



Productivity in Common Operating Systems

Productivity in Common Operating Systems

Unix Essentials

LESTER HIRAKI

TORONTO METROPOLITAN UNIVERSITY
TORONTO



Productivity in Common Operating Systems Copyright © 2022 by Lester Hiraki is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/), except where otherwise noted.

Contents

Introduction	1
Lester Hiraki	

Part I. Main Body

1. The Hierarchical File System	3
<i>Hierarchical File System</i>	3
2. Common Unix Commands	12
3. Input and Output: Redirection and Pipes	20
<i>Filters</i>	20
<i>Redirection</i>	21
<i>Pipes</i>	27
<i>Making your script executable</i>	30
4. Shell Variables, Quotes, Command Substitution	31
<i>Shell Variables</i>	31
<i>read</i>	37
<i>Quotes</i>	39
<i>Command substitution</i>	42
5. Control Structures - Part 1 - branching	44
<i>test</i>	44
<i>Control Structures</i>	46

6. Control Structures - Part 2 - Loops	59
<i>Computer Science Loop Concepts</i>	59
<i>Loops in Unix</i>	62
7. Quick Reference Guide	76
<i>Common Notations</i>	76
<i>FAQs</i>	78
Acknowledgements	79

Introduction

LESTER HIRAKI

Welcome to Productivity in Common Operating Systems!

The goal of this book is to provide the interested learner with the essentials to work in a Unix environment.

The focus is on the user's perspective to enable the user to be productive in a Unix environment. Topics include understanding and navigating the file system, using common commands, and automating tasks. Emphasizing the user's perspective, the scope of this book does not include topics such as system administration, installation, or networking.

To gain the most out of this book, it is recommended that the learner have access to a Unix or Unix-like system, specifically with command line access, so as to be able to practice commands and programming.

This book is intended for adoption in the freshman or sophomore year of a technical program (e.g. computer science, engineering, STEM, etc.). No prior knowledge or experience with Unix is expected; however, familiarity with computer programming (coding and debugging) is strongly recommended.

UNIX is a registered trademark of The Open Group. The Open Group is not affiliated with this resource. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. In this book, Unix (mixed case) refers to Unix-like operating systems such as Linux as well as UNIX.

1. The Hierarchical File System

Pre-amble

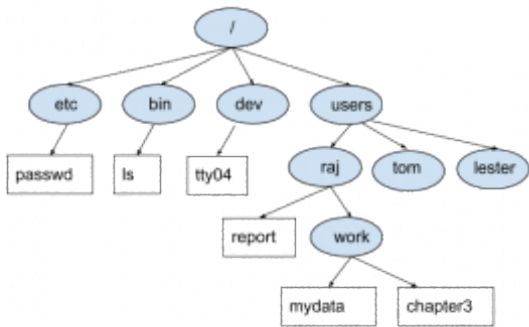
Unix is primarily a command line oriented operating system. Most commands are an action which is performed on an object, typically a file or directory. In order to be productive in a Unix environment, one must be intimately familiar with the concept of the hierarchical file system. Mastering this concept is fundamental to successful work in a Unix environment and is the key takeaway of this section.

Hierarchical File System

The Unix file system is that of an inverted tree. Imagine a tree with leaves and branches but turned upside down with the root or main trunk at the top. The main trunk branches off to smaller branches and eventually leaves. By analogy, the root directory typically contains several directories (folders) which in turn contain other directories (subdirectories) and/or files. Just as a tree branch can have smaller branches or leaves, a leaf cannot have other branches or leaves. Similarly, the difference between a directory and a file is that a directory can contain other directories or files, but a file cannot contain other directories or files — a file is a terminal node.

The figure below shows the typical layout of a Unix file system. While systems vary greatly in size, most will have at least

these directories. The convention in this diagram uses an ellipse to depict a directory and a rectangle to depict a file.



Typical hierarchical file structure of a Unix system

Explanation of common subdirectories:

Directory	Remarks
etc	contains operation and administrative files
bin	contains executable commands
dev	contains the devices connected to the system (printers, terminals, etc...); these devices still appear as files
users*	contains user files and directories. *The directory name is not standard and varies between systems. Other common variations are things like "home". Some larger systems will even have more than one top-level user directories such as "faculty", "staff", "classof31", etc.

How to Specify a File or Directory in Unix

As most Unix commands act on files or directories, it is necessary to be able to specify such an entity.

There are two methods to specify a file or directory, absolute and relative:

Absolute

To specify a file or directory using the absolute method, start with the root directory (/) and write each directory that is encountered on the path to the directory or file being specified. Separate each directory with a "/" forward slash character.

Examples

Specify the etc directory in the above system.

`/etc`

Specify the file named passwd in the above system.

`/etc/passwd`

Specify the file named ls in the above system.

```
/bin/ls
```

Specify the file named mydata in the above system.

```
/users/raj/work/mydata
```

Key Takeaways

The absolute specification always starts with a “/” (forward slash).

Relative

When working on Unix, the user will always “be” at some logical position within the hierarchy. This position is termed the “current working directory” or simply the “current directory”. The

specification of a relative path is relative to this current directory position. Note that it is possible to change one's current directory while working; this will be discussed in a later chapter.

To specify a file or directory using the relative method, start with the current directory and write each directory that is encountered on the path to the directory or file being specified. Separate each directory with a "/" forward slash character.

Examples

Specify the file mydata with current directory:
`/users/raj/`

`work/mydata`

Specify the file mydata with current directory:
`/users/raj/work/`

`mydata`

Specify the file report with current directory:
`/users/raj/work`

`../report`

As the file **report** is not contained in the current directory, it is necessary to go up one level first (to raj) to be able to reach the file **report**. Here the double dots mean “parent directory” or one level up.

Key Takeaways

The relative specification never starts with a “/” (forward slash).

When should one use absolute vs. relative specifications? In many cases both are acceptable. One may notice that a relative specification usually requires less typing. Who likes more typing? An absolute path is preferred when the user or pro-

grammar values portability allowing the specification to be used from any position on the system by any user.

Points to Consider

- Names for files and directories are case-sensitive. Thus report, Report, and REPORT are all distinct.
- Names may include any letters, digits, and some special characters (period, comma, underscore, etc.) but not /, <, >, &, :, |.
- Names may be up to 255 characters in length.
- There is no requirement for file extensions (few characters after a period). All of the following are valid names:
 - `report`
 - `letter_to_bob.text`
 - `forecast.July,Version1`
 - `notes.doc`
- Names must be unique within a directory (no duplicates allowed). This is automatically enforced by the operating system. In the `raj` directory, it would not be possible to create another file or directory called `work`. Consider the analogy of human families: Two siblings would not share the same name, but a cousin, uncle, or grandparent could share the same name without conflict.



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=25#h5p-1>

Home Directory

The home directory is a private area for the user files and directories. Each user will have a directory name matching the login name assigned by the system administrator.

While it is correct and possible to specify a home directory using the absolute and relative methods aforementioned, there exists an abbreviation consisting of a ~ (tilde) followed by the user's login name. Advantage: This avoids having to know the name of the user file area which is non-standard: some installations call it users, others call it home, still others have their own conventions.

Examples

Eg. Raj's home directory would be

`~raj`

Eg. A file in his directory would be

`~raj/report`

2. Common Unix Commands

While there are hundreds of Unix commands, fortunately it is not necessary to know all of them. In fact, one can achieve a significant level of productivity knowing just a couple of dozen. Here are some of the most common and useful commands.

All commands are case sensitive.

Spaces are a BIG DEAL in Unix

When issuing commands, in order for Unix to tell when a command finishes and when parameters and file names start and end, every item (or token) on a command line must be separated by whitespace (one or more space characters). One of the most common causes of frustration is failure to put whitespace between items on the command line, or putting whitespace where it should not be.

Just as in English, there is a big difference in meaning between “no table” and “notable”, so is the case in Unix.

Right:

```
ls /etc
```

Wrong:

```
ls/etc
```

Unix commands are a single word requesting an action. Sometimes the action is standalone, but most actions are applied to some object like a file or directory. At the point in the command where it expects the name of a file, say, this is where you specify the file in the form of an absolute or relative reference as described in the previous chapter.

Command	What does it do?	Example usage
ls	list directory contents	<p>List files (and directories) in my current directory: ls</p> <p>List files in top level etc directory: ls /etc</p> <p>List all files beginning with g (wildcard *): ls g*</p> <p>List and give details about files: ls -l</p> <p>List all files including hidden: ls -a</p>
cat	concatenate files and print on the standard output	<p>Show contents of file report: cat report</p> <p>Show contents of files chapter1 and chapter2: cat chapter1 chapter2</p>
head	output the first part of files	Show the first few lines of file logfile: head logfile
tail	output the last part of files	Show the last few lines of file logfile: tail logfile
cp	copy files and directories	Copy file report to report_v2 in the work directory: cp report work/report_v2
mv	move (rename) files	<p>Rename file report to presentation: mv report presentation</p> <p>Move file mydata to yourdata one level up: mv mydata ../yourdata</p>

Command	What does it do?	Example usage
rm	remove files (permanent)	Delete file report in current directory: rm report Caution: There is no recycling bin in Unix. Consider all deletions as permanent. Exercise care when using wildcards (e.g. *).
cd	change the working directory	Change directory to work directory (one level down): cd work Change directory to /bin directory: cd /bin Change directory to parent (one level up): cd .. Change directory to home directory of user ahmed: cd ~ahmed
pwd	print name of current/working directory	Are you lost? pwd
mkdir	make directories	Create new directory called unix_exercises: mkdir unix_exercises
rmdir	remove empty directories	Remove (permanently delete) the directory unix_exercises: rmdir unix_exercises
who	show who is logged on	Display who else is logged in right now: who
whoami	print effective userid	Display one's own userid (short login name): whoami

Command	What does it do?	Example usage
date	print the system date and time	<code>date</code>
man	an interface to the on-line reference manuals	Display manual entry for ls command: <code>man ls</code> Keyword search for commands related to "directory": <code>man -k directory</code>

Key Takeaways

- Spaces are a BIG DEAL in Unix: They are needed between commands, parameters, and filenames.
- All commands are case-sensitive (usually all lowercase)

More Key Takeaways

- Command options (e.g. -l, or -d, etc.) are specific to the command. For example, while both the ls and cp command both have a "-l" option, the

option means different things in each command. Command options are **not** mix and match.

- Command options may be listed separately or combined. The following are equivalent:

- `ls -ld`
- `ls -l -d`

More Unix Commands

All the commands below require some sort of input, typically a file, but the commands do **not** modify the input file. The outputs will contain portions of the input files, but the input files are never changed.

Command	What does it do?	Example usage
cut	Print selected parts of lines from each FILE to standard output.	display columns 1-10 and 20-23 of myfile <pre>cut -c1-10,20-23 myfile</pre> display the 3rd and 5th fields of the /etc/passwd file <pre>cut -f3,5 -d: /etc/passwd</pre>
paste	merge lines of files If cat joins vertically, think of paste as a horizontal version of cat.	display file1, file2, and file3 side-by-side <pre>paste file1 file2 file3</pre>
wc	print newline, word, and byte counts for each file	display the number of lines, words, and characters for the file chapter3 <pre>wc chapter3</pre> display only the number of lines <pre>wc -l chapter3</pre>
grep	print lines matching a pattern	display lines matching the string Total in the file sales <pre>grep Total sales</pre>
sort	sort lines of text files	display a sorted version of the file namelist <pre>sort namelist</pre>

3. Input and Output: Redirection and Pipes

By design most Unix commands are small and simple in their functionality. To solve beyond the trivial requires the use of several steps or commands. How to sequence and combine commands in Unix requires an understanding of how input and output is managed. The next sections will introduce these concepts and show how more complex problems may be solved.

Filters

One may think of a filter as a black box with an input and an output. Most Unix commands can be thought of as a filter. The inputs and outputs have been given formal names. The input is named standard input (STDIN); the output is named standard output (STDOUT); and there is a secondary output named standard error (STDERR) which will be discussed in more detail later. These inputs and outputs are associated with file descriptors or stream numbers 0, 1, and 2 respectively.

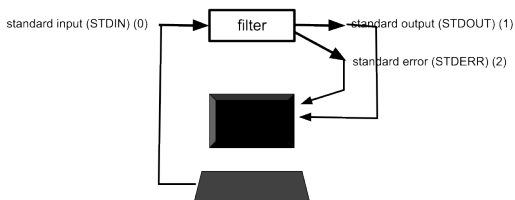
By default, Unix commands read from standard input and print to standard output. Any error messages are sent to standard error.

Examples of Unix commands as filters

Command	Description
cut	reads from standard input and passes selected portions (columns, fields) to standard output
grep	reads from standard input and prints matching lines to standard output
head & tail	reads from standard input and prints the first (last) few lines to standard output
cat	a transparent filter: reads from standard input and prints the same to standard output
wc	reads from standard input and prints summary information to standard output
...	not an exhaustive list

Redirection

When working interactively in a Unix session, the default setup is to have standard input draw from the keyboard, and standard output (as well as standard error) directed at the screen. In other words, a command reading from standard input will wait for keystrokes. A command printing to standard output will have its output appear on the terminal screen.

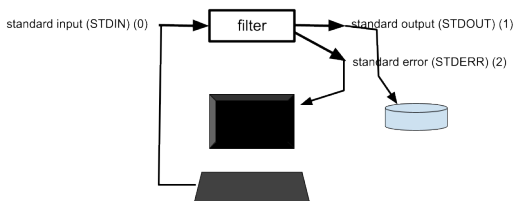


Default association of keyboard and screen to filter.

While this arrangement works well, there will come situations where the user will want to save the output of a command to a file, or substitute a file for keyboard input. This is accomplished through the concept of redirection where one of the inputs or outputs is associated with a file.

Redirection of standard output (> operator)

The user can save the standard output of any Unix command by redirecting standard output to a file using the > operator. Graphically, the concept is illustrated as follows.

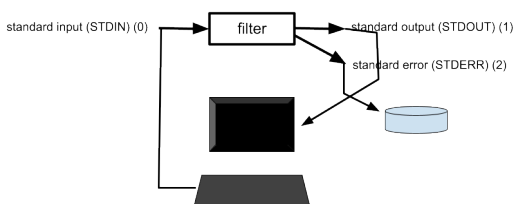


Redirection of Standard Output

Operator syntax	Examples and explanation
cmd > some_file	e.g. cat chapter1 > book <ul style="list-style-type: none"> if the file already exists it will overwrite it
cmd >> some_file	e.g. cat chapter3 >> book <ul style="list-style-type: none"> the >> operator appends an existing file

Redirecting standard error (2> operator)

The user can save the standard error of any Unix command by redirecting standard error to a file using the 2> operator. Graphically, the concept is illustrated as follows.

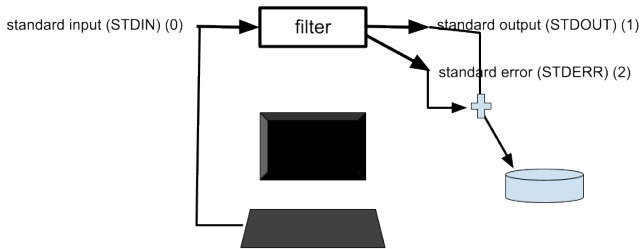


*Redirection
of Standard
Error*

Operator syntax	Examples and explanation
cmd 2> some_file	e.g. cat chapter1 chapter4 2> errors <ul style="list-style-type: none"> • saves error messages in file errors but standard output is displayed on the screen
cmd > some_file 2> another_file	e.g. cat chapter1 > book 2> errors <ul style="list-style-type: none"> • saves standard output to file book, saves standard error messages in file errors

Merging two streams (>& operator)

The user can save both standard output and standard error of any Unix command. This is accomplished by first redirecting standard error to a file, and then merging standard error with standard output. The syntax of the merge operator is `m>&n` where stream `m` is merged with wherever stream `n` is already going. Graphically, the concept is illustrated as follows. Here stream 2 (standard error) is merged with stream 1 (standard output).

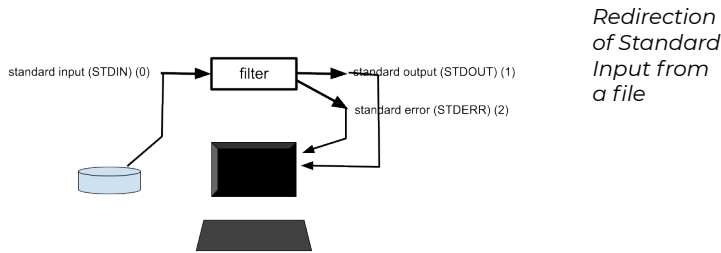


Standard Output and Standard Error are merged (combined) and re-directed to a file

Operator syntax	Examples and explanation
<code>cmd > out_file 2>&1</code>	<p>e.g.</p> <pre>cat text1 junk text3 > both 2>&1</pre> <ul style="list-style-type: none"> • saves standard output and standard error to file both • Explanation: Unix parses left to right. First standard output is redirected to file "both". Then the merge operator (>&) merges (blends) standard error (stream 2) with where standard output (stream 1) is already going (to file "both"). • Caution: How about <pre>cat text1 junk text3 > both 2> both</pre> <ul style="list-style-type: none"> • This is an incorrect method of trying to merge two streams. This causes a race condition (two streams competing for the same file) and correct results are not guaranteed.

Redirecting standard input (< operator)

The user can redirect standard input from a file instead of the keyboard to any Unix command using the < operator. Graphically, the concept is illustrated as follows.



Operator syntax	Examples and explanation
<code>cmd < some_file</code>	<p><code>cat < appendix</code></p> <p>While it may appear that the “<” in the above command does anything (works the same without the “<”), the reason is that <code>cat</code> is smart and knows to look for input on the command line. <code>cat</code> is specifically designed to inspect the command line for input. It is necessary to use the “<” for redirection of standard input.</p> <p>Consider a more basic example of a simple script that reads from the keyboard. To substitute a file, it would be necessary to use a command, where the file “keystrokes” contains what the user typed.</p> <p><code>myscript < keystrokes</code></p> <p>Here “myscript” represents a user-written Unix script that reads from standard input.</p>

Pipes


It is often the case that a problem in Unix is solved with multiple commands. Typically the output of the first command is saved in a file which is then used as input to a subsequent command. The use of a pipe is considered a refinement of this approach potentially simplifying the solution.

Problem: To determine the number of entries in a directory

Method 1

Graphical view	code	explanation
<pre>ls -> file_list -> wc</pre>	<pre>ls /etc > file_list wc -l file_list rm file_list</pre>	<ul style="list-style-type: none">• save output of ls command in file_list• run wc command to count lines• delete temporary file file_list

Method 2

Graphical view	code	explanation
<pre>ls ->  -> wc</pre>	<pre>ls /etc wc -l</pre>	<ul style="list-style-type: none">• save output of ls command is sent directly to input of wc command• no temporary file needed

Definition:

A pipe connects STDOUT of previous command to
STDIN of next command

You can use a pipe multiple times creating a pipeline.

e.g.

```
cmd1 | cmd2 | cmd3 | cmd4
```

Building a pipeline should be an iterative process. Condense stepwise as you know the solutions work, otherwise there might be errors that might be difficult to detect from a single pipeline

Start out like this:

```
cmd1 > out1  
cmd2 < out1 > out2  
cmd3 < out2
```

...

Key Takeaways

1. Redirection: Use between a command and a file
2. Pipe: Use between commands

Making your script executable

1. create file containing unix commands
2. Once per file, type either:
 - `chmod u+x myscript`
 - `chmod 700 myscript`
3. To run, type `./myscript`

4. Shell Variables, Quotes, Command Substitution

Shell Variables

Unix is an operating system but also the name given to its scripting (programming) language. In order to be useful, a computer language needs to be able to store data. Unix supports this capability with shell variables.

There are three kinds of shell variables:

1. special
2. environment
3. program

Special variables

Special variables are unlike what you may have seen in other programming languages. Rather special variables are created and set by the operating system automatically. Consider the following examples:

Positional parameters (\$1, \$2, ...)

Recall that the cat program allows the user to specify inputs

as command-line arguments: (In the following examples, the leading dollar sign represents the command prompt; do not type the leading dollar sign.)

```
$ cat ch1 ch2 ch3
```

Imagine that you would like to write your own script allowing the user to specify inputs as command-line arguments as in.

```
$ myscript apple cherry
```

Example

```
$ cat arg_demo
#!/bin/bash
echo The 1st argument is $1
echo The 2nd argument is $2
$
$ ./arg_demo apple cherry
The 1st argument is apple
The 2nd argument is cherry
```

number of command line arguments (\$#)

The special variable \$# holds the number of command-line arguments specified when your script was run.

Examples

```
$ cat numeg
#!/bin/bash
echo The number of command line arguments is $#
$
$ ./numeg apple cherry
The number of command line arguments is 2
$
```

return code of previous command (\$?)

Every Unix command generates a return code typically indicating success or failure. The value of this return code is stored in the special variable `$?` . Thus each time a Unix command is run the `$?` variable is updated automatically to hold the return code of the most recently executed Unix command. The use of the return code is important and will become apparent in the chapter on control structures.

Environment variables

Environment variables hold information about the users's current settings and configuration. By convention, they are typically all UPPERCASE. To display them, type `'env'`

Example

```
lhiraki@metis:~$ env
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
EDITOR=vi
LANGUAGE=en_CA:en
LPDEST=eng206c
lhiraki@metis:~$
```

Here is an abridged list of environment variables. Some of the typical environment variables include SHELL (current shell), PATH (list of directories the operating system will search in order to find a command), EDITOR (preferred editor), LANGUAGE (preferred display language), LPDEST (preferred printer destination).

Program variables

Program variables are the type of variable that typically come to mind when one thinks of variables in a computer programming language. Program variables are variables which you as the programmer create and set. As Unix is a prototyping language, it is common to dispense with many of the formalities required for variable declarations, etc.

Program variables hold a string, i.e. text, and can be used anywhere text could appear in a program such as a filename, part of the file name, or even a Unix command itself. A variable can hold only one value at a time.

The naming convention for program variables is to use lowercase. Specifically uppercase program variable names should be avoided so as to prevent confusion with environment variables and accidentally overwriting an environment variable.

Eg. 1 An easy way to create and set a program variable is using an assignment statement.

```
temp_name=/usr/temp  
cp myfile $temp_name
```

Assignment statements copy the value to the right of the equal sign to the variable on the left. Important: Assignment statements must not contain spaces, especially around the equal sign.

Eg. 2

```
month=september  
echo the current month is $month
```

When to use \$ with variables

When writing to a variable, i.e. setting or changing its contents, do not use the dollar sign. When reading a variable, i.e. accessing its contents, you must use the dollar sign.

Key Takeaway – When to use \$ with variables:

Writing to a variable: **no** dollar sign

Reading from a variable: **use** dollar sign

Editorial Remark:

One of the key factors affected software maintenance costs is code readability. Most of the time (cost) of maintaining software is spent in having designers read and understand existing code. One

way to control and reduce costs (business competitiveness) is to ease readability. By choosing variables which reflect their contents, it makes it easier to understand the code.

Choose self-describing variable names:

Good variable names:	Bad variable names:
<ul style="list-style-type: none">• sum• total• x_value• length• count	<ul style="list-style-type: none">• a,b,c....• var1, var2, var3....

read

The read command collects characters from standard input (STDIN) and stores them in a variable. The read command is typically used in an interactive script to collect user input after an appropriate prompt message. As the read command draws from standard input, and standard input can be redirected from a file, it is possible to prepare inputs (answers) in a file and run the script in a non-interactive fashion.

Example using read command

```
$ cat readeg
echo -n 'What is your name? '
read name
echo Hello $name, pleased to meet you!
$
$ ./readeg
What is your name? Mohammed
Hello Mohammed, pleased to meet you!
$
```

What does the option “-n” do in the echo command above?

Hint: type “man echo”. To exit the manual, press “q”.

Key Takeaways

read vs. command line arguments

- Both are ways the user can supply input to a program. Command line arguments are placed on the command line before pressing <ENTER> to run the program. The values are accessed within one's program using \$1, \$2, etc.
- The "read" command causes the program to wait for keyboard input (if STDIN has not been redirected from a file). The input is stored in and later accessed from a program variable.

Which method should you use? Refer first to any program requirements. (Does it say, "Prompt the user to enter ..." or "specify as a command line argument"?)

Quotes

Single quotes:

Problem:

```
$ grep Al Shaji employee_list
grep: can't open Shaji
```

Solution:

```
$ grep 'Al Shaji' employee_list
```

Single quotes causes Unix to take everything within the single

quotes literally. This is how you would prevent interpretation of characters which would normally have special meaning, for example the space character separating command line arguments.

Double quotes

- Recognize \$, \, ` (backtick or backquote)

Exercise 1: Try this out and see the difference double quotes makes

```
heading='      Name      Addr      Phone '  
echo $heading  
echo "$heading"
```

Exercise 2: Try this out and see the difference double quotes makes

```
read operator # enter asterisk *
```

```
echo $operator  
echo "$operator"
```

The double quotes are similar to single quotes in that Unix takes what is within the quotes literally. However, double quotes are “smarter” in that variables and selected meta-characters are interpreted and expanded in spite of the usual literal nature of quotes.

Home directory potential tricky issue (tilde is protected i.e. not expanded within quotes).

Instead of:

```
datafile=~jasmin/rawdata  
cat $datafile # produces "No such file or directory" error
```

Say (solution 1):

```
datafile=~jasmin/rawdata  
cat $datafile
```

Say (solution 2):

```
datafile=~jasmin/rawdata  
eval cat $datafile
```

Command substitution

Sometimes a programmer wishes to run a command and use its output at some point within a program. While it is possible to redirect output to a temporary file, load the contents of the file, and then promptly delete it, for small tasks, it is more convenient and efficient to use the technique of *command substitution* and avoid extra disc access.

Eg. 1 Newbie's first guess

```
today=date  
echo $today
```

Eg. 2 This is the way to do it.

```
today=$(date) (old version today=`date`)  
echo $today
```


Syntax:

```
$(unix_command)
```

The mechanics of command substitution works as follows:

1. Unix will run the command within the parentheses as if it were typed at the keyboard. The command may include options, command line arguments, or even be a script.
2. Instead of being displayed on the screen, the standard output (STDOUT) of the command is substituted at the exact position of the call (`$(unix_command)`).

The command substitution may be made anywhere in a program; however, it is often used on the right-hand side of an assignment statement (to save the output in a variable), or within an echo statement.



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=35#h5p-10>

5. Control Structures - Part 1 - branching

In order to support control structures (branching, looping, etc.), all computer languages need some method of evaluating a condition. In Unix, the command to evaluate a condition is the test command. The test command is often at the heart of most control structures in Unix.

test

What does the test command do? It evaluates a condition and sets the special shell variable `$_`, the return code. Much to the confusion of new users, the test command is silent in that it does not print anything to standard output. Thus when running the test command, it appears as if to do nothing. To check the return code, one may simply print its value with an echo statement.

Example 1: Check if a file is readable.

```
$ test -r myfile
$ echo $_
0
```

How does one interpret the return code?

Unix convention:

0 (zero)	means TRUE
not 0 (not zero)	means FALSE

Example 2: Compare if two strings are equal.

```
$ test "this" = "that"
$ echo $?
1
```

Synonym to the test command: [

For reasons of readability, many programmers will use the synonym or abbreviation for the test command which is the left square bracket: [. In all examples above, replace the word test with the left square bracket. Note that a matching right square bracket needs to be added for syntactic reasons. As with all Unix commands, spaces are a big deal and a space is required after the left square bracket and before the right square bracket.

Example 3: Synonym to the test command [

```
$ [ -r myfile ] # note the space after the left bracket
$ echo $?
0
```

Control Structures

if

The if statement in Unix is the basic two-way branch.

Syntax

```
if unix_statements
then
    actions_true
else # optional
```

```
        actions_false
    fi
```

How does the if statement work? Here is the sequence of operations. The if statement will:

1. Run all the unix_statements.
2. Check the return code (\$?) of the last statement in the list (just prior to the keyword “then”.
3. If the return code is true, the “then” clause is executed (actions_true). If the return code is false, the “else” clause is executed (actions_false). The else clause is optional (can be left out if not needed).

Example of if

```
#!/bin/bash
comfort=20
temperature=18
if echo The current temperature is $temperature
    echo Temperature for comparison is $comfort
    [ $temperature -lt $comfort ]
then
    echo It is cold.
else
    echo It is warm.
```

fi



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=37#h5p-5>

Nested conditions are supported in Unix. Any statement in the “then” or “else” clause can itself be another if statement.

elif

Although the if statement is primarily a two-way branch (e.g. true or false), a multi-way branch (e.g. red, yellow, green) can be coded using a set of nested if statements. Some computer languages support an “else-if”-type clause; Unix is one of them.

There is an “else-if” clause called “elif” which requires a statement just like the if clause.

Example: elif

```
if [ $temperature -lt $comfort ]
then
    echo It is cold.
elif [ $temperature -eq $comfort ]
then
    echo It is perfect.
else
    echo It is warm.
fi
```

Note that the “elif” clause is actually a clause of the main “if” and not a nested if statement. Thus, there is only one “fi” (end if) required for each opening “if” regardless of how many “elif” clauses there are.



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=37#h5p-6>



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshi-raki/?p=37#h5p-7>

case

Many computer languages support a multi-way branch. Unix is included as one of them.

Simplified syntax:

```
case $variable
in
    val1) action1;;
    val2) action2;;
    *) default action;;
esac
```

The keywords are case, in, and esac. The double semi-colon is a syntactic requirement to separate the inner clauses of the case statement.

Example: Flexible command line processing.

Allow your user to run your script in various ways.
Accommodate all of the following invocations.

```
$ ./myscript # user omits filename; give second chance
$ ./myscript chapter3 # preferred syntax; just proceed
$ ./myscript chapter3 chapter5 # multiple arguments not supported
inform user
```

Place this code snippet at the beginning of your script like

```
$ cat myscript
case $# in
  0) echo Enter file name:
      read arg1;;
  1) arg1=$1;;
  *) echo invalid number of arguments
      echo "Syntax: $0 filename"
      exit 1;;
esac
# rest of program continues after esac
$
```

General syntax:

```
case match_string_expr in
  match_pattern) action1;;
  match_pattern) action2;;
  ...
esac
```

Example: Demonstrate pattern matching use in case statement. Print a message about the length of the current month.

```
lhiraki@metis:~/test$ cat case_month
case $(date '+%m') in
01|03|05|07|08|10|12)
    echo This is a long month;;
04|06|09|11)
    echo This is a short month;;
02)
    echo This is the shortest month;;
*)
    echo Something wrong with date command;;
esac

# Example run in September
```

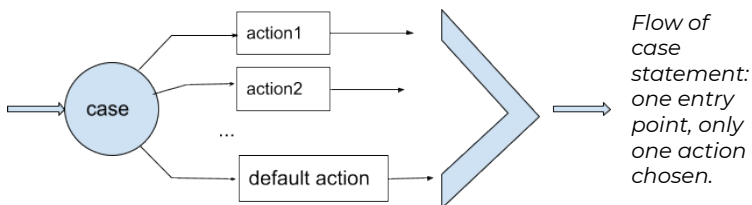
```
lhiraki@metis:~/test$ ./case_month
This is a short month
lhiraki@metis:~/test$
```

The date command is called with an option to return the numerical value of the month (e.g. Jan=01, Feb=02, etc.). Depending on the month, a message is printed regarding the length of the month. Months with the same number of days are grouped together using a pattern with the OR (vertical bar) syntax.

Defensive programming

Well the previous example is rather trivial and the date command has been well tested over the years, it is considered good practice to always have a default clause even if you think you have covered all possible conditions.

Summary



The `match_string_expr` is matched against each `match_pattern` in the order coded. At the first match, the corresponding action is taken. After one action is completed, the case statement terminates and execution continues after the `esac` (end case). There is no “fall-through”. The case statement will not execute multiple actions.



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=37#h5p-8>

shift

When processing multiple command-line arguments, it may be necessary to manipulate them to facilitate processing.

The `shift` command moves all command-line arguments one position to the left. For example the second command line argument is moved into the first position; the third command line argument is moved into the second position, and so forth.

Example of shift command

```
lhiraki@thebe:~/test$ cat shifter
```

```
#!/bin/bash
echo The 1st arg is $1
echo The 2nd arg is $2
shift
echo The 2nd arg is $1
shift
echo The 3rd arg is $1
```

Execution results in:

```
lhiraki@thebe:~/test$ ./shifter apple pear grape
The 1st arg is apple
The 2nd arg is pear
The 2nd arg is pear
The 3rd arg is grape
lhiraki@thebe:~/test$
```

The exit command

The exit command does two things:

1. It terminates the current shell (or script), returning control to the calling shell, if any.
2. It sets the return code (\$?) for your script.

Example – exit command usage

The trivial but illustrative script `exit_example` shows the `exit` command setting return code for the script to 3 and then terminating the script. Control returns to the calling program, in this case just back to the operating system prompt.

```
$ cat exit_example
#!/bin/bash
exit 3
echo This line never executed.
```

```
$ ./exit_example
$ echo $?
3
$
```

Note that one must inspect the return code (`$?`) immediately after running `exit_example`. The return code is updated (overwritten) by each Unix command executed.

The `exit` command is typically used to terminate a script midway through often due to an error condition. The other primary use of the `exit` command is for a sub-script to communicate information back to the calling script.

Example – parent/child script relation

parent	child
<pre>./child ret_val=\$? case \$ret_val</pre>	<pre>... exit 2 ... exit 0</pre>

Note: parent-child relationship can be used for team project where multiple people can work on the same file at the same time.

Shebang line

To specify which interpreter Unix should use when executing your script, as the first line of the file place the path to the interpreter after “#!”. While the number sign character (#) normally introduces a comment, when used with the exclamation mark at the beginning of a file, Unix will load the interpreter specified in the path to run the rest of the script file.

This is especially important to make your script portable in Unix environments. If you write your script in bash, and you give your script for someone else to use who works in a c-shell or Korn shell environment, your script may not work properly.

To ensure that the script is run under bash, you must specify the shebang line.

Careful: The shebang line must be the first line of the file, not just the first line of text, or the first line of code. A common mistake is to have a blank line as the first line, or some comments above the shebang line. Unix does not look beyond the first line of the file in order to identify the expected interpreter.

Right	Wrong	Wrong
1:#!/bin/bash 2:#comments 3:#comments 4:code begins here	1:#comments 2:#comments 3:#!/bin/bash 4:code begins here	1: 2:#!/bin/bash 3: 4:code begins h



An interactive H5P element has been excluded from this version of the text. You can view it online here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=37#h5p-9>

6. Control Structures - Part 2 - Loops

Eventually you will need to repeat actions. A practical way to repeat actions without repeating code is achieved with the construct of a loop.

Computer Science Loop Concepts

The construct of a loop allows for the repetition of actions without repeating code.

General Loop (Concept)

The most general flow of control is illustrated in the flowchart below (General). Observe that there can be a set of actions (Block 1) which is performed regardless on entry into the loop. Then a conditional check is made to determine continuation of the loop. If satisfied, further actions within the loop (Block 2) will be performed.

Not all computer languages support the General

loop construct (Unix does support a General loop). If a computer language does not support the General loop construct, it will usually support one or more of the specific cases of the General loop. For example, C Language supports both a While and a Do-While construct but not the General loop construct.

While Loop (Concept)

The While Loop, as a concept, is a special case of a General Loop which has no actions for Block 1. In this case, the condition is checked as a first action on entry into the loop construct. See flowchart below (While).

Do-While Loop (Concept)

The Do-While Loop, as a concept, is a special case of a General Loop which has no actions for Block 2. In this case, the Block 1 code is executed on entry into the loop construct, and the condition is checked at the end of the loop construct. See flowchart below (Do-While).

General	While		Do-V



An interactive H5P element has been excluded from this version of the text. You can view it online here:

<https://pressbooks.library.torontomu.ca/opsyshiraki/?p=41#h5p-3>



An interactive H5P element has been excluded from this version of the text. You can view it online

here:

<https://pressbooks.library.torontomu.ca/opsyshi-raki/?p=41#h5p-4>

Loops in Unix

There are three commands in Unix to support loops.

- while
- until
- for

while command

The while command in Unix implements the General Loop construct discussed earlier.

general syntax	relation to flow chart
<pre>while list1 do list2 done</pre>	<pre>while block1 check do block 2 done</pre>

Unix runs the list1 and looks at the return code (\$?) of the last command in list. (If there's only one command in the list1, then that's the last command.) If the return code of this last command is TRUE, then list2 (loop body) is run. Once the keyword "done" is reached, control returns to re-run list1 and the return code of the last command in list1 is checked. If the return code of this last command is FALSE, then the while loop terminates, and control continues after the keyword "done".

E.g. 1: Count from 1 to 12

```
month=1
while echo Checking limit against month ${month}
  [ ${month} -le 12 ]    # The test is the last command in li
do
  echo Performing action for month ${month}
  month=$(( ${month} + 1))
done
echo Value of month outside loop is ${month}
```

Notes and observations

1. The last command of list1 is a test command using the left bracket synonym.
2. The -le is a test command option for the numeric comparison less than or equal to.
3. As Unix program variables hold strings, to perform arithmetic operations, one must use the $((..$

-)) syntax with dollar sign and double parentheses.
4. The formal spelling of Unix variables requires enclosure in set braces {}. Where there is no ambiguity, it is common practice in Unix to omit the set braces.
 5. Try running this code and see that the value of month never exceeds 12 in the line “echo Performing action ...”.

Preamble: Redirecting input from a file

```
$ cat novel
It was a
dark and
stormy
$ read oneline < novel
$ echo $oneline
It was a
$ read oneline < novel
$ echo $oneline
It was a
$
```

Be reminded that redirection operators in Unix (<, >, etc.) apply only to the current command. Once the command is over, redirection is restored to the default configuration (STDIN from keyboard, STDOUT to screen, etc.). Thus, observe that

each time the read command is run, redirection is applied anew, and the first line of the file is read in.

Eg 2a: How to read every line of a file (a non-functional example)

The objective is to read a file one line at a time. Here is a newbie's first attempt.

```
$ cat badwhile
while read wholeline < novel
do
    something $wholeline           #e.g. echo $wholeline
done
$ ./badwhile
It was a
It was a
It was a
It was a
...
```

What happened? The redirection operator applies to the read command. Each time the read command is run, redirection is applied anew and only the first line of the file is ever read.

How can one fix this problem?

Eg 2b: How to read every line of a file? (a functional example)

To have redirection apply to the entire duration of the while loop and not just the read command, it is necessary to establish redirection and associate it with the while loop.

Here is a template which you can use to read lines from a file one-at-a-time.

```
while read wholeline
do
    something $wholeline #e.g. echo $wholeline
done < inputfile
```

To make this work, one must position the redirection operator after the command to which it should be applied. The correct position is after the while command, in particular the keyword done. Unix is smart enough to look ahead for redirection when it starts executing the while command. Any commands within the while command which draw from STDIN (e.g. read) will draw from the redirected file. (If you are having difficulty understanding redirection, refer to the chapter on Redirection and Pipes.)

*Eg 3: How to process multiple command line arguments
(Method 1)*

Recall that the cat command will accept any number of command line arguments like this:

```
cat ch1 ch2 ch3 ...
```

Let's say that you want to write your own Unix script that will accept any number of command line arguments:

```
mycmd parm1 parm2 ...
```

The key requirement here is that you will not know beforehand how many command-line arguments the user will supply when the user runs your program. Your program must be flexible enough to handle zero or more command line arguments.

Here is a template of code to solve this problem:

```
while [ $1 ]
do
    someprocess $1
    shift
done
```

Explanatory notes:

1. The last command in the while list is a test command (using the left square bracket synonym).

2. With no specific test operator (e.g. -r, etc.), the default behaviour of the test command is to check if the string is null. The return code (\$?) is TRUE for a non-null string (something is there), and FALSE for a null string.
3. The shift command slides all command line parameters to the left, in this case moving the next unprocessed command line argument into the \$1 position. (The shift command also has the side-effect of decrementing the \$# variable.)
4. Once there are no more command line arguments, the loop terminates. Note that if there were no command line arguments to begin with (\$1 is NULL), the loop immediately terminates without entry into the body.

Exercise: Password entry loop

Write a bash script which will continually prompt a user to enter a password matching the control flow in the example below. If the user enters the correct password, grant access. If the user enters the wrong password, give the user another try (unlimited). Avoid code duplication.

Sample dialog:

```
$ ./whilecmdlist
What is the password? happy
--- Sorry, that is not the right password.
What is the password? access
--- Sorry, that is not the right password.
What is the password? Strawberry
Welcome to the system.
$
```

until command

The until command in Unix implements the General Loop construct discussed earlier.

The syntax is identical to the while command with the only difference that the logic of the conditional test is reversed. If the last command in `list1` is TRUE, the loop terminates.

Here's the same explanation for the until command:

Unix runs the `list1` and looks at the return code (`$?`) of the last command in `list1`. (If there's only one command in the `list1`, then that's the last command.) If the return code of this last command is FALSE, then `list2` (loop body) is run. Once the keyword "done" is reached, control returns to re-run `list1` and the return code of the last command in `list1` is checked. If the return code of this last command is TRUE, then the until loop terminates, and control continues after the keyword "done".

for command

The for command in Unix operates quite differently than what

is common in other computer languages like BASIC or Pascal. In particular, it is not a counted loop, rather one should think “iterative substitution”.

Syntax:

```
    for name in word ...  
do  
    list  
done
```

The list of words is expanded to create a list of items. Each of these items is substituted into the variable name one-at-a-time and the list is run.

E.g. 1: Multiple file rename problem

Consider the problem of adding a suffix to several files.

Those familiar with a DOS or Windows Power Shell environment could use a command like this:

```
ren assig* assig*.bak
```

Unfortunately a similar command does not work in Unix:

```
mv assig* assig*.backup
```

Nevertheless, this problem can be solved with the simple use of a for command:

```
for filevar in assig*
do
    mv ${filevar} ${filevar}.backup
done
```

Explanatory notes

1. The word list is the wildcard file specification `assig*`. On expansion, for the purposes of this example, assume that the result is three matching files: `assig1`, `assig2`, `assig3`.
2. Each of these words is substituted one-at-a-time in the loop variable `filevar`, and the body of the loop is run (code between `do` and `done`).
3. When the body of the loop is run with the variable substitutions, the effective commands generated are:

- `mv assig1 assig1.backup`
- `mv assig2 assig2.backup`
- `mv assig3 assig3.backup`

Additional Notes

1. There is no specific string concatenation operator in Unix. Simply placing text adjacently achieves the desired result. Here the string `".backup"` is appended to the `"assig1"`, etc.

2. As stated earlier, while the use of set braces { } is often omitted when spelling variables, here it helps to clarify the distinction between the variable name and surrounding text.

E.g. 2: Count the number of entries in a directory

Print the number of entries in the /etc directory.

```
count=0
for files in $(ls /etc)
do
    count=$(( $count +1 ))
done
echo The number of entries is $count
```

Explanatory notes

1. The word list consists of a command substitution call of the ls command on the /etc directory.
2. Expansion of this word list is the names of the entries in the /etc directory.
3. Each name is substituted into the loop variable files.
4. The body of the loop consists of incrementing a counter which will eventually hold the number of

entries.

5. Outside the loop, the total number of entries is printed in an appropriate message.

Eg 3: How to process multiple command line arguments (Method 2)

Let's say that you want to write your own Unix script that will accept any number of command line arguments:

```
mycmd parm1 parm2 ...
```

Here is another method, this one using the for command

```
for cmdarg in $*  
do  
    myprocess $cmdarg  
done
```

Explanatory notes

1. The special variable `$*` expands to all command line arguments present starting with `$1`.

2. Each of these words is substituted into the loop variable `cmdarg`.
3. The body of the loop is run against each command line argument (`myprocess` is a fictitious Unix command or script).
4. Unlike the `while` command example, there is no need for the `shift` command.
5. The value of `$#` is intact after the completion of the `for` loop.

break

The `break` statement causes control to exit the current `for`, `while`, or `until` loop. One must be judicious in its use to avoid “spaghetti” code. Except in rare circumstances, adherence to structure programming principles leads to clearer and less error-prone code. In many cases the use of `break` should be avoided.

<i>Poor (and rampant) use of break</i>	<i>Better code</i>
<pre> i=0 while true do if [\$i -eq 5] then echo Reached 5 break fi i=\$((\$i + 1)) done </pre>	<pre> i=0 while [\$i -ne 5] do i=\$((\$i + 1)) done echo Reached 5 </pre>
<p><i>Observations</i></p> <p>The use of an infinite loop (while true) is generally unnecessary (except perhaps in server processes) and should be avoided because it requires unstructured exit points leading to "spaghetti" code. Here, the break statement is used to jump out midway through the loop contributing to poor program structure.</p>	<p><i>Observations</i></p> <p>The purpose and limit of the loop is coded clearly in the while condition. Termination of the loop naturally follows coded instructions. There is no jumping out from the middle of the loop. The code is clearer, shorter, and better structured with a single entry and exit point for the loop.</p>

7. Quick Reference Guide

Common Notations

Here are some commonly used but easy-to-forget notations and syntax.

Notation	Typical usage with mini-example	Where to get more info
single quotes (')	Creates literal string protecting special characters. <code>grep 'Al Shaji' namelist</code>	Shell Variables
double quotes (")	Like single quotes but interprets \$, \ for variables. <code>echo "Price: \$value"</code>	
\$(cmd)	Command substitution. Runs cmd and substitutes output of cmd at position of call. <code>today=\$(date)</code>	man bash
set braces {}	Formal variable specification; substrings. <code>echo \${alpha:3:5}</code> 3 is offset (zero-based) and 5 is length	
double parentheses (())	Arithmetic expression. <code>a=\$((\$a + 1))</code>	man bash
left square bracket [Synonym for test command. <code>if [\$op = MR]</code> Closing right bracket is needed.	man test

Notation	Typical usage with mini-example	Where to get more info
<code>[[string =~ regex]]</code>	<p>Checks to see if regular expression (regex) is contained in string</p> <p>Example</p> <p><code>[[importing =~ port]]</code> will return TRUE</p> <p>Can be used with variables, too.</p> <p>Example</p> <p><code>[[\$title =~ \$word]]</code> returns TRUE if \$word is contained within \$title</p>	man bash
<p>Character classes in regular expressions</p> <p><code>[abc]</code></p>	<p>Square brackets represent a single character pattern. Thus,</p> <p><code>r[ao]n</code> matches ran, ron, or run.</p> <p><code>[Pp]olish</code> matches Polish or polish</p>	man regex.7

FAQs

Q1a: How do make my script executable?

Q1b: I'm getting an error message, "-bash: ./myscript: Permission denied"

A1: You need to issue the chmod command, either:

- `chmod 700 myscript`
- `chmod u+x myscript`

Acknowledgements

I would first like to thank my wife, Sylvie, for her holistic support.

There are many people in the Toronto Metropolitan University (formerly Ryerson University) community who supported me academically, technically and editorially. While it is not possible to mention everyone who influenced the development of this book, I wish to acknowledge the following individuals:

Anne-Marie Brinsmead, Maryam Davoudpour, Alex Ferworn, Greg Gay, Shannon Koumphol, Ann Ludbrook, Allen Pader, Mehrdad Tirandazian, Muhammad Waqas, Sally Wilson, and Leonora Zefi.

Without their support and endorsement, this book would not have been created.