like `names.dat`
  - Before the program exits, it will display the contents of the file, as well as a count of the number of records in the file
  - The project will be fully commented

Course Project Solution

Chapter 5

Conditional Code

True and False

- When every UNIX command completes, it invisibly returns a value to the program that started it (usually the shell) informing that program of the "status" of completion of the command
- This value is a number, and is known as the *exit status* of a command
- This value is usually ignored (by the shell and by the user)
- Typically, an exit status of `0` means that the program completed with no error conditions arising, while an exit status of some other number means that some error has occurred

True and False (cont.)

- This exit status is stored in the built-in variable called "?", and can be examined at any time with the command:

  ```
  echo $?
  ```

- The contents of this variable are updated every time a program runs (including the `echo` command above)
- It is often useful for shell programmers to think of an exit status of `0` as being equivalent to the logical term *True*, and for any other exit status to be equivalent to *False*
- Note that this is exactly the *opposite* to the way some programming languages (e.g. C) treat true and false
- There even exist UNIX programs called `true` and `false` that demonstrate the use of these conventions

Conditional Command Execution

- It is possible to specify that a command in a script will only run if particular condition is met
- Such conditions are *always* expressed in terms of the exit status of another program, as follows:

```
command1 && command2
```

  means that `command2` will only run if `command1` completes with an exit status of `0`

```
command3 || command4
```

  means that `command4` will only run if `command3` completes with an exit status that is *not* `0`

Conditional Command Execution

- For example:

```
ls file1 && cp file1 /tmp
cp abc xyz && echo The file was copied okay
diff fileA fileB || echo The files are different
ls file2 || exit
```

- The only problem with these constructs is that they are very limited:
  - You can only perform *one* command if the condition is met (however, it *is* possible to group commands)
  - You cannot specify a second command to be run if the condition is *not* met

The `if` Statement

- A much more powerful (and readable) shell programming construct is the `if` statement
- It's form is as follows:

```
if command1
then
    command2
    command3
    ...
fi
```

- `command2`, `command3`, etc will only run if `command1` completes with an exit status of `0` (*true*)

The `if` Statement (cont.)

- For example:

```
if diff file1 file2 > /dev/null
then
    echo The files are the same
    rm file2
    exit
fi
```

The `else` Clause

- The `if` statement is a powerful language construct, but we still have not seen a way to either:
  - execute commands on the condition that a given command returns a *non*-zero exit status
  - execute commands if a given condition is *not* met
- There is an optional component to the `if` statement, known as the `else` clause, that will facilitate solutions to both of these problems, as follows:

```
if command1
then
    one set of commands
else
    another set of commands
fi
```

The `else` Clause (cont.)

- For example:

```
if diff file1 file2 > /dev/null
then
    echo The files are the same
    rm file2
else
    echo The files are different!
    echo Please review the differences:
    diff file1 file2
fi
```

The `else` Clause (cont.)

- We now have a way to execute commands if a given command returns a *non*-zero exit status:

```
if ls file1 > /dev/null
then
    :         #  ":" is the "do nothing" command
else
    echo The file does not exist – exiting...
    exit
fi
```

The `elif` Clause

- Often we need to write a conditional code construct in which there are more than two mutually exclusive options
- The `if` statement also offers the `elif` clause (short for *else if*), as follows:

```
if command1
then
    command set 1
elif command2
then
    command set 2
else
    command set 3
fi
```

The `elif` Clause (cont.)

- For example:

```
if ls $file > /dev/null 2>&1
then
    echo Sorry, the file already exists
elif who > $file
    echo $file now contains the user list
else
    echo Could not create $file
fi
```

- The `elif` clauses can be repeated indefinitely (however, there can only be one `else` clause)

Using `test`

- Many programming languages support the notion of comparing two values (i.e. comparing two variables, or comparing a variable with a value)
- The values may be compared and checked for:
  - equality
  - *in*equality
  - one is *greater than* the other
  - or *less than*, etc
- The Bourne shell does not natively support such comparisons, but there is another UNIX program, called `test`, that does. `test` is used extensively in shell script development

Using `test` (cont.)

- `test` is used as follows:

```
$ var1=10
$ test $var1 = 20
$ echo $?
1
```

- The sole purpose of `test` is to return an exit status appropriate to the condition being tested. This exit status is consistent with the notion of *true* and *false*. In other words, in the above example, `$var1 = 20` is considered to be *false* (1)

Using `test` (cont.)

- `test` was specifically designed for use with the `if` statement:

```
if test $var1 -gt $max
then
    echo That value is too large
fi
```

Using `test` (cont.)

- Notes:
  - In the test `$var1 = 20` example, you *must* have spaces around both sides of the "=" (contrast this with *Assigning Variables*!)
  - If a variable has not been set, or is set to nothing (e.g. `x=`), then checking this variable using `test` will cause a syntax error. This can be remedied by enclosing the variable in double quotes:

```
test "$var1" = 20
```

Using `test` (cont.)

- `test` has many useful options:
  - `test` *`value1 = value2`*
    returns *true* (`0`) if the values are equal
  - `test` *`value1 != value2`*
    returns *true* (`0`) if the values are different
  - `test` *`value1 -gt value2`*
    returns *true* (`0`) if the *`value1`* and *`value2`* are both integer (numeric) values and *`value1`* is greater than *`value2`*. Similar options include `-lt` (less than), `-ge` (greater than or equal to), and `-le` (less than or equal to)
  - `test` *`value`*
    returns *true* (`0`) if the value is non-empty
  - `test -z value`
    returns *true* (`0`) if the value is empty (zero-length)

Using `test` (cont.)

- `test` also offers many useful options for checking files and directories:
  - `test –f` *`filename`*
    returns *true* (`0`) if the given file exists and is a regular file (i.e. not a directory, device, etc)
  - `test –d` *`filename`*
    returns *true* (`0`) if the given file exists and is a directory
  - `test –s` *`filename`*
    returns *true* (`0`) if the given file exists and has a file-size greater than 0
  - `test –r|w|x` *`filename`*
    returns *true* (`0`) if the given file exists and is readable | writable | executable by the current process

Using `test` (cont.)

- The following options can be used in combination with the options mentioned above:
  - `test ! expression`
    returns *true* (`0`) if the expression is considered *false*
    (`expression` is one of the options mentioned above)
    (the "`!`" character is read as "not")
  - `test expression1 –a expression2`
    returns *true* (`0`) if the *both* expressions are true (and)
  - `test expression1 –o expression2`
    returns *true* (`0`) if the *either* expression is true (or)

Using `test` (cont.)

- Interestingly, another name (an alias) for `test` is "[", meaning that our earlier example could have been written:

```
if [ $var1 -gt $max ]
then
    echo That value is too large
fi
```

- Note the (mandatory) use of the closing "]"
- This is the way `test` is primarily used

The `case` Statement

- Once common use of the `if`/`elif`/`else` statement is to compare the value of a variable to a known set of values. If there are more than two or three values, this can involve considerable code:

```
if [ $var1 = val1 ]
then
    code for case 1
elif [ $var1 = val2 ]
then
    code for case 2
elif [ $var1 = val3 ]
then
    code for case 3
elif [ $var1 = val4 ]
then
    etc
```

The `case` Statement (cont.)

- A language construct called the `case` statement was created to make writing this sort of code easier
- The `case` statement looks like this:

```
case $var1 in
    val1)
        code for case 1
        ;;
    val2)
        code for case 2
        ;;
    val3)
        code for case 3
        ;;
esac
```

The `case` Statement (cont.)

- The `case` statement has a similar construct to the `else` clause of the `if` statement – simply create a case called *****

```
case $var1 in
      val1)
          code for case 1
          ;;
      ...
      *)
          code for any case that
          is not covered above
          ;;
  esac
```

The `case` Statement (cont.)

- It is possible to cause the same code to be executed in many different cases, as follows:

```
case $var1 in
      val1|val2|val3)
          code for cases 1-3
          ;;
      ...
```

- It is possible to use wildcard characters to match values to variables, as follows:

```
case $var1 in
      d*)
          code for anything that starts with d
          ;;
      ...
```

The `case` Statement (cont.)

- Notes on using `case`:
  - Syntactically speaking, the `case` statement is complicated – there are many language elements to remember and get right:
    - `case`
    - `in`
    - `esac`
    - `)`
    - `;;`
    - `|`
    - wildcards

    Nevertheless, it is usually preferable to using a series of `if…elif…elif…else` clauses
  - Unlike the `if` statement, the `case` statement has nothing to do with *true*/*false* and exit statuses.

Chapter Exercises

- Write a script that asks the user for their age, then compares that age to a variable called `retirement_age`. If the age they entered is greater, print out a message telling them they should be retired. Otherwise, tell them that they are still quite young
  - Make sure you correctly handle the case where the user simply presses ENTER and doesn't type in an age
- Write a script that reads the name of a directory from the keyboard and then checks to see if the name they entered is indeed a directory, and can be written to (try to do this test on one line). If all is okay, the script should create a file in the given directory called `hello`

Course Project

- Modify your contacts database script such that the user is presented with a menu of choices, as follows:
  1. Create a record
  2. View records
  3. Search for records
  4. Delete records that match a pattern
- Use a case statement to determine what choice they made, and perform the appropriate code
  - Ensure that you handle the case where the user doesn't enter anything or where they enter an invalid choice
  - Choices 3 and 4 can do nothing for the moment
  - Remember that is it now possible to view the file before anything has been written to it

Exercise Solutions

Chapter 6

Loops

The `while` Loop

- The `while` loop *repeats* the execution of a code block in the same way that the `if` statement *conditionally* executes a code block
- The syntax for the while loop is as follows:

```
while command
do
    code block
done
```

- The given code block will be repeatedly executed until the given command returns an exit status that is non-zero

The `while` Loop (cont.)

- For example:

```
read answer
while [ "$answer" != chickens ]
do
    echo That answer is incorrect
    echo Please try again
    read answer
done
```

- It is actually possible to use a while loop to process each line of the output of a command. For example:

```
who | while read user term time
do
    echo $user has been on $term since $time
done
```

`break` and `continue`

- The `break` and `continue` statements can be used to further control the execution of any loop (not just the `while` loop)
- The `break` statement will cause the loop to immediately terminate. Execution will recommence with the next line after the `done`
- The `continue` statement will cause the loop to abandon the current iteration of the loop and begin the next one (the loop condition is retested)

`break` and `continue` (cont.)

- For example:

```
while  [ "$filename" ]
do
   if [ ! -d $filename ]
   then
      echo Must be a directory
      continue
   fi
   if [ `ls $filename | wc –l` -gt 100 ]
   then
      echo stopping - There was a huge directory
      break
   fi
   # process the directory

   read filename
done
```

Numerical Calculations

- As mentioned before, the Bourne shell has no notion of a number (only strings), and as such is incapable of doing numerical calculations
- However, there is a UNIX program called `expr` which was designed specifically for this purpose
- It works like this:

```
$ expr 3 * 4
12
$ a=15
$ b=3
$ c=`expr $a / $b`
$ echo $c
5
```

Numerical Calculations (cont.)

- With this tool, we can cause loops to repeat a certain number of times, as follows:

```
read count
i = 1
while [ $i –le $count ]
do
    echo This is loop $i of $count
    i=`expr $i + 1`
done
```

The `for` Loop

- There is a second type of loop available in the Bourne shell, called the `for` loop
- It causes a variable to be set to a given sequence of values, and then executes a code block once for each value, as follows:

```
for var1 in Bread Meat Dairy Vegetables Fruit
do
   echo One of the main food groups is $var1
done
```

- Note that is different from most programming languages, where the `for` loop is used to execute code a precise number of times (based on a minimum and maximum)

The `for` Loop (cont.)

- The `for` loop is often used to run through a series of filenames, as follows:

```
for fname in *.txt
do
   [ -s $fname ] && cp $fname backups
done
```

Chapter Exercises

- Create a script that examines all the files in the current directory and displays a total which represents the sum of the sizes of each file that contains the word "foo"
- Modify the course project such that the main menu is redisplayed after each action, until the user types "q"

Exercise Solutions

Chapter 7

Text Processing

About Filters

- A *filter* is a UNIX command-line program that has the following properties:
  1. It takes either standard input or the contents of one or more files
  2. It performs some processing on the above data
  3. It produces output based upon that input
- For example, `wc` is a filter. The processing it performs is counting lines, words and characters. The output it produces is the numbers that represent those counts
- `ls`, however, is *not* a filter, because it takes no input
- Filters are used to process the data produced by other programs and the data in files

Common Filters

- The following programs are filters that are used regularly in UNIX shell scripting:

Filter	Processing done to Standard Input
`cat`	None
`more`	Pagination
`grep`	Removal of lines that do not contain certain text
`sort`	Sorting
`wc`	Counting of lines, words and/or characters
`tee`	Duplication – write to files *and* screen
`sed`	Basic editing
`awk`	Anything

grep

- `grep`, in its simplest form, is a program that displays lines of text from its input that contain a certain pattern
- Put another way, it removes – or *filters out* – lines of text from its input that do *not* contain the pattern (`grep` is thus a filter in every sense of the word)
- Its usage is as follows:

```
grep pattern [files]
```

- For example:

```
grep sincerely *.txt
```

or

```
who | grep root
```

Regular Expressions

- `grep` stands for "global regular expression parser"
- A *regular expression* is a term used to describe a set of special text-matching patterns, for example:
  - `^abc` matches any line of text that *begins* with `abc`
  - `abc$` matches any line of text that *ends* with `abc`
  - `^$` matches a blank line
  - `a*` matches any sequence of zero or more `a`'s
  - `a+` matches any sequence of one or more `a`'s
  - `c[aou]t` matches `cat`, `cot` or `cut`
  - `c.t` matches a `c`, followed by *any one character*, followed by a `t`
  - `X[a-zA-Z0-9]*X` matches any sequence (even zero-length) of letters or digits surrounded by a pair of `X`'s
  - and many more – check the `man` page for `grep` or `regexp`

Regular Expressions (cont.)

- Many programs use regular expressions, including
  - `grep`
  - `sed`
  - `awk`
  - `vi and ed/ex`

sort

- `sort` is a simple filter whose processing consists of sorting lines of text from its input
- For example:

```
sort datafile
```

or

```
who | sort
```

- Note: Many novice shell programmers make the following mistake:

```
sort file1 > file1
```

- The correct technique would be:

```
sort file1 > file2
mv file2 file1
```

sed

- `sed` (short for "stream editor") is a program for performing basic editing tasks on the output of another program or on a file (similarly to `sort`, the file itself is not changed)
- The most basic form of `sed` is as follows:

```
sed action [files]
```

- `sed` can perform several actions at a time, as follows:
  - `sed -e action1 -e action2 [files]`
  - `sed -f scriptfile [files]`
- Actions specified on the command line are almost always enclosed in single quotes to prevent shell interpretation of special characters

sed (cont.)

- The most common "action" is text substitution (using regular expressions, if necessary):
  - `s/foo/bar/` – change the first occurrence of `foo` on each line to `bar`
  - `s/foo/bar/g` – change *all* occurrences of `foo` to `bar`
- A range of line numbers can be specified to restrict these actions:
  - `1,10s/foo/bar/` or `40,$s/foo/bar/`
- Note that $ refers to the last line in the file
- Another common action is *deleting* lines:
  - `11,20d` – delete the second 10 lines of the input
  - `/hopscotch/d` – delete all lines with `hopscotch` in them

sed (cont.)

- It is also possible to delete all lines *except* for lines that match certain criteria:

```
3,$!d
/ducks/!d
```

- Note this second option is similar to `grep`'s action
- There are many more actions and options available in `sed` – check the `man` page for details

awk

- awk (which stands for *Aho, Weinberger and Kernighan* – its creators) is a text-processing tool and programming language. It is capable of virtually any conceivable text processing
- As with sed, its simplest usage is as follows:

```
awk action [files]
```

- where action is a sequence of statements enclosed in { }, each separated by a ; For example:

```
who | awk '{print $1, "is on terminal", $2}'
```

- $1, $2, $3, etc are the tokens from each line of input. Tokens are separated by spaces and tabs

awk (cont.)

- With the `-F` option, it is possible to specify the character(s) used to separate tokens:

```
awk -F : '{print $1, "home:", $6}' /etc/passwd
```

- It is possible to perform different actions on lines that match certain (regular expression) patterns:

```
awk '/Australia/ {print $1}' database
```

- It is possible to perform arithmetic on variables within awk, for example:

```
awk '{print $1,($3+$4)/$5}' database
```

- Complex sets of actions may be put in a separate script file and executed thus:

```
awk –f scriptfile inputfile
```

awk (cont.)

- And of course, it is possible to create standalone awk scripts, as follows:

```
#!/bin/awk –f
{print $1, "home:", $6}
```

  and then run it as follows:

```
myawkscript /etc/passwd
```

- awk offers a more powerful print facility, known as `printf` (similar to that used in C and C++):

```
awk 'printf("%-12s%-20s\n", $1, $6)' database
```

- Note the need for the `\n` on the end of the format string

Chapter Exercises

- Create a script that uses `grep` to find all occurrences of the following patterns in the supplied file `greptest`:
  - Lines that start with a T
  - Blank lines
  - Lines that have two or more a's anywhere in them
  - lines that have a two-or-more-digit number in them
  - (advanced) lines that have the pattern `[x,y]` in them, where `x` and `y` are any numbers

Chapter Exercises (cont.)

- Write a script that produces the following output:

```
Frederick    Aug 25 12:55
Peter        Aug 24 11:01
Joan         Aug 25  9:19
```

  The data should come from the `who` program and should be sorted alphabetically by username

- Modify the course project so that the lines in `names.dat` are printed out neatly (without the `":"`) and in *Surname* order (check the `man` page for sort to learn how to sort by a field other than the start field)

Exercise Solutions

Chapter 8

Functions

Program Structure

- Most shell scripts adhere to the following code structure:
  1. interpreter
  2. opening comment
  3. important variables
  4. functions
  5. main body of code
- An example of this is on the next page:

Program Structure (cont.)

```
#!/bin/sh
####################################
#  Program to back up a directory
####################################
dir=/home/user1/docs
backupdir=/backups/user1

backup_one_file()
{
    cp $1 $backupdir
    echo $1 has been backed up
}

for file in $dir/*
do
    [ -s $file ] && backup_one_file $file
done
```

Defining and Calling a Function

- A *function* is a named code block that may be run from any point in the program, simply by invoking its name
- Functions must be defined before they can be used
- A function is defined as follows:

```
function_name()
{
    code block
    ...
}
```

- Once a function has been defined in this way, it may be used at any time as follows:

```
function_name
```

Function Parameters

- It is possible to "pass" functions data on their command-lines that they can use in their processing. Each item of data passed to a function is called a *function parameter*
- For example:

```
backup_one_file()
{
   cp $1 $backupdir
   echo $1 has been backed up
}
```

- This function might be called as follows:

```
backup_one_file report.txt
backup_one_file logo.gif
```

Function Parameters (cont.)

- Each parameter is numbered: $1 up to $9
- Note that, unlike many programming languages, there is no reference to a function's parameters on that function's declaration line
- Some handy "shortcut" variables are created for us by the shell:
  - $* represents *all* the parameters
  - $# represents the number of parameters (e.g. 4)

Function Parameters (cont.)

- The `for` loop can be written in such as way as to loop for each parameter in a function's command-line
- For example (assuming each parameter is a filename):

```
backup_these_files()
{
    for fname       # short for "for fname in $*"
    do
        cp $fname /backup
        echo $fname has been backed up
    done
}

backup_these_files *.txt *.doc
```

Function Return Values

- Functions can return information to the calling code in two ways:
  1. output (e.g. standard output)
  2. an exit status
- Function *output*:
  - This is the combined output of all programs run within the code block of the function (e.g. who, ls, echo, etc)
  - Usually this output is simply left to go to the screen, but it may be sent to the following places:
    - Another program: `sample_func | wc -l`
    - A file: `sample_func > file1`
    - Stored in a variable: `output=`sample_func``

Function Return Values (cont.)

- Function *exit status*:
  - *All* UNIX commands have an exit status, even functions
  - If you do not specify an exit status in your function, the exit status of the last command that your function executes will be used
  - An exit status may be specified using the `return` statement:

```
return 1
```

  - For example:

```
valid_file()
{
   [ ! -r $1 ] && return 1
   grep LOGFILE $1 > /dev/null || return 1
   return 0
}
```

Function Return Values (cont.)

- This exit status may then be used in the normal manner:

```
if valid_file /logs/May21.log
then
   ...
```

- Note: The `return` statement is similar to the `exit` statement, but they should not be confused
  - The `return` statement terminates the execution of the current *function*
  - The `exit` statement terminates the execution of the current *shell*, which usually means that the shell script terminates and the user is left back at their login shell
  - Both the `exit` statement and the `return` statement take an optional parameter – a number that represents the exit status to return to the calling environment

Functions in Other Files

- As a programmer, one often creates code that is suitable to for reuse
- Any functions that you write that are suitable for reuse may be placed in a separate file and "included" in future scripts that you write using the `.` command, as follows:

```
. /home/user1/libs/script_library1
. /home/user1/libs/script_library2

func_from_first_library
func_from_second_library
```

Case Study: The `yesno` Function

- Imagine a function with the following features:
  1. Its parameters are treated as the words of a "yes/no"-style question to ask the user
  2. It asks the question, then reads an answer
  3. The answer must be one of `y`, `Y`, `yes`, `YES`, `Yes`, `n`, `N`, `No`, etc
  4. If a valid answer is not given, the function repeats steps 2 & 3
  5. The function returns an exit status of "true" (0) if they answer *yes* or "false" (1) if they answered *no*
- The function could be used as follows:

```
if yesno Do you really wish to quit now
then
    exit
fi
```

Chapter Exercises

- Write a function that calculates the average of all of its parameters and prints the average to Standard Output
  - Make sure you handle the case where no parameters are given
- Modify the course project such that:
  - The code block that gets executed for every menu option is a function (even the ones not implemented yet)
  - Copy in the code for the `yesno` function, and use it to:
    - Confirm that they want to exit the program (when they type q)
    - Confirm the addition of each record (by showing them what they're about to add)
    - After each record has been added, ask them if they would like to add another one
  - Create a `pause` function and call it after each menu option completes

Exercise Solutions

Chapter 9

Command-line Parameters

Using Command-line Parameters

- Any UNIX command can receive parameters via the command-line
- These command-line parameters are made available to the shell-script programmer through the $1…$9 variables
- These variables, along with $* and $#, may be used exactly the same way as they are used with *functions*

Advanced Command-line Control

- It is possible to reference individual command-line parameters beyond `$9` by using the `shift` command
- When `shift` is used:
  - The original value in `$1` is lost (so make sure you either use this value first, or store it in another variable. e.g. `first=$1`)
  - `$1` assumes the value of the second parameter
  - `$2` assumes the value of the third parameter

  and so on, down to:
  - `$9` assumes the value of the tenth parameter
- It is even possible to specify, as a parameter to `shift`, the number of shell script parameters you want to shift:

```
shift 4
```

Using `set --`

- While it is not possible to set an individual command-line parameter within the script (`$1`…`$9` are read-only), it *is* possible to set them all collectively, using the `set --` construct:

```
set -- file1.txt fred
```

- This sets `$1` to `file1.txt` and `$2` to `fred`
- `set --` is commonly used with the back-quotes feature:

```
set -- `who | grep fred`
```

- This is especially useful in setting *default* command-line parameters should the user omit them

Using IFS

- Typically, when setting command-line parameters using `set --`, the parameters set are separated by spaces (and/or tabs). Occasionally this is not what we desire
- We can use the `IFS` variable (internal field separator) to designate the character(s) that will be used to separate one command-line parameter from the next, for example:

```
IFS=|
```

- Setting the command-line parameters would then be (for example):

```
set -- Parameter 1|Parameter 2|Parameter 3
```

Usage Messages

- The great majority of UNIX commands have a *usage message* – a couple of lines of text that are displayed if the user incorrectly specifies command-line parameters
- For example:

```
Usage: grep [OPTION]... PATTERN [FILE]...
```

- The `[ ]` brackets specify parameters that are optional
- If is recommended that if your script will have enduring or public use, you create a usage message for it

```
if [ $# -lt 2 ]
then
   echo Usage: myscript username filename ...
   exit 2
fi
```

Usage Messages (cont.)

- Note: The usage message should go to Standard Error. This can be achieved by adding a `1>&2` to the end of the `echo` statement
- Suppose the script is called `myscript`. Suppose further that it is copied (or even linked) to another location and is given a different name. It is a nuisance to have to remember to change the usage message to correspond to this (in the case of linked scripts it is actually not possible)
- There is a solution to this problem: The special variable `$0` contains the name of the script as it was entered on the command-line (e.g. `myscript`)

Usage Messages (cont.)

- Thus, it is a simple matter to replace the reference to the name of the script in the usage message with `$0`:

```
echo Usage: $0 username filename ... 1>&2
```

- Notes:
  - `$0` cannot be `shift`-ed
  - The user can often see a usage message like the following:

```
Usage: /usr/local/bin/myscript username filename …
```

  - This can be remedied (if you choose) as follows:

```
echo Usage: `basename $0` username filename … 1>&2
```

Chapter Exercises

- Write a reusable function called `usage` that takes at least one parameter (the script name) and:
  - `echo`'s the script name and any other given parameters to Standard Error
  - Beeps, using the `ctrl-g` character
  - Exits the program with an exit status of 2
  - Ensure that `basename` is used on the script name
  - It might be called as follows:

```
usage $0 username filename ...
```

Chapter Exercises (cont.)

- Modify the course project so that it required one command-line parameter – the name of the data file
  - Use your new `usage` function to display the usage message if the user does not specify this parameter
  - If they specify a file and the file exists, check that you can write to it. If not, exit the program. If the file does *not* exist, ask the user if you can create it. If not, exit the program. If you *can*, check that you were able to successfully create it before proceeding

Exercise Solutions

Chapter 10

Advanced Scripting

Debugging

- Sometimes your script will not work properly, and it is not immediately obvious why not
- In situations like this it is necessary to use the Bourne shell's built-in simple debugging facility
- When the `-x` option is used to run a script (for example, `sh -x myscript`) the script echoes each command-line it is about to run just before it runs it
  - All special characters (such as wildcards, variable substitutions, etc) will have already been processed by the shell before the command-line is displayed

Default Values for Variables

- We frequently write code like the following:

```
if [ -z "$var1" ]    # test if var1 is not set
then
    var1="Some Default Value"
fi
```

- The above code my be abbreviated thus:

```
${var1:="Some Default Value"}
```

- The only problem with the above is that, as a command line, `Some Default Value` is treated as the name of a program. So we usually write:

```
: ${var1:="Some Default Value"}
```

or

```
var2=${var1:="Some Default Value"}
```

Temporary Files

- What is the problem with putting the following line into a shell script?

```
who > /tmp/who_results
```

  (hint: what if this was a very commonly used script?)

- The answer is: Many people may be running the script at the same time, and the same temp file would be used for all of them (this is especially significant in CGI programming)
- The challenge is to come up with a filename that is *guaranteed* to be unique to this instance of the script

Temporary Files (cont.)

- The secret is to use the $$ variable: $$ contains the *process ID* of the current shell (the shell running the script)
- $$ could be used as the name of the file, or it is also possible to append (or prepend) a string. For example:

```
tmpfile=/tmp/myscript.$$
who > $tmpfile
...
rm $tmpfile
```

Preventing Abnormal Termination

- Many of us will be familiar with the notion of "killing" a process (usually to terminate it), using the `kill` program. For example:

```
kill 12345
```

- When a process is "killed", what actually happens is the process is sent a *signal*, and if the process is not designed to handle that signal, the process simply terminates (aborts)
- It is possible to program a shell script (in fact *any* program) to handle signals in a meaningful manner

Preventing Abnormal Termination

- There are three possible reasons for handling signals:
  - Preventing the program from terminating (simply ignoring the signal)
  - Terminate the program, but in a graceful manner – for example, removing any temporary files, restoring the system to a known good state, etc
  - Causing the program to perform some custom action upon receipt of a particular signal

Preventing Abnormal Termination

- There are over 20 possible signals that could be received by a program. Most are very rare. However, the following signals occur reasonably often:
  - 1 (`SIGHUP`, short for "hang-up") Occurs when the user's terminal is disconnected or the login shell is terminated (modem hangs up, `telnet` session is interrupted, etc)
  - 2 (`SIGINT`) Occurs when the user presses the "interrupt" key on their keyboard (usually `Ctrl-C`)
  - 3 (`SIGQUIT`) Occurs when the user presses the "quit" key on their keyboard (usually `Ctrl-\`)
  - 15 (`SIGTERM`) Occurs when UNIX is in the process of shutting down, or if the process is manually "killed"

Preventing Abnormal Termination

- Two other signals, `SIGUSR1` and `SIGUSR2` (10 and 12 on Linux) are available to developers for custom processing in their programs
- Notes:
  - *Any* signal may be sent to a process using the `kill` program:

    ```
    kill -10 12345
    ```

  - It is not possible for any process to handle signal 9 (`SIGKILL`)
- In order to handle a signal, the script should contain (generally near the beginning of the script) a `trap` command in the following format:

  ```
  trap command signal ...
  ```

Preventing Abnormal Termination

- For example:

```
trap ls 1 2
```

- This will cause the `ls` program to be run whenever signals 1 or 2 are received (use “” to do nothing)
- The command specified above can even be the name of a function, and parameters can be specified:

```
graceful_exit()
{
    echo "Exiting, please wait..."
    # clean up temp files, etc
    exit $1
}
trap "graceful_exit 2" 1 2 15
```

Chapter Exercises

- Create a script that does nothing at all (except `sleep`) until a `SIGUSR1` is received, at which point it displays the current date and time, and a calendar of the current month, and continues doing nothing (use `kill -l` to see which signal `SIGUSR1` is)
- Modify the course project:
  - Implement the "Search" and "Delete" options. The "Search" option should prompt the user for a pattern to search for, and display all the records that contain that pattern. The "Delete" option should do all of the above and then (if there were any matches) ask the user if they want to delete all those records
  - A temporary file will probably be necessary for the "Delete" option. Make sure it is named appropriately

Chapter Exercises

- If either `SIGHUP` (1) or `SIGTERM` (15) is received, exit immediately
- If either `SIGINT` (2) or `SIGQUIT` (3) is received, ask the user the standard question about quitting ("do you really wish to quit now?")
- Try to avoid repeated code (i.e. Rather than copying and pasting a piece of code, turn that code into a separate function)

Exercise Solutions

The End