

Adventures with the Linux Command Line

First Internet Edition

William Shotts

A LinuxCommand.org Book

Copyright ©2014-2021, William Shotts



Except where otherwise noted, this work is licensed under
<http://creativecommons.org/licenses/by-nc-nd/3.0/>

This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. To view a copy of this license, visit the link above or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042.

Linux® is the registered trademark of Linus Torvalds. All other trademarks belong to their respective owners.

This book is part of the LinuxCommand.org project, a site for Linux education and advocacy devoted to helping users of legacy operating systems migrate into the future. You may visit the LinuxCommand.org project at <https://linuxcommand.org>.

Release History

Version	Date	Description
21.10	October 1, 2021	First Internet Edition.

Table of Contents

“Amaze your friends! Baffle your enemies!”.....	v
What this book is about.....	v
Who should read this book.....	vi
What’s in the first Internet edition.....	vi
How to read this book.....	vi
Acknowledgments.....	vii
1 Midnight Commander.....	1
Features.....	1
Availability.....	1
Invocation.....	1
Screen Layout.....	2
Using the Keyboard and Mouse.....	3
Navigation and Browsing.....	3
Viewing and Editing Files.....	6
Tagging Files.....	9
We Need a Playground.....	10
Power Features.....	18
The User Menu.....	23
Summing Up.....	27
Further Reading.....	27
2 Terminal Multiplexers.....	29
Some Historical Context.....	29
GNU Screen.....	29
tmux.....	36
byobu.....	42
Summing Up.....	46
Further Reading.....	46
3 Less Typing.....	49
Aliases and Shell Functions.....	49
Command Line Editing.....	50
Completion.....	53
Programmable Completion.....	54
Summing Up.....	55
Further Reading.....	56
4 More Redirection.....	57
What’s Really Going On.....	57
Duplicating File Descriptors.....	58
exec.....	58

/dev/tty.....	60
Noclobber.....	60
Summing Up.....	61
Further Reading.....	61
5 tput.....	63
Availability.....	63
What it Does/How it Works.....	63
Reading Terminal Attributes.....	65
Controlling the Cursor.....	66
Text Effects.....	67
Clearing the Screen.....	70
Making Time.....	72
Summing Up.....	75
Further Reading.....	75
6 dialog.....	77
Features.....	77
Availability.....	80
How it Works.....	81
Before and After.....	85
Limitations.....	89
Summing Up.....	89
Further Reading.....	89
7 AWK.....	91
History.....	91
Availability.....	91
So, What's it Good For?.....	91
How it Works.....	92
Invocation.....	94
The Language.....	94
Summing Up.....	113
Further Reading.....	113
8 Power Terminals.....	115
A Typical Modern Terminal.....	115
Past Favorites.....	117
Modern Power Terminals.....	121
Terminals for Other Platforms.....	135
Summing Up.....	139
Further Reading.....	140
9 Other Shells.....	143
The Evolution of Shells.....	143
Modern Implementations.....	144

Changing to Another Shell.....	152
Summing Up.....	153
Further Reading.....	153
10 Vim, with Vigor.....	155
Let's Get Started.....	155
Getting Help.....	156
Starting a Script.....	158
Using the Shell.....	159
Buffers.....	160
Tabs.....	160
Color Schemes.....	162
Marks and File Marks.....	163
Visual Mode.....	164
Indentation.....	164
Power Moves.....	167
Text Formatting.....	169
Macros.....	172
Registers.....	173
Insert Sub-Modes.....	174
Mapping.....	175
Snippets.....	176
Finishing Our Script.....	177
Using External Commands.....	178
File System Management and Navigation.....	180
One Does Not Live by Code Alone.....	182
More .vimrc Tricks.....	184
Summing Up.....	185
Further Reading.....	186
11 source.....	187
Configuration Files.....	187
Function Libraries.....	188
Let's Not Forget .bashrc.....	190
Security Considerations and Other Subtleties.....	193
Summing Up.....	194
Further Reading.....	194
12 Coding Standards Part 1: Our Own.....	195
A Coding Standard of Our Own.....	196
Summing Up.....	208
Further Reading.....	208
13 Coding Standards Part 2: new_script.....	211
Installing new_script.....	211
Options and Arguments.....	211

Creating Our First Template.....	212
Looking at the Template.....	214
Summing Up.....	217
Further Reading.....	218
14 SQL.....	219
A Little Theory: Tables, Schemas, and Keys.....	219
Database Engines/Servers.....	221
sqlite3.....	221
Creating a Table and Inserting Our Data.....	222
Creating and Deleting Tables.....	224
Data Types.....	224
Inserting Data.....	225
Doing Some Queries.....	225
Controlling the Output.....	226
Sorting Output.....	228
Subqueries.....	229
Updating Tables.....	232
Deleting Rows.....	234
Adding and Deleting Columns.....	235
Joins.....	237
Views.....	238
Indexes.....	238
Triggers and Stored Procedures.....	240
Performing Backups.....	242
Generating Your Own Datasets.....	242
Summing Up.....	245
Further Reading.....	245
Index.....	247

“Amaze your friends! Baffle your enemies!”

And the story continues.

A long time ago (shortly after I finished college in 1977) I got my first computer, a TRS-80 model 1. In the early days of personal computing, many computer peripherals, such as printers and floppy disk drives, were very costly and as a result, my dad (an electrical engineer) and I would cruise electronic surplus stores looking for deals on devices we could attach to our new computer.

One day, as we were searching a large warehouse near the University of Maryland, I came across a store display featuring a small, clear, plastic box containing a battery, a few computer chips, and several randomly blinking LEDs. While the little device served no useful purpose, it did have blinking lights. Above it hung a handwritten sign that read simply:

Amaze Your Friends! Baffle Your Enemies!

The excitement pervasive in the early days of personal computing is hard to explain to people today. The computers of that period seem so laughably primitive by today's standards but it was a revolution nonetheless and there were many explorers mapping the new, uncharted territory of personal empowerment and technical innovation.

People entering the computer field now are at a disadvantage compared to those of us who came up in the 1970s and 1980s. The early computers were very simple, slow, and had tiny memories. All the attributes you need if you really want to understand how computers work. Today, computers are so fast, and software so large and complex that you can't see the computer underneath anymore and that's a shame. You can't see the beauty of what they do.

However, we are now in the midst of another revolution, extremely low-cost computing. Devices like the Raspberry Pi single board computer offer the opportunity to work on systems more simple and basic compared to contemporary desktop and mobile devices. But make no mistake, these low-cost computers are powerful. In fact, a \$35 Raspberry Pi compares favorably to the \$30,000 Unix workstations I used in the early 1990s.

What this book is about

This volume is a sequel/supplement to my first book, *The Linux Command Line* (TLCL) and as such, we will be referring back to the first book frequently, so if you don't already have a copy, please download one from LinuxCommand.org or, if you prefer, pickup a printed copy from your favorite bookseller or library. This time around we are going to

build on our experience with the command line and add some more tools and techniques to our repertoire. Like the first book, this second volume is not a book about Linux system administration, rather it is a collection of topics that I consider both fun and interesting. We will cover many tools that will be of interest to budding system administrators, but the tools were chosen for other reasons. Sometimes they were chosen because they are “classic” Unix, others because they are just “something you should know,” but mostly topics were chosen because I find them fun and interesting. Personal computing, after all, should be about doing things that are fun and interesting just as it was in the early days.

Who should read this book

This is a book for explorers and creators looking for adventure. I think computers are the coolest things ever and if you share that feeling of excitement with every new thing you can get your computer to do then you have come to the right place. Many people today come into the computer field only in hopes of developing enough skill to get a job. There is nothing wrong with that of course. Everyone needs to earn a decent living, but there is more to life than that. There is beauty and there is love, and if you are wise (and lucky) you will find these things in your career. Computers are powerful tools that, in the right hands, can improve the human condition. I think it’s a worthy goal to leave the world a little better than the way you found it. I hope you do too.

What's in the first Internet edition

For the most part, you can think of this book as an expansion of Part 3 of TLCL. In fact, I considered some of these topics for inclusion in the first book, but ran out of space for them. That being said, this is definitely a work-in-progress. Future editions will contain more chapters and the existing chapters will contain additional content and the chapters will likely appear in a different order. Typography and layout will improve too.

How to read this book

This book is not as linear as TLCL so feel free to skip around. Some adventures are prerequisites for later ones. If an adventure requires an earlier one, it will be indicated. A few of the adventures call for supplemental material (typically code samples and datasets) that can be downloaded from LinuxCommand.org.

Acknowledgments

I would once again like to thank my ever-faithful editor Karen Shotts for her nitpicking my text. Also a big shout out to my many readers who made my first book such a success. If you find a typo or a bug in my code please drop me a note at bshotts@users.sourceforge.net for possible correction in a future edition. Thanks.

And as always, use your powers only for good. Let the adventures begin!

1 Midnight Commander

At the beginning of Chapter 4 in TLCL there is a discussion of GUI-based file managers versus the traditional command line tools for file manipulation such as `cp`, `mv`, and `rm`. While many common file manipulations are easily done with a graphical file manager, the command line tools provide additional power and flexibility.

In this adventure we will look at Midnight Commander, a character-based directory browser and file manager that bridges the two worlds of the familiar graphical file manager and the common command line tools.

The design of Midnight Commander is based on a common concept in file managers: dual directory panes where the listings of two directories are shown at the same time. The idea is that files are moved or copied from the directory shown in one pane to the directory shown in the other. Midnight Commander can do this, and much, much more.

Features

Midnight Commander is quite powerful and boasts an extensive set of features:

- Performs all the common file and directory manipulations such as copying, moving, renaming, linking, and deleting.
- Allows manipulation of file and directory permissions.
- Can treat remote systems (via FTP or SSH) as though they were local directories.
- Can treat archive files (like `.tar` and `.zip`) as though they were local directories.
- Allows creation of a user-defined “hotlist” of frequently used directories.
- Can search for files based on file name or file contents, and treat the search results like a directory.

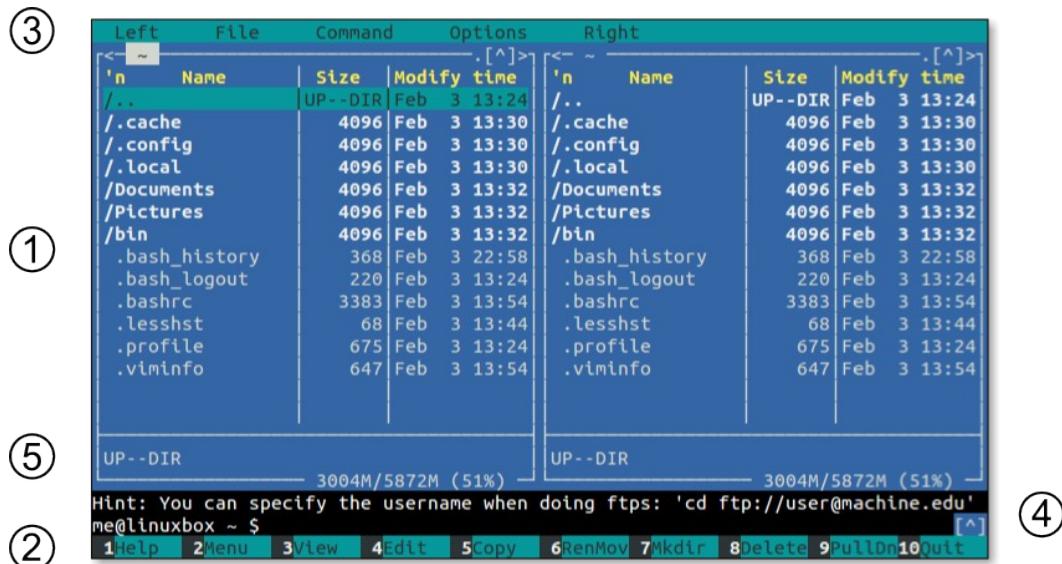
Availability

[Midnight Commander](#) is part of the GNU project. It is installed by default in some Linux distributions, and is almost always available in every distribution’s software repositories as the package “mc”.

Invocation

To start Midnight Commander, enter the command `mc` followed optionally by either 1 or 2 directories to browse at start up.

Screen Layout



Midnight Commander screen layout

1. Left and Right Directory Panels

The center portion of the screen is dominated by two large *directory panels*. One of the two panels (called the *current panel*) is active at any one time. To change which panel is the current panel, press the `Tab` key.

2. Function Key Labels

The bottom line on the display contains function key (F1-F10) shortcuts to the most commonly used functions.

3. Menu Bar

The top line of the display contains a set of pull-down menus. These can be activated by pressing the `F9` key.

4. Command Line

Just above the function key labels there is a shell prompt. Commands can be entered in the usual manner. One especially useful command is `cd` followed by a directory pathname. This will change the directory shown in the current directory panel.

5. Mini-Status Line

At the very bottom of the directory panel and above the command line is the *mini-status line*. This area is used to display supplemental information about the currently selected item such as the targets of symbolic links.

Using the Keyboard and Mouse

Being a character-based application with a lot of features means Midnight Commander has a lot of keyboard commands, some of which it shares with other applications; others are unique. This makes Midnight Commander a bit challenging to learn. Fortunately, Midnight Commander also supports mouse input on most terminal emulators (and on the console if the `gpm` package is installed), so it's easy to pick up the basics. Learning the keyboard commands is needed to take full advantage of the program's features, however.

Another issue when using the keyboard with Midnight Commander is interference from the window manager and the terminal emulator itself. Many of the function keys and Alt-key combinations that Midnight Commander uses are intercepted for other purposes by the terminal and window manager.

To work around this problem, Midnight Commander allows the `Esc` key to function as a Meta-key. In cases where a function key or Alt-key combination is not available due to interference from outside programs, use the `Esc` key instead. For example, to input the `F1` key, press and release the `Esc` key followed by the “1” key (use “0” for `F10`). The same method works with troublesome Alt-key combinations. For example, to enter `Alt-t`, press and release the `Esc` key followed by the “t” key. To close dialog boxes in Midnight Commander, press the `Esc` key twice.

Navigation and Browsing

Before we start performing file operations, it's important to learn how to use the directory panels and navigate the file system.

As we can see, there are two directory panels, the left panel and the right panel. At any one time, one of the panels is active and is called the *current panel*. The other panel is conveniently called the *other panel* in the Midnight Commander documentation.

The current panel can be identified by the highlighted bar in the directory listing, which can be moved up and down with the arrow keys, `PgUp`, `PgDn`, etc. Any file or directory which is highlighted is said to be *selected*.

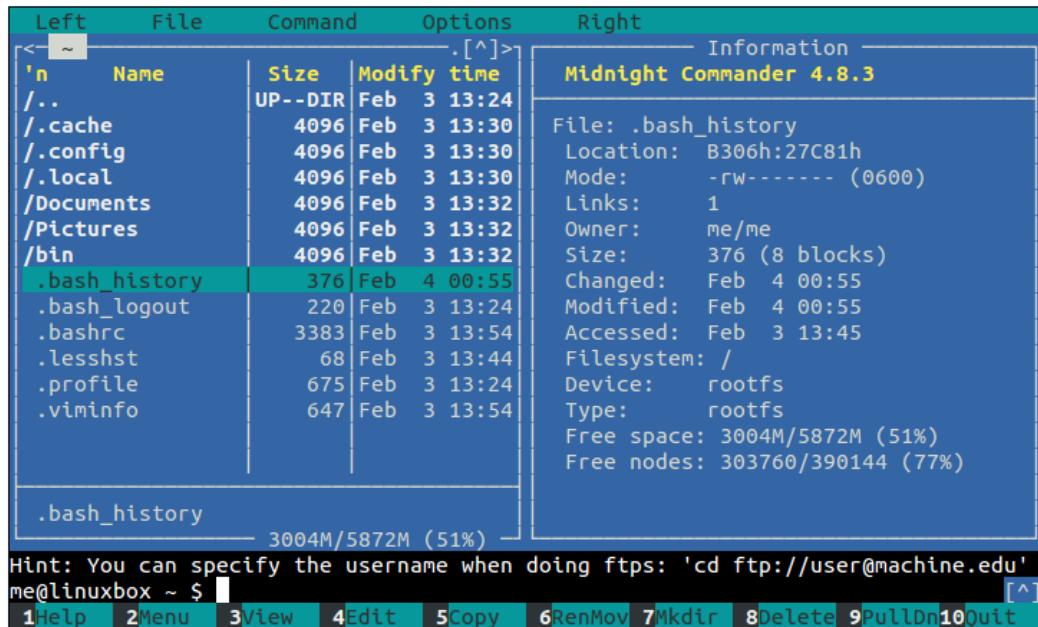
Select a directory and press `Enter`. The current directory panel will change to the selected directory. Highlighting the topmost item in the listing selects the parent directory. It is also possible to change directories directly on the command line below the directory panels. To do so, simply enter `cd` followed by a path name as usual.

Pressing the `Tab` key switches the current panel.

Changing the Listing Format

The directory listing can be displayed in several different formats. Pressing `Alt-t` cycles through them. There is a dual column format, a format resembling the output of `ls -l`, and others.

There is also an “information mode.” This will display detailed file system information in the other panel about the selected item in the current panel. To invoke this mode, type `Ctrl-x i`. To return the other panel to its normal state, type `Ctrl-x i` again.



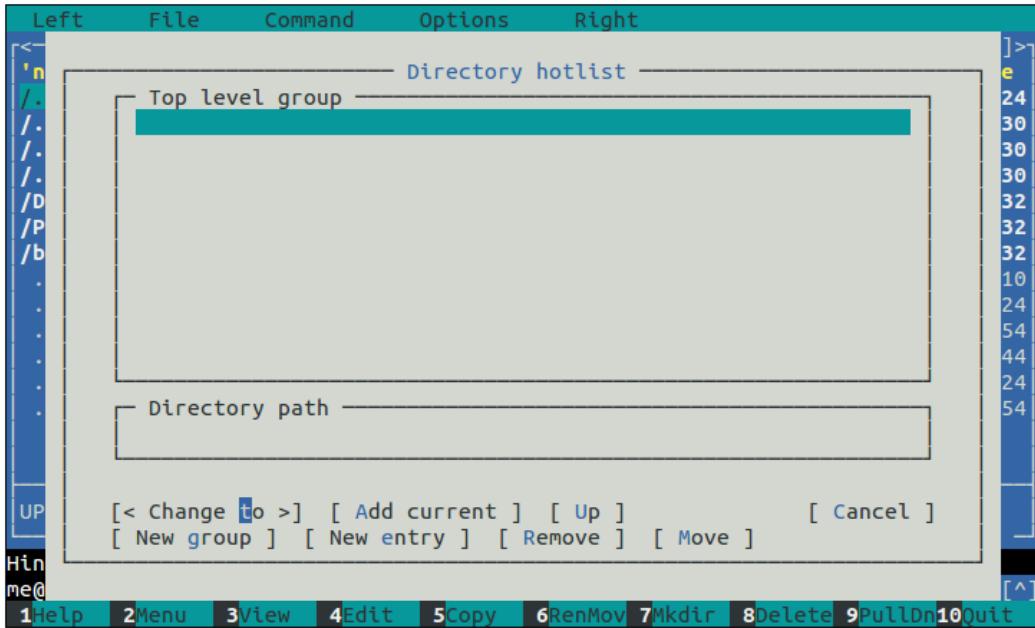
Directory panel in information mode

Setting the Directory on the Other Panel

It is often useful to select a directory in the current panel and have its contents listed on the other panel; for example, when moving files from a parent directory into a subdirectory. To do this, select a directory and type `Alt-o`. To force the other panel to list the same directory as the current panel, type `Alt-i`.

The Directory Hotlist

Midnight Commander can store a list of frequently visited directories. This “hotlist” can be displayed by pressing `Ctrl-\`.



Directory hotlist

To add a directory to the hotlist while browsing, select a directory and type `Ctrl-x h`.

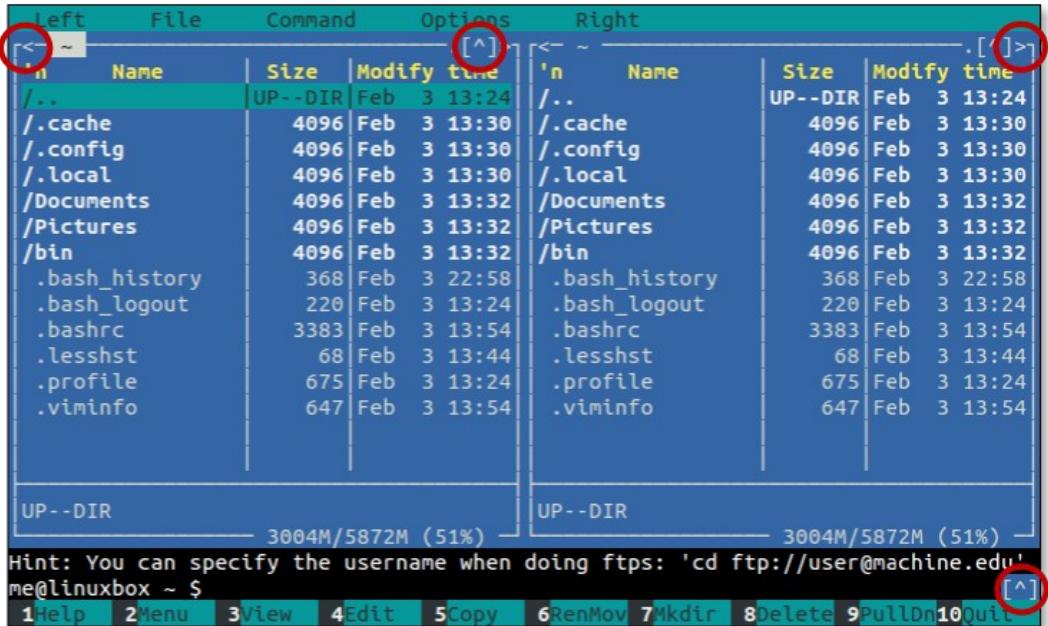
Directory History

Each directory panel maintains a list of directories that it has displayed. To access this list, type `Alt-H`. From the list, a directory can be selected for browsing. Even without the history list display, we can traverse the history list forward and backward by using the `Alt-u` and `Alt-y` keys respectively.

Using the Mouse

We can perform many Midnight Commander operations using the mouse. A directory panel item can be selected by clicking on it and a directory can be opened by double clicking. Likewise, the function key labels and menu bar items can be activated by clicking on them. What is not so apparent is that the directory history can be accessed and traversed. At the top of each directory panel there are small arrows (circled in the image below). Clicking on them will show the directory history (the up arrow) and move forward and backward through the history list (the right and left arrows).

There is also an arrow to the extreme lower right edge of the command line which reveals the command line history.



Directory and command line history mouse controls

Viewing and Editing Files

An activity often performed while directory browsing is examining the content of files. Midnight Commander provides a capable file viewer which can be accessed by selecting a file and pressing the F3 key.

```
/home/me/.bashrc          649/3383  19%
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

1Help 2UnWrap 3Quit 4Hex 5Goto 6 7Search 8Raw 9Format 10Quit
```

File viewer

As we can see, when the file viewer is active, the function key labels at the bottom of the screen change to reveal viewer features. Files can be searched and the viewer can quickly go to any position in the file. Most importantly, files can be viewed in either ASCII (regular text) or hexadecimal, for those cases when we need a really detailed view.

	0x0000000000																0%
00000000	2B	20	7E	2F	2E	62	61	73	68	72	63	3A	20	65	78	65	# ~./.bashrc: exe
00000010	63	75	74	65	64	20	62	79	20	62	61	73	68	28	31	29	cuted by bash(1)
00000020	20	66	6F	72	20	6E	6F	6E	2D	6C	6F	67	69	6E	20	73	for non-login s
00000030	68	65	6C	6C	73	2E	0A	23	20	73	65	65	20	2F	75	73	hells..# see /us
00000040	72	2F	73	68	61	72	65	2F	64	6F	63	2F	62	61	73	68	r/share/doc/bash
00000050	2F	65	78	61	6D	70	6C	65	73	2F	73	74	61	72	74	75	/examples/startu
00000060	70	2D	66	69	6C	65	73	20	28	69	6E	20	74	68	65	20	p-files (in the
00000070	70	61	63	6B	61	67	65	20	62	61	73	68	2D	64	6F	63	package bash-doc
00000080	29	0A	23	20	66	6F	72	20	65	78	61	6D	70	6C	65	73).# for examples
00000090	0A	0A	23	20	49	66	20	6E	6F	74	20	72	75	6E	66	..# If not runni	
000000A0	6E	67	20	69	6E	74	65	72	61	63	74	69	76	65	6C	79	ng interactively
000000B0	2C	20	64	6F	6E	27	74	20	64	6F	20	61	6E	79	74	68	, don't do anyth
000000C0	69	6E	67	0A	5B	20	2D	7A	20	22	24	50	53	31	22	20	ing.[-z "\$PS1"
000000D0	5D	20	26	26	20	72	65	74	75	72	6E	0A	0A	23	20	64] && return..# d
000000E0	6F	6E	27	74	20	70	75	74	20	64	75	70	6C	69	63	61	on't put duplica
000000F0	74	65	20	6C	69	6E	65	73	20	6F	72	20	6C	69	6E	65	te lines or line
00000100	73	20	73	74	61	72	74	69	6E	67	20	77	69	74	68	20	s starting with
00000110	73	70	61	63	65	20	69	6E	20	74	68	65	20	68	69	73	space in the his
00000120	74	6F	72	79	2E	0A	23	20	53	65	65	20	62	61	73	68	tory..# See bash
00000130	28	31	29	20	66	6F	72	20	6D	6F	72	65	20	6F	70	74	(1) for more opt
00000140	69	6F	6E	73	0A	48	49	53	54	43	4F	4E	54	52	4F	4C	ions.HISTCONTROL
00000150	3D	69	67	6E	6F	72	65	62	6F	74	68	0A	0A	23	20	61	=ignoreboth..# a
1 Help	2 Edit	3 Quit	4 Ascii	5 Goto	6 Save	7 HxSrch	8 Raw	9 Format	10 Quit								

File viewer in hexadecimal mode

It is also possible to put the other panel into “quick view” mode to view the currently selected file. This is especially nice if we are browsing a directory full of text files and want to rapidly view the files, as each time a new file is selected in the current panel, it’s instantly displayed in the other. To start quick view mode, type `Ctrl-x q`.

```

Left      File      Command      Options      Right
[...]
/home/me/.bashrc          11%
# ~/.bashrc: executed by bash(1) for n
on-login shells.
# see /usr/share/doc/bash/examples/sta
rtup-files (in the package bash-doc)
# for examples

# If not running interactively, don't
do anything
[ -z "$PS1" ] && return

# don't put duplicate lines or lines s
tarting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't ov
erwrite it

```

[<-- ~ -->]	Name	Size	Modify time
'n		UP--DIR	Feb 3 13:24
/..		4096	Feb 3 13:30
/.cache		4096	Feb 3 13:30
/.config		4096	Feb 3 13:30
/.local		4096	Feb 3 13:30
/Documents		4096	Feb 3 13:32
/Pictures		4096	Feb 3 13:32
/bin		4096	Feb 3 13:32
/playground		4096	Feb 10 18:08
.bash_history		438	Feb 10 18:22
.bash_logout		220	Feb 3 13:24
.bashrc		3383	Feb 3 13:54
.lesshst		68	Feb 3 13:44
.profile		675	Feb 3 13:24
.viminfo		647	Feb 3 13:54
	.bashrc		2995M/5872M (51%)

Hint: Completion works on all input lines in all dialogs. Just press M-Tab.

me@linuxbox ~ \$

1 Help 2 UnWrap 3 View 4 Hex 5 Goto 6 7 Search 8 Raw 9 PullDn 10 [^]

Quick view mode

Once in quick view mode, we can press `Tab` and the focus changes to the other panel in quick view mode. This will change the function key labels to a subset of the full file viewer. To exit the quick view mode, press `Tab` to return to the directory panel and press `Alt-i`.

Editing

Since we are already viewing files, we will probably want to start editing them too. Midnight Commander accommodates us with the `F4` key, which invokes a text editor loaded with the selected file. Midnight Commander can work with the editor of your choice. On Debian-based systems we are prompted to make a selection the first time we press `F4`. Debian suggests `nano` as the default selection, but various flavors of `vim` are also available along with Midnight Commander's own built-in editor, `mcedit`. We can try out `mcedit` on its own at the command line for a taste of this editor.

```

/home/me/.bashrc  [---]  0 L:[ 1+ 0  1/106] *(0   /3383b) 0035 0x023
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

```

1 Help 2 Save 3 Mark 4 Replace 5 Copy 6 Move 7 Search 8 Delete 9 Pull Dn 10 Quit

mcedit

Tagging Files

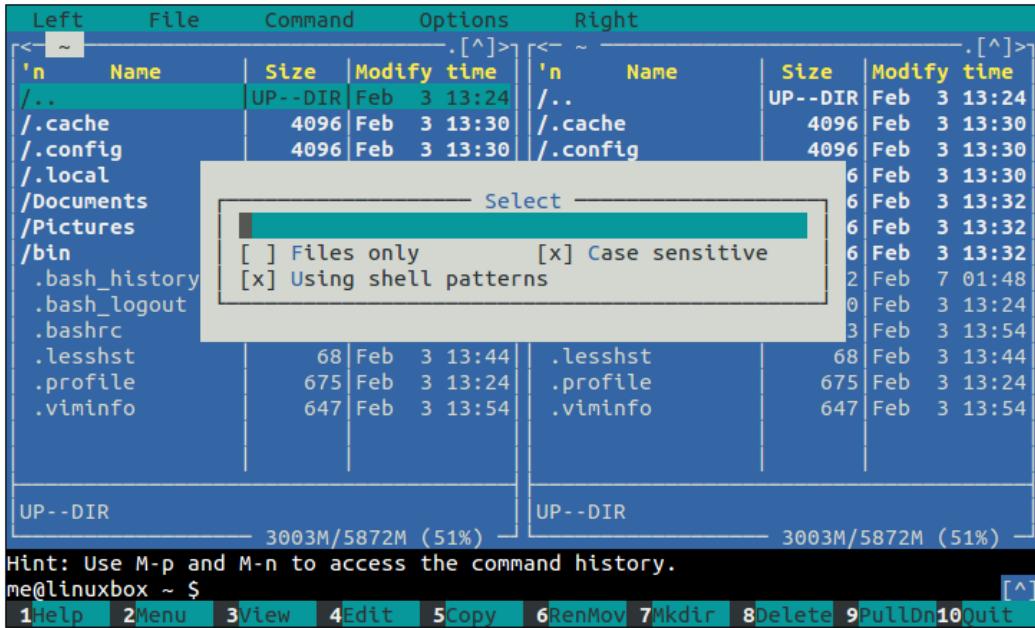
We have already seen how to select a file in the current directory panel by simply moving the highlight, but operating on a single file is not of much use. After all, we can perform those kinds of operations more easily by entering commands directly on the command line. However, we often want to operate on multiple files. This can be accomplished through *tagging*. When a file is tagged, it is marked for some later operation such as copying. This is why we choose to use a file manager like Midnight Commander. When one or more files are tagged, file operations (such as copying) are performed on the tagged files and selection has no effect.

Tagging Individual Files

To tag an individual file or directory, select it and press the `Insert` key. To untag it, press the `Insert` key again.

Tagging Groups of Files

To tag a group of files or directories according to a selection criteria, such as a wildcard pattern, press the `+` key. This will display a dialog where the pattern may be specified.



File tagging dialog

This dialog stores a history of patterns. To traverse it, use Ctrl up and down arrows.

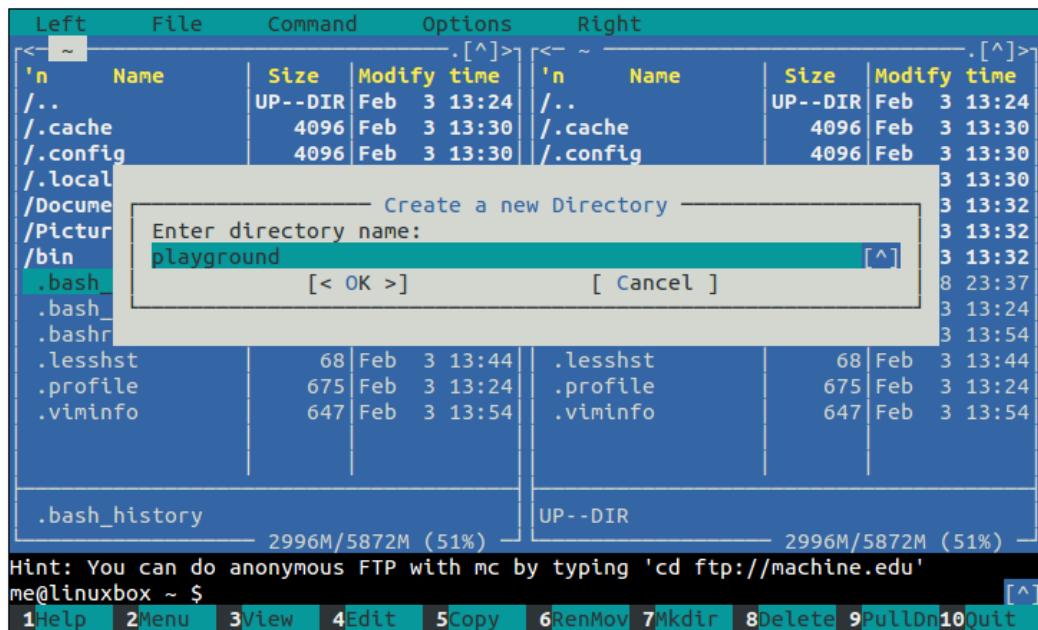
It is also possible to un-tag a group of files. Pressing the / key will cause a pattern entry dialog to display.

We Need a Playground

To explore the basic file manipulation features of Midnight Commander, we need a “playground” like we had in Chapter 4 of TLCL.

Creating Directories

The first step in creating a playground is creating a directory called, aptly enough, `playground`. First, we will navigate to our home directory, then press the F7 key.



Create Directory dialog

Type “playground” into the dialog and press Enter. Next, we want the other panel to display the contents of the playground directory. To do this, highlight the `playground` directory and press Alt-o.

Now let’s put some files into our playground. Press Tab to switch the current panel to the playground directory panel. We’ll create a couple of subdirectories by repeating what we did to create `playground`. Create subdirectories `dir1` and `dir2`. Finally, using the command line, we will create a few files:

```
me@linuxbox: ~/playground $ touch file1 file2 "ugly file"
```

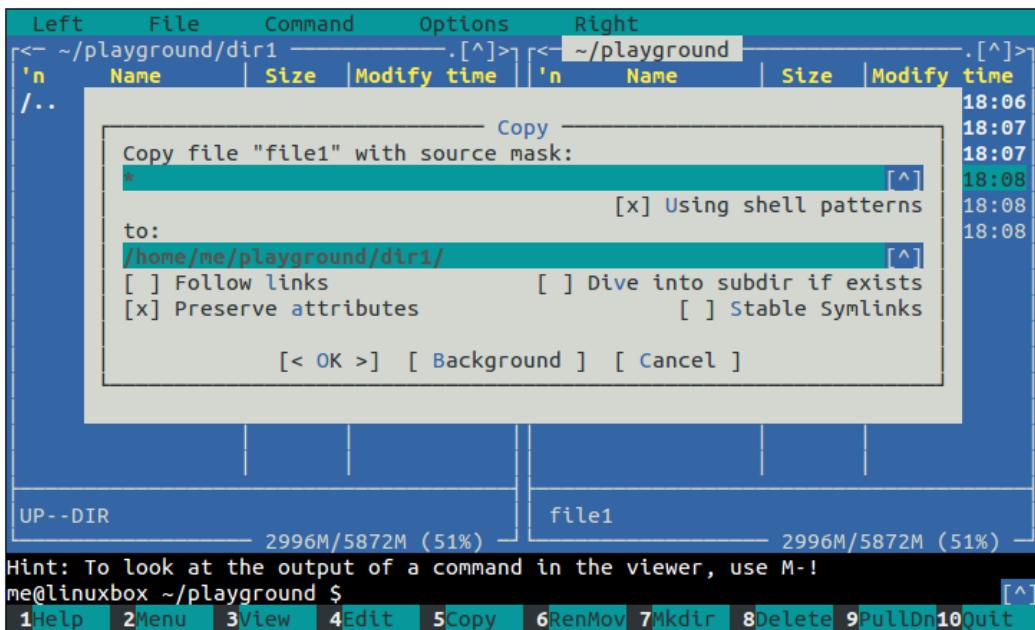
Left	File	Command	Options	Right
'n Name	Size	Modify time	.[>]	'n Name
..	UP--DIR	Feb 3 13:24	[<]	..
./cache	4096	Feb 3 13:30		/dir1
./config	4096	Feb 3 13:30		/dir2
./local	4096	Feb 3 13:30		file1
/Documents	4096	Feb 3 13:32		file2
/Pictures	4096	Feb 3 13:32		ugly file
/bin	4096	Feb 3 13:32		
/playground	4096	Feb 10 18:08		
.bash_history	400	Feb 8 23:37		
.bash_logout	220	Feb 3 13:24		
.bashrc	3383	Feb 3 13:54		
.lesshst	68	Feb 3 13:44		
.profile	675	Feb 3 13:24		
.viminfo	647	Feb 3 13:54		
.bash_history	2996M	5872M (51%)		/dir2
				2996M/5872M (51%)
Hint: Setting the CDPATH variable can save you keystrokes in cd commands.				
me@linuxbox ~ /playground \$ [^]				
1 Help	2 Menu	3 View	4 Edit	5 Copy
6 RenMov	7 Mkdir	8 Delete	9 PullDn	10 Quit

The playground

Copying and Moving Files

Okay, here is where things start to get weird.

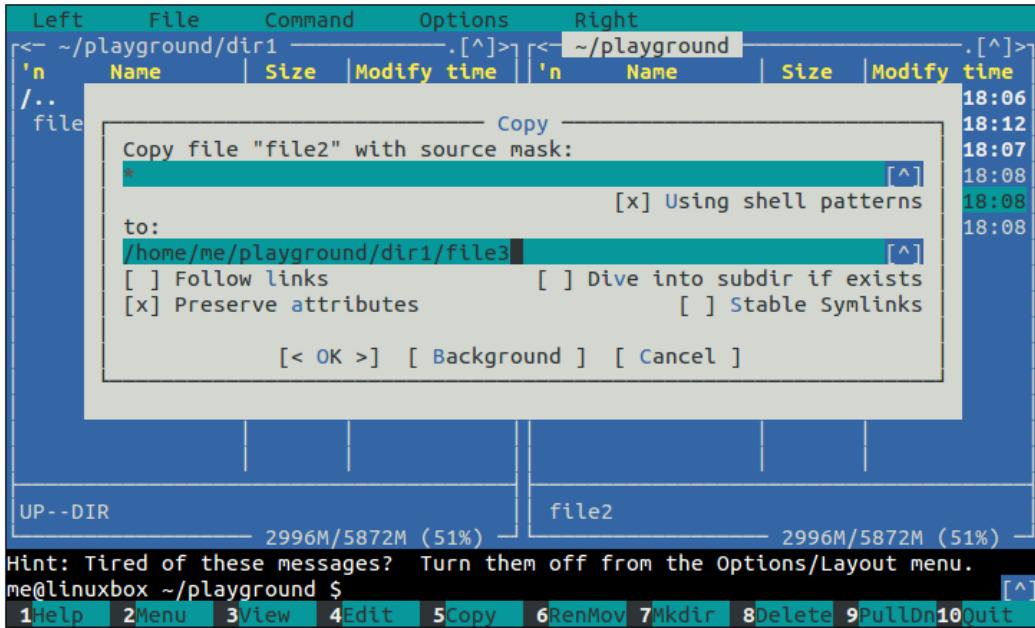
Select `dir1`, then press `Alt-o` to display `dir1` in the other panel. Select the file `file1` and press F5 to copy (The F6-RenMov command is similar). We are now presented with this formidable-looking dialog box:



Copy dialog

To see Midnight Commander's default behavior, just press `Enter` and `file1` is copied into directory `dir1` (i.e., the file is copied from the directory displayed in current panel to the directory displayed in the other panel).

That was straightforward, but what if we want to copy `file2` to a file in `dir1` named `file3`? To do this, we select `file2` and press `F5` again and enter the new filename into the Copy dialog:

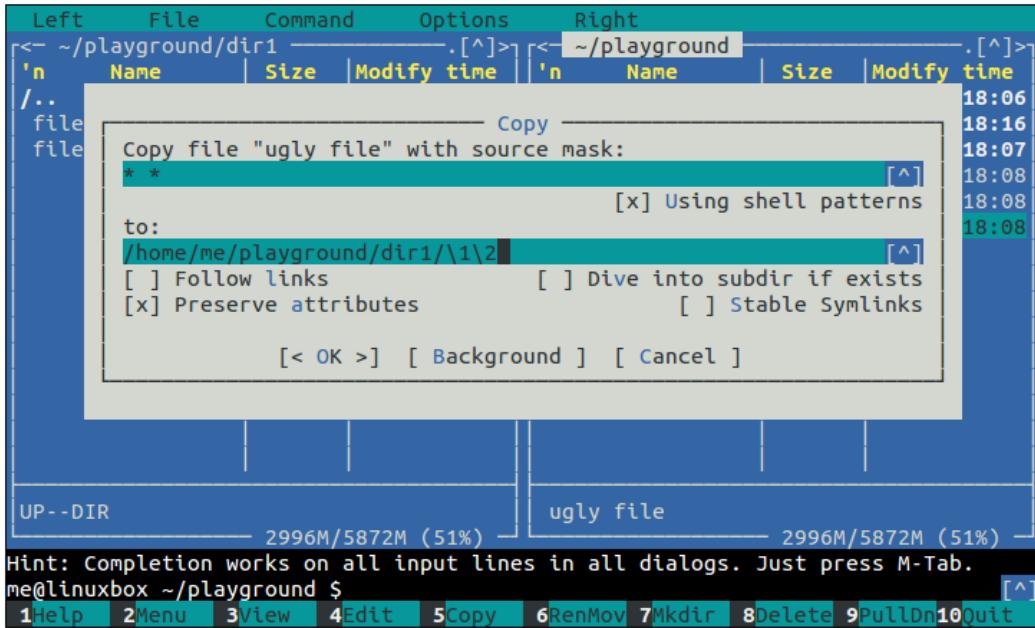


Renaming a file during copy

Again, this is pretty straightforward. But let's say we tagged a group of files and wanted to copy and rename them as they are copied (or moved). How would we do that? Midnight Commander provides a way of doing it, but it's a little strange.

The secret is the source mask in the copy dialog. At first glance, it appears that the source mask is simply a file selection wildcard, but first appearances can be deceiving. The mask does filter files as we would expect, but only in a limited way. Unlike the range of wildcards available in the shell, the wildcards in the source mask are limited to "?" (for matching single characters) and "*" (for matching multiple characters). What's more, the wildcards have a special property.

It works like this: let's say we had a file name with an embedded space such as "ugly file" and we want to copy (or move) it to `dir1` as the file "uglyfile", instead. Using the source mask, we could enter the mask "`* *`" which means break the source file name into two blocks of text separated by a space. This wildcard pattern will match the file `ugly file`, since its name consists of two strings of characters separated by a space. Midnight Commander will associate each block of text with a number starting with 1, so block 1 will contain "ugly" and block 2 will contain "file". Each block can be referred to by a number as with regular expression grouping. So to create a new file name for our target file without the embedded space, we would specify "\1\2" in the "to" field of the copy dialog like so:



Using grouping

The “?” wildcard behaves the same way. If we make the source mask “????? ?????” (which again matches the file `ugly file`), we now have eight pieces of text that we can rearrange at will. For example, we could make the “to” mask “\8\7\6\5\4\3\2\1”, and the resulting file name would be “`elifylgu`”. Pretty neat.

Midnight Commander can also perform case conversion on file names. To do this, we include some additional escape sequences in the to mask:

- `\u` Converts the next character to uppercase.
- `\U` Converts all characters to uppercase until another sequence is encountered.
- `\l` Converts the next character to lowercase.
- `\L` Converts all characters to lowercase until another sequence is encountered.

So if we wanted to change the name `ugly file` to camel case, we could use the mask “`u\L\1\u\L\2`” and we would get the name `UglyFile`.

Creating Links

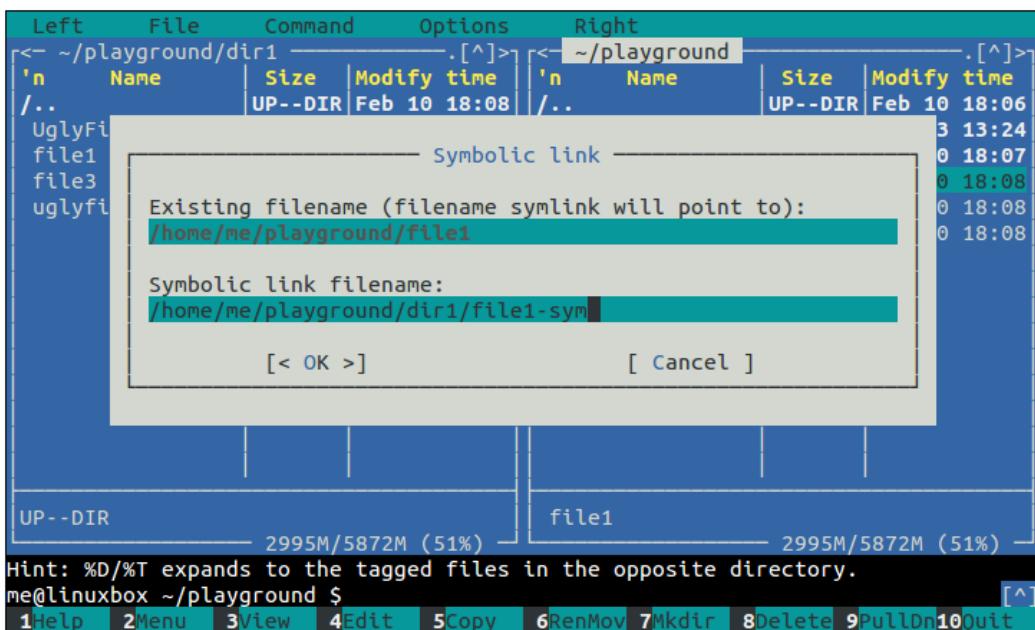
Midnight Commander can create both hard and symbolic links. They are created using these 3 keyboard commands which cause a dialog to appear where the details of the link can be specified:

- `ctrl-x 1` creates a hard link, in the directory shown in the current panel.

- `Ctrl-x s` creates a symbolic link in the directory shown in the other panel, using an absolute directory path.
- `Ctrl-x v` creates a symbolic link in the directory shown in the other panel, using a relative directory path.

The two symbolic link commands are basically the same. They differ only in the fact that the paths suggested in the Symbolic Link dialog are absolute or relative.

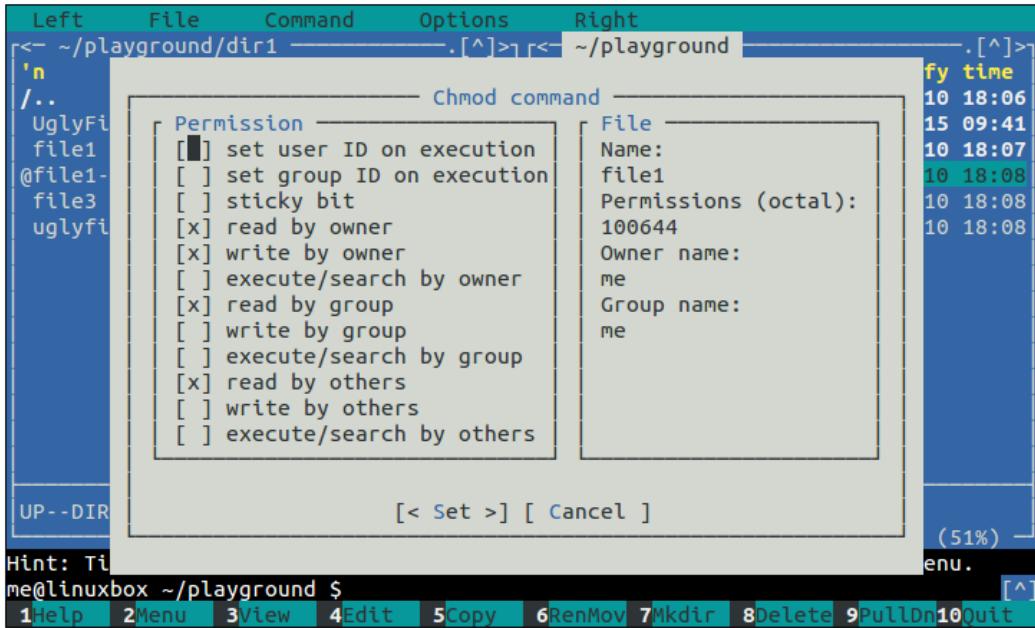
We'll demonstrate creating a symbolic link by creating a link to `file1`. To do this, we select `file1` in the current panel and type `Ctrl-x s`. The Symbolic Link dialog appears and we can either enter a name for the link or we can accept the program's suggestion. For the sake of clarity, we will change the name to `file1-sym`.



Symbolic link dialog

Setting File Modes and Ownership

File modes (i.e., permissions) can be set on the selected or tagged files by typing `Ctrl-x c`. Doing so will display a dialog box in which each attribute can be turned on or off. If Midnight Commander is being run with superuser privileges, file ownership can be changed by typing `Ctrl-x o`. A dialog will be displayed where the owner and group owner of selected/tagged files can be set.



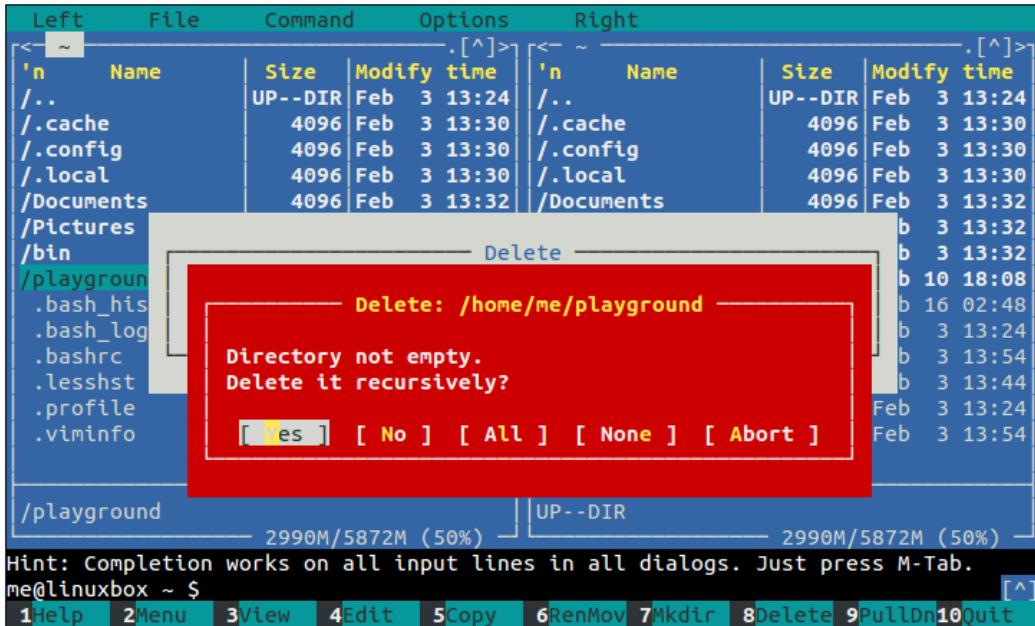
Chmod dialog

To demonstrate changing file modes, we will make `file1` executable. First, we will select `file1` and then type `Ctrl-x c`. The Chmod command dialog will appear, listing the file's mode settings. By using the arrow keys we can select the check box labeled "execute/search by owner" and toggle its setting by using the space bar.

Deleting Files

Pressing the `F8` key deletes the selected or tagged files and directories. By default, Midnight Commander always prompts the user for confirmation before deletion is performed.

We're done with our playground for now, so it's time to clean up. We will enter `cd` at the shell prompt to get the current panel to list our home directory. Next, we will select `playground` and press `F8` to delete the playground directory.



Delete confirmation dialog

Power Features

Beyond basic file manipulation, Midnight Commander offers a number of additional features, some of which are very interesting.

Virtual File Systems

Midnight Commander can treat some types of archive files and remote hosts as though they are local file systems. Using the `cd` command at the shell prompt, we can access these.

For example, we can look at the contents of tar files. To try this out, let's create a compressed tar file containing the files in the `/etc` directory. We can do this by entering this command at the shell prompt:

```
me@linuxbox ~ $ tar czf etc.tgz /etc
```

Once this command completes (there will be some “permission denied” errors but these don't matter for our purposes), the file `etc.tgz` will appear among the files in the current panel. If we select this file and press `Enter`, the contents of the archive will be displayed in the current panel. Notice that the shell prompt does not change as it does with ordinary directories. This is because while the current panel is displaying a list of files like before, Midnight Commander cannot treat the virtual file system in the same way as a real one. For example, we cannot delete files from the tar archive, but we can copy files from the archive to the real file system.

Virtual file systems can also treat remote file systems as local directories. In most versions of Midnight Commander, both FTP and FISH (Fil es transferred over SHell) protocols are supported and, in some versions, SMB/CIFS as well.

As an example, let's look at the software library FTP site at Georgia Tech, a popular repository for Linux software. Its name is `ftp.glibc.gatech.edu`. To connect with `/pub` directory on this site and browse its files, we enter this `cd` command:

```
me@linuxbox ~ $ cd ftp://ftp.glibc.gatech.edu/pub
```

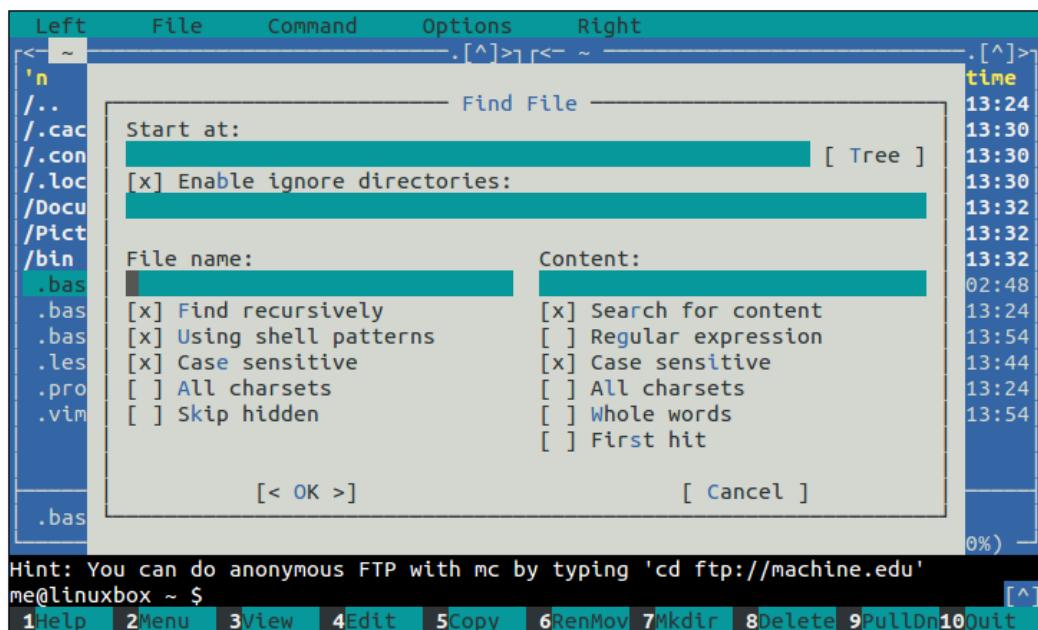
Since we don't have write permission on this site, we cannot modify any any files there, but we can copy files from the remote server to our local file system.

The FISH protocol is similar. This protocol can be used to communicate with any Unix-like system that runs a secure shell (SSH) server. If we have write permissions on the remote server, we can operate on the remote system's files as if they were local. This is extremely handy for performing remote administration. The `cd` command for FISH protocol looks like this:

```
me@linuxbox ~ $ cd sh://user@remotehost/dir
```

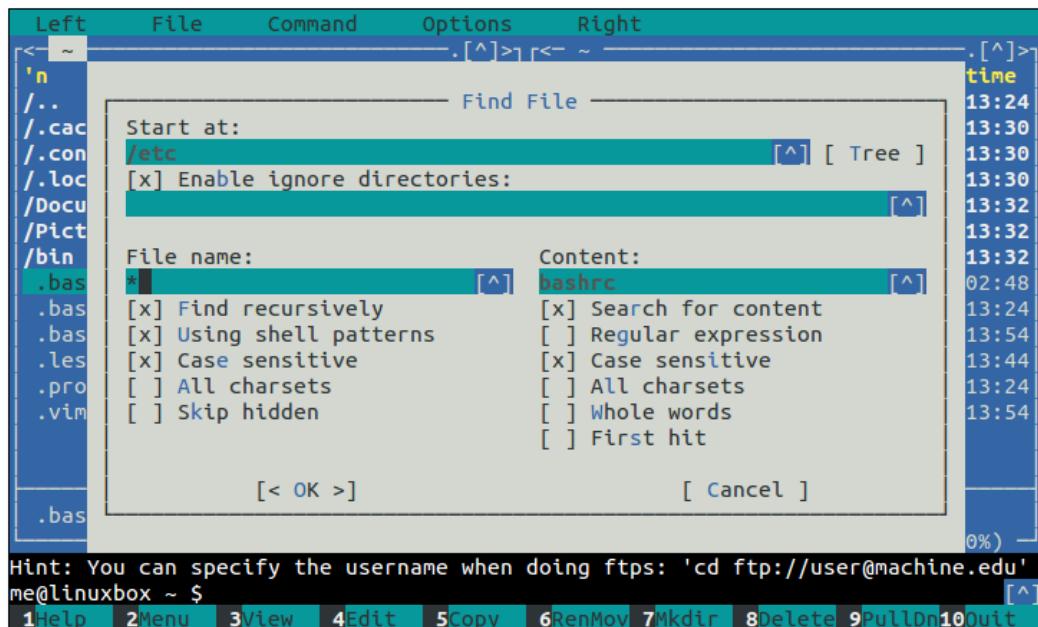
Finding Files

Midnight Commander has a useful file search feature. When invoked by pressing `Alt-?`, the following dialog will appear:



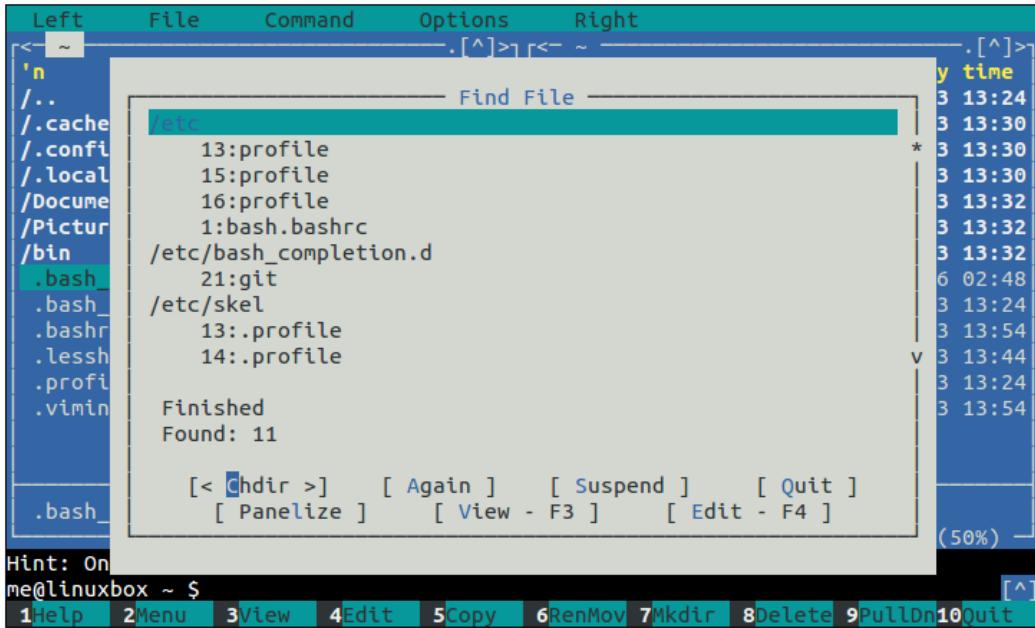
Find dialog

On this dialog we can specify: where the search is to begin, a colon-separated list of directories we would like to skip during our search, any restriction on the names of the files to be searched, and the content of the files themselves. This feature is well-suited to searching large trees of source code or configuration files for specific patterns of text. For example, let's look for every file in /etc that contains the string "bashrc". To do this, we would fill in the dialog as follows:



Search for files containing “bashrc”

Once the search is completed, we will see a list of files which we can view and/or edit.

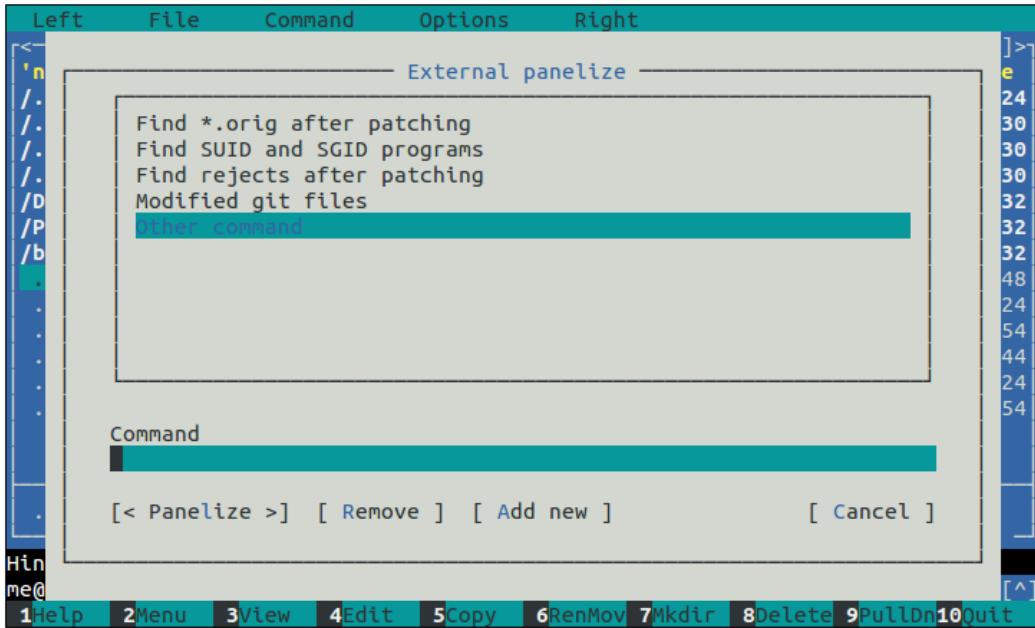


Search results

Panelizing

There is a button at the bottom of the search results dialog labeled “Panelize.” If we click it, the search results become the contents of the current panel. From here, we can act on the files just as we can with any others.

In fact, we can create a panelized list from any command line program that produces a list of path names. For example, the `find` program. To do this, we use Midnight Commander’s “External Panelize” feature. Type `Ctrl-x !` and the External Panelize dialog appears:



External panelize dialog

On this dialog we see a predefined list of panelized commands. Midnight Commander allows us to store commands for repeated use. Let's try it by creating a panelized command that searches the system for every file whose name has the extension .JPG starting from the current panel directory. Select “Other command” from the list and type the following command into the “Command” field:

```
find . -type f -name "*.JPG"
```

After typing the command we can either press `Enter` to execute the command or, for extra fun, we can click the “Add new” button and assign our command a name and save it for future use.

Subshells

We may, at any time, move from the Midnight Commander to a full shell session and back again by pressing `Ctrl-o`. The subshell is a copy of our normal shell, so whatever environment our usual shell establishes (aliases, shell functions, prompt strings, etc.) will be present in the sub-shell as well. If we start a long-running command in the sub-shell and press `Ctrl-o`, the command is suspended until we return to the sub-shell. Note that once a command is suspended, Midnight Commander cannot execute any further external commands until the suspended command terminates.

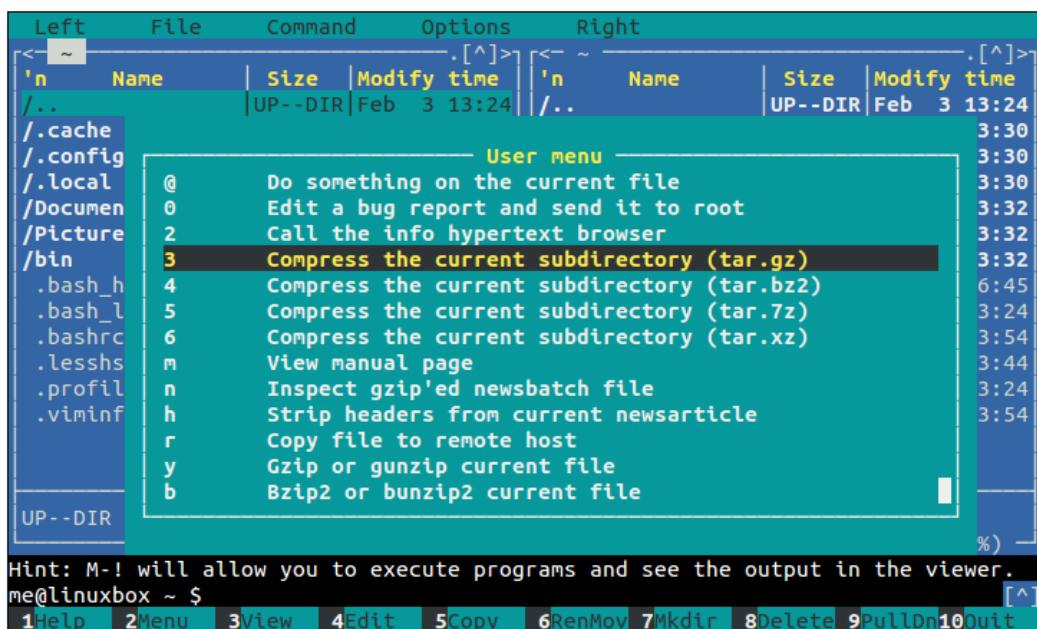
The User Menu

So far we have avoided discussion of the mysterious F2 command. This is the user menu, which may be Midnight Commander's most powerful and useful feature. The user menu is, as the name suggests, a menu of user-defined commands.

When we press the F2 key, Midnight Commander looks for a file named `.mc.menu` in the current directory. If the file does not exist, Midnight Commander looks for `~/.config/mc/menu`. If that file does not exist, then Midnight Commander falls back to a system-wide menu file named `/usr/share/mc/mc.menu`.

The neat thing about this scheme is that each directory can have its own set of user menu commands, so that we can create commands appropriate to the contents of the current directory. For example, if we have a “Pictures” directory, we can create commands for processing images; if we have a directory full of HTML files, we can create commands for managing a web site, and so on.

So, after we press F2 the first time, we are presented with the default user menu that looks something like this:



The User Menu

Editing the User Menu

The default user menu contains several example entries. These are by no means set in stone. We are encouraged to edit the menu and create our own entries. The menu file is ordinary text and it can be edited with any text editor, but Midnight Commander provides

a menu editing feature found in the “Command” pulldown menu. The entry is called “Edit menu file.”

If we select this entry, Midnight Commander offers us a choice of “Local” and “User.” The Local entry allows us to edit the `.mc.menu` file in the current directory while selecting User will cause us to edit the `~/.config/mc/menu` file. Note that if we select Local and the current directory does not contain a menu file, Midnight Commander will copy the default menu file into current directory as a starting point for our editing.

Menu File Format

Some parts of the user menu file format are pretty simple; other parts, not so much. We’ll start with the simple parts first.

A menu file consists of one or more entries. Each entry contains:

- A single character (usually a letter) that will act as a hot key for the entry when the menu is displayed.
- Following the hot key, on the same line, is the description of the menu entry as it will appear on the menu.
- On the following lines are one or more commands to be performed when the menu entry is selected. These are ordinary shell commands. Any number of commands may be specified, so quite sophisticated operations are possible. Each command must be indented by at least one space or tab.
- A blank line to separate one menu entry from the next.
- Comments may appear on their own lines. Each comment line starts with a `#` character.

Here is an example user menu entry that creates an HTML template in the current directory:

```
# Create a new HTML file
H  Create a new HTML file
{ echo "<html>"
echo "\t<head>\n\t</head>"
echo "\t<body>\n\t</body>"
echo "</html>"; } > new_page.html
```

Notice the absence of the `-e` option on the `echo` commands used in this example. Normally, the `-e` option is required to interpret the backslash escape sequences like `\t` and `\n`. The reason they are omitted here is that Midnight Commander does not use `bash` as the shell when it executes user menu commands. It uses `sh` instead. Different distributions use different shell programs to emulate `sh`. For example, Red Hat-based distributions use `bash` but Debian-based distributions like Ubuntu and Raspberry Pi OS

use `dash` instead. `dash` is a compact shell program that is `sh` compatible but lacks many of the features found in `bash`. The `dash` man page describes the features of that shell.

This command will reveal which program is actually providing the `sh` emulation (i.e., is symbolically linked to `sh`):

```
me@linuxbox ~ $ ls -l /bin/sh
```

Macros

With that bit of silliness out of the way, let's look at how we can get a user menu entry to act on currently selected or tagged files. First, it helps to understand a little about how Midnight Commander executes user menu commands. It's done by writing the commands to a file (essentially a shell script) and then launching `sh` to execute the contents of the file. During the process of writing the file, Midnight Commander performs *macro substitution*, replacing embedded symbols in the menu entry with alternate values. These macros are single alphabetic characters preceded by a percent sign. When Midnight Commander encounters one of these macros, it substitutes the value the macro represents. Here are the most commonly used macros:

Macro	Meaning
<code>%f</code>	Selected file's name
<code>%x</code>	Selected file's extension
<code>%b</code>	Selected file's name stripped of extension (basename)
<code>%d</code>	Name of the current directory
<code>%t</code>	The list of tagged files
<code>%s</code>	If files are tagged, they are used, else the selected file is used.

List of common macros

Let's say we wanted to create a user menu entry that would resize a JPEG image using the ever-handy `convert` program from the ImageMagick suite. Using macros, we could write a menu entry like this, which would act on the currently selected file:

```
# Resize an image using convert
R Resize image to fit within 800 pixel bounding square
size=800
convert "%f" -resize ${size}x${size} "%b-$size.%x"
```

Using the `%b` and `%x` macros, we are able to construct a new output file name for the resized image. There is still one potential problem with this menu entry. It's possible to run the menu entry command on a directory, or a non-image file (Doing so would not be good).

We could include some extra code to ensure that `%f` is actually the name of an image file, but Midnight Commander also provides a method for only displaying menu entries appropriate to the currently selected (or tagged) file(s).

Conditionals

Midnight Commander supports two types of *conditionals* that affect the behavior of a menu entry. The first, called an *addition conditional* determines if a menu entry is displayed. The second, called *default conditional* sets the default entry on a menu.

A conditional is added to a menu entry just before the first line. A conditional starts with either a + (for an addition) or a = (for a default) followed by one or more *sub-conditions*. Sub-conditions are separated by either a | (meaning or) or a & (meaning and) allowing us to express some complex logic. It is also possible to have a combined addition and default conditional by beginning the conditional with =+ or +=. Two separate conditionals, one addition and one default, are also permitted preceding a menu entry.

Let's look at sub-conditions. They consist of one of the following:

Sub-condition	Description
f <i>pattern</i>	Match currently selected file
F <i>pattern</i>	Match last selected in other panel
d <i>pattern</i>	Match currently selected directory
D <i>pattern</i>	Match last selected directory in other panel
t <i>type</i>	Type of currently selected file
T <i>type</i>	Type of last selected file in other panel
x <i>filename</i>	File is executable
! <i>sub-cond</i>	Negate result of sub-condition

List of sub-conditions

pattern is either a shell pattern (i.e., wildcards) or a regular expression according to the global setting configured in the Options/Configuration dialog. This setting can be overridden by adding `shell_patterns=0` as the first line of the menu file. A value of 1 forces use of shell patterns, while a value of 0 forces regular expressions instead.

type is one or more of the following:

Type	Description
r	regular file
d	directory
n	not a directory
l	link
x	executable file
t	tagged
c	character device
b	block device
f	FIFO (pipe)
s	socket

List of file types

While this seems really complicated, it's not really that bad. To change our image resizing entry to only appear when the currently selected file has the extension `.jpg` or `.JPG`, we would add one line to the beginning of the entry (regular expressions are used in this example):

```
#    Resize an image using convert
+
+ f \.jpg$ | f \.JPG$
R  Resize image to fit within 800 pixel bounding square
  size=800
  convert "%f" -resize ${size}x${size} "%b-${size}.%x"
```

The conditional begins with `+` meaning that it's an addition condition. It is followed by two sub-conditions. The `|` separating them signifies an “or” relationship between the two. So, the finished conditional means “display this entry if the selected file name ends with `.jpg` or the selected file name ends with `.JPG`.”

The default menu file contains many more examples of conditionals. It's worth a look.

Summing Up

Even though it takes a little time to learn, Midnight Commander offers a lot of features and facilities that make file management easier when using the command line. This is particularly true when operating on a remote system where a graphical user interface may not be available. The user menu feature is especially good for specialized file management tasks. With a little configuration, Midnight Commander can become a powerful tool in our command line arsenal.

Further Reading

- The *Midnight Commander man page* is extensive and discusses even more features than we have covered here.
- midnight-commander.org is the official site for the project.

2 Terminal Multiplexers

It's easy to take the terminal for granted. After all, modern terminal emulators like gnome-terminal, konsole, and the others included with Linux desktop environments are feature-rich applications that satisfy most of our needs. But sometimes we need more. We need to have multiple shell sessions running in a single terminal. We need to display more than one application in a single terminal. We need to move a running terminal session from one computer to another. In short, we need a *terminal multiplexer*.

Terminal multiplexers are programs that can perform these amazing feats. In this adventure, we will look at three examples: GNU screen, tmux, and byobu.

Some Historical Context

If we were to go back in time to say, the mid-1980s, we might find ourselves staring at a computer terminal; a box with an 80-column wide, 24-line high display and a keyboard connected to a shared, central Unix computer via an RS-232 serial connection and, possibly, an acoustic-coupler modem and a telephone handset. On the display screen there might be a shell prompt not unlike the prompt we see today during a Linux terminal session. However, unlike today, the computer terminal of the 1980s did not have multiple windows or tabs to display multiple applications or shell sessions. We only had one screen and that was it. Terminal multiplexers were originally developed to help address this limitation. A terminal multiplexer allows multiple sessions and applications to be displayed and managed on a single screen. While modern desktop environments and terminal emulator programs support multiple windows and tabbed terminal sessions, which mitigate the need of terminal multiplexers for some purposes, terminal multiplexers still offer some features that will greatly enhance our command-line experience.

GNU Screen

GNU screen goes way back. First developed in 1987, screen appears to be the first program of its type and it defined the basic feature set found in all subsequent terminal multiplexers.

Availability

As its name implies, GNU screen is part of the GNU Project. Though it is rarely installed by default, it is available in most distribution repositories as the package “screen”.

Invocation

We can start using GNU screen by simply entering the `screen` command at the shell prompt. Once the command is launched, we will be presented with a shell prompt.

Multiple Windows

At this point, screen is running and has created its first *window*. The terminology used by screen is a little confusing. It is best to think of it this way: screen manages a *session* consisting of one or more *windows* each containing a shell or other program. Furthermore, screen can divide a terminal display into multiple *regions*, each displaying the contents of a window.

Whew! This will start to make sense as we move forward.

In any case, we have screen running now, and it's displaying its first window. Let's enter a command in the current window:

```
me@linuxbox: ~ $ top
```

top - 18:45:46 up 22 days, 2:32, 7 users, load average: 0.15, 0.83, 0.98												
Tasks: 119 total, 1 running, 115 sleeping, 0 stopped, 3 zombie												
%Cpu(s): 10.0 us, 8.3 sy, 0.0 ni, 81.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st												
KiB Mem: 448180 total, 417992 used, 30188 free, 13780 buffers												
KiB Swap: 102396 total, 4 used, 102392 free, 186144 cached												
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	
25788	me	20	0	4672	1392	1024	R	1.6	0.3	0:00.43	top	
2273	bshott	20	0	105m	8044	5452	S	1.3	1.8	281:34.93	lxpanel	
10888	bshott	20	0	4764	2904	1168	S	1.3	0.6	29:34.91	tmux	
2227	root	20	0	40912	32m	5268	S	1.0	7.4	314:00.60	Xorg	
7	root	20	0	0	0	0	S	0.3	0.0	32:05.08	rcu_preempt	
2271	bshott	20	0	15624	5048	2696	S	0.3	1.1	24:21.51	openbox	
2325	bshott	20	0	269m	16m	8860	S	0.3	3.8	178:19.80	geany	
2337	bshott	20	0	118m	12m	6824	S	0.3	2.8	98:14.53	lxterminal	
24582	me	20	0	9808	1520	884	S	0.3	0.3	0:00.07	sshd	
1	root	20	0	2144	672	568	S	0.0	0.1	1:07.28	init	
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd	
3	root	20	0	0	0	0	S	0.0	0.0	10:46.14	ksoftirqd/0	
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H	
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh	
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_sched	
10	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	khelper	
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kdevtmpfs	

Initial screen window

So far, so good. Now, let's create another window. To do this, we type `Ctrl-a` followed by the character “c”. Our terminal screen should clear and we should see a new shell prompt. So what just happened to our first window with `top` running in it? It's still there, running in the background. We can return to the first window by typing `Ctrl-a p` (think “p” for “previous”).

Before we go any further, let's talk about the keyboard. Controlling screen is pretty simple. Every command consists of `Ctrl-a` (called the “command prefix” or “escape sequence”) followed by another character. We have already seen two such commands: `Ctrl-a c` to create a new window, and `Ctrl-a p` to switch from the current window to the previous one. Typing the command `Ctrl-a ?` will display a list of all the commands.

GNU screen has several commands for switching from one window to another. Like the “previous” command, there is a “next” command `Ctrl-a n`. Windows are numbered, starting with 0, and may be chosen directly by typing `Ctrl-a` followed by a numeral from 0 to 9. It is also possible list all the windows by typing `Ctrl-a "`. This command will display a list of windows, where we can choose a window.

Num	Name	Flags
0	bash	\$
1	bash	\$

Screen window list

As we can see, windows have names. The default name for a window is the name of the program the window was running at the time of its creation, hence both of our windows are named “bash”. Let’s change that. Since we are running `top` in our first window, let’s make its name reflect that. Switch to the first window using any of the methods we have discussed, and type the command `Ctrl-a A` and we will be prompted for a window name. Simple.

Okay, so we have created some windows, how do we destroy them? A window is destroyed whenever we terminate the program running in it. After all windows are destroyed, screen itself will terminate. Since both of our windows are running bash, we need only exit each respective shell to end our screen session. In the case of a program that refuses to terminate gracefully, `Ctrl-a k` will do the trick.

Let’s terminate the shell running `top` by typing `q` to exit `top` and then enter `exit` to terminate `bash`, thereby destroying the first window. We are now taken to the remaining window still running its own copy of `bash`. We can confirm this by typing `Ctrl-a "` to view the window list again.

It's possible to create windows and run programs without an underlying shell. To do this, we enter `screen` followed by the name of the program we wish to run, for example:

```
me@linuxbox: ~ $ screen vim ~/.bashrc
```

We can even do this in a screen window. Issuing a `screen` command in a screen window does not invoke a new copy of screen. It tells the existing instance of screen to carry out an operation like creating a new window.

Copy and Paste

Given that GNU screen was developed for systems that have neither a graphical user interface nor a mouse, it makes sense that screen would provide a way of copying text from one screen window to another. It does this by entering what is called *scrollback mode*. In this mode, screen allows the text cursor to move freely throughout the current window and through the contents of the *scrollback buffer*, which contains previous contents of the window.

We start scrollback mode by typing `Ctrl-a [`. In scrollback mode we can use the arrow keys and the `Page Up` and `Page Down` keys to navigate the scrollback buffer. To copy text, we first need to mark the beginning and end of the text we want to copy. This is done by moving the text cursor to the beginning of the desired text and pressing the space bar. Next, we move the cursor to the end of the desired text (which is highlighted as we move the cursor) and press the space bar again to mark the end of the text to be copied. Marking text exits scrollback mode and copies the marked text into screen's internal buffer. We can now paste the text into any screen window. To do this, we go to the desired window and type `Ctrl-a]`.

```

SCREEN(1)                                     SCREEN(1)

NAME
    screen - screen manager with VT100/ANSI terminal emulation

SYNOPSIS
    screen [ -options ] [ cmd [ args ] ]
    screen -r [[pid.]tty[.host]]
    screen -r sessionowner/[[pid.]tty[.host]]

DESCRIPTION
    Screen is a full-screen window manager that multiplexes a physical terminal between several processes (typically interactive shells). Each virtual terminal provides the functions of a DEC VT100 terminal and, in addition, several control functions from the ISO 6429 (ECMA 48, ANSI X3.64) and ISO 2022 standards (e.g. insert/delete line and support for multiple character sets). There is a scrollback history buffer for each virtual terminal and a copy-and-paste mechanism that allows moving text regions between windows.

    When screen is called, it creates a single window with a shell in it (or the specified command) and then gets out of your way so that you can use the program as you normally would. Then, at any time, you can
Manual page screen(1) line 1 (press h for help or q to quit)

```

Text marked for copying

Multiple Regions

GNU screen can also divide the terminal display into separate regions, each providing a view of a screen window. This allows us to view 2 or more windows at the same time. To split the terminal horizontally, type the command `Ctrl-a S`, to split it vertically, type `Ctrl-a |`. Newly created regions are empty (i.e., they are not associated with a window). To display a window in a region, first move the focus to the new region by typing `Ctrl-a Tab` and then either create a new window, or chose an existing window to display using any of the window selection commands we have already discussed. Regions may be further subdivided to smaller regions and we can even display the same window in more than one region.

```

top - 18:52:29 up 22 days, 2:39, 8 users, load average: 0.59, 0.43, 0.71
Tasks: 122 total, 1 running, 118 sleeping, 0 stopped, 3 zombie
%Cpu(s): 14.6 us, 15.0 sy, 0.0 ni, 70.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 448180 total, 420312 used, 27868 free, 13780 buffers
KiB Swap: 102396 total, 4 used, 102392 free, 186232 cached

      PID USER      PR  NI    VIRT   RES   SHR S %CPU %MEM     TIME+ COMMAND
10888 bshotts  20   0  4764 2908 1168 S  2.6  0.6 29:42.91 tmux
30365 me  20   0  4672 1368 1028 R  1.3  0.3  0:00.57 top
 2227 root    20   0 40912 32m 5268 S  1.0  7.4 314:04.72 Xorg
 2273 bshotts 20   0 105m 8044 5452 S  1.0  1.8 281:38.40 lxpanel
 1 bash
TOP(1)                               User Commands                         TOP(1)

NAME
      top - display Linux processes

SYNOPSIS
      top [-hv|-bcHisS -d delay -n limit -u|U user -p pid -w [cols]]
      The traditional switches '-' and whitespace are optional.

Manual page top(1) line 1 (press h for help or q to quit)
 0 bash

```

Regions

Using multiple regions is very convenient when working with large terminal displays. For example, if we split the display into two horizontal regions, we can edit a script in one region and perform testing of the script in the other. Or we could read a man page in one region and try out a command in the other.

There are two commands for deleting regions: `Ctrl-a Q` removes all regions except the current one, and `Ctrl-a X` removes the current region. Note that removing a region does not remove its associated window. Windows continue to exist until they are destroyed.

Detaching Sessions

Perhaps the most interesting feature of screen is its ability to detach a session from the terminal itself. Just as it is able to display its windows on any region of the terminal, screen can also display its windows on any terminal or no terminal at all.

For example, we could start a screen session on one computer, say at the office, detach the session from the local terminal, go home and log into our office computer remotely, and reattach the screen session to our home computer's terminal. During the intervening time, all jobs on our office computer have continued to execute.

There are a number of commands used to manage this process.

- `screen -list` lists the screen sessions running on a system. If there is more than one session running, the `pid.tty.host` string shown in the listing can be appended to the `-d/-D` and `-r/-R` options below to specify a particular session.

- `screen -d -r` detaches a screen session from the previous terminal and reattaches it to the current terminal.
- `screen -D -R` detaches a screen session from the previous terminal, logs the user off the old terminal and attaches the session to the new terminal creating a new session if no session existed. According to the screen documentation, this is the author's favorite.

The `-d/-D` and `-r/-R` options can be used independently, but they are most often used together to detach and reattach an existing screen session in a single step.

We can demonstrate this process by opening two terminals. Launch screen on the first terminal and create a few windows. Now, go to the second terminal and enter the command `screen -D -R`. This will cause the first terminal to vanish (the user is logged off) and the screen session to move to the second terminal fully intact.

Customizing Screen

Like many of the interactive GNU utilities, screen is very customizable. During invocation, screen reads the `/etc/screenrc` and `~/.screenrc` files if they exist. While the list of customizable features is extensive (many having to do with terminal display control on a variety of Unix and Unix-like platforms), we will concern ourselves with key bindings and startup session configuration since these are the most commonly used.

First, let's look at a sample `.screenrc` file:

```
# This is a comment

# Set some key bindings

bind k          # Un-bind the "k" key (set it to do nothing)
bind K kill    # Make `Ctrl-a K` destroy the current window
bind } history # Make `Ctrl-a }` copy and paste the current
               # command line

# Define windows 7, 8, and 9 at startup

screen -t "mdnghit cmdr" 7 mc
screen -t htop 8 htop
screen -t syslog 9 tailf /var/log/syslog
```

As we can see, the format is pretty simple. The `bind` directive is followed by the key and the screen command it is to be bound to. A complete list of the screen commands can be found in the screen man page. All of the screen commands we have discussed so far are simply key bindings like those in the example above. We can redefine them at will.

The three lines at the end of our example `.screenrc` file create windows at startup. The commands set the window title (the `-t` option), a window number, and a command for the window to contain. This way, we can set up a screen session to be automatically built

when we start screen which contains a complete multi-window, command-line environment running all of our favorite programs.

tmux

Despite its continuing popularity, GNU screen has been criticized for its code complexity (to the point of being called “unmaintainable”) and its resource consumption. In addition, it is reported that screen is no longer actively developed. In response to these concerns, a newer program, tmux, has attracted widespread attention.

tmux is modern, friendlier, more efficient, and generally superior to screen in most ways. Conceptually, tmux is very similar to screen in that it also supports the concept of sessions, windows and regions (called *panes* in tmux). In fact, it even shares a few keyboard commands with screen.

Availability

tmux is widely available, though not as widely as screen. It’s available in most distribution repositories. The package name is “tmux”.

Invocation

The program is invoked with the command `tmux new` to create a new session. We can optionally add `-s <session_name>` to assign a name to the new session and `-n <window_name>` to assign a name to the first window. If no option to the `new` command is supplied, the `new` itself may be omitted; it will be assumed. Here is an example:

```
me@linuxbox: ~ $ tmux new -s "my session" -n "window 1"
```

Once the program starts, we are presented with a shell prompt and a pretty status bar at the bottom of the window.



Initial tmux window

Multiple Windows

tmux uses the keyboard in a similar fashion to screen, but rather than using `Ctrl-a` as the command prefix, tmux uses `Ctrl-b`. This is good since `Ctrl-a` is used when editing the command line in bash to move the cursor to the beginning of the line.

Here are the basic commands for creating windows and navigating them:

Command	Description
<code>Ctrl-b ?</code>	Show the list of key bindings (i.e., help)
<code>Ctrl-b c</code>	Create a new window
<code>Ctrl-b n</code>	Go to next window
<code>Ctrl-b p</code>	Go to previous window
<code>Ctrl-b 0</code>	Go to window 0. Numbers 1-9 are similar.
<code>Ctrl-b w</code>	Show window list. The status bar lists windows, too.
<code>Ctrl-b ,</code>	Rename the current window

tmux window commands

Multiple Panes

Like screen, tmux can divide the terminal display into sections called panes. However, unlike the implementation of regions in screen, panes in tmux do not merely provide viewports to various windows. In tmux they are complete pseudo-terminals associated with the window. Thus a single tmux window can contain multiple terminals.

Command	Description
Ctrl-b "	Split pane horizontally
Ctrl-b %	Split pane vertically
Ctrl-b arrow	Move to adjoining pane
Ctrl-b Ctrl-arrow	Resize pane by 1 character
Ctrl-b Alt-arrow	Resize pane by 5 characters
Ctrl-b x	Destroy current pane

tmux pane commands

We can demonstrate the behavior of panes by creating a session and a couple of windows. First, we will create a session, name it, and name the initial window:

```
me@linuxbox: ~ $ tmux new -s PaneDemo -n Window0
```

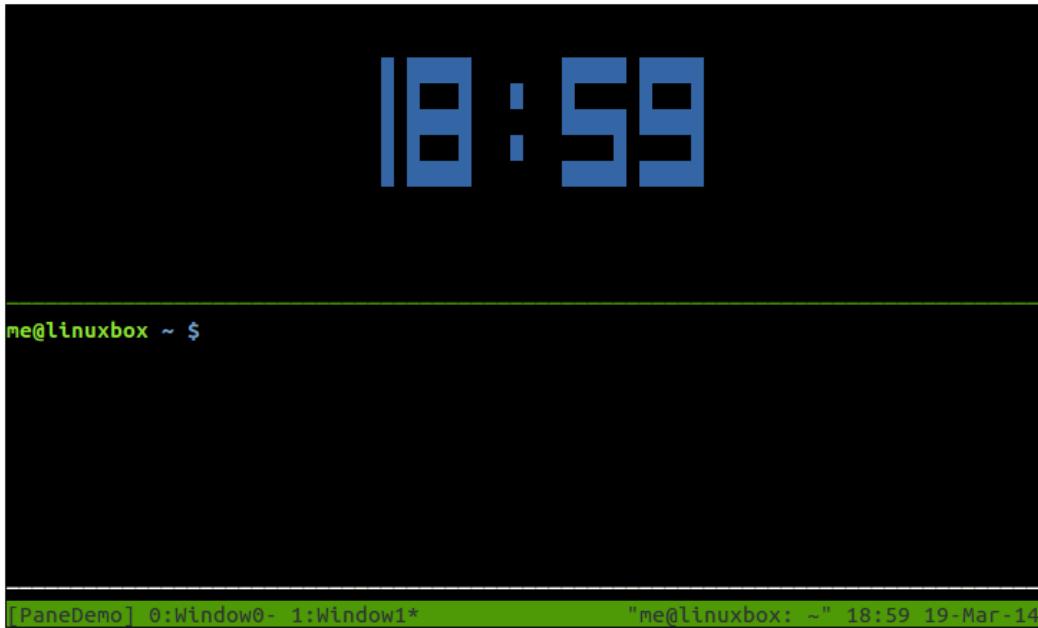
Next, we will create a second window and give it a name:

```
me@linuxbox: ~ $ tmux neww -n Window1
```

We could have done this second step with `Ctrl-b` commands, but seeing the command-line method prepares us for something coming up a little later.

Assuming that all has gone well, we now find ourselves in a tmux session named “PaneDemo” and a window named “Window1”. Now we will split the window in two horizontally by typing `Ctrl-b "`. We still have only two windows (Window0 and Window1), but now have two shell prompts on Window1. We can switch back and forth between the two panes by typing `Ctrl-b` followed by up arrow or down arrow.

Just for fun, let’s type `Ctrl-b t` and a digital clock appears in the current pane. It’s just a cute thing that `tmux` can do.



Multiple panes

We can terminate the clock display by typing `q`. If we move to the first window by typing `Ctrl-b 0`, we see that the panes remain associated with Window1 and have no effect on Window0.

Returning to Window1, let's adjust the size of the panes. We do this by typing `Ctrl-b Alt-arrow` to move the boundary up or down by 5 lines. Typing `Ctrl-b Ctrl-arrow` will move the boundary by 1 line.

It's possible to break a pane out into a new window of its own. This is done by typing `Ctrl-b !`.

`Ctrl-b x` is used to destroy a pane. Note that, unlike screen, destroying a pane in tmux also destroys the pseudo-terminal running within it, along with any associated programs.

Copy Mode

Like screen, tmux has a copy mode. It is invoked by typing `Ctrl-b [`. In copy mode, we can move the cursor freely within the scrollback buffer. To mark text for copying, we first type `Ctrl-space` to begin selection, then move the cursor to make our selection. Finally, we type `Alt-w` to copy the selected text.

Admittedly, this procedure is a little awkward. A little later we'll customize tmux to make the copy mode act more like the vim's visual copying mode.

TMUX(1) BSD General Commands Manual TMUX [0/0]

NAME
tmux - terminal multiplexer

SYNOPSIS
tmux [-28lquivV] [-c shell-command] [-f file] [-L socket-name]
[-S socket-path] [command [flags]]

DESCRIPTION
tmux is a terminal multiplexer: it enables a number of terminals to be created, accessed, and controlled from a single screen. tmux may be detached from a screen and continue running in the background, then later reattached.

When tmux is started it creates a new session with a single window and displays it on screen. A status line at the bottom of the screen shows information on the current session and is used to enter interactive commands.

A session is a single collection of pseudo terminals under the management of the tmux process. The tmux manual page (man tmux) contains detailed information about sessions, windows, panes, and other concepts.

Text marked for copying

As with the digital clock, we return to normal mode by typing “q”. Now we can paste our copied text by typing `Ctrl-b`].

Detaching Sessions

With tmux it's easier to manage multiple sessions than with screen. First, we can give sessions descriptive names, either during creation, as we saw with our "PaneDemo" example above, or by renaming an existing session with `Ctrl-b $`. Second, it's easy to switch sessions on-the-fly by typing `Ctrl-b s` and choosing a session from the presented list.

While we are in a session, we can type `Ctrl-b d` to detach it and, in essence, put tmux into the background. This is useful if we want to create new a session by entering the `tmux new` command.

If we start a new terminal (or log in from a remote terminal) and wish to attach an existing session to it, we can issue the command `tmux ls` to display a list of available sessions. To attach a session, we enter the command `tmux attach -d -t <session_name>`. The “-d” option causes the session to be detached from its previous terminal. Without this option, the session will be attached to both its previous terminal and the new terminal. If only one session is running, a `tmux attach` will connect to it and leave any existing connections intact.

Customizing tmux

As we would expect, tmux is *extremely* configurable. When tmux starts, it reads the files `/etc/tmux.conf` and `~/.tmux.conf` if they exist. It is also possible to start tmux with the `-f` option and specify an alternate configuration file. This way, we can have multiple custom configurations.

The number of configuration commands is extensive, just as it is with screen. The tmux man page has the full list.

As an example, here is a hypothetical configuration file that changes the command prefix key from `Ctrl-b` to `Ctrl-a` and creates a new session with 4 windows:

```
# Sample tmux.conf file

# Change the command prefix from Ctrl-b to Ctrl-a
unbind-key C-b
set-option -g prefix C-a
bind-key C-a send-prefix

#####
# Create session with 4 windows
#####

# Create session and first window
new-session -d -s MySession

# Create second window and vertically split it
new-window
split-window -d -h

# Create third window (and name it) running Midnight Commander
new-window -d -n MdnghtCmdr mc

# Create fourth window (and name it) running htop
new-window -d -n htop htop

# Give focus to the first window in the session
select-window -t 0
```

Since this configuration creates a new session, we should launch tmux by entering the command `tmux attach` to avoid the default behavior of automatically creating a new session. Otherwise, we end up with an additional and unwanted session.

Here's a useful configuration file that remaps the keys used to create panes and changes copy and paste to behave more like vim.

```
# Change bindings for pane-splitting from " and % to | and -
unbind ''
unbind %
bind | split-window -h
bind - split-window -v

# Enable mouse control (clickable windows, panes, resizable panes)
set -g mouse on

# Set color support to allow visual mode highlighting to work in vim
```

```
set -g default-terminal "screen-256color"

# Make copy work like vi
# Start copy ^b-[ 
# Use vi movement keys (arrows, etc.)
# Select with v, V
# Yank and end copy mode with y
# Paste with ^b-]
# View all vi key bindings with ^b-: followed with list-keys -T copy-mode-vi
set-window-option -g mode-keys vi
bind-key -T copy-mode-vi 'v' send -X begin-selection
bind-key -T copy-mode-vi 'y' send -X copy-selection-and-cancel
```

byobu

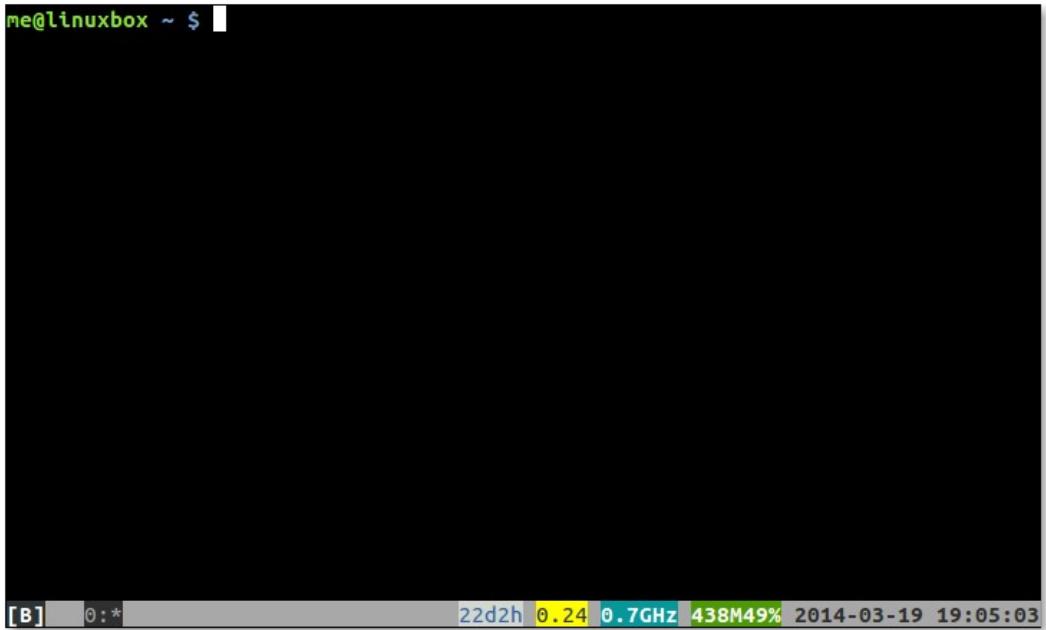
byobu (pronounced “BEE-oh-boo”) from the Japanese word for “a folding, decorative, multi-panel screen” is not a terminal multiplexer *per se*, but rather, it is a wrapper around either GNU screen or tmux (the default is tmux). It aims to create a simplified user interface with an emphasis on presenting useful system information on the status bar.

Availability

byobu was originally developed by Canonical employee Dustin Kirkland, and as such is usually found in Ubuntu and other Debian-based distributions. Recent versions are more portable than the initial release, and it is beginning to appear in a wider range of distributions. It is distributed as the package “byobu”.

Invocation

byobu can be launched simply by entering the command `byobu` followed optionally by any options and commands to be passed to the backend terminal multiplexer (i.e., tmux or screen). For this adventure, we will confine our discussion to the tmux backend as it supports a larger feature set.



Initial byobu window

Usage

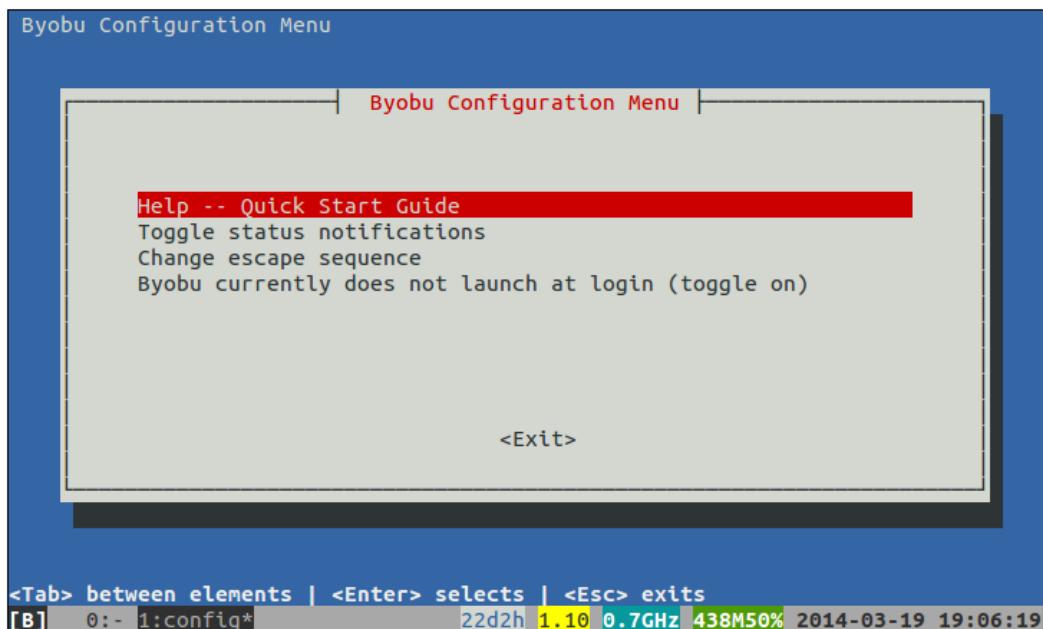
Unlike screen and tmux, byobu doesn't use a command prefix such as `Ctrl-a` to start a command. byobu relies extensively on function keys instead. This makes byobu somewhat easier to learn, but in exchange, it gives up some of the power and flexibility of the underlying terminal multiplexer. That said, byobu still provides an easy-to-use interface for the most useful features and it also provides a key (`F12`) which acts as command prefix for tmux commands. Below is an excerpt from the help file supplied with byobu when using tmux as the backend:

F1	* Used by X11 *
Shift-F1	Display this help
F2	Create a new window
Shift-F2	Create a horizontal split
Ctrl-F2	Create a vertical split
Ctrl-Shift-F2	Create a new session
F3/F4	Move focus among windows
Shift-F3/F4	Move focus among splits
Ctrl-F3/F4	Move a split
Ctrl-Shift-F3/F4	Move a window
Alt-Up/Down	Move focus among sessions
Shift-Left/Right/Up/Down	Move focus among splits
Ctrl-Shift-Left/Right	Move focus among windows
Ctrl-Left/Right/Up/Down	Resize a split
F5	Reload profile, refresh status
Shift-F5	Toggle through status lines
Ctrl-F5	Reconnect ssh/gpg/dbus sockets
Ctrl-Shift-F5	Change status bar's color randomly
F6	Detach session and then logout
Shift-F6	Detach session and do not logout
Ctrl-F6	Kill split in focus

F7	Enter scrollback history
Alt-PageUp/PageDown	Enter and move through scrollback
F8	Change the current window's name
Shift-F8	Toggle through split arrangements
Ctrl-F8	Restore a split-pane layout
Ctrl-Shift-F8	Save the current split-pane layout
F9	Launch byobu-config window
F10	* Used by X11 *
F11	* Used by X11 *
Alt-F11	Expand split to a full window
Shift-F11	Join window into a horizontal split
Ctrl-F11	Join window into a vertical split
F12	Escape sequence
Shift-F12	Toggle on/off Byobu's keybindings
Ctrl-Shift-F12	Modrian squares

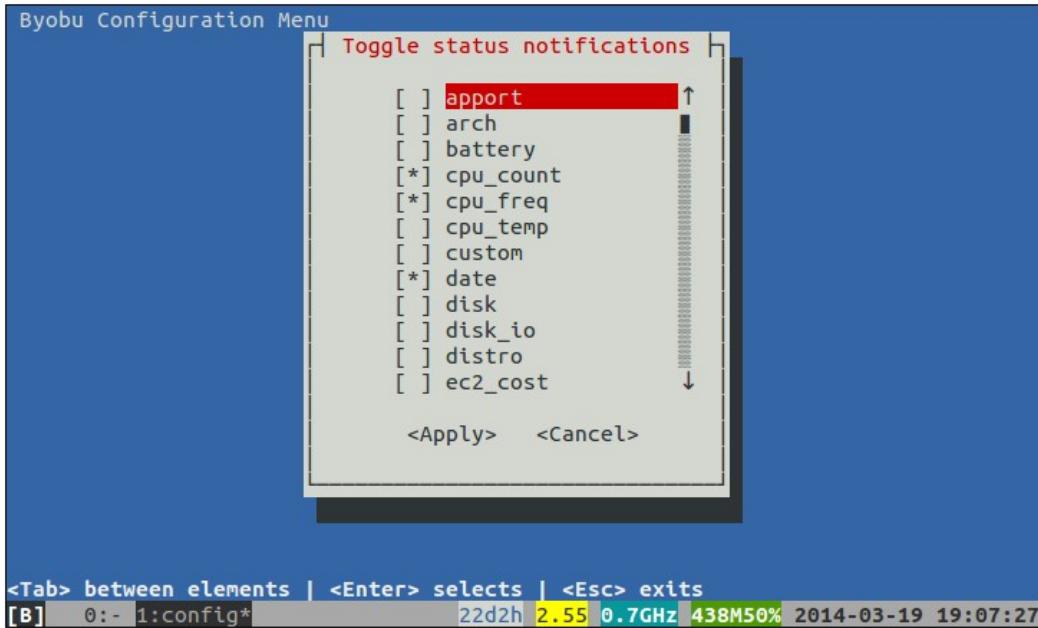
As we can see, most of the commands here correspond to features we have already seen in tmux. There are, however, a couple of interesting additions.

First is the F9 key, which brings up a menu screen:



byobu menu

The choices are pretty self-explanatory, though the “Change escape sequence” item is only relevant when using screen as the backend. If we choose “Toggle status notifications” we get to a really useful feature in byobu; the rich and easily configured status bar.



Status notifications

Here we can choose from a wide variety of system status information to be displayed. Very useful if we are monitoring remote servers.

The second is the `Shift-F12` key, which disables byobu from interpreting the function keys as commands. This is needed in cases where a text-based application (such as Midnight Commander) needs the function keys. Pressing `Shift-F12` a second time re-enables the function keys for byobu. Unfortunately, byobu gives no visual indication of the state of the function keys, making this feature rather confusing to use in actual practice.

Copy Mode

byobu provides an interface to the copy mode of its backend terminal multiplexer. For tmux, it's slightly simplified from normal tmux, but works about the same. Here are the key commands:

Command	Description
Alt-PgUp	Enter copy mode
Space	Start text selection
Enter	End text selection, copy text, and exit copy mode
Alt-Insert	Paste selected text

byobu copy commands

Detaching Sessions

To detach a session and log off, press the `F6` key. To detach without logging off, type `Shift-F6`. To attach, simply enter the `byobu` command and the previous session will be reattached. If more than one session is running, we are prompted to select a session. While we are in a session, we can type `Alt-Up` and `Alt-Down` to move from session to session.

Customizing byobu

The local configuration file for byobu is located in either `~/.byobu/.tmux.conf` or `~/.config/byobu/.tmux.conf`, depending on the distribution. If one doesn't work, try the other. The configuration details are the same as for tmux.

Summing Up

We have seen how a terminal multiplexer can enhance our command-line experience by providing multiple windows and sessions, as well as multiple regions on a single terminal display. So, which one to choose? GNU screen has the benefit of being almost universally available, but is now considered by many as obsolete. tmux is modern and well supported by active development. byobu builds on the success of tmux with a simplified user interface, but if we rely on applications that need access to the keyboard function keys, byobu becomes quite tedious. Fortunately, many Linux distributions make all three available, so it's easy to try them all and see which one satisfies the needs at hand.

Further Reading

The man pages for screen and tmux are richly detailed. Well worth reading. The man page for byobu is somewhat simpler.

GNU Screen

- Official site: <https://www.gnu.org/software/screen/>
- A helpful entry in the Arch Wiki: https://wiki.archlinux.org/index.php/GNU_Screen
- A Google search for “screenrc” yields many sample `.screenrc` files
- Also look for sample files in `/usr/share/doc/screen/examples`

tmux

- Official site: <https://www.gigastudio.com.ua>
- The tmux FAQ: <https://github.com/tmux/tmux/wiki/FAQ>
- A helpful entry in the Arch Wiki: <https://wiki.archlinux.org/index.php/tmux>
- A Google search for “tmux.conf” yields many sample `.tmux.conf` files
- Also look for sample files in `/usr/share/doc/tmux/examples`

byobu

- Official site: <https://www.byobu.org>
- Answers to many common questions: <https://askubuntu.com/tags/byobu/hot>

3 Less Typing

Since the beginning of time, Man has had an uneasy relationship with his keyboard. Sure, keyboards make it possible to express our precise wishes to the computer, but in our fat-fingered excitement to get stuff done, we often suffer from typos and digital fatigue.

In this adventure, we will travel down the carpal tunnel to the land of less typing. We covered some of this in TLCL, but here we will look a little deeper.

Aliases and Shell Functions

The first thing we can do to reduce the number of characters we type is to make full use of *aliases* and *shell functions*. Aliases were created for this very purpose and they are often a very effective solution. Shell functions perform in many ways like aliases but allow a full range of shell script-like capabilities such as programmatic logic, and option and argument processing.

Most Linux distributions provide some set of default alias definitions and it's easy to add more. To see the aliases we already have, we enter the `alias` command without arguments:

```
me@linuxbox: ~ $ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias ls='ls --color=auto'
```

On this example system, we see `alias` is used to activate color output for some commonly used commands. It is also common to create aliases for various forms of the `ls` command:

```
alias ll='ls -l'
alias la='ls -A'
alias l='ls -CF'
alias l.='ls -d .*'
alias lm='ls -l | less'
```

Aliases are good for lots of things, for example, here's one that's useful for Debian-style systems:

```
alias update='sudo apt-get update && sudo apt-get upgrade'
```

Aliases are easy to create. It's usually just a matter of appending them to our `.bashrc` file. Before creating a new alias, it's a good idea to first test the proposed name of the alias with the `type` command to check if the name is already being used by another program or alias.

While being easy, aliases are somewhat limited. In particular, aliases can't handle complex logic or accept positional parameters. For that we need to use shell functions.

As we recall from TLCL, shell functions are miniature shell scripts that we can add to our `.bashrc` file to perform anything that we may otherwise do with a shell script. Here is an example function that displays a quick snapshot of a system's health:

```
status() {
    echo -e "\nuptime:"
    uptime
    echo -e "\ndisk space:"
    df -h 2> /dev/null
    echo -e "\ninodes:"
    df -i 2> /dev/null
    echo -e "\nblock devices:"
    blkid
    echo -e "\nmemory:"
    free -m
    if [[ -r /var/log/syslog ]]; then
        echo -e "\nsyslog:"
        tail /var/log/syslog
    fi
    if [[ -r /var/log/messages ]]; then
        echo -e "\nmessages:"
        tail /var/log/messages
    fi
} | less
```

Unlike aliases, shell functions can accept positional parameters:

```
params() {
    local argc=0
    while [[ -n $1 ]]; do
        argc=$((++argc))
        echo "Argument $argc = $1"
        shift
    done
}
```

Command Line Editing

Aliases and shell functions are all well and good, provided we know in advance the operations we wish to perform, but what about the rest of the time? Most command line operations we perform are on-the-fly, so other techniques are needed.

As we saw in Chapter 8 of TLCL, the bash shell includes a library called `readline` to handle keyboard input during interactive shell sessions. This includes text typed at the shell prompt and keyboard input using the `read` builtin when the `-e` option is specified. The readline library supports a large number of commands that can be used to edit what we type at the command line. Since readline is from the GNU project, many of the commands are taken from the emacs text editor.

Control Commands

Before we get to the actual editing commands, let's look at some commands that are used to control the editing process.

Command	Description
Enter	Pressing the enter key causes the current command line to be accepted. Note that the cursor location within the line does not matter (i.e., it doesn't have to be at the end). If the line is not empty, it is added to the command history.
Esc	Meta-prefix. If the Alt key is unavailable, the Esc key can be used in its place. For example, if a command calls for Alt-r but another program intercepts that command, press and release the Esc key followed by the r key.
Ctrl-g	Abort the current editing command.
Ctrl-_	Incrementally undo changes to the line.
Alt-r	Revert all changes to the line (i.e., complete undo).
Ctrl-l	Clear the screen.
Alt-num	Where num is a number. Some commands accept a numeric argument. For those commands that accept it, type this first followed by the command.

Editing control commands

Moving Around

Here are some commands to move the cursor around the current command line. In the readline documentation, the current cursor location is referred to as the *point*.

Command	Description
Right	Move forward one character.
Left	Move backward one character.
Alt-f	Move forward one word.
Alt-b	Move backward one word.
Ctrl-a	Move to the beginning of the line.
Ctrl-e	Move to the end of the line.

Cursor movement commands

Using Command History

In order to save typing, we frequently reuse previously typed commands stored in the command history. We can move up and down the history list and the history list can be searched.

Command	Description
Up	Move to previous history list entry.
Down	Move to next history list entry.

Command	Description
Alt-<	Move to the beginning of the history list.
Alt->	Move to the end of the history list.
Ctrl-r	Perform an incremental history search starting at the current position and moving up the history list. After a command is typed, a prompt appears and with each succeeding character typed, the position within the list moves to the next matching line. This is probably the most useful of the history search commands.
Ctrl-s	Like Ctrl-r except the search is performed moving down the history list.
Alt-p	Perform a non-incremental search moving up the history list.
Alt-n	Perform a non-incremental search moving down the history list.
Alt-Ctrl-y	Insert the first argument from the previous history entry. This command can take a numeric argument. When a numeric argument is given, the nth argument from the previous history entry is inserted.
Alt-.	Insert the last argument from the previous history entry. When a numeric argument is given, behavior is the same as Alt-Ctrl-y above.

History commands

Changing Text

Command	Description
Ctrl-d	Delete the character at the point.
Ctrl-t	Transpose characters. Exchange the character at the point with the character preceding it.
Alt-t	Transpose words. Exchange the word at the point with the word preceding it.
Alt-u	Change the current word to uppercase.
Alt-l	Change the current word to lowercase.
Alt-c	Capitalize the current word.

Editing commands

Cutting and Pasting

As with vim, cutting and pasting in readline are referred to as “killing” and “yanking.” The clipboard is called the *kill-ring* and is implemented as a *circular buffer*. This means that it contains multiple entries (i.e., each kill adds a new entry). The latest entry is referred to as the “top” entry. It is possible to “rotate” the kill-ring to bring the previous entry to the top and delete the latest entry. However, this feature is rarely used.

Mostly, the kill commands are used to simply delete text rather than save it for later yanking.

Command	Description
Alt-d	Kill from the point to the end of the current word. If the point is located in whitespace, kill to the end of the next word.
Alt-Backspace	Kill the word before the point.
Ctrl-k	Kill from the point to end of line.
Ctrl-u	Kill from the point to the beginning of the line.
Ctrl-y	Yank the “top” entry from the kill-ring.
Alt-y	Rotate the kill-ring and yank the new “top” entry.

Copy and delete commands

Editing in Action

In Chapter 4 of TLCL, we considered the danger of using a wildcard with the `rm` command. It was suggested that we first test the wildcard with the `ls` command to see the result of the expansion. We then recall the command from the history and edit the line to replace the “ls” with “rm”. So, how do we perform this simple edit?

First, the beginner’s way: we recall the command with the up arrow, use the left arrow repeatedly to move the cursor to the space between the “ls” and the wildcard, backspace twice, then type “rm” and `Enter`.

That’s a lot of keystrokes.

Next, the tough-guy’s way: we recall the command with the up arrow, type `Ctrl-a` to jump to the beginning of the line, type `Alt-d` to kill the current word (the “ls”), type “rm” and `Enter`.

That’s better.

Finally, the super-tough-guy’s way: type “rm” then `Alt-.` to recall the last argument (the wildcard) from the previous command, then `Enter`.

Wow.

Completion

Another trick that readline can perform is called *completion*. This is where readline will attempt to automatically complete something we type.

For example, let’s imagine that our current working directory contains a single file named `foo.txt` and we want to view it with `less`. So we begin to type the command `less foo.txt` but instead of typing it all out, we just type `less f` and then press the `Tab` key. Pressing `Tab` tells readline to attempt completion on the file name and remainder of the command is completed automatically.

This will work as long as the “clue” given to readline is not ambiguous. If we had two files in our imaginary directory named “foo.txt” and “foo1.txt”, a successful completion would not take place since “less f” could refer to either file. What happens instead is readline makes the next best guess by completing as far as “less foo” since both possible answers contain those characters. To make a full completion, we need to type either `less foo.` for `foo.txt` or `less foo1` for `foo1.txt`.

If we have typed an ambiguous clue, we can view a list of all possible completions to get guidance as what to type next. In the case of our imaginary directory, pressing `Tab` a second time will display all of the file names beginning with “foo” so that we can see what more needs to be typed to remove the ambiguity.

Besides file name completion, readline can complete command names, environment variable names, user home directory names, and network host names:

Completion	Description
Command names	Completion on the first word of a line will complete the name of an available command. For example, typing “lsu” followed by <code>Tab</code> will complete as <code>lsusb</code> .
Variables	If completion is attempted on a word beginning with “\$”, environment variable names will be used. For example, typing “echo \$TE” will complete as <code>echo \$TERM</code> .
User names	To complete the name of a user’s home directory, precede the user’s name with a “~” and press ‘Tab’. For example: <code>ls ~ro</code> followed by <code>Tab</code> will complete to <code>ls ~root/</code> . It is also possible to force completion of a user name without the leading ~ by typing <code>Alt-~</code> . For example “who ro” followed by <code>Alt-~</code> will complete to <code>who root</code> .
Host names	Completion on a word starting with “@” causes host name completion, however this feature rarely works on modern systems since they tend to use DHCP rather than listing host names in the <code>/etc/hosts</code> file.
File names	In all other cases, completion is attempted on file and path names.

Completion types

Programmable Completion

Bash includes some builtin commands that permit the completion facility to be programmed on a command-by-command basis. This means it’s possible to set up a custom completion scheme for individual commands; however, doing this is beyond the scope of this adventure. We will instead talk about an optional package that uses these builtins to greatly extend the native completion facility. Called *bash-completion*, this package is installed automatically for some distributions (for example, Ubuntu) and is

generally available for others. To check for the package, examine the `/etc/bash-completion.d` directory. If it exists, the package is installed.

The `bash-completion` package adds support for many command line programs, allowing us to perform completion on both command options and arguments. The `ls` command is a good example. If we type “`ls -`” then the `Tab` key a couple of times, we will see a list of possible options to the command:

```
me@linuxbox: ~ $ ls --  
--all --ignore= --ignore-backups  
--almost-all --indicator-style= --inode  
--author --literal --no-group  
--block-size= --numeric-uid-gid  
--classify --quote-name --quoting-style= --recursive  
--color --reverse --show-control-chars  
--color= --dereference --dereference-command-line  
--context --dereference-command-line-symlink-to-dir  
--deref --directory --size  
--dired --escape --sort  
--dirc --file-type --sort= --tabsize= --time= --time-style= --version  
--dirc --format= --size  
--group-directories-first --help --version  
--help= --hide= --width=
```

An option can be completed by typing a partial option followed by `Tab`. For example, typing “`ls -ver`” then `Tab` will complete to “`ls -version`”.

The `bash-completion` system is interesting in its own right as it is implemented by a series of shell scripts that make use of the `complete` and `compgen` bash builtins. The main body of the work is done by the `/etc/bash_completion` (or `/usr/share/bash-completion/bash_completion` in newer versions) script along with additional scripts for individual programs in either the `/etc/bash-completion.d` directory or the `/usr/share/bash-completion/completions` directory. These scripts are good examples of advanced scripting technique and are worthy of study.

Summing Up

This adventure is a lot to take in and it might not seem all that useful at first, but as we continue to gain experience and practice with the command line, learning these labor-saving tricks will save us a lot of time and effort.

Further Reading

- “The beginning of time” actually has meaning in Unix-like operating systems such as Linux. It’s January 1, 1970. See: https://en.wikipedia.org/wiki/Unix_time for details.
- Aliases and shell functions are discussed in Chapters 5 and 26, respectively, of *The Linux Command Line*: <https://linuxcommand.org/tlcl.php>.
- The READLINE section of the bash man page describes the many keyboard shortcuts available on the command line.
- The HISTORY section of the bash man page covers the command line history features of `bash`.
- The official home page of the bash-completion project: <https://github.com/scop/bash-completion>
- For those readers interested in learning how to write their own bash completion scripts, see this tutorial at the Linux Documentation Project: <https://tldp.org/LDP/abs/html/tabexpansion.html>.

4 More Redirection

As we learned in Chapter 6 of TLCL, I/O redirection is one of the most useful and powerful features of the shell. With redirection, our commands can send and receive streams of data to and from files and devices, as well as allow us to connect different programs together into pipelines.

In this adventure, we will look at redirection in a little more depth to see how it works and to discover some additional features and useful redirection techniques.

What's Really Going On

Whenever a new program is run on the system, the kernel creates a table of *file descriptors* for the program to use. File descriptors are pointers to files. By convention, the first 3 entries in the table (descriptors 0, 1, and 2) are used as standard input (stdin), standard output (stdout), and standard error (stderr). Initially, all three descriptors point to the terminal device (which the system treats as a read/write file), so that standard input comes from the keyboard and standard output and standard error go to the terminal display.

When a program is started as a child process of another (for instance, when we run an executable program in the shell), the newly launched program inherits a copy of the parent's file descriptor table. Redirection is the process of manipulating the file descriptors so that input and output can be routed from/to different files.

The shell hides the presence of file descriptors in common redirections such as:

command > file

Here we redirect standard output to a file, but the full syntax of the redirection operator includes an optional file descriptor. We could write the above statement this way and it would have exactly the same effect:

command 1> file

As a convenience, the shell assumes we want to redirect standard output if the file descriptor is omitted. Likewise, the following two statements are equivalent when referring to standard input:

command < file

command 0< file

Duplicating File Descriptors

It is sometimes desirable to write more than one output stream (for example standard output and standard error) to the same file. To do this, we would write something like this:

```
command > file 2>&1
```

We'll add the assumed file descriptor to the first redirection to make things a little clearer:

```
command 1> file 2>&1
```

This is an example of *duplication*. When we read this statement, we see that file descriptor 1 is changed from pointing to the terminal device to instead pointing to *file*. This is followed by the second redirection that causes file descriptor 2 to be a duplicate (i.e., it points to the same file) of file descriptor 1. When we look at things this way, it's easy to see why the order of redirections is important. For example, if we reverse the order:

```
command 2>&1 1> file
```

file descriptor 2 becomes a duplicate of file descriptor 1 (which points to the terminal) and then file descriptor 1 is set to point to *file*. The final result is file descriptor 1 points to *file* while file descriptor 2 still points to the terminal.

exec

Before we go any farther, we need to take a brief detour and talk about a shell builtin that we didn't cover in TLCL. This builtin is named `exec` and it does some interesting things. Its main purpose is to terminate the shell and launch another program in its place. This is often used in startup scripts that initiate system services. However, it is not common in scripts used for other purposes.

Usage of `exec` is described below:

```
exec [program] [redirections]
```

program is the name of the program that will start and take the place of the shell.

redirections are the redirections to be used by the new program.

One feature of `exec` is useful for our study of redirection. If *program* is omitted, any specified redirections are performed on the current shell. For example, if we included this near the beginning of a script:

```
exec 1> output.txt
```

from that point on, every command using standard output would send its data to *output.txt*. It should be noted that if this trick is performed by a script, it is no longer

possible to redirect that script's output at runtime using the command line. For example, if we had the following script:

```
#!/bin/bash

# exec-test - Test external redirection and exec

exec 1> ~/foo1.txt
echo "Boo."

# End of script
```

and tried to invoke it with redirection:

```
me@linuxbox ~ $ ./exec-test > ~/foo2.txt
```

the attempted redirection would have no effect. The word “Boo” would still be written to the file `foo1.txt`, not `foo2.txt` as specified on the command line. This is because the redirection performed inside the script via `exec` is performed after the redirection on the command line, and thus, takes precedence.

Another way we can use `exec` is to open and close additional file descriptors. While we most often use descriptors 0, 1, and 2, it is possible to use others. Here are examples of opening and closing file descriptor 3:

```
# Open fd 3
exec 3> some_file.txt

# Close fd 3
exec 3>&-
```

It's easy to open and use file descriptors 3-9 in the shell, and it's even possible to use file descriptors 10 and above, though the `bash` man page cautions against it.

So why would we want to use additional file descriptors? That's a little hard to answer. In most cases we don't need to. We *could* open several descriptors in a script and use them to redirect output to different files, but it's just as easy to specify (using shell variables, if desired) the names of the files to which we want to redirect since most commands are going to send their data to standard output anyway.

There is one case in which using an additional file descriptor would be helpful. It's the case of a filter program that accepts standard input and sends its filtered data to standard output. Such programs are quite common, for example `sort` and `grep`. But what if we want to create a filter program that also writes stuff on the terminal display while it was filtering? We can't use standard output to do it, because standard output is being used to output the filtered data. We could use standard error to display stuff on the screen, but let's say we wanted to keep it restricted to just error messages (this is good for logging). Using `exec`, we could do something like this:

```
#!/bin/bash

# counter-exec - Count number of lines in a pipe
```

```

exec 3> /dev/tty # open fd 3 and point to controlling terminal

count=0
while read; do  # read line from stdin
    echo "$REPLY" # send line to stdout
    ((count++))
    printf "\b\b\b\b\b\b%06d" $count >&3
done
echo " Lines Counted" >&3

exec 3>&- # close fd 3

```

This program simply copies standard input to standard output, but it displays a running count of the number of lines that it has copied. If we invoke it this way, we can see it in action:

```
me@linuxbox ~ $ find /usr/share | ./counter-exec > ~/find_list.txt
```

In this pipeline example, we generate a list of files using `find`, and then count them before writing the list in a file named `find_list.txt`.

The script works by reading a line from the standard input and writing the `REPLY` variable (which contains the line of text from `read`) to standard output. The `printf` format specifier contains a series of six backspaces and a formatted integer that is always six digits long padded with leading zeros.

/dev/tty

The mysterious part of the script above is the `exec`. The `exec` is used to open a file using file descriptor 3 which is set to point to `/dev/tty`. `/dev/tty` is one of several *special files* that we can access from the shell. Special files are usually not “real” files in the sense that they are files that exists on a physical disk. Rather, they are virtual like the files in the `/proc` directory. The `/dev/tty` file is a device that always points to a program’s *controlling terminal*, that is, the terminal that is responsible for launching the program. If we run the command `ps aux` on our system, we will see a listing of every process. At the top of the listing is a column labeled “TTY” (short for “Teletype” reflecting its historical roots) that contains the name of the controlling terminal. Most entries in this column will contain “?” meaning that the process has no controlling terminal (the process was not launched interactively), but others will contain a name like “pts/1” which refers to the device `/dev/pts/1`. The term “pty” means *pseudo-terminal*, the type of terminal used by terminal emulators rather than actual physical terminals.

Noclobber

When the shell encounters a command with output redirection, such as:

command > file

the first thing that happens is that the output stream is started by either creating *file* or, if *file* already exists, truncating it to zero length. This means that if *command* completely fails or doesn't even exist, *file* will end up with zero length. This can be a safety issue for new users who might overwrite (or truncate) a valuable file.

To avoid this, we can do one of two things. First we can use the “>>” operator instead of “>” so that output will be appended to the end of *file* rather than the beginning. Second, we can set the “noclobber” shell option which prevents redirection from overwriting an existing file. To activate this, we enter:

```
set -o noclobber
```

Once we set this option, attempts to overwrite an existing file will cause the following error:

```
bash: file: cannot overwrite existing file
```

The effect of the `noclobber` option can be overridden by using the `>|` redirection operator like so:

command `>|` *file*

To turn off the `noclobber` option we enter this command:

```
set +o noclobber
```

Summing Up

While this adventure may be more on the “interesting” side than the “fun” side, it does provide some useful insight into how redirection actually works and some of the interesting ways we can use it. In a later adventure, we will put this new knowledge to work expanding the power of our scripts.

Further Reading

- A good visual tutorial can be found at The Bash Hackers Wiki: https://wiki.bash-hackers.org/howto/redirection_tutorial
- For a little background on file descriptors, see this Wikipedia article: https://en.wikipedia.org/wiki/File_descriptor
- This *Linux Journal* article covers using `exec` to manage redirection: <https://www.linuxjournal.com/content/bash-redirections-using-exec>
- *The Linux Command Line* covers redirection in Chapters 6 (main discussion), 25 (here documents), 28 (here strings), and 36 (command grouping, subshells, process substitution, named pipes).
- The REDIRECTION section of the `bash` man page, of course, has all the details.

5 tput

While our command line environment is certainly powerful, it can be somewhat lacking when it comes to visual appeal. Our terminals cannot create the rich environment of the graphical user interface, but it doesn't mean we are doomed to always look at plain characters on a plain background.

In this adventure, we will look at `tput`, a command used to manipulate our terminal. With it, we can change the color of text, apply effects, and generally brighten things up. More importantly, we can use `tput` to improve the human factors of our scripts. For example, we can use color and text effects to better present information to our users.

Availability

`tput` is part of the `ncurses` package and is supplied with most Linux distributions.

What it Does/How it Works

Long ago, when computers were centralized, interactive computer users communicated with remote systems by using a physical terminal or a terminal emulator program running on some other system. In their heyday, there were many kinds of terminals and they all used different sequences of control characters to manage their screens and keyboards.

When we start a terminal session on our Linux system, the terminal emulator sets the TERM environment variable with the name of a *terminal type*. If we examine TERM, we can see this:

```
[me@linuxbox ~]$ echo $TERM  
xterm
```

In this example, we see that our terminal type is named “xterm” suggesting that our terminal behaves like the classic X terminal emulator program `xterm`. Other common terminal types are “linux” for the Linux console, and “screen” used by terminal multiplexers such as `screen` and `tmux`. While we will encounter these 3 types most often, there are, in fact, thousands of different terminal types. Our Linux system contains a database called *terminfo* that describes them. We can examine a typical *terminfo* entry using the `infocmp` command followed by a terminal type name:

```
[me@linuxbox ~]$ infocmp screen
# Reconstructed via infocmp from file: /lib/terminfo/s/screen
screen|VT 100/ANSI X3.64 virtual terminal,
    am, km, mir, msgr, xenl,
    colors#8, cols#80, it#8, lines#24, ncv@, pairs#64,
    acsc=++\,\,\-\-..00`aaaffgghhiijjkkllmmmnnooppqqrsssttuuvvwwwxxxxyyzz{{||}}~~,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
    clear=\E[H\E[J, cnorm=\E[34h\E[?25h, cr=^M,
    csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%cD, cubl=^H,
    cud=\E[%p1%dB, cudl=^J, cuf=\E[%p1%dC, cuf1=\E[C,
    cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%cA, cuul=\EM,
```

```

cvvis=\E[34l, dch=\E[%p1%dP, dch1=\E[P, dl=\E[%p1%dM,
dl1=\E[M, ed=\E[J, el=\E[K, el1=\E[1K, enacs=\E(B\E)0,
flash=\Eg, home=\E[H, ht=^I, hts=\EH, ich=\E[%p1%d@,
il=\E[%p1%dL, il1=\E[L, ind=^J, is2=\E)0, kbs=\177,
kcbt=\E[Z, kcub1=\EOD, kcud1=\EOB, kcuf1=\EOC, kcuu1=\EOA,
kdch1=\E[3~, kend=\E[4~, kf1=\EOP, kf10=\E[21~,
kf11=\E[23~, kf12=\E[24~, kf2=\EOQ, kf3=\EOR, kf4=\EOS,
kf5=\E[15~, kf6=\E[17~, kf7=\E[18~, kf8=\E[19~, kf9=\E[20~,
khome=\E[1~, kich1=\E[2~, kmous=\E[M, knp=\E[6~, kpp=\E[5~,
nel=\EE, op=\E[39;49m, rc=\E8, rev=\E[7m, ri=\EM, rmacs=^O,
rmcup=\E[?10491, rmir=\E[4l, rmkx=\E[?11\E>, rmso=\E[23m,
rmul=\E[24m, rs2=\Ec\E[?10001\E[?25h, sc=\E7,
setab=\E[4%p1%dm, setaf=\E[3%p1%dm,
sgr=\E[0%?%p6%t;1%;%?%p1%t;3%;%?%p2%t;4%;%?%p3%t;7%;%?%p4%t;5%;m%?%p9%t\016%e\017%;,
sgr0=\E[m\017, smacs=^N, smcup=\E[?1049h, smir=\E[4h,
smkx=\E[?1h\E=, smso=\E[3m, smul=\E[4m, tbc=\E[3g,

```

The example above is the terminfo entry for the terminal type “screen”. What we see in the output of `infocmp` is a comma-separated list of *terminal capability names* or *capnames*. Some of the capabilities are standalone - like the first few in the list - while others are assigned cryptic values. Standalone terminal capabilities indicate something the terminal can do. For example, the capability “am” indicates the terminal has an automatic right margin. Terminal capabilities with assigned values contain strings, which are interpreted as commands by the terminal. The values starting with “\E” (which represents the escape character) are sequences of control codes that cause the terminal to perform an action such as moving the cursor to a specified location, or setting the text color.

The `tput` command can be used to test for a particular capability or to output the assigned value. Here are some examples:

```
tput longname
```

This outputs the full name of the current terminal type. We can specify another terminal type by including the `-T` option. Here, we will ask for the full name of the terminal type named “screen”:

```
tput -T screen longname
```

We can inquire values from the terminfo database, like the number of supported colors and the number of columns in the current terminal:

```
tput colors
tput cols
```

We can test for particular capability. For example, to see if the current terminal supports “bce” (background color erase - meaning that clearing or erasing text will be done using the currently defined background color) we type:

```
tput bce && echo "True"
```

We can send instructions to the terminal. For example, to move the cursor to the position 20 characters to the right and 5 rows down:

```
tput cup 5 20
```

There are many different terminal types defined in the terminfo database and there are many terminal capnames. The terminfo man page contains a complete list. Note, however, that in general practice, there are only a relative handful of capnames supported by all of the terminal types we are likely to encounter on Linux systems.

Reading Terminal Attributes

For the following capnames, `tput` outputs a value to stdout:

Capname	Description
<code>longname</code>	Full name of the terminal type
<code>lines</code>	Number of lines in the terminal
<code>cols</code>	Number of columns in the terminal
<code>colors</code>	Number of colors available

Capability names

The `lines` and `cols` values are dynamic. That is, they are updated as the size of the terminal window changes. Here is a handy alias that creates a command to view the current size of our terminal window:

```
alias term_size=`echo "Rows=$(tput lines) Cols=$(tput cols)"'
```

If we define this alias and execute it, we will see the size of the current terminal displayed. If we then change the size of the terminal window and execute the alias a second time, we will see the values have been updated.

One interesting feature we can use in our scripts is the SIGWINCH signal. This signal is sent each time the terminal window is resized. We can include a signal handler (i.e., a trap) in our scripts to detect this signal and act upon it:

```
#!/bin/bash
# term_size2 - Dynamically display terminal window size

redraw() {
    clear
    echo "Width = $(tput cols) Height = $(tput lines)"
}

trap redraw WINCH

redraw
while true; do
    :
done
```

With this script, we start an empty infinite loop, but since we set a trap for the SIGWINCH signal, each time the terminal window is resized the trap is triggered and the new terminal size is displayed. To exit this script, we type `Ctrl-c`.



```
Width = 80 Height = 24
```

term_size2

Controlling the Cursor

The capnames below output strings containing control codes that instruct the terminal to manipulate the cursor:

Capname	Description
sc	Save the cursor position
rc	Restore the cursor position
home	Move the cursor to upper left corner (0,0)
cup <row> <col>	Move the cursor to position row, col
cud1	Move the cursor down 1 line
cuu1	Move the cursor up 1 line
civis	Set to cursor to be invisible
cnorm	Set the cursor to its normal state

Cursor control capnames

We can modify our previous script to use cursor positioning and to place the window dimensions in the center as the terminal is resized:

```
#!/bin/bash
# term_size3 - Dynamically display terminal window size
#               with text centering

redraw() {
    local str width height length

    width=$(tput cols)
    height=$(tput lines)
    str="Width = $width Height = $height"
```

```

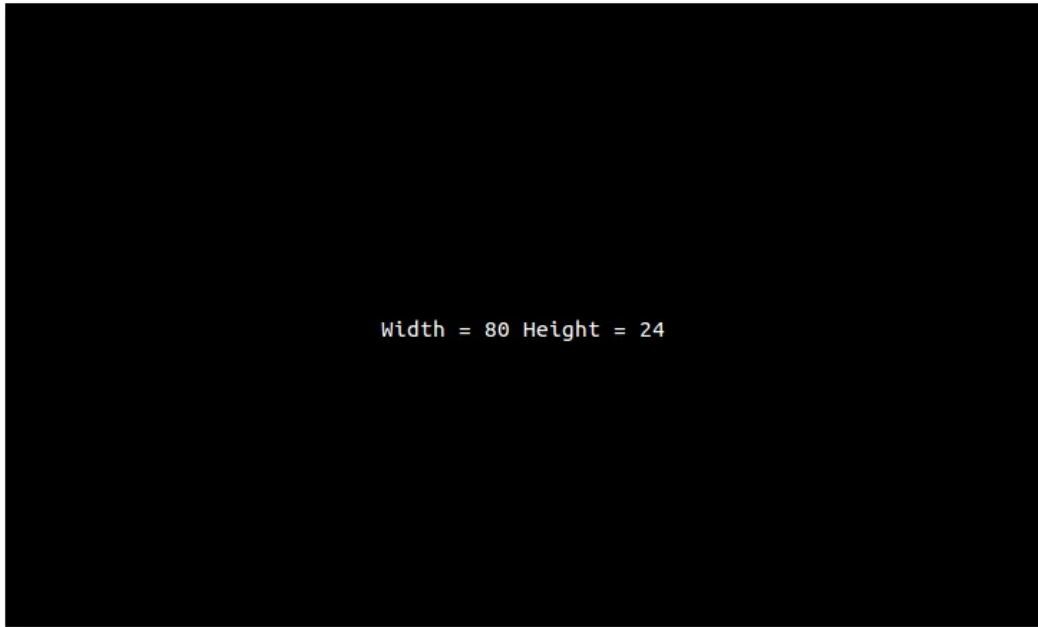
length=${#str}
clear
tput cup $((height / 2)) ${((width / 2) - (length / 2)))
echo "$str"
}

trap redraw WINCH

redraw
while true; do
:
done

```

As in the previous script, we set a trap for the SIGWINCH signal and start an infinite loop. The redraw function in this script is a bit more complicated, since it has to calculate the center of the terminal window each time its size changes.



term_size3

Text Effects

Like the capnames used for cursor manipulation, the following capnames output strings of control codes that affect the way our terminal displays text characters:

Capname	Description
bold	Start bold text
smul	Start underlined text
rmul	End underlined text
rev	Start reverse video
blink	Start blinking text
invis	Start invisible text
smsq	Start “standout” mode

Capname	Description
rms0	End “standout” mode
sgr0	Turn off all attributes
setaf <value>	Set foreground color
setab <value>	Set background color

Text effects capnames

Some capabilities, such as underline and standout, have capnames to turn the attribute both on and off while others only have a capname to turn the attribute on. In these cases, the `sgr0` capname can be used to return the text rendering to a “normal” state. Here is a simple script that demonstrates the common text effects:

```
#!/bin/bash

# tput_characters - Test various character attributes

clear

echo "tput character test"
echo "=====
echo

tput bold; echo "This text has the bold attribute.";      tput sgr0
tput smul; echo "This text is underlined (smul).";       tput rmul

# Most terminal emulators do not support blinking text (though xterm
# does) because blinking text is considered to be in bad taste ;-)
tput blink; echo "This text is blinking (blink).";         tput sgr0

tput rev;   echo "This text has the reverse attribute";    tput sgr0

# Standout mode is reverse on many terminals, bold on others.
tput sms0; echo "This text is in standout mode (sms0)."; tput rms0

tput sgr0
echo
```

```
tput character test
=====
This text has the bold attribute.
This text is underlined (smul).
This text is blinking (blink).
This text has the reverse attribute
This text is in standout mode (smso).

me@linuxbox ~ $
```

tput_characters

Text Color

Most terminals support 8 foreground text colors and 8 background colors (though some support as many as 256). Using the `setaf` and `setab` capabilities, we can set the foreground and background colors. The exact rendering of colors is a little hard to predict. Many desktop managers impose “system colors” on terminal windows, thereby modifying foreground and background colors from the standard. Despite this, here are what the colors should be:

Value	Color
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Not used
9	Reset to default color

Text colors

The following script uses the `setaf` and `setab` capabilities to display the available foreground/background color combinations:

```
#!/bin/bash
```

```
# tput_colors - Demonstrate color combinations.

for fg_color in {0..7}; do
    set_foreground=$(tput setaf $fg_color)
    for bg_color in {0..7}; do
        set_background=$(tput setab $bg_color)
        echo -n $set_background$set_foreground
        printf ' F:%s B:%s ' $fg_color $bg_color
    done
    echo $(tput sgr0)
done
```

```
me@linuxbox ~ $ tput_colors
F:0 B:1 F:0 B:2 F:0 B:3 F:0 B:4 F:0 B:5 F:0 B:6 F:0 B:7
F:1 B:0 F:1 B:2 F:1 B:3 F:1 B:4 F:1 B:5 F:1 B:6 F:1 B:7
F:2 B:0 F:2 B:1 F:2 B:3 F:2 B:4 F:2 B:5 F:2 B:6 F:2 B:7
F:3 B:0 F:3 B:1 F:3 B:2 F:3 B:4 F:3 B:5 F:3 B:6 F:3 B:7
F:4 B:0 F:4 B:1 F:4 B:2 F:4 B:3 F:4 B:5 F:4 B:6 F:4 B:7
F:5 B:0 F:5 B:1 F:5 B:2 F:5 B:3 F:5 B:4 F:5 B:6 F:5 B:7
F:6 B:0 F:6 B:1 F:6 B:2 F:6 B:3 F:6 B:4 F:6 B:5 F:6 B:7
F:7 B:0 F:7 B:1 F:7 B:2 F:7 B:3 F:7 B:4 F:7 B:5 F:7 B:6
me@linuxbox ~ $
```

tput_colors

Clearing the Screen

These capnames allow us to selectively clear portions of the terminal display:

Capnam Description

e

smcup	Save screen contents
rmcup	Restore screen contents
el	Clear from the cursor to the end of the line
elli	Clear from the cursor to the beginning of the line
ed	Clear from the cursor to the end of the screen
clear	Clear the entire screen and home the cursor

Screen erasure capnames

Using some of these terminal capabilities, we can construct a script with a menu and a separate output area to display some system information:

```
#!/bin/bash
```

```

# tput_menu: a menu driven system information program

BG_BLUE=$(tput setab 4)
BG_BLACK=$(tput setab 0)
FG_GREEN=$(tput setaf 2)
FG_WHITE=$(tput setaf 7)

# Save screen
tput smcup

# Display menu until selection == 0
while [[ $REPLY != 0 ]]; do
    echo -n ${BG_BLUE}${FG_WHITE}
    clear
    cat <<- _EOF_
        Please Select:

        1. Display Hostname and Uptime
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit

_EOF_

    read -p "Enter selection [0-3] > " selection

    # Clear area beneath menu
    tput cup 10 0
    echo -n ${BG_BLACK}${FG_GREEN}
    tput ed
    tput cup 11 0

    # Act on selection
    case $selection in
        1) echo "Hostname: $HOSTNAME"
           uptime
           ;;
        2) df -h
           ;;
        3) if [[ $(id -u) -eq 0 ]]; then
               echo "Home Space Utilization (All Users)"
               du -sh /home/* 2> /dev/null
           else
               echo "Home Space Utilization ($USER)"
               du -s $HOME/* 2> /dev/null | sort -nr
           fi
           ;;
        0) break
           ;;
        *) echo "Invalid entry."
           ;;
    esac
    printf "\n\nPress any key to continue."
    read -n 1
done

# Restore screen
tput rmcup
echo "Program terminated."

```

```
Please Select:  
1. Display Hostname and Uptime  
2. Display Disk Space  
3. Display Home Space Utilization  
0. Quit
```

Enter selection [0-3] > 2

Filesystem	Size	Used	Avail	Use%	Mounted on
rootfs	5.8G	4.0G	1.6G	73%	/
/dev/root	5.8G	4.0G	1.6G	73%	/
devtmpfs	215M	0	215M	0%	/dev
tmpfs	44M	260K	44M	1%	/run
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	88M	0	88M	0%	/run/shm
/dev/mmcblk0p5	60M	9.6M	50M	17%	/boot

Press any key to continue.

tput_menu

Making Time

For our final exercise, we will make something useful; a large character clock. To do this, we first need to install a program called `banner`. The `banner` program accepts one or more words as arguments and displays them like so:

This program has been around for a long time and there are several different implementations. On Debian-based systems (such as Ubuntu) the package is called “sysvbanner”, on Red Hat-based systems the package is called simply “banner”. Once we have `banner` installed we can run this script to display our clock:

```
#!/bin/bash

# tclock - Display a clock in a terminal

BG_BLUE=$(tput setab 4)
FG_BLACK=$(tput setaf 0)
FG_WHITE=$(tput setaf 7)

terminal_size() { # Calculate the size of the terminal

    terminal_cols=$(tput cols)
    terminal_rows=$(tput lines)"
```

```

}

banner_size() {

    # Because there are different versions of banner, we need to
    # calculate the size of our banner's output

    banner_cols=0
    banner_rows=0

    while read; do
        [[ ${#REPLY} -gt $banner_cols ]] && banner_cols=${#REPLY}
        ((++banner_rows))
    done < <(banner "12:34 PM")
}

display_clock() {

    # Since we are putting the clock in the center of the terminal,
    # we need to read each line of banner's output and place it in the
    # right spot.

    local row=$clock_row

    while read; do
        tput cup $row $clock_col
        echo -n "$REPLY"
        ((++row))
    done < <(banner "$(date '+%I:%M %p') ")
}

# Set a trap to restore terminal on Ctrl-c (exit).
# Reset character attributes, make cursor visible, and restore
# previous screen contents (if possible).

trap 'tput sgr0; tput cnorm; tput rmcup || clear; exit 0' SIGINT

# Save screen contents and make cursor invisible
tput smcup; tput civis

# Calculate sizes and positions
terminal_size
banner_size
clock_row=$((($terminal_rows - $banner_rows) / 2))
clock_col=$((($terminal_cols - $banner_cols) / 2))
progress_row=$((($clock_row + $banner_rows + 1)))
progress_col=$((($terminal_cols - 60) / 2))

# In case the terminal cannot paint the screen with a background
# color (tmux has this problem), create a screen-size string of
# spaces so we can paint the screen the hard way.

blank_screen=
for ((i=0; i < ($terminal_cols * $terminal_rows); ++i)); do
    blank_screen+="$blank_screen"
done

# Set the foreground and background colors and go!
echo -n ${BG_BLUE}${FG_WHITE}
while true; do

```

```

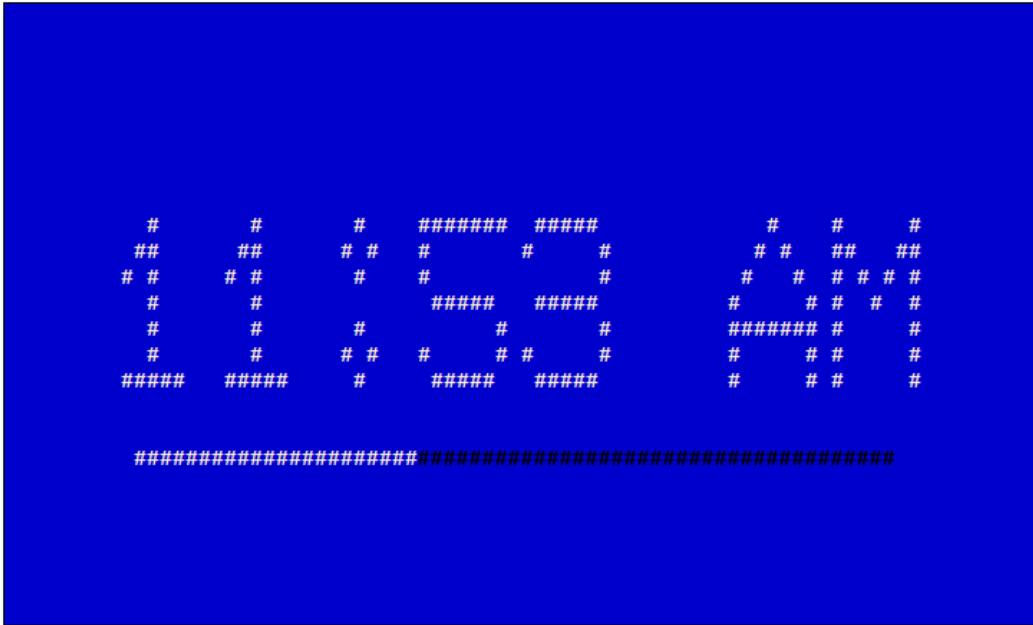
# Set the background and draw the clock

if tput bce; then # Paint the screen the easy way if bce is supported
    clear
else # Do it the hard way
    tput home
    echo -n "$blank_screen"
fi
tput cup $clock_row $clock_col
display_clock

# Draw a black progress bar then fill it in white
tput cup $progress_row $progress_col
echo -n ${FG_BLACK}
echo -n "#####
tput cup $progress_row $progress_col
echo -n ${FG_WHITE}

# Advance the progress bar every second until a minute is used up
for ((i = $(date +%S); i < 60; ++i)); do
    echo -n "#"
    sleep 1
done
done

```



tclock script in action

Our script paints the screen blue and places the current time in the center of the terminal window. This script does not dynamically update the display's position if the terminal is resized (that's an enhancement left to the reader). A progress bar is displayed beneath the clock and it is updated every second until the next minute is reached, when the clock itself is updated.

One interesting feature of the script is how it deals with painting the screen. Terminals that support the “bce” capability erase using the current background color. So, on

terminals that support bce, this is easy. We simply set the background color and then clear the screen. Terminals that do not support bce always erase to the default color (usually black).

To solve this problem, our this script creates a long string of spaces that will fill the screen. On terminal types that do not support bce (for example, screen) the background color is set, the cursor is moved to the home position and then the string of spaces is drawn to fill the screen with the desired background color.

Summing Up

Using `tput`, we can easily add visual enhancements to our scripts. While it's important not to get carried away, lest we end up with a garish, blinking mess, adding text effects and color can increase the visual appeal of our work and improve the readability of information we present to our users.

Further Reading

- The terminfo man page contains the entire list of terminal capabilities defined terminfo database.
- On most systems, the `/lib/terminfo` and `/usr/share/terminfo` directories contain the all of the terminals supported by terminfo.
- [Bash Hacker's Wiki](#) has a good entry on the subject of text effects using `tput`. The page also has some interesting example scripts.
- [Greg's Wiki](#) contains useful information about setting text colors using `tput`.
- [Bash Prompt HOWTO](#) discusses using `tput` to apply text effects to the shell prompt.

6 dialog

If we look at contemporary software, we might be surprised to learn that the majority of code in most programs today has very little to do with the real work for which the program was intended. Rather, the majority of code is used to create the user interface. Modern graphical programs need large amounts of CPU time and memory for their sophisticated eye candy. This helps explain why command line programs usually use so little memory and CPU compared to their GUI counterparts.

Still, the command line interface is often inconvenient. If only there were some way to emulate common graphical user interface features on a text display.

In this adventure, we're going to look at `dialog`, a program that does just that. It displays various kinds of *dialog boxes* that we can incorporate into our shell scripts to give them a much friendlier face. `dialog` dates back a number of years and is now just one member of a family of programs that attempt to solve the user interface problem for command line users. The More Redirection adventure is a suggested prerequisite to this adventure.

Features

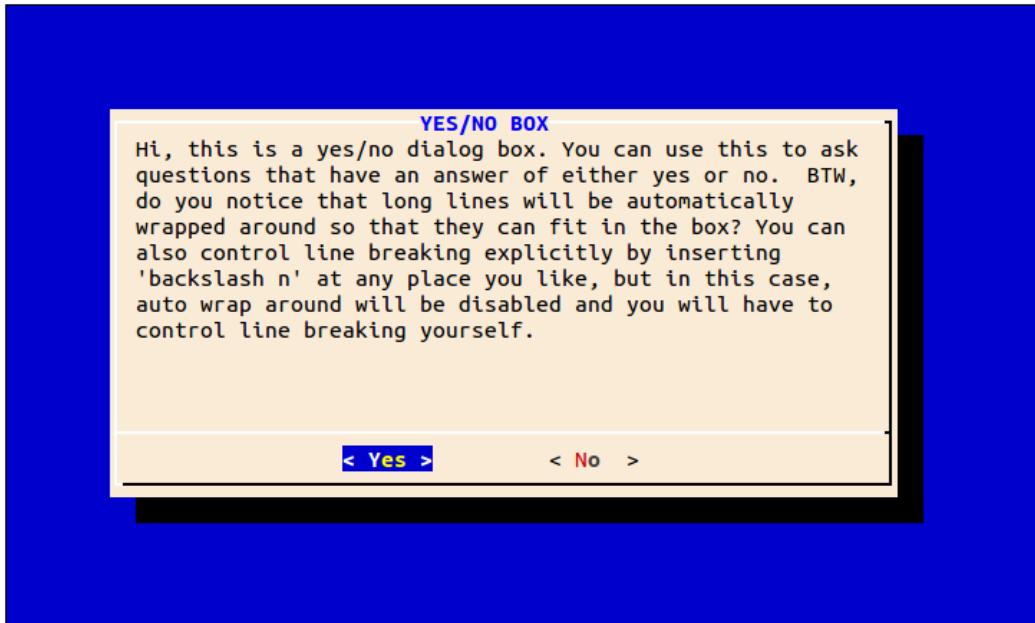
`dialog` is a fairly large and complex program (it has almost 100 command line options), but compared to the typical graphical user interface, it's a real lightweight. Still, it is capable of many user interface tricks. With `dialog`, we can generate the following types of dialog boxes (version 1.2 shown):

Dialog	Option	Description
Build List	--buildlist	Displays two lists, side-by-side. The list on the left contains unselected items, the list on the right selected items. The user can move items from one list to the other.
Calendar	--calendar	Displays a calendar and allow the user to select a date.
Checklist	--checklist	Presents a list of choices and allow the user to select one or more items.
Directory Select	--dselect	Displays a directory selection dialog.
Edit Box	--editbox	Displays a rudimentary text file editor.
Form	--form	Allows the user to enter text into multiple fields.
File Select	--fselect	A file selection dialog.
Gauge	--gauge	Displays a progress indicator showing the percentage of completion.
Info Box	--infobox	Displays a message (with an optional timed pause) and terminates.
Input Box	--inputbox	Prompts the user to enter/edit a text field.
Menu Box	--menubox	Displays a list of choices.

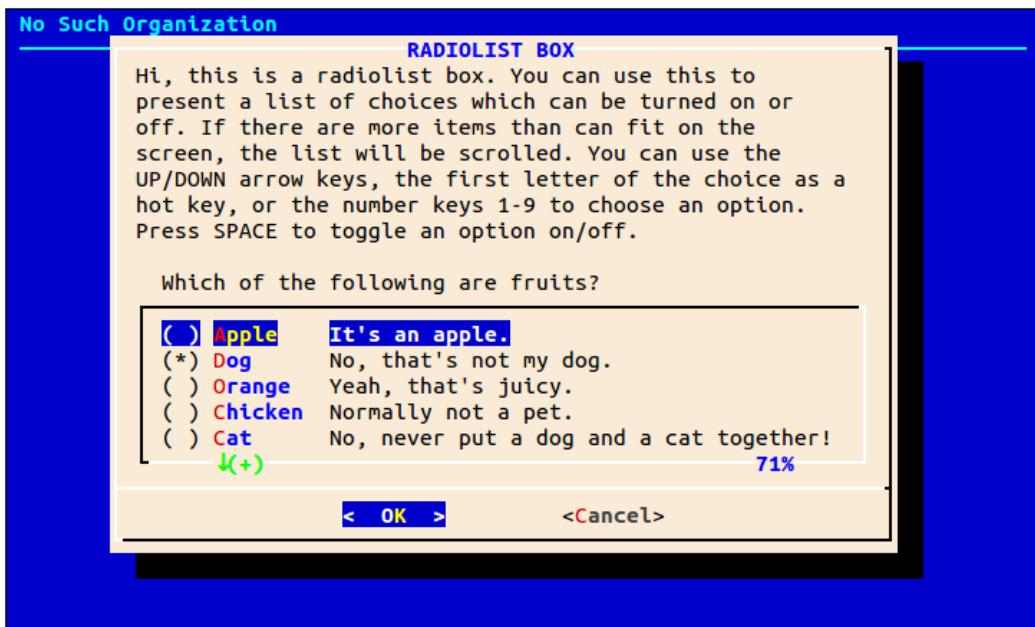
Dialog	Option	Description
Message Box	--msgbox	Displays a text message and waits for the user to respond.
Password Box	--passwordbox	Similar to an input box, but hides the user's entry.
Pause	--pause	Displays a text message and a countdown timer. The dialog terminates when the timer runs out or when the user presses either the OK or Cancel button.
Program Box	--programbox	Displays the output of a piped command. When the command completes, the dialog waits for the user to press an OK button.
Progress Box	--progressbox	Similar to the program box except the dialog terminates when the piped command completes, rather than waiting for the user to press OK.
Radio List Box	--radiolist	Displays a list of choices and allows the user to select a single item. Any previously selected item becomes unselected.
Range Box	--rangebox	Allows the user to select a numerical value from within a specified range using a keyboard-based slider.
Tail Box	--tailbox	Displays a text file with real-time updates. Works like the command <code>tail -f</code> .
Text Box	--textbox	A simple text file viewer. Supports many of the same keyboard commands as <code>less</code> .
Time Box	--timebox	A dialog for entering a time of day.
Tree View	--treeview	Displays a list of items in a tree-shaped hierarchy.
Yes/No Box	--yesno	Displays a text message and gives the user a chance to respond with either "Yes" or "No."

Supported dialog boxes

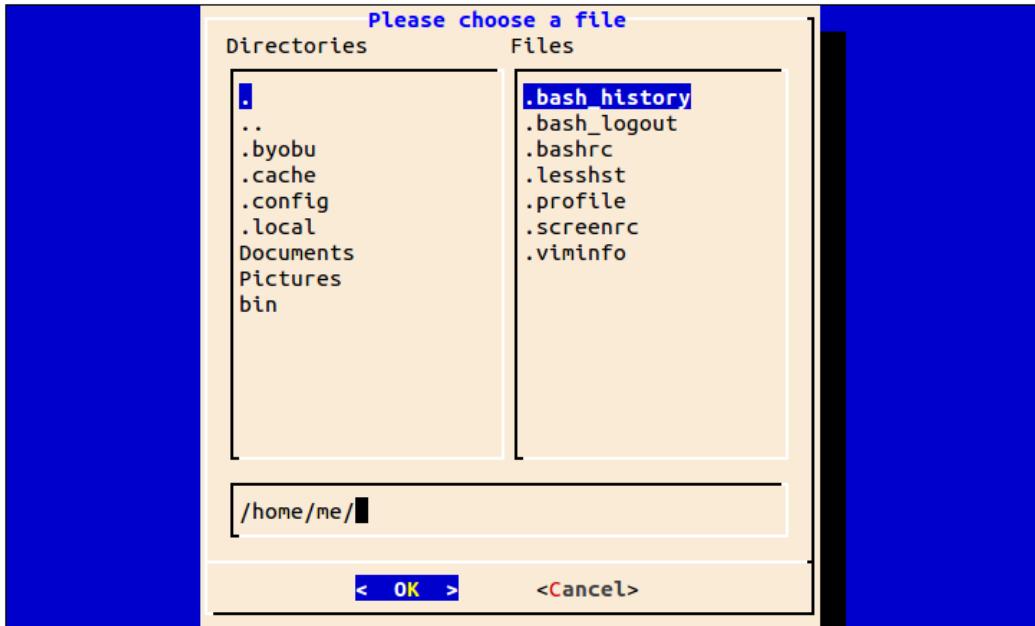
Here are some examples:



Screen shot of the yesno dialog



Screen shot of the radiolist dialog



Screen shot of the fselect dialog

Availability

dialog is available from most distribution repositories as the package “dialog”. Besides the program itself, the dialog package includes a fairly comprehensive man page and a large set of sample programs that demonstrate the various dialog boxes it can display. After installation on a Debian-based system, these sample programs can be found in the /usr/share/doc/dialog/examples directory. Other distributions are similar.

By the way, using [Midnight Commander](#) to browse the examples directory is a great way to run the example programs and to study the scripts themselves:

Left	File	Command	Options	Right
<- ...share/doc/dialog/examples -.[^>]				<- ...share/doc/dialog/examples -.[^>]
'n Name	Size	Modify time		'n Name
*checkbox10	815	Jan 13 2010		/..
*editbox	1039	Jan 13 2010		*calendar
*editbox-utf8	852	Jan 13 2010		*calendar-stdout
*editbox2	1075	Jan 13 2010		*calendar2
*form1	1428	Oct 4 2011		*calendar2-stdout
*form2	2192	Jan 13 2010		*checkbox
*fselect	214	Jan 13 2010		*checkbox-utf8
*fselect-stdout	199	Jan 13 2010		*checkbox10
*fselecti	322	Jan 13 2010		*editbox
*fselecti-stdout	388	Jan 13 2010		*editbox-utf8
*fselect2	212	Jan 13 2010		*editbox2
*fselect2-stdout	198	Jan 13 2010		*form1
*gauge	285	Jan 13 2010		*form2
*gauge0-input-fd	390	Jan 13 2010		*fselect
*gauge2	468	Jan 13 2010		*fselect-stdout
*fselect				*calendar-stdout
	984M/5872M (16%)			984M/5872M (16%)

Browsing the examples with Midnight Commander

How it Works

On the surface, `dialog` appears straightforward. We launch `dialog` followed by one or more common options (options that apply regardless of the desired dialog box) and then the box option and its associated parameters. The tricky part of using `dialog` is getting data out of it.

The data that `dialog` takes in (such as a string entered into a input box) is normally returned on standard error. This is because `dialog` uses standard output to display text on the terminal when it is drawing the dialog box itself. There are a couple of techniques we can use to handle the returned data. Let's take a look at them.

Method 1: Store the Results in a Temporary File

The first method is to use a temporary file. The sample programs supplied with `dialog` provide some examples (this script has been modified from the original for clarity):

```
#!/bin/bash

# inputbox - demonstrate the input dialog box with a temporary file

# Define the dialog exit status codes
: ${DIALOG_OK=0}
: ${DIALOG_CANCEL=1}
: ${DIALOG_HELP=2}
: ${DIALOG_EXTRA=3}
: ${DIALOG_ITEM_HELP=4}
: ${DIALOG_ESC=255}
```

```

# Create a temporary file and make sure it goes away when we're done
tmp_file=$(tempfile 2>/dev/null) || tmp_file=/tmp/test$$
trap "rm -f $tmp_file" 0 1 2 5 15

# Generate the dialog box
dialog --title "INPUT BOX" \
    --clear \
    --inputbox \
"Hi, this is an input dialog box. You can use \n
this to ask questions that require the user \n
to input a string as the answer. You can \n
input strings of length longer than the \n
width of the input box, in that case, the \n
input field will be automatically scrolled. \n
You can use BACKSPACE to correct errors. \n\n
Try entering your name below:" \
16 51 2> $tmp_file

# Get the exit status
return_value=$?

# Act on it
case $return_value in
    $DIALOG_OK)
        echo "Result: `cat $tmp_file`";;
    $DIALOG_CANCEL)
        echo "Cancel pressed.";;
    $DIALOG_HELP)
        echo "Help pressed.";;
    $DIALOG_EXTRA)
        echo "Extra button pressed.";;
    $DIALOG_ITEM_HELP)
        echo "Item-help button pressed.";;
    $DIALOG_ESC)
        if test -s $tmp_file ; then
            cat $tmp_file
        else
            echo "ESC pressed."
        fi
    ;;
esac

```

The first part of the script defines some constants that are used to represent the six possible exit status values supported by `dialog`. They are used to tell the calling script which button on the dialog box (or alternately, the Esc key) was used to terminate the dialog. The construct used to do this is somewhat interesting. First, each line begins with the null command ":" which is a command that does nothing. Yes, really. It intentionally does nothing, because sometimes we need a command (for syntax reasons) but don't actually want to do anything. Following the null command is a parameter expansion. The expansion is similar in form to one we covered in Chapter 34 of TLCL:

`$ {parameter:=value}`

This sets a default value for a parameter (variable) that is either unset (it does not exist at all), or is set, but empty. The author of the example code is being very cautious here and

has removed the colon from the expansion. This changes the meaning of the expansion to mean that a default value is set only if the parameter is unset rather than unset or empty.

The next part of the example creates a temporary file named `tmp_file` by using the `tempfile` command, which is a program used to create a temporary file in a secure manner. Next, we set a trap to make sure that the temporary file is deleted if the program is somehow terminated. Neatness counts!

At last, we get to the `dialog` command itself. We start off setting a title for the input box and specify the `--clear` option to tell `dialog` that we want to erase any previous dialog box from the screen. Next, we indicate the type of dialog box we want and its required arguments. These include the text to be displayed above the input field, and the desired height and width of the box. Though the example specifies exact dimensions for the box, we could also specify zero for both values and `dialog` will attempt to automatically determine the correct size.

Since `dialog` normally outputs its results to standard error, we redirect its file descriptor to our temporary file for storage.

The last thing we have to do is collect the exit status of the command in a variable (`return_value`) so that we can figure out which button the user pressed to terminate the dialog box. At the end of the script, we look at this value and act accordingly.

Method 2: Use Command Substitution and Redirection

The second method of receiving data from `dialog` involves redirection. In the script that follows, we pass the results from `dialog` to a variable rather than a file. To do this, we need to first perform some redirection.

```
#!/bin/bash

# inputbox - demonstrate the input dialog box with redirection

# Define the dialog exit status codes
: ${DIALOG_OK=0}
: ${DIALOG_CANCEL=1}
: ${DIALOG_HELP=2}
: ${DIALOG_EXTRA=3}
: ${DIALOG_ITEM_HELP=4}
: ${DIALOG_ESC=255}

# Duplicate (make a backup copy of) file descriptor 1
# on descriptor 3
exec 3>&1

# Generate the dialog box while running dialog in a subshell
result=$(dialog \
--title "INPUT BOX" \
--clear \
--inputbox \
```

```

"Hi, this is an input dialog box. You can use \n
this to ask questions that require the user \n
to input a string as the answer. You can \n
input strings of length longer than the \n
width of the input box, in that case, the \n
input field will be automatically scrolled. \n\n
You can use BACKSPACE to correct errors. \n\n
Try entering your name below:" \
16 51 2>&1 1>&3)

# Get dialog's exit status
return_value=$?

# Close file descriptor 3
exec 3>&-

# Act on the exit status
case $return_value in
    $DIALOG_OK)
        echo "Result: $result";;
    $DIALOG_CANCEL)
        echo "Cancel pressed.";;
    $DIALOG_HELP)
        echo "Help pressed.";;
    $DIALOG_EXTRA)
        echo "Extra button pressed.";;
    $DIALOG_ITEM_HELP)
        echo "Item-help button pressed.";;
    $DIALOG_ESC)
        if test -n "$result" ; then
            echo "$result"
        else
            echo "ESC pressed."
        fi
        ;;
esac

```

At first glance, the redirection may seem nonsensical. First, we duplicate file descriptor 1 (stdout) to descriptor 3 using `exec` (this was covered in [More Redirection](#)) to create a backup copy of descriptor 1.

The next step is to perform a command substitution and assign the output of the `dialog` command to the variable `result`. The command includes redirections of descriptor 2 (stderr) to be the duplicate of descriptor 1 and lastly, descriptor 1 is restored to its original value by duplicating descriptor 3 which contains the backup copy. What might not be immediately apparent is why the last redirection is needed. Inside the subshell, standard output (descriptor 1) does not point to the controlling terminal. Rather, it is pointing to a pipe that will deliver its contents to the variable `result`. Since `dialog` needs standard output to point to the terminal so that it can display the input box, we have to redirect standard error to standard output (so that the output from `dialog` ends up in the `result` variable), then redirect standard output back to the controlling terminal.

So, which method is better, temporary file or command substitution? Probably command substitution, since it avoids file creation.

Before and After

Now that we have a basic grip on how to use `dialog`, let's apply it to a practical example.

Here we have an “ordinary” script. It’s a menu-driven system information program similar to one discussed in Chapter 29 of TLCL:

```
#!/bin/bash

# while-menu: a menu-driven system information program

DELAY=3 # Number of seconds to display results

while true; do
    clear
    cat << _EOF_
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

_EOF_

read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
    case $REPLY in
        1)
            echo "Hostname: $HOSTNAME"
            uptime
            sleep $DELAY
            continue
            ;;
        2)
            df -h
            sleep $DELAY
            continue
            ;;
        3)
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/* 2> /dev/null
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME 2> /dev/null
            fi
            sleep $DELAY
            continue
            ;;
        0)
            break
            ;;
    esac
else
    echo "Invalid entry."
    sleep $DELAY
fi
done
```

```

echo "Program terminated." 

Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] > ■

```

A script displaying a text menu

The script displays a simple menu of choices. After the user enters a selection, the selection is validated to make sure it is one of the permitted choices (the numerals 0-3) and if successfully validated, a `case` statement is used to carry out the selected action. The results are displayed for the number of seconds defined by the `DELAY` constant, after which the whole process is repeated until the user selects the menu choice to exit the program.

Here is the script modified to use `dialog` to provide a new user interface:

```

#!/bin/bash

# while-menu-dialog: a menu driven system information program

DIALOG_CANCEL=1
DIALOG_ESC=255
HEIGHT=0
WIDTH=0

display_result() {
    dialog --title "$1" \
        --no-collapse \
        --msgbox "$result" 0 0
}

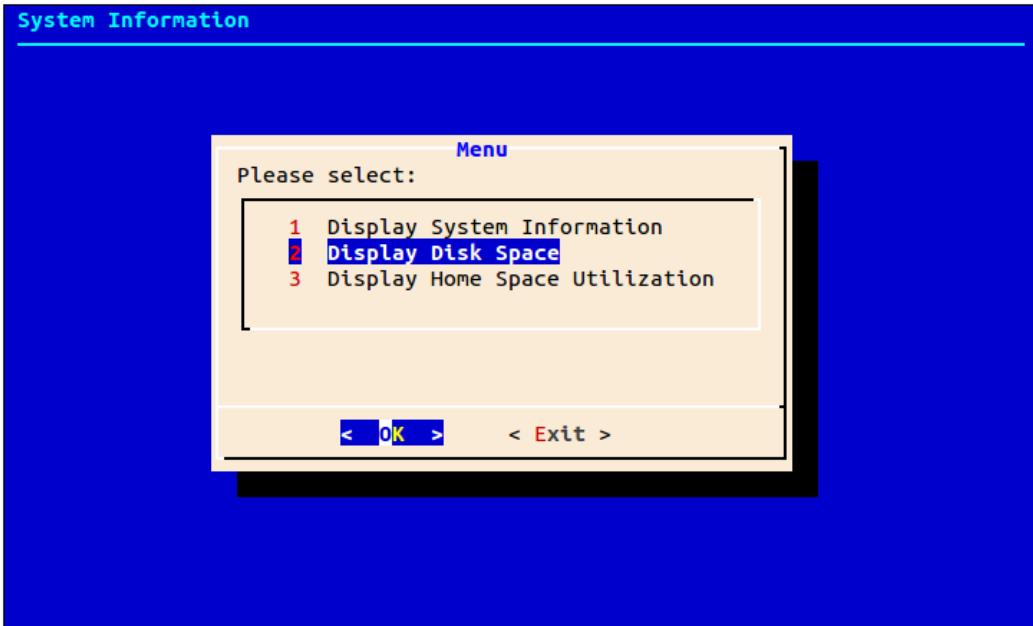
while true; do
    exec 3>&1
    selection=$(dialog \
        --backtitle "System Information" \
        --title "Menu" \
        --clear \
        --cancel-label "Exit" \

```

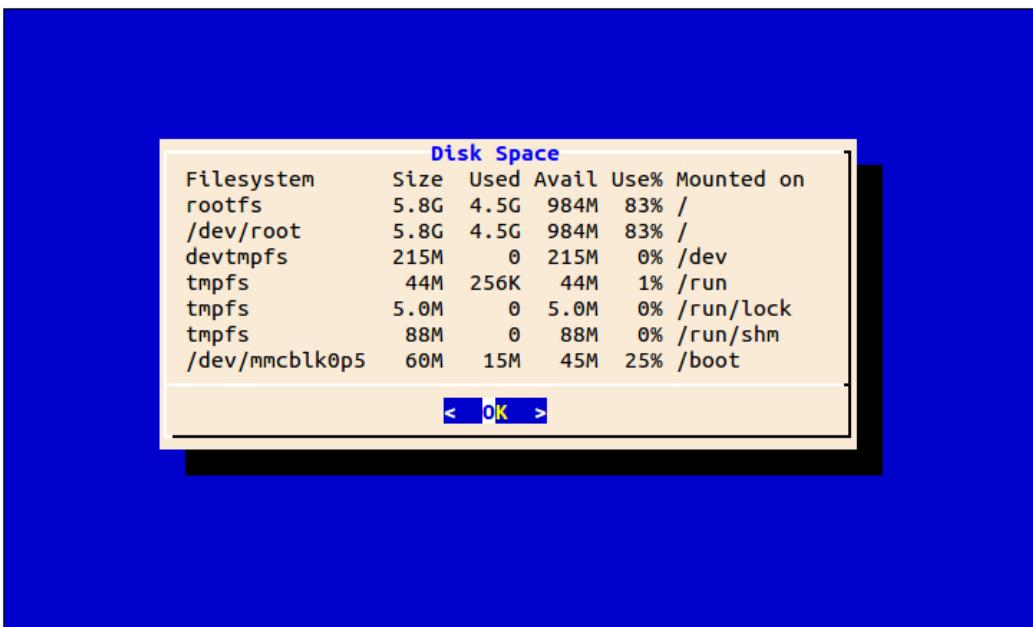
```

--menu "Please select:" $HEIGHT $WIDTH 4 \
"1" "Display System Information" \
"2" "Display Disk Space" \
"3" "Display Home Space Utilization" \
2>&1 1>&3)
exit_status=$?
exec 3>-
case $exit_status in
$DIALOG_CANCEL)
    clear
    echo "Program terminated."
    exit
;;
$DIALOG_ESC)
    clear
    echo "Program aborted." >&2
    exit 1
;;
esac
case $selection in
1 )
    result=$(echo "Hostname: $HOSTNAME"; uptime)
    display_result "System Information"
;;
2 )
    result=$(df -h)
    display_result "Disk Space"
;;
3 )
    if [[ $(id -u) -eq 0 ]]; then
        result=$(du -sh /home/* 2> /dev/null)
        display_result "Home Space Utilization (All Users)"
    else
        result=$(du -sh $HOME 2> /dev/null)
        display_result "Home Space Utilization ($USER)"
    fi
;;
esac
done

```



Script displaying a dialog menu



Displaying results with a msgbox

As we can see, the script has some structural changes. First, we no longer have to validate the user's selection. The menu box only allows valid choices. Second, there is a function defined near the beginning to display the output of each selection.

We also notice that several of `dialog`'s common options have been used:

- `--no-collapse` prevents `dialog` from reformatting message text. Use this when the exact presentation of the text is needed.
- `--backtitle` sets the title of the background under the dialog box.
- `--clear` clears the background of any previous dialog box.
- `--cancel-label` sets the string displayed on the “cancel” button. In this script, it is set to “Exit” since that is a better description of the action taken when it is selected.

Limitations

While it’s true that `dialog` can produce many kinds of dialog boxes, care must be taken to remember that `dialog` has significant limitations. Some of the dialog boxes have rather odd behaviors compared to their traditional GUI counterparts. For example, the edit box used to edit text files cannot perform cut and paste and files to be edited cannot contain tab characters. The behavior of the file box is more akin to the shell’s tab completion feature than to a GUI file selector.

Summing Up

The shell is not really intended for large, interactive programs, but using `dialog` can make small to moderate interactive programs possible. It provides a useful variety of dialog boxes, allowing many types of user interactions which would be very difficult to implement with the shell alone. If we keep our expectations modest, `dialog` can be a great tool.

Further Reading

- The `dialog` man page is well-written and contains a complete listing of its numerous options.
- `dialog` normally includes a large set of example programs which can be found in the `/usr/share/doc/dialog` directory.
- The `dialog` project home page can be found at <https://invisible-island.net/dialog/>

7 AWK

One of the great things we can do in the shell is embed other programming languages within the body of our scripts. We have seen hints of this with the stream editor `sed`, and the arbitrary precision calculator program `bc`. By using the shell's single quoting mechanism to isolate text from shell expansion, we can freely express other programming languages, provided we have a suitable language interpreter to execute them.

In this adventure, we are going to look at one such program, `awk`.

History

The AWK programming language is truly one of the classic tools used in Unix. It dates back to the very earliest days of the Unix tradition. It was originally developed in the late 1970's at AT&T Bell Telephone Laboratories by Alfred Aho, Peter Weinberger, and Brian Kernighan. The name "AWK" comes from the last names of the three authors. It underwent major improvements in 1985 with the release of `nawk` or "new awk." It is that version that we still use today, though it is usually just called `awk`.

Availability

`awk` is a standard program found in most every Linux distribution. Two free/open source versions of the program are in common use. One is called `mawk` (short for Mike's awk, named for its original author, Mike Brennan) and `gawk` (GNU awk). Both versions fully implement the 1985 `nawk` standard as well as add a variety of extensions. For our purposes, either version is fine, since we will be focusing on the traditional `nawk` features. In most distributions, the name `awk` is symbolically linked to either `mawk` or `gawk`.

So, What's it Good For?

Though AWK is fairly general purpose, it is really designed to create *filters*, that is, programs that accept standard input, transform data, and send it to standard output. In particular, AWK is very good at processing *columnar data*. This makes it a good choice for developing report generators, and tools that are used to re-format data. Since it has strong regular expression support, it's good for very small text extraction and reformatting problems, too. Like `sed`, many AWK programs are just one line long.

In recent years, AWK has fallen a bit out of fashion, being supplanted by other, newer, interpreted languages such as *Perl* and *python*, but AWK still has some advantages:

- It's easy to learn. The language is not overly complex and has a syntax much like the C programming language, so learning it will be useful in the future when we study other languages and tools.
- It really excels at solving certain types of problems.

How it Works

The structure of an AWK program is somewhat unique among programming languages. Programs consist of a series of one or more *pattern* and *action* pairs. Before we get into that though, let's look at what the typical AWK program does.

We already know that the typical AWK program acts as a filter. It reads data from standard input, and outputs filtered data on standard output. It reads data one *record* at a time. By default, a record is a line of text terminated by a newline character. Each time a record is read, AWK automatically separates the record into *fields*. Fields are, again by default, separated by whitespace. Each field is assigned to a variable, which is given a numeric name. Variable \$1 is the first field, \$2 is the second field, and so on. \$0 signifies the entire record. In addition, a variable named NF is set containing the number of fields detected in the record.

Pattern/action pairs are tests and corresponding actions to be performed on each record. If the pattern is true, then the action is performed. When the list of patterns is exhausted, the AWK program reads the next record and the process is repeated.

Let's try a really simple case. We'll filter the output of an `ls` command:

```
me@linuxbox ~ $ ls -l /usr/bin | awk '{print $0}'
```

The AWK program is contained within the single quotes following the `awk` command. Single quotes are important because we do not want the shell to attempt any expansion on the AWK program, since its syntax has nothing to do with the shell. For example, \$0 represents the value of the entire record the AWK program read on standard input. In AWK, the \$ means “field” and is not a trigger for parameter expansion as it is in the shell.

Our example program consists of a single action with no pattern present. This is allowed and it means that every record matches the pattern. When we run this command, it simply outputs every line of input much like the `cat` command.

If we look at a typical line of output from `ls -l`, we see that it consists of 9 fields, each separated from its neighbor by one or more whitespace characters:

```
-rwxr-xr-x 1 root root      265 Apr 17 2012 zxpdf
```

Let's add a pattern to our program so it will only print lines with more than 9 fields:

```
me@linuxbox ~ $ ls -l /usr/bin | awk 'NF > 9 {print $0}'
```

We now see a list of symbolic links in `/usr/bin` since those directory listings contain more than 9 fields. This pattern will also match entries with file names containing embedded spaces, since they too will have more than 9 fields.

Special Patterns

Patterns in AWK can have many forms. There are conditional expressions like we have just seen. There are also regular expressions, as we would expect. There are two special patterns called BEGIN and END. The BEGIN pattern carries out its corresponding action before the first record is read. This is useful for initializing variables, or printing headers at the beginning of output. Likewise, the END pattern performs its corresponding action after the last record is read from the input file. This is good for outputting summaries once the input has been processed.

Let's try a more elaborate example. We'll assume for the moment that the directory does not contain any file names with embedded spaces (though this is *never* a safe assumption). We could use the following script to list symbolic links:

```
#!/bin/bash

# Print a directory report

ls -l /usr/bin | awk '
    BEGIN {
        print "Directory Report"
        print "====="
    }

    NF > 9 {
        print $9, "is a symbolic link to", $NF
    }

    END {
        print "====="
        print "End Of Report"
    }
'
```

In this example, we have 3 pattern/action pairs in our AWK program. The first is a BEGIN pattern and its action that prints the report header. We can spread the action over several lines, though the opening brace “{” of the action must appear on the same line as the pattern.

The second pattern tests the current record to see if it contains more than 9 fields and, if true, the 9th field is printed, followed by some text and the final field in the record. Notice how this was done. The NF variable is preceded by a “\$”, thus it refers to the NFth field rather than the value of NF itself.

Lastly, we have an END pattern. Its corresponding action prints the “End Of Report” message once all of the lines of input have been read.

Invocation

There are three ways we can run an AWK program. We have already seen how to embed a program in a shell script by enclosing it inside single quotes. The second way is to place the awk script in its own file and call it from the `awk` program like so:

```
awk -f program_file
```

Lastly, we can use the *shebang* mechanism to make the AWK script a standalone program like a shell script:

```
#!/usr/bin/awk -f

# Print a directory report

BEGIN {
    print "Directory Report"
    print "====="
}

NF > 9 {
    print $9, "is a symbolic link to", $NF
}

END {
    print "====="
    print "End Of Report"
}
```

The Language

Let's take a look at the features and syntax of AWK programs.

Program Format

The formatting rules for AWK programs are pretty simple. Actions consist of one or more statements surrounded by braces (`{ }{ }`) with the starting brace appearing on the same line as the pattern. Blank lines are ignored. Comments begin with a pound sign (#) and may appear at the end of any line. Long statements may be broken into multiple lines using line continuation characters (a backslash followed immediately by a newline). Lists of parameters separated by commas may be broken after any comma. Here is an example:

```
BEGIN { # The action's opening brace must be on same line as the pattern

    # Blank lines are ignored

    # Line continuation characters can be used to break long lines
    print \
        $1, # Parameter lists may be broken by commas
        $2, # Comments can appear at the end of any line
        $3

    # Multiple statements can appear on one line if separated by
    # a semicolon
    print "String 1"; print "String 2"
```

```
} # Closing brace for action
```

Patterns

Here are the most common types of patterns used in AWK:

BEGIN and END

As we saw earlier, the BEGIN and END patterns perform actions before the first record is read and after the last record is read, respectively.

relational-expression

Relational expressions are used to test values. For example, we can test for equivalence:

```
$1 == "Fedora"
```

or for relations such as:

```
$3 >= 50
```

It is also possible to perform calculations like:

```
$1 * $2 < 100
```

/regular-expression/

AWK supports extended regular expressions like those supported by egrep. Patterns using regular expression can be expressed in two ways. First, we can enclose a regular expression in slashes and a match is attempted on the entire record. If a finer level of control is needed, we can provide an expression containing the string to be matched using the following syntax:

expression ~ /regexp/

For example, if we only wanted to attempt a match on the third field in a record, we could do this:

```
$3 ~ /^[567]/
```

From this, we can think of the “~” as meaning “matches” or “contains”, thus we can read the pattern above as “field 3 matches the regular expression ^[567]”.

pattern logical-operator pattern

It is possible to combine patterns together using the logical operators || and &&, meaning OR and AND, respectively. For example, if we want to test a record to see if the first field is a number greater than 100 and the last field is the word “Debit”, we can do this:

```
$1 > 100 && $NF == "Debit"
```

! pattern

It is also possible to negate a pattern, so that only records that do not match a specified pattern are selected.

pattern, pattern

Two patterns separated by a comma is called a *range pattern*. With it, once the first pattern is matched, every subsequent record matches until the second pattern is matched. Thus, this type of pattern will select a range of records. Let's imagine that we have a list of records and that the first field in each record contains a sequential record number:

```
0001    field    field    field  
0002    field    field    field  
0003    field    field    field
```

and so on. And let's say that we want to extract records 0050 through 0100, inclusive. To do so, we could use a range pattern like this:

```
$1 == "0050", $1 == "0100"
```

Fields and Records

The AWK language is so useful because of its ability to automatically separate fields and records. While the default is to separate records by newlines and fields by whitespace, this can be adjusted. The `/etc/passwd` file, for example, does not separate its fields with whitespace; rather, it uses colons (:). AWK has a built in variable named FS (field separator) that defines the delimiter separating fields in a record. Here is an AWK program that will list the user ID and the user's name from the file:

```
BEGIN { FS = ":" }  
{ print $1, $5 }
```

This program has two pattern/action pairs. The first action is performed before the first record is read and sets the input field separator to be the colon character.

The second pair contains only an action and no pattern. This will match every record. The action prints the first and fifth fields from each record.

The FS variable may contain a regular expression, so really powerful methods can be used to separate fields.

Records are normally separated by newlines, but this can be adjusted too. The built-in variable RS (record separator) defines how records are delimited. A common type of record consists of multiple lines of data separated by one or more blank lines. AWK has a shortcut for specifying the record separator in this case. We just define RS to be an empty string:

```
RS = ""
```

Note that when this is done, newlines, in addition to any other specified characters, will always be treated as field separators regardless of how the FS variable is set. When we process multi-line records, we will often want to treat each line as a separate field, so doing this is often desirable:

```
BEGIN { FS = "\n"; RS = "" }
```

Variables and Data Types

AWK treats data as either a string or a number, depending on its context. This can sometimes become an issue with numbers. AWK will often treat numbers as strings unless something specifically “numeric” is done with them.

We can force AWK to treat a string of digits as a number by performing some arithmetic on it. This is most easily done by adding zero to the number:

```
n = 105 + 0
```

Likewise, we can get AWK to treat a string of digits as a string by concatenating an empty string:

```
s = 105 ""
```

String concatenation in AWK is performed using a space character as an operator - an unusual feature of the language.

Variables are created as they are encountered (no prior declaration is required), just like the shell. Variable names in AWK follow the same rules as the shell. Names may consist of letters, numbers, and underscore characters. Like the shell, the first character of a variable name must not be a number. Variable names are case sensitive.

Built-in Variables

We have already looked at a few of AWK’s built-in variables. Here is a list of the most useful ones:

FS - Field separator

This variable contains a regular expression that is used to separate a record into fields. Its initial value separates fields with whitespace. AWK supports a shortcut to return this variable to its original value:

```
FS = " "
```

The value of FS can also be set using the -F option on the command line. For example, we can quickly extract the user name and UID fields from the /etc/passwd file like this:

```
awk -F: '{print $1, $3}' /etc/passwd
```

NF - Number of fields

This variable updates each time a record is read. We can easily access the last field in the record by referring to \$NF.

NR - Record number

This variable increments each time a record is read, thus it contains the total number of records read from the input stream. Using this variable, we could easily simulate a `wc -l` command with:

```
awk 'END {print NR}'
```

or number the lines in a file with:

```
awk '{print NR, $0}'
```

OFS - Output field separator

This string is used to separate fields when printing output. The default is a single space. Setting this can be handy when reformatting data. For example, we could easily change a table of values to a CSV (comma separated values) file by setting OFS to equal “,”. To demonstrate, here is a program that reads our directory listing and outputs a CSV stream:

```
ls -l | awk 'BEGIN {OFS = ","}  
NF == 9 {print $1,$2,$3,$4,$5,$6,$7,$8,$9}'
```

We set the pattern to only match input lines containing 9 fields. This eliminates symbolic links and other weird file names from the data to be processed.

Each line of the resulting output would resemble this:

```
-rwxr-xr-x,1,root,root,100984,Jan,11,2015,a2p
```

If we had omitted setting OFS, the print statement would use the default value (a single space):

```
ls -l | awk 'NF == 9 {print $1,$2,$3,$4,$5,$6,$7,$8,$9}'
```

Which would result in each line of output resembling this:

```
-rwxr-xr-x 1 root root 100984 Jan 11 2015 a2p
```

ORS - Output record separator

This is the string used to separate records when printing output. The default is a newline character. We could use this variable to easily double-space a file by setting ORS to equal two newlines:

```
ls -l | awk 'BEGIN {ORS = "\n\n"} {print}'
```

RS - Record separator

When reading input, AWK interprets this string as the end of record marker. The default value is a newline.

FILENAME

If AWK is reading its input from a file specified on the command line, then this variable contains the name of the file.

FNR - File record number

When reading input from a file specified on the command line, AWK sets this variable to the number of the record read from that file.

Arrays

Single-dimensional arrays are supported in AWK. Data contained in array elements may be either numbers or strings. Array indexes may also be either strings (for *associative arrays*) or numbers.

Assigning values to array elements is done like this:

```
a[1] = 5      # Numeric index  
a["five"] = 5 # String index
```

Though AWK only supports single dimension arrays (like bash), it also provides a mechanism to simulate multi-dimensional arrays. When assigning an array index, it is possible to use this form to represent more than one dimension:

```
a[j,k] = "foo"
```

When AWK sees this construct, it builds an index consisting of the strings `j` and `k` separated by the contents of the built-in variable `SUBSEP`. By default, `SUBSEP` is set to “\034” (character 34 octal, 28 decimal). This ASCII control code is fairly obscure and thus unlikely to appear in ordinary text, so it’s pretty safe for AWK to use.

Note that both `mawk` and `gawk` implement language extensions to support multi-dimensional arrays in a more formal way. Consult their respective documentation for details. If a portability is needed, use the method above rather than the implementation-specific feature.

We can delete arrays and array elements this way:

```
delete a[i]      # delete a single element  
delete a          # delete array a
```

Arithmetic and Logical Expressions

AWK supports a pretty complete set of arithmetic and logical operators:

Operators

Assignment	= += -= *= /= %= ^= ++ --
Relational	< > <= >= == !=
Arithmetic	+ - * / % ^
Matching	~ ! ~
Array	in
Logical	&&

Arithmetic and logical operators

These operators behave like those in the shell; however, unlike the shell, which is limited to integer arithmetic, AWK arithmetic is floating point. This makes AWK a good way to do more complex arithmetic than the shell alone.

Arithmetic and logical expressions can be used in both patterns and actions. Here's an example that counts the number of lines containing exactly 9 fields:

```
ls -l /usr/bin | awk 'NF == 9 {count++} END {print count}'
```

This AWK program consists of 2 pattern/action pairs. The first one matches lines where the number of fields is equal to 9. The action creates and increments a variable named `count`. Each time a line with exactly 9 fields is encountered in the input stream, `count` is incremented by 1.

The second pair matches when the end of the input stream is reached and the resulting action prints the final value of `count`.

Using this basic form, let's try something a little more useful; a program that calculates the total size of the files in the list:

```
ls -l /usr/bin | awk 'NF >= 9 {total += $5} END {print total}'
```

Here is a slight variation (with shortened variable names to make it a little more concise) that calculates the average size of the files:

```
ls -l /usr/bin | awk 'NF >= 9 {c++; t += $5} END {print t / c}'
```

Flow Control

AWK has many of the same flow control statements that we've seen previously in the shell (with the notable exception of case, though we can think of an AWK program as one big case statement inside a loop) but the syntax more closely resembles that of the C programming language. Actions in AWK often contain complex logic consisting of various statements and flow control instructions. A statement in this context can be a simple statement like:

```
a = a + 1
```

Or a compound statement enclosed in braces such as:

```
{a = a + 1; b = b * a}
```

if (expression) statement

if (expression) statement else statement

The *if/then/else* construct in AWK behaves the way we would expect. AWK evaluates an expression in parenthesis and if the result is non-zero, the statement is carried out. We can see this behavior by executing the following commands:

```
awk 'BEGIN {if (1) print "true"; else print "false"}'  
awk 'BEGIN {if (0) print "true"; else print "false"}'
```

Relational expressions such as `(a < b)` will also evaluate to 0 or 1.

In the example below, we construct a primitive report generator that counts the number of lines that have been output and, if the number exceeds the length of a page, a formfeed character is output and the line counter is reset:

```
ls -1 /usr/bin | awk '  
BEGIN {  
    line_count = 0  
    page_length = 60  
}  
  
{  
    line_count++  
    if (line_count < page_length)  
        print  
    else {  
        print "\f" $0  
        line_count = 0  
    }  
}  
'
```

While the above might be the most obvious way to code this, our knowledge of how evaluations are actually performed, allows us to code this example in a slightly more concise way by using some arithmetic:

```
ls -1 /usr/bin | awk '  
BEGIN {  
    page_length = 60  
}  
  
{  
    if (NR % page_length)  
        print  
    else  
        print "\f" $0  
}  
'
```

Here we exploit the fact that the page boundaries will always fall on even multiples of the page length. If `page_length` equals 60 then the page boundaries will fall on lines 60, 120, 180, 240, and so on. All we have to do is calculate the remainder (modulo) on the

number of lines processed in the input stream (`NR`) divided by the page length and see if the result is zero, and thus an even multiple.

AWK supports an expression that's useful for testing membership in an array:

```
(var in array)
```

where `var` is an index value and `array` is an array variable. Using this expression tests if the index `var` exists in the specified array. This method of testing for array membership avoids the problem of inadvertently creating the index by testing it with methods such as:

```
if (array[var] != "")
```

When the test is attempted this way, the array element `var` is created, since AWK creates variables simply by their use. When the `(var in array)` form is used, no variable is created.

To test for array membership in a multi-dimensional array, the following syntax is used:

```
((var1,var2) in array)
```

for (expression ; expression ; expression) statement

The `for` loop in AWK closely resembles the corresponding one in the C programming language. It is comprised of 3 expressions. The first expression is usually used to initialize a counter variable, the second defines when the loop is completed, and the third defines how the loop is incremented or advanced at each iteration. Here is a demonstration using a `for` loop to print fields in reverse order:

```
ls -l | awk '{s = ""; for (i = NF; i > 0; i--) s = s $i OFS; print s}'
```

In this example we create an empty string named `s`, then begin a loop that starts with the number of fields in the current input line (`i = NF`) and counts down (`i--`) until we reach the first field (`i > 0`). Each iteration of the loop causes the current field and the output field separator to be concatenated to the string `s` (`s = s $i OFS`). After the loop concludes, we print the resulting value of string `s`.

for (var in array) statement

AWK has a special flow control statement for traversing the indexes of an array. Here is an example of what it does:

```
awk 'BEGIN {for (i=0; i<10; i++) a[i]="foo"; for (i in a) print i}'
```

In this program, we have a single `BEGIN` pattern/action that performs the entire exercise without the need for an input stream. We first create an array `a` and add 10 elements, each containing the string "foo". Next, we use `for (i in a)` to loop through all the indexes in the array and print each index. It is important to note that the order of the arrays in memory is *implementation dependent*, meaning that it could be anything, so we cannot rely on the results being in any particular order. We'll look at how to address this problem a little later.

Even without sorted order, this type of loop is useful if we need to process every element in an array. For example, we could delete every element of an array like this:

```
for (i in a) delete a[i]
```

while (expression) statement

do statement while (expression)

The *while* and *do* loops in AWK are pretty straightforward. We determine a condition that must be maintained for the loop to continue. We can demonstrate this using our field reversal program (we'll type it out in multiple lines to make the logic easier to follow):

```
ls -l | awk '{
    s = ""
    i = NF
    while (i > 0) {
        s = s $i OFS
        i--
    }
    print s
}'
```

The *do* loop is similar to the *while* loop; however the *do* loop will always execute its statement at least once, whereas the *while* loop will only execute its statement if the initial condition is met.

break ***continue*** ***next***

The *break*, *continue*, and *next* keywords are used to “escape” from loops. *break* and *continue* behave like their corresponding commands in the shell. *continue* tells AWK to stop and continue with the next iteration of the current loop. *break* tells AWK to exit the current loop entirely. The *next* keyword tells AWK to skip the remainder of the current program and begin processing the next record of input.

exit expression

As with the shell, we can tell AWK to exit and provide an optional expression that sets AWK's exit status.

Regular Expressions

Regular expressions in AWK work like those in `egrep`, a topic we covered in Chapter 19 of TLCL. It is important to note that back references are not supported and that some

versions of AWK (most notably mawk versions prior to 1.3.4) do not support POSIX character classes.

Regular expressions are most often used in patterns, but they are also used in some of the built-in variables such as FS and RS, and they have various roles in the string functions which we will discuss shortly.

Let's try using some simple regular expressions to tally the different file types in our directory listing (we'll make clever use of an associative array too).

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {t["Regular Files"]++}
$1 ~ /^[dD]/{t["Directories"]++}
$1 ~ /^[lL]/{t["Symbolic Links"]++}
END {for (i in t) print i ":" t[i]}
'
```

In this program, we use regular expressions to identify the first character of the first field and increment the corresponding element in array `t`. Since we can use strings as array indexes in AWK, we spell out the file type as the index. This makes printing the results in the END action easy, as we only have to traverse the array with `for (i in t)` to obtain both the name and the accumulated total for each type.

Output Functions

print expr1, expr2, expr3,...

As we have seen, `print` accepts a comma-separated list of arguments. An argument can be any valid expression; however, if an expression contains a relational operator, the entire argument list must be enclosed in parentheses.

The commas are important, because they tell AWK to separate output items with the output field separator (OFS). If omitted, AWK will interpret the members of the argument list as a single expression of string concatenation.

printf(format, expr1, expr2, expr3,...)

In AWK, `printf` is like the corresponding shell built-in (see TLCL Chapter 21 for details). It formats its list of arguments based the contents of a *format string*. Here is an example where we output a list of files and their sizes in kilobytes:

```
ls -l /usr/bin | awk '{printf("%-30s%8.2fK\n", $9, $5 / 1024)}'
```

Writing to Files and Pipelines

In addition to sending output to stdout, we can also send output to files and pipelines.

```
ls -l /usr/bin | awk '
$1 ~ /^-/ {print $0 > "regfiles.txt"}
$1 ~ /^[dD]/{print $0 > "directories.txt"}'
```

```
$1 ~ /^-/ {print $0 > "symlinks.txt"}  
'
```

Here we see a program that writes separate lists of regular files, directories, and symbolic links.

AWK also provides a `>>` operator for appending to files, but since AWK only opens a file once per program execution, the `>` causes AWK to open the file at the beginning of execution and truncate the file to zero length much like we see with the shell. However, once the file is open, it stays open and each subsequent write appends contents to the file. The `>>` operator behaves in the same manner, but when the file is initially opened it is not truncated and all content is appended (i.e., it preserves the contents of an existing file).

AWK also allows output to be sent to pipelines. Consider this program, where we read our directory into an array and then output the entire array:

```
ls -l /usr/bin | awk '  
$1 ~ /^-/ {a[$9] = $5}  
END {for (i in a)  
    {print a[i] "\t" i}}  
'
```

If we run this program, we notice that the array is output in a seemingly random “implementation dependent” order. To correct this, we can pipe the output through `sort`:

```
ls -l /usr/bin | awk '  
$1 ~ /^-/ {a[$9] = $5}  
END {for (i in a)  
    {print a[i] "\t" i | "sort -nr"}}  
'
```

Reading Data

As we have seen, AWK programs most often process data supplied from standard input. However, we can also specify input files on the command line:

```
awk 'program' file...
```

Knowing this, we can, for example, create an AWK program that simulates the `cat` command:

```
awk '{print $0}' file1 file2 file3
```

or `wc`:

```
awk '{chars += length($0); words += NF}  
END {print NR, words, chars + NR}' file1
```

This program has a couple of interesting features. First, it uses the AWK string function `length` to obtain the number of characters in a string. This is one of many string functions that AWK provides, and we will talk more about them in a bit. The second feature is the `chars + NR` expression at the end. This is done because `length($0)` does

not count the newline character at the end of each line, so we have to add them to make the character count come out the same as real `wc`.

Even if we don't include filenames on the command line for AWK to input, we can tell AWK to read data from a file specified from within a program. Normally we don't need to do this, but there are some cases where this might be handy. For example, if we wanted to insert one file inside of another, we could use the `getline` function in AWK. Here's an example that adds a header and footer to an existing body text file:

```
awk 'BEGIN { while (getline <"header.txt" > 0) { print $0 } } {print} END { while (getline <"footer.txt" > 0) { print $0 } }' < body.txt > finished_file.txt
```

`getline` is quite flexible and can be used in a variety of ways:

`getline`

In its most basic form, `getline` reads the next record from the current input stream. `$0`, `NF`, `NR`, and `FNR` are set.

`getline var`

Reads the next record from the current input stream and assigns its contents to the variable `var`. `var`, `NR`, and `FNR` are set.

`getline <file`

Reads a record from `file`. `$0` and `NF` are set. It's important to check for errors when reading from files. In the earlier example above, we specified a while loop as follows:

```
while (getline <"header.txt" > 0)
```

As we can see, `getline` is reading from the file `header.txt`, but what does the “`> 0`” mean? The answer is that, like most functions, `getline` returns a value. A positive value means success, zero means EOF (end of file), and a negative value means some other file-related problem, such as file not found has occurred. If we did not check the return value, we might end up with an infinite loop.

getline var <file

Reads the next record from *file* and assigns its contents to the variable *var*. Only *var* is set.

command | getline

Reads the next record from the output of *command*. $\$0$ and NF are set. Here is an example where we use AWK to parse the output of the `date` command:

```
awk 'BEGIN { "date" | getline print $4 }'
```

command | getline var

Reads the next record from the output of *command* and assigns its contents to the variable *var*. Only *var* is set.

String Functions

As one would expect, AWK has many functions used to manipulate strings and what's more, many of them support regular expressions. This makes AWK's string handling very powerful.

gsub(*r, s, t*)

Globally replaces any substring matching regular expression *r* contained within the target string *t* with the string *s*. The target string is optional. If omitted, $\$0$ is used as the target string. The function returns the number of substitutions made.

index(*s1, s2*)

Returns the leftmost position of string *s2* within string *s1*. If *s2* does not appear within *s1*, the function returns 0.

length(*s*)

Returns the number of characters in string *s*.

match(*s, r*)

Returns the leftmost position of a substring matching regular expression *r* within string *s*. Returns 0 if no match is found. This function also sets the internal variables `RSTART` and `RLENGTH`.

split(s, a, fs)

Splits string *s* into fields and stores each field in an element of array *a*. Fields are split according to field separator *fs*. For example, if we wanted to break a phone number such as 800-555-1212 into 3 fields separated by the “-” character, we could do this:

```
phone="800-555-1212"
split(phone, fields, "-")
```

After doing so, the array *fields* will contain the following elements:

```
fields[1] = "800"
fields[2] = "555"
fields[3] = "1212"
```

sprintf(fmt, exprs)

This function behaves like `printf`, except instead of outputting a formatted string, it returns a formatted string containing the list of expressions to the caller. Use this function to assign a formatted string to a variable:

```
area_code = "800"
exchange = "555"
number = "1212"
phone_number = sprintf("(%s) %s-%s", area_code, exchange, number)
```

sub(r, s, t)

Behaves like `gsub`, except only the first leftmost replacement is made. Like `gsub`, the target string *t* is optional. If omitted, `$0` is used as the target string.

substr(s, p, l)

Returns the substring contained within string *s* starting at position *p* with length *l*.

Arithmetic Functions

AWK has the usual set of arithmetic functions. A word of caution about math in AWK: it has limitations in terms of both number size and precision of floating point operations. This is particularly true of `mawk`. For tasks involving extensive calculation, `gawk` would be preferred. The `gawk` documentation provides a good discussion of the issues involved.

atan2(y, x)

Returns the arctangent of *y/x* in radians.

cos(x)

Returns the cosine of *x*, with *x* in radians.

exp(x)

Returns the exponential of x , that is e^x .

int(x)

Returns the integer portion of x . For example if $x = 1.9$, 1 is returned.

log(x)

Returns the natural logarithm of x . x must be positive.

rand()

Returns a random floating point value n such that $0 \leq n < 1$. This is a value between 0 and 1 where a value of 0 is possible but not 1. In AWK, random numbers always follow the same sequence of values unless the seed for the random number generator is first set using the `srand()` function (see below).

sin(x)

Returns the sine of x , with x in radians.

sqrt(x)

Returns the square root of x .

srand(x)

Sets the seed for the random number generator to x . If x is omitted, then the time of day is used as the seed. To generate a random integer in the range of 1 to n , we can use code like this:

```
srand()
# Generate a random integer between 1 and 6 inclusive
dice_roll = int(6 * rand()) + 1
```

User Defined Functions

In addition to the built-in string and arithmetic functions, AWK supports user-defined functions much like the shell. The mechanism for passing parameters is different, and more like traditional languages such as C.

Defining a function

A typical function definition looks like this:

```
function name(parameter-list) {
    statements
```

```
        return expression  
    }
```

We use the keyword `function` followed by the name of the function to be defined. The name must be immediately followed by the opening left parenthesis of the parameter list. The parameter list may contain zero or more comma-separated parameters. A brace delimited code block follows with one or more statements. To specify what is returned by the function, the `return` statement is used, followed by an expression containing the value to be returned. If we were to convert our previous dice rolling example into a function, it would look like this:

```
function dice_roll() {  
    return int(6 * rand()) + 1  
}
```

Further, if we wanted to generalize our function to support different possible maximum values, we could code this:

```
function rand_integer(max) {  
    return int(max * rand()) + 1  
}
```

and then change `dice_roll` to make use of our generalized function:

```
function dice_roll() {  
    return rand_integer(6)  
}
```

Passing Parameters to Functions

As we saw in the example above, we pass parameters to the function, and they are operated upon within the body of the function. Parameters fall into two general classes. First, there are the *scalar variables*, such as strings and numbers. Second are the arrays. This distinction is important in AWK because of the way that parameters are passed to functions. Scalar variables are *passed by value*, meaning that a copy of the variable is created and given to the function. This means that scalar variables act as local variables within the function and are destroyed once the function exits. Array variables, on the other hand, are *passed by reference* meaning that a pointer to the array's starting position in memory is passed to the function. This means that the array is not treated as a local variable and that any change made to the array persists once the program exits the function. This concept of passed by value versus passed by reference shows up in a lot of programming languages so it's important to understand it.

Local Variables

One interesting limitation of AWK is that we cannot declare local variables within the body of a function. There is a workaround for this problem. We can add variables to the parameter list. Since all scalar variables in the parameter list are passed by value, they will be treated as if they are local variables. This does not apply to arrays, since they are always passed by reference. Unlike many other languages, AWK does not enforce the parameter list, thus we can add parameters that are not used by the caller of the function.

In most other languages, the number and type of parameters passed during a function call must match the parameter list specified by the function's declaration.

By convention, additional parameters used as local variables in the function are preceded by additional spaces in the parameter list like so:

```
function my_funct(param1, param2, param3,      local1, local2)
```

These additional spaces have no meaning to the language, they are there for the benefit of the human reading the code.

Let's try some short AWK programs on some numbers. First we need some data. Here's a little AWK program that produces a table of random integers:

```
# random_table.awk - generate table of random numbers

function rand_integer(max) {
    return int(max * rand()) + 1
}

BEGIN {
    srand()
    for (i = 0; i < 100; i++) {
        for (j = 0; j < 5; j++) {
            printf("      %5d", rand_integer(99999))
        }
        printf("\n", "")
    }
}
```

If we store this in a file, we can run it like so:

```
me@linuxbox ~ $ awk -f random_table.awk > random_table.dat
```

And it should produce a file containing 100 rows of 5 columns of random integers.

Convert a File Into CSV Format

One of AWK's many strengths is file format conversion. Here we will convert our neatly arranged columns of numbers into a CSV (comma separated values) file.

```
awk 'BEGIN {OFS=","} {print $1,$2,$3,$4,$5}' random_table.dat
```

This is a very easy conversion. All we need to do is change the output field separator (`OFS`) and then print all of the individual fields. While it is very easy to write a CSV file, reading one can be tricky. In some cases, applications that write CSV files (including many popular spreadsheet programs) will create lines like this:

```
word1, "word2a, word2b", word3
```

Notice the embedded comma in the second field. This throws the simple AWK solution (`FS=","`) out the window. Parsing this kind of file can be done (`gawk`, in fact has a

language extension for this problem), but it's not pretty. It is best to avoid trying to read this type of file.

Convert a File Into TSV Format

A frequently available alternative to the CSV file is the TSV (tab separated value) file. This file format uses tab characters as the field separators:

```
awk 'BEGIN {OFS="\t"} {print $1,$2,$3,$4,$5}' random_table.dat
```

Again, writing these files is easy to do. We just set the output field separator to a tab character. In regards to reading, most applications that write CSV files can also write TSV files. Using TSV files avoids the embedded comma problem we often see when attempting to read CSV files.

Print the Total for Each Row

If all we need to do is some simple addition, this is easily done:

```
awk '
{
    t = $1 + $2 + $3 + $4 + $5
    printf("%s = %6d\n", $0, t)
}
' random_table.dat
```

Print the Total for Each Column

Adding up the column is pretty easy, too. In this example, we use a loop and array to maintain running totals for each of the five columns in our data file:

```
awk '
{
    for (i = 1; i <= 5; i++) {
        t[i] += $i
    }
    print
}
END {
    print " ==="
    for (i = 1; i <= 5; i++) {
        printf(" %7d", t[i])
    }
    printf("\n", "")
}
' random_table.dat
```

Print the Minimum and Maximum Value in Column 1

```
awk '
BEGIN {min = 99999}
$1 > max {max = $1}
$1 < min {min = $1}
END {print "Max =", max, "Min =", min}
```

```
' random_table.dat
```

One Last Example

For our last example, we'll create a program that processes a list of pathnames and extracts the extension from each file name to keep a tally of how many files have that extension:

```
# file_types.awk - sorted list of file name extensions and counts

BEGIN {FS = "."
types[$NF]++}

END {
    for (i in types) {
        printf("%6d %s\n", types[i], i) | "sort -nr"
    }
}
```

To find the 10 most popular file extensions in our home directory, we can use the program like this:

```
find ~ -name "*.*" | awk -f file_types.awk | head
```

Summing Up

We really have to admire what an elegant and useful tool the authors of AWK created during the early days of Unix. So useful that its utility continues to this day. We have given AWK a brief examination in this adventure. Feel free to explore further by delving deeper into the documentation of the various AWK implementations. Also, searching the web for “AWK one-liners” will reveal many useful and clever tricks possible with AWK.

Further Reading

- The `nawk` man page provides a good reference for the baseline version of AWK. An online version is available at <https://linux.die.net/man/1/nawk>
- Many useful AWK programs are just one line long. Eric Pement has compiled an extensive list: <http://www.pement.org/awk/awk1line.txt>
- In addition to its man page, `gawk` has its own book titled *Gawk: Effective AWK Programming* available at: <https://www.gnu.org/software/gawk/manual/>
- Peteris Krumins has a nice blog post listing a variety of helpful tips for AWK users: <https://catonmat.net/ten-awk-tips-tricks-and-pitfalls>

8 Power Terminals

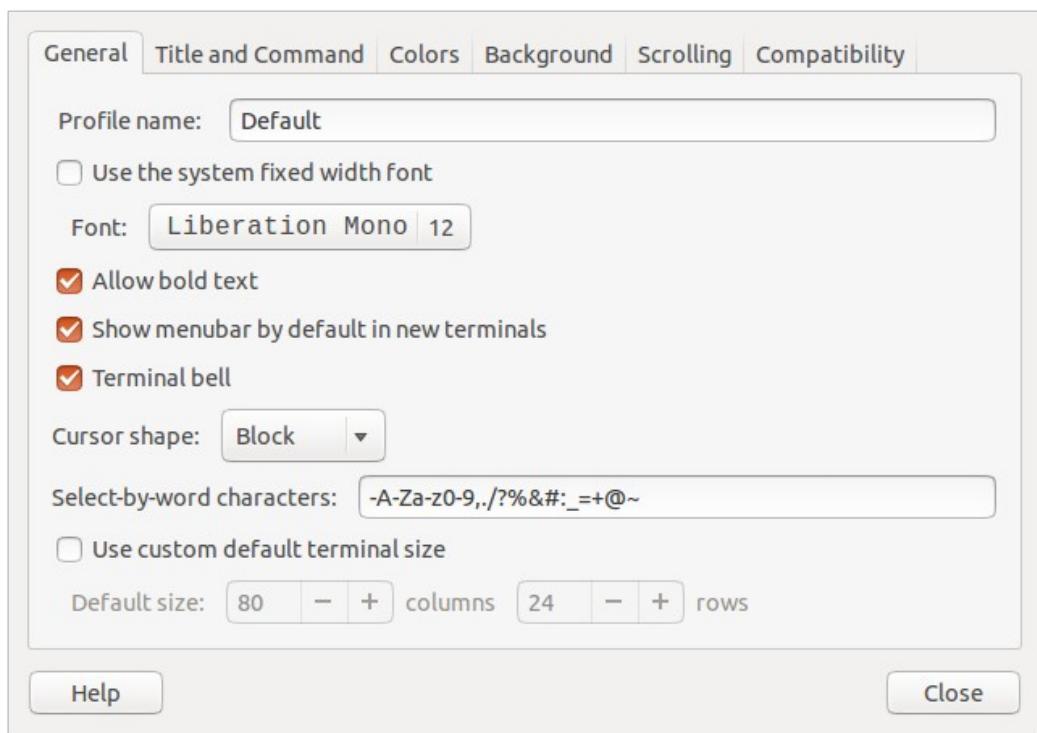
Over the course of our many lessons and adventures, we have learned a lot about the shell, and explored many of the common command line utilities found on Linux systems. There is, however, one program we have overlooked, and it may be among the most important and most frequently used of them all—our terminal emulator.

In this adventure, we are going to dig into these essential tools and look at a few of the different terminal programs and the many interesting things we can do with them.

A Typical Modern Terminal

Graphical desktop environments like GNOME, KDE, LXDE, XFCE, etc. all include terminal emulators as standard equipment. We can think of this as a safety feature because, if the desktop environment suffers from some lack of functionality (and they all do), we can still access the shell and actually get stuff done.

Modern terminal emulators are quite flexible and can be configured in many ways:



gnome-terminal preferences dialog

Size

Terminal emulators display a window that can be adjusted to any size from the sublime to the ridiculous. Many terminals allow configuration of a default size.

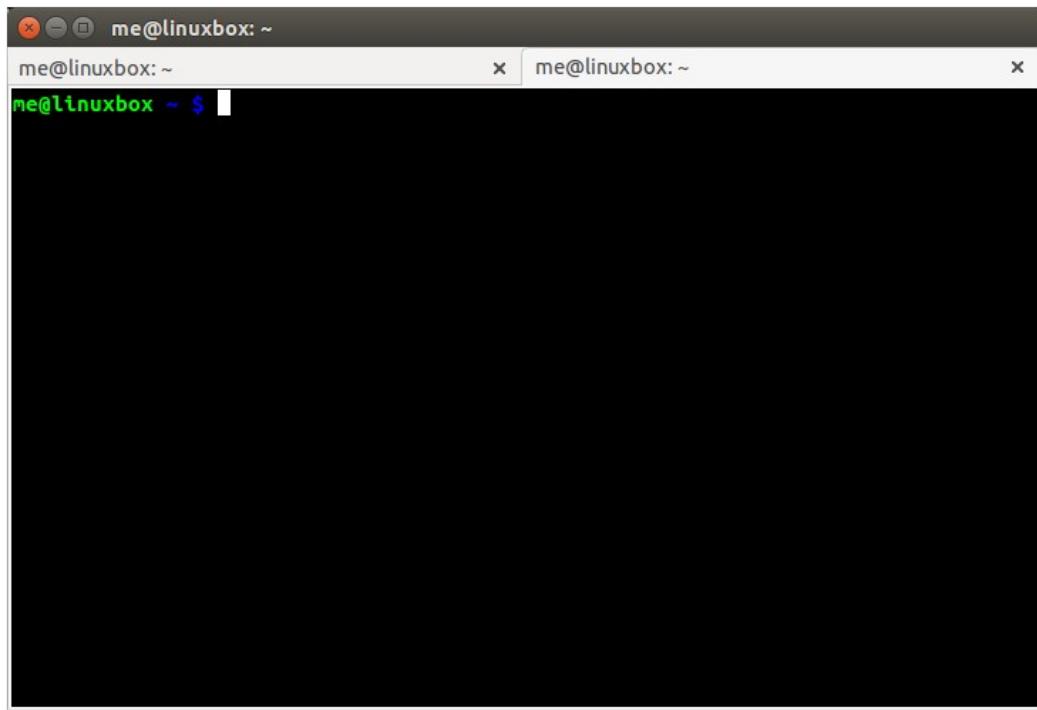
The “normal” size for a terminal is 80 columns by 24 rows. These dimensions were inherited from the size of common hardware terminals, which, in turn, were influenced by the format of IBM punch cards (80 columns by 12 rows). Some applications expect 80 by 24 to be the minimum size, and will not display properly when the size is smaller. Making the terminal larger, on the other hand, is preferable in most situations, particularly when it comes to terminal height. 80 columns is a good width for reading text, but having additional height provides us with more context when working at the command line.

Another common width is 132 columns, derived from the width of wide fan-fold computer paper. Though this is too wide for comfortable reading of straight text (for example, a man page), it’s fine for other purposes, such as viewing log files.

The 80-column default width has implications for the shell scripts and other text-based programs we write. We should format our printed output to fit within the limits of an 80-character line for best effect.

Tabs

A single terminal window with the ability to contain several different shell sessions is a valuable feature found in most modern terminal emulators. This is accomplished through the use of *tabs*.



gnome-terminal with tabs

Tabs are a fairly recent addition to terminal emulators, first appearing around 2003 in both GNOME's `gnome-terminal` and KDE's `konsole`.

Profiles

Another feature found in some modern terminals is multiple configuration profiles. With this feature, we can have separate configurations for different tasks. For example, if we are responsible for maintaining a remote server, we might have a separate profile for the terminal that we use to manage it.

Fonts, Colors, and Backgrounds

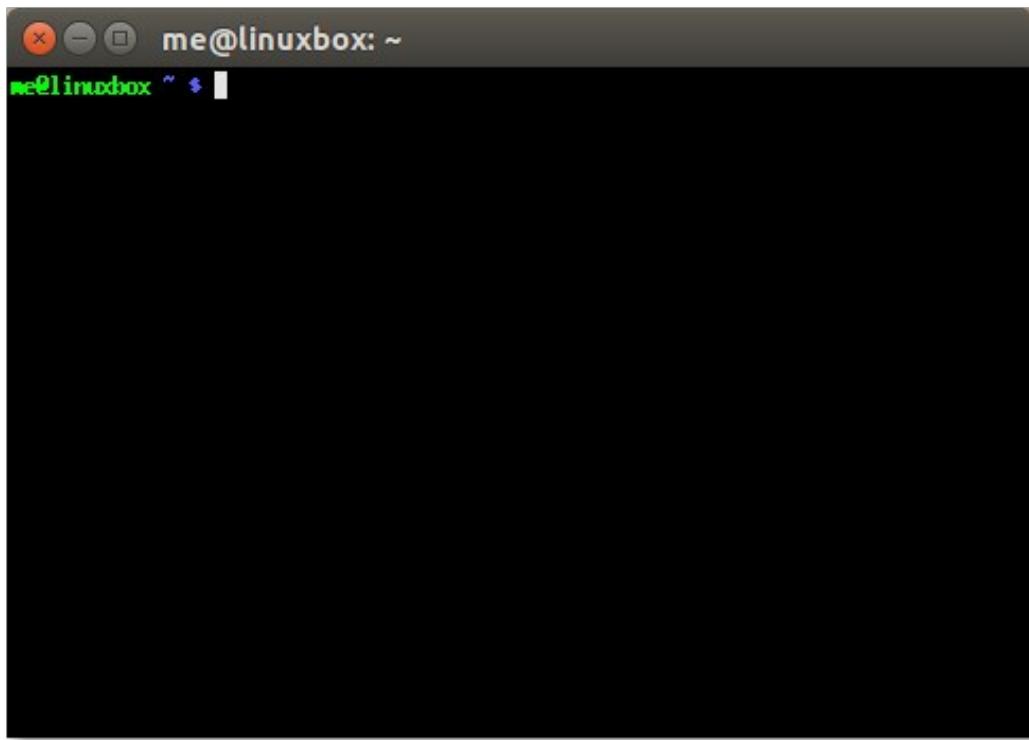
Most terminal emulators allow us to select fonts, colors, and backgrounds for our terminal sessions. The three most important criteria for selecting fonts, colors, and backgrounds are: 1. legibility, 2. legibility, and 3. legibility. Many people post screen shots of their Linux desktops online, and there is a great fascination with "stylish" fonts, faint colors, and pseudo-transparent terminal windows, but we use our terminals for very serious things, so we should treat our terminals very seriously, too. No one wants to make a mistake while administering a system because they misread something on the screen. Choose wisely.

Past Favorites

When the first graphical environments began appearing for Unix in the mid-1980s, terminal emulators were among the first applications that were developed. After all, the GUIs of the time had very little functionality and people still needed to do their work. Besides, the graphical desktop allowed users to display multiple terminal windows- a powerful advantage at the time.

xterm

The granddaddy of all graphical terminals is `xterm`, the standard terminal emulator for the X Window System. Originally released in 1984, it's still under active maintenance. Since it is a standard part of X, it is included in many Linux distributions. `xterm` was very influential, and most modern terminal programs emulate its behavior in one way or another.

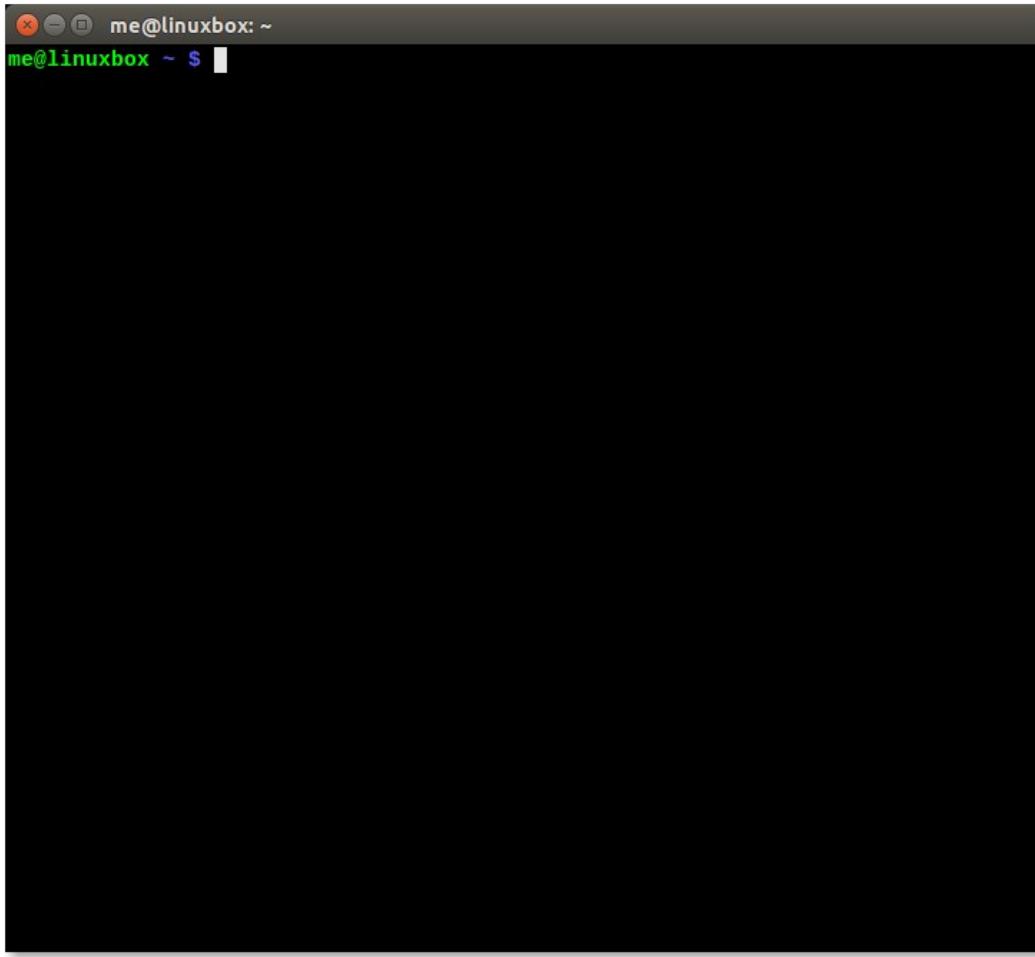


xterm with default configuration

In its default configuration, `xterm` looks rather small and pathetic, but almost everything about `xterm` is configurable. When we say “configurable,” we don’t mean there is a pretty “Preferences” dialog. This is Unix! Like many early X applications, it relies on an `Xresources` file for its configuration. This file can be either global (`/etc/X11/Xresources`) or local to the user (`~/.Xresources`). Each item in this file consists of an application class and a setting. If we create the file `~/.Xresources` with the following content:

```
XTerm.vt100.geometry: 80x35  
XTerm.vt100.faceName: Liberation Mono:size=11  
XTerm.vt100.cursorBlink: true
```

then we get a terminal like this:



Configured xterm

A complete list of the Xresources configuration values for `xterm` appears in its man page.

While `xterm` does not appear to have menus, it actually has 3 different ones, which are made visible by holding the `Ctrl` key and pressing a mouse button. Different menus appear according to which button is pressed. The scroll bar on the side of the terminal has a behavior like ancient X applications. Hint: after enabling the scroll bar with the menu, use the middle mouse button to drag the slider.

Though `xterm` offers neither tabs nor profiles, it does have one strange extra feature: it can display a Tektronix 4014 graphics terminal emulator window. The Tektronix 4014 was an early and very expensive storage tube graphics display that was popular with computer aided design systems in the 1970s. It's extremely obscure today. The normal `xterm` text window is called the VT window. The name comes from the DEC VT220, a popular computer terminal of the same period. `xterm`, and most terminals today, emulate this terminal to a certain extent. `xterm` is not quite the same as the VT terminal, and it has its own specific `terminfo` entry (see the `tput` adventure for some background on

`terminfo`). Terminals set an environment variable named `TERM` that is used by X and `terminfo` to identify the terminal type, and thus send it the correct control codes. To see the current value of the `TERM` variable, we can do this:

```
me@linuxbox ~ $ echo $TERM
```

Even if we are using a modern terminal, such as `gnome-terminal`, we will notice that the `TERM` variable is often set to “`xterm`” or “`xterm-color`”. That’s how much influence `xterm` had. We still use it as the standard.

rxvt

By the standards of the time, `xterm` was a heavyweight program but, as time went by, some of its features were rarely used such as the Tektronix emulation. Around 1990, in an attempt to create a simpler, lighter terminal emulator, Robert Nation wrote `rxvt` as part of the FVWM window manager, an early desktop environment for Unix-like systems.

`rxvt` has a smaller feature set than `xterm` and emulates the DEC VT102 terminal rather than the more advanced VT220. `rxvt` sets the `TERM` variable to “`rxvt`”, which is widely supported. Like `xterm`, `rxvt` has menus that are displayed by holding the `Ctrl` key and pressing different mouse buttons.

`rxvt` is still under active maintenance, and there is a popular modern implementation forked from the original called `urxvt()` by Mark Lehmann, which supports Unicode (multi-byte characters used to express a wider range of written languages than ASCII). One interesting feature in `urxvt` is a daemon mode that allows launching multiple terminal windows all sharing the same instance of the program- a potential memory saver.



urxvt with default configuration

Like `xterm`, `rxvt` uses Xresources to control its configuration. The default `rxvt` configuration is very spare. Adding the following settings to our Xresources file will make it more palatable (`urxvt` shown):

```
URxvt.geometry: 80x35
URxvt.saveLines: 10000
URxvt.scrollBar: false
URxvt.foreground: white
URxvt.background: black
URxvt.secondaryScroll: true
URxvt.font: xft:liberation mono:size=11
URxvt.cursorBlink: true
```

Modern Power Terminals

Most modern graphical desktop environments include a terminal emulator program. Some are more feature-rich than others. Let's look at some of the most powerful and popular ones.

gnome-terminal

The default terminal application for GNOME and its derivatives is `gnome-terminal`. Possibly the world's most popular terminal app, it's a good, full-featured program. It has many features we expect in modern terminals, like multiple tabs and profile support. It also allows many kinds of customization.

Tabs

Busy terminal users will often find themselves working in multiple terminal sessions at once. It may be to perform operations on several machines at the same time, or to manage a complex set of tasks on a single system. This problem can be addressed either by opening multiple terminal windows, or by having multiple tabs in a single window.

The File menu in `gnome-terminal` offers both choices (well, in older versions anyway). In newer versions, use the keyboard shortcut `Ctrl-Shift-T` to open a tab. Tabs can be rearranged with the mouse, or can be dragged out of the window to create a new window. With `gnome-terminal`, we can even drag a tab from one terminal window to another.

Keyboard Shortcuts

Since, in an ideal universe, we never lift our fingers from the keyboard, we need ways of controlling our terminal without resorting to a mouse. Fortunately, `gnome-terminal` offers a large set of keyboard shortcuts for common operations. Here are some of the most useful ones, defined by default:

Shortcut	Action
<code>Ctrl-Shift-N</code>	New Window
<code>Ctrl-Shift-W</code>	Close Window
<code>F11</code>	View terminal full screen
<code>Shift-PgUp</code>	Scroll up
<code>Shift-PgDn</code>	Scroll down
<code>Shift-Home</code>	Scroll to the beginning
<code>Shift-End</code>	Scroll to the end
<code>Ctrl-Shift-T</code>	New Tab
<code>Ctrl-Shift-Q</code>	Close Tab
<code>Ctrl-PgUp</code>	Next Tab
<code>Ctrl-PgDn</code>	Previous Tab
<code>Alt-n</code>	Where n is a number in the range of 1 to 9, go to tab n

Gnome-terminal keyboard shortcuts

Keyboard shortcuts are also user configurable.

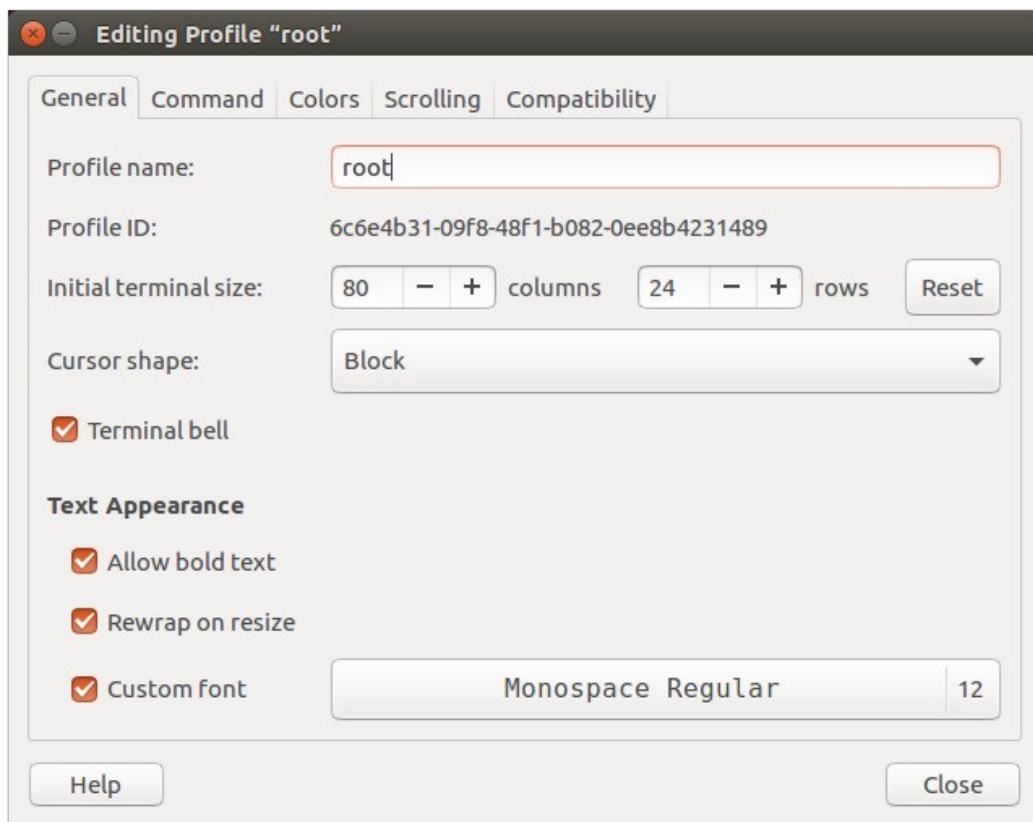
While it is well known that `Ctrl-c` and `Ctrl-v` cannot be used in the terminal window to perform copy and paste, `Ctrl-Shift-C` and `Ctrl-Shift-V` will work in their place with `gnome-terminal`.

Profiles

Profiles are one of the great, unsung features of many terminal programs. This may be because their advantages are perhaps not intuitively obvious. Profiles are particularly useful when we want to visually distinguish one terminal session from another. This is especially true when managing multiple machines. In this case, having a different

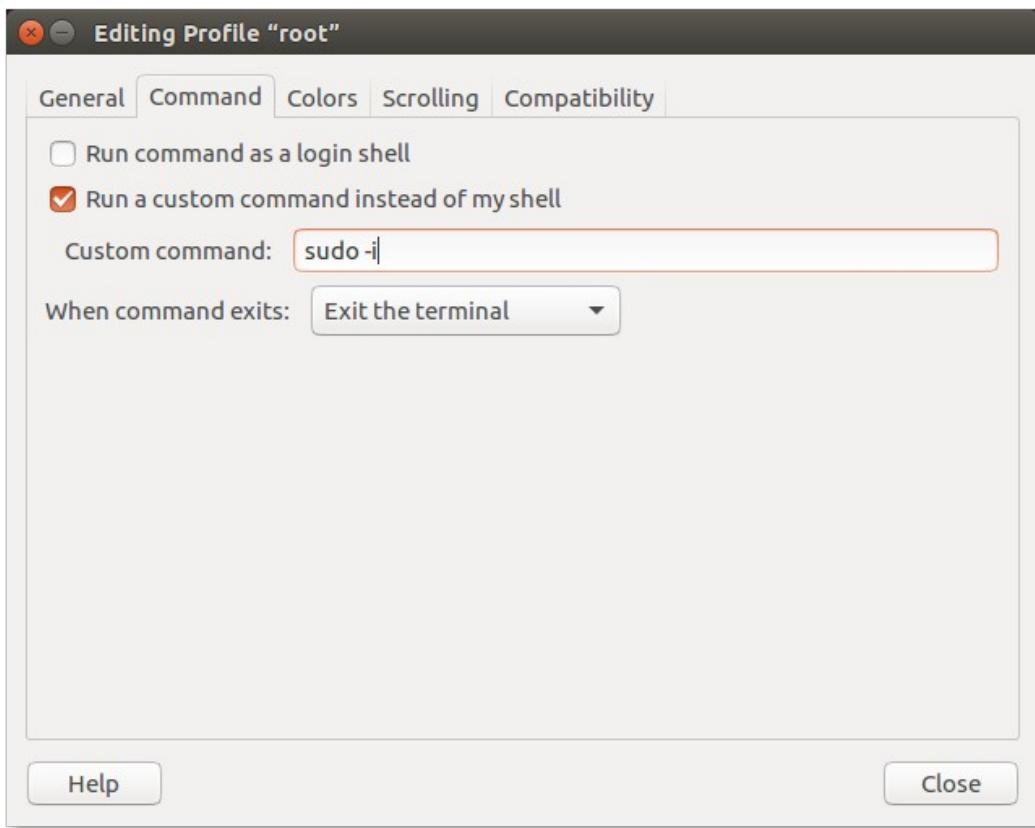
background color for the remote system's session may help us avoid typing a command into the wrong session. We can even incorporate a default command (like `ssh`) into a profile to facilitate the connection to the remote system.

Let's make a profile for a root shell. First, we'll go to the File menu and select "New Profile..." and when the dialog appears enter the name "root" as our new profile:



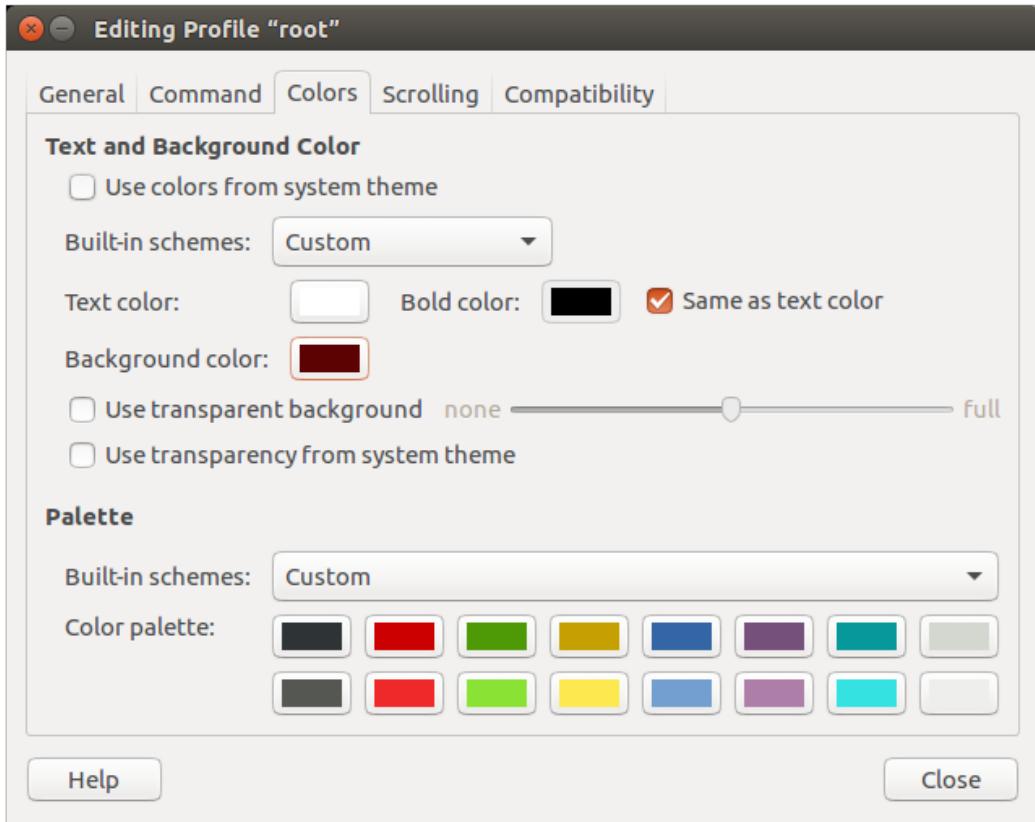
gnome-terminal new profile dialog

Next, we'll configure our new profile and choose the font and default size of the terminal window. Then we will choose a command for the terminal window when it is opened. To create a root shell, we can use the command `sudo -i`. We will also make sure to specify that the terminal should exit when the command exits.



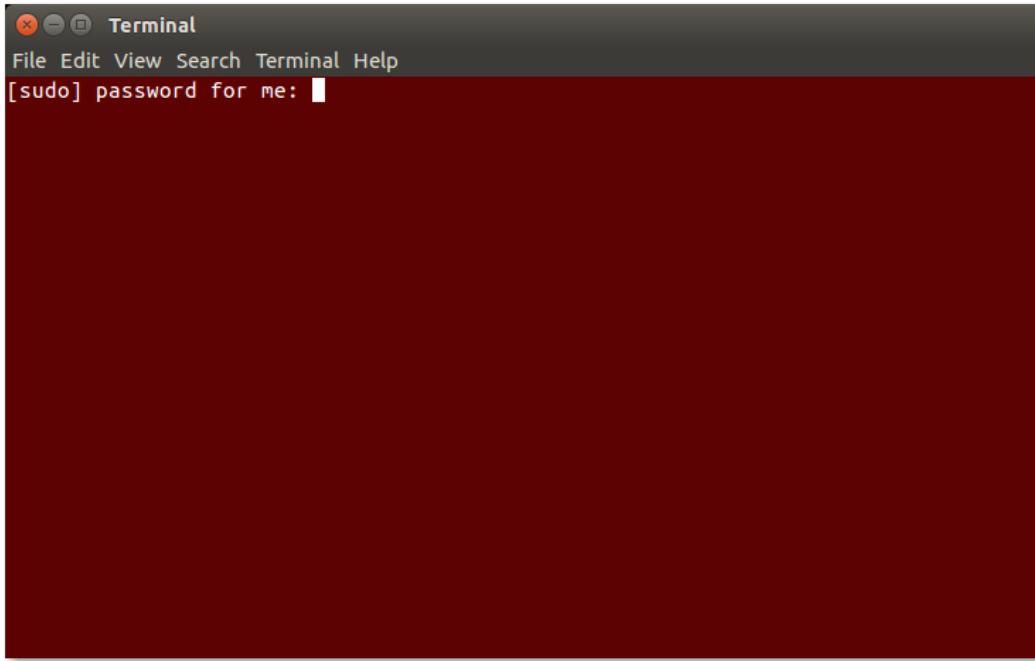
Setting the command in the configuration dialog

Finally, we'll select some colors. How about white text on a dark red background? That should convey an appropriate sense of gravity when we use a root shell.



Setting the colors in configuration dialog

Once we finish our configuration, we can test our shell:



Root profile gnome-terminal

We can configure terminal profiles for any command line program we want: Midnight Commander, tmux, whatever.

Here is another example. We will create a simple man page viewer. With this terminal profile, we can have a dedicated terminal window to only display man pages. To do this, we first need to write a short script to prompt the user for the name of which command to look up, and display the man page in a (nearly) endless loop:

```
#!/bin/bash

# man_view - simple man page viewer

while true; do
    echo -en "\nPlease enter a command name (q to quit) -> "
    read
    [[ "$REPLY" == "q" ]] && break
    [[ -n "$REPLY" ]] && { man $REPLY || sleep 3; }
    clear
done
```

We'll save this file in our `~/bin` directory and use it as our custom command for our terminal profile.

Next, we create a new terminal profile and name it “man page”. Since we are designing a window for man pages, we can play with the window size and color. We'll set the window tall and a little narrow (for easier reading) and set the colors to green text on a black background for that retro terminal feeling:

```
Terminal
File Edit View Search Terminal Help
LS(1) User Commands LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

    Mandatory arguments to long options are mandatory for short options too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        do not list implied . and ..

    --author
        with -l, print the author of each file

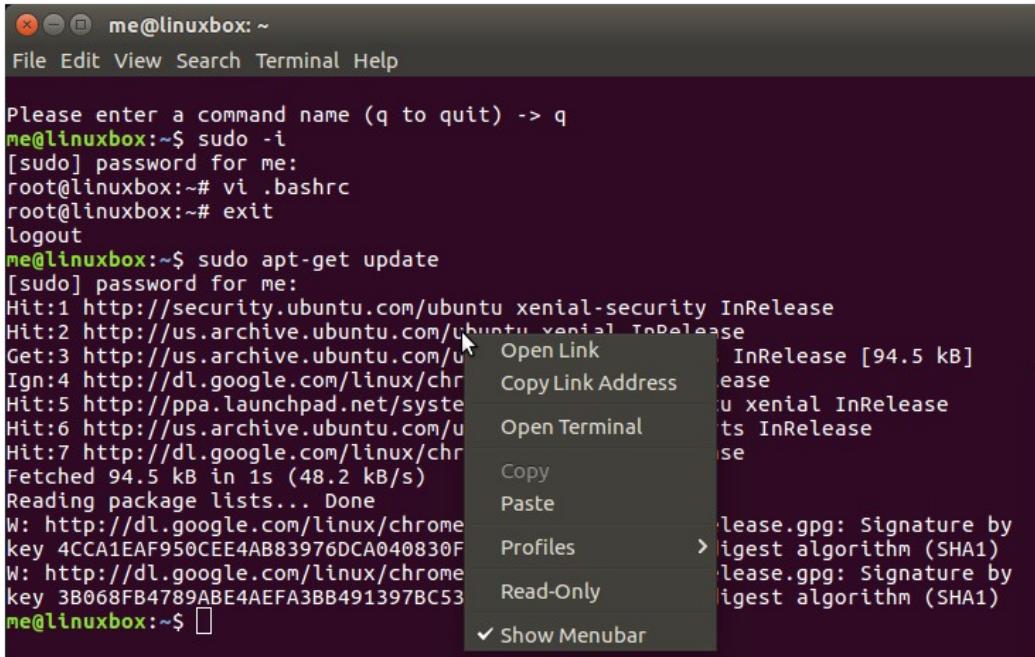
    -b, --escape
        print C-style escapes for nongraphic characters

    --block-size=SIZE
Manual page ls(1) line 1 (press h for help or q to quit)
```

Man page gnome-terminal window

Opening Hyperlinks and Email Addresses

One of the neat tricks `gnome-terminal` can do is copy and/or open URLs. When it detects a URL in the stream of displayed text, it displays it with an underline. Right-clicking on the link displays a menu of operations:



gnome-terminal URL context menu

Resetting the Terminal

Sometimes, despite our best efforts, we do something dumb at the terminal, like attempting to display a non-text file. When this happens, the terminal emulator will dutifully interpret the random bytes as control codes and we'll notice that the terminal screen fills with garbage and nothing works anymore. To escape this situation, we must reset the terminal. `gnome-terminal` provides a function for this located in its Terminal menu.

konsole

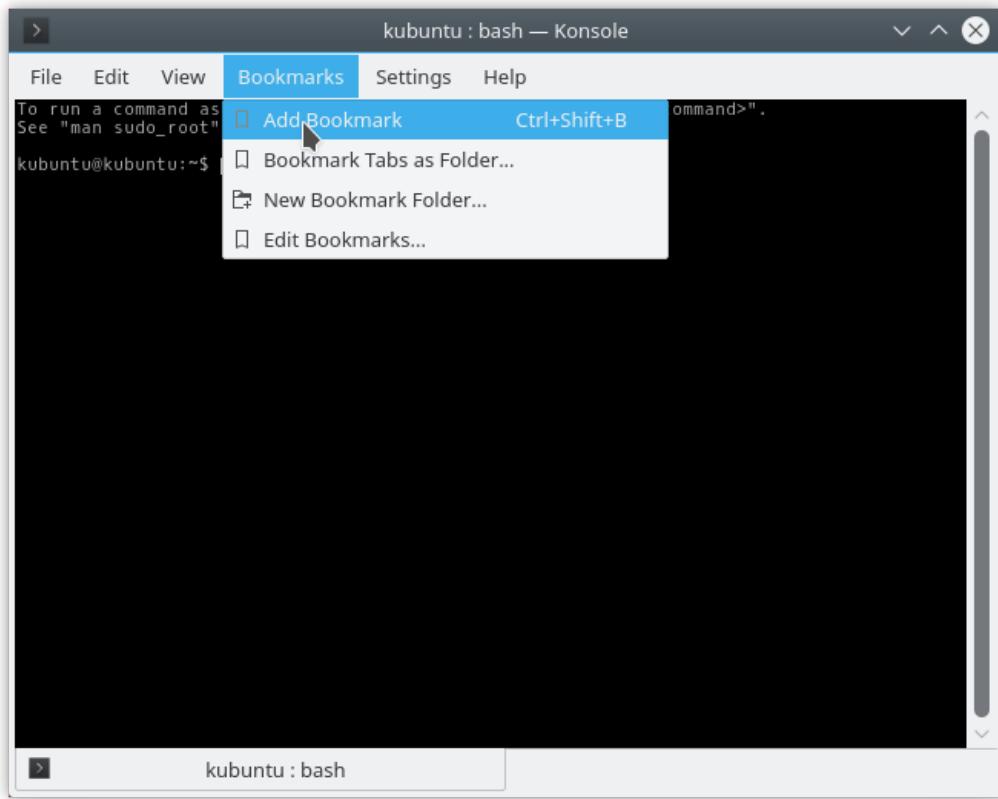
`konsole`, the default terminal application for the KDE desktop, has a feature set similar to that of `gnome-terminal`. This, of course, makes sense since `konsole` directly “competes” with `gnome-terminal`. For instance, both `gnome-terminal` and `konsole` support tabs and profiles in a similar fashion.

`konsole` does have a couple of unique features not found in `gnome-terminal`. `konsole` has bookmarks, and `konsole` can split the screen into regions allowing more than one view of the same terminal session to be displayed at the same time.

Bookmarks

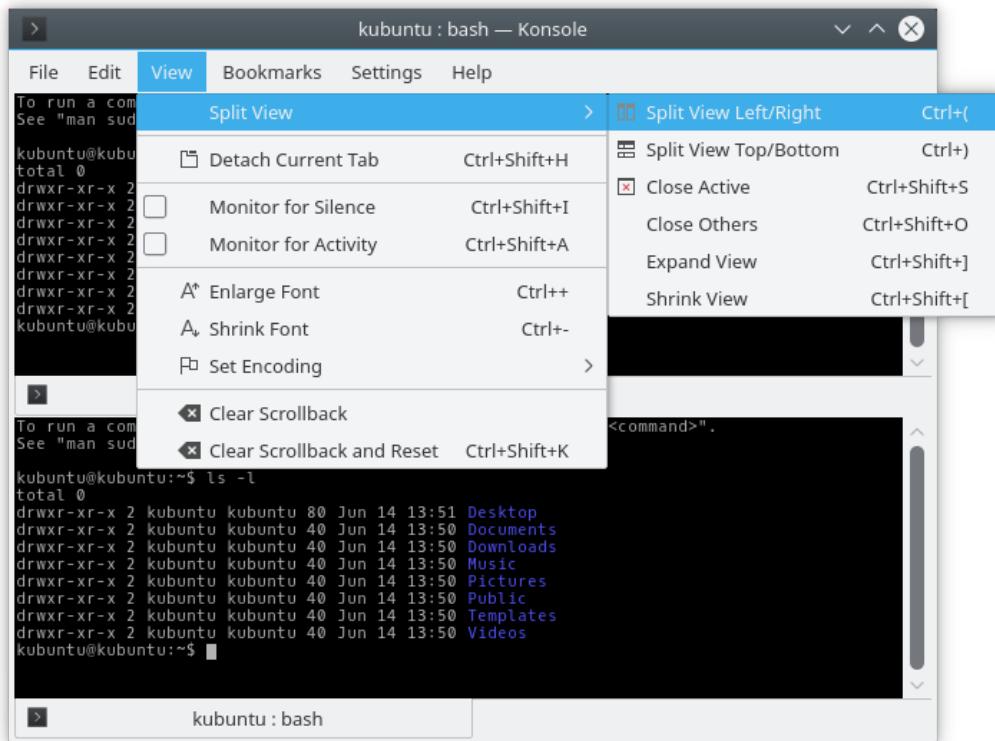
`konsole` allows us to store the location of directories as bookmarks. Locations may also include remote locations accessible via `ssh`. For example, we can define a bookmark

such as `ssh:me@remotehost`, and it will attempt to connect with the remote system when the bookmark is used.



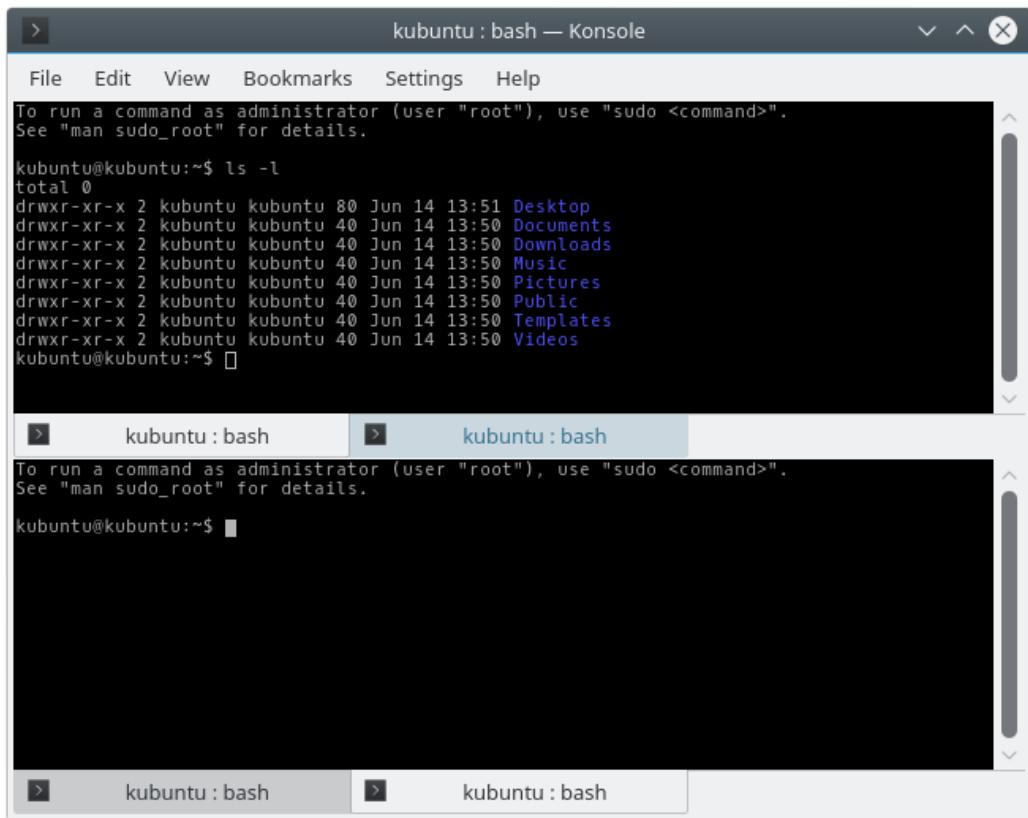
konsole bookmarks menu

Split View



konsole's split view feature

konsole's unique split view feature allows us to have two views of a single terminal session. This seems odd at first glance, but is useful when examining long streams of output. For example, if we needed to copy text from one portion of a long output stream to the command line at the bottom, this could be handy. Further, we can get views of different terminal sessions, by using tabs in conjunction with split views, since while the tabs will appear in all of the split views, they can be switched independently in each view:



konsole with tabs and split view

quake

`gnome-terminal` has spawned a couple of programs that reuse many of its internal parts to create different terminal applications. The first is `quake`, a terminal that borrows a design feature from a popular first-person shooter game. When running, `quake` normally hides in the background, but when the F12 key is pressed, the terminal window “rolls down” from the top of the screen to reveal itself. This can be handy if terminal use is intermittent, or if screen real estate is at a premium.

`quake` shares many of the configuration options with `gnome-terminal`, as well as the ability to configure what key activates it, which side of the screen it rolls from, and its size.

Though `quake` supports tabs, it does not (as of this writing) support profiles. However, we can approximate profiles with a little clever scripting:

```
#!/bin/bash

# gtab - create pseudo-profiles for quake

if [[ $1 == "" ]]; then
    quake --new-tab=. --show
```

```

    exit
fi

case $1 in
root) # Create a root shell tab
    guake --new-tab=. --fgcolor=\#ffffff --bgcolor=\#5e0000
    guake --show    # Switch to new fg/bg colors
    guake --rename-current-tab=root
    guake --execute-command='sudo -i; exit'
;;
man) # Create a manual page viewer tab
    guake --new-tab=. --fgcolor=\#00ef00 --bgcolor=\#000000
    guake --show    # Switch to new fg/bg colors
    guake --rename-current-tab="man viewer"
    guake --execute-command='man_view; exit'
;;
*)
    echo "No such tab. Try either 'root' or 'man'" >&2
    exit 1
;;
esac

```

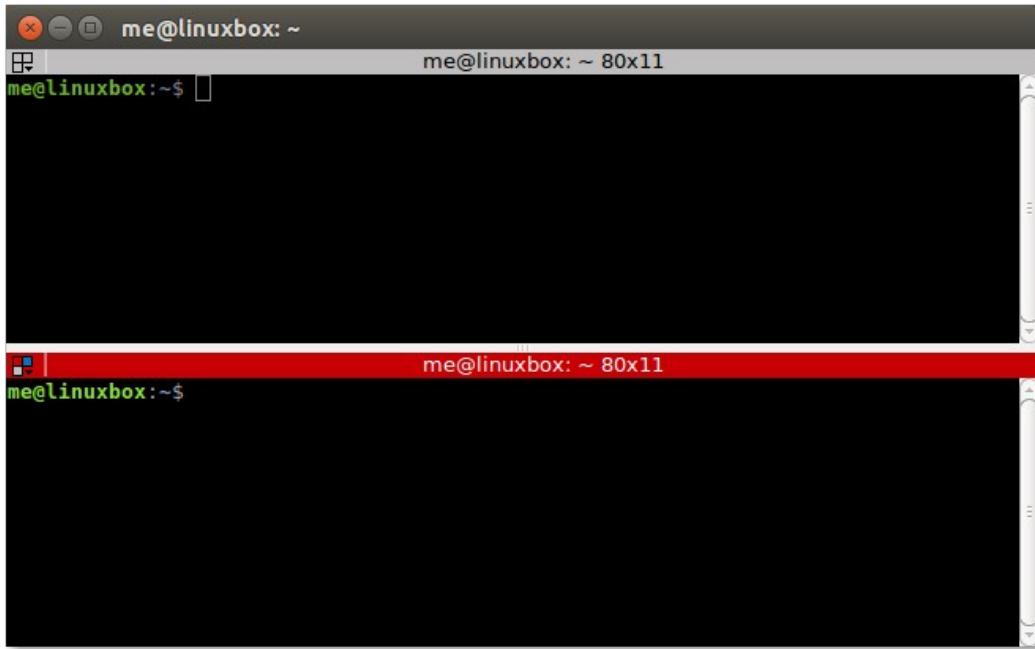
After saving this script, we can open new tabs in `guake` by entering the command `gtab` followed by an optional profile, either “root” or “man” to duplicate what we did with the `gnome-terminal` profiles above. Entering `gtab` without an option simply opens a new tab in the current working directory.

As we can see, `guake` has a number of interesting command line options that allow us to program its behavior.

For KDE users, there is a similar program called `yakuake`.

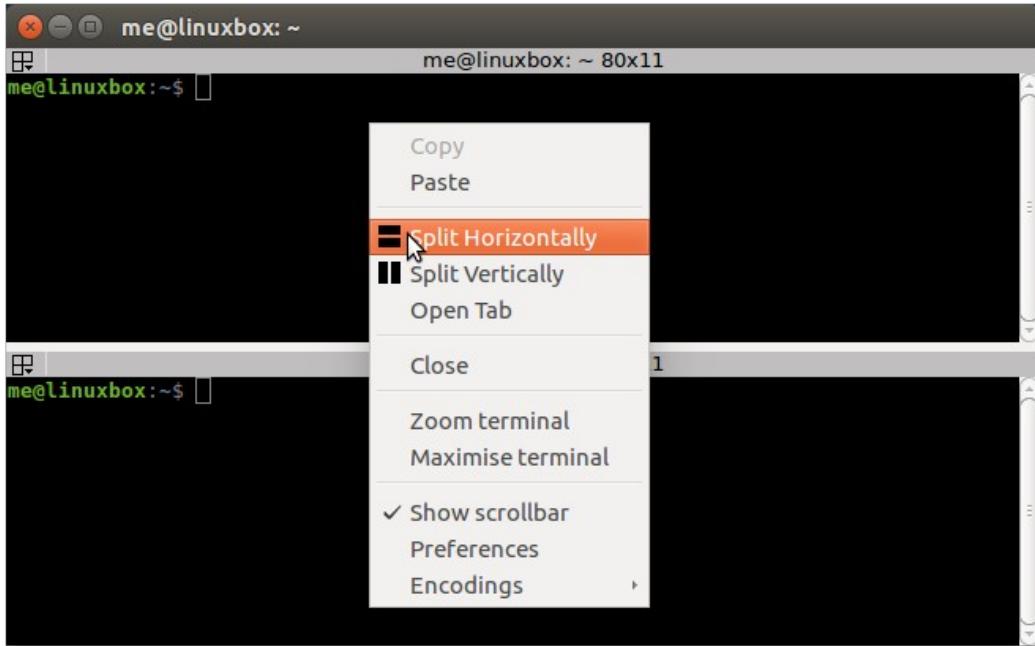
terminator

Like `guake`, `terminator` builds on the `gnome-terminal` code to create a very popular alternative terminal. The main feature addition is split window support.



terminator with split screens

By right-clicking in the `terminator` window, `terminator` displays its menu where we can see the options for splitting the current terminal either vertically or horizontally.

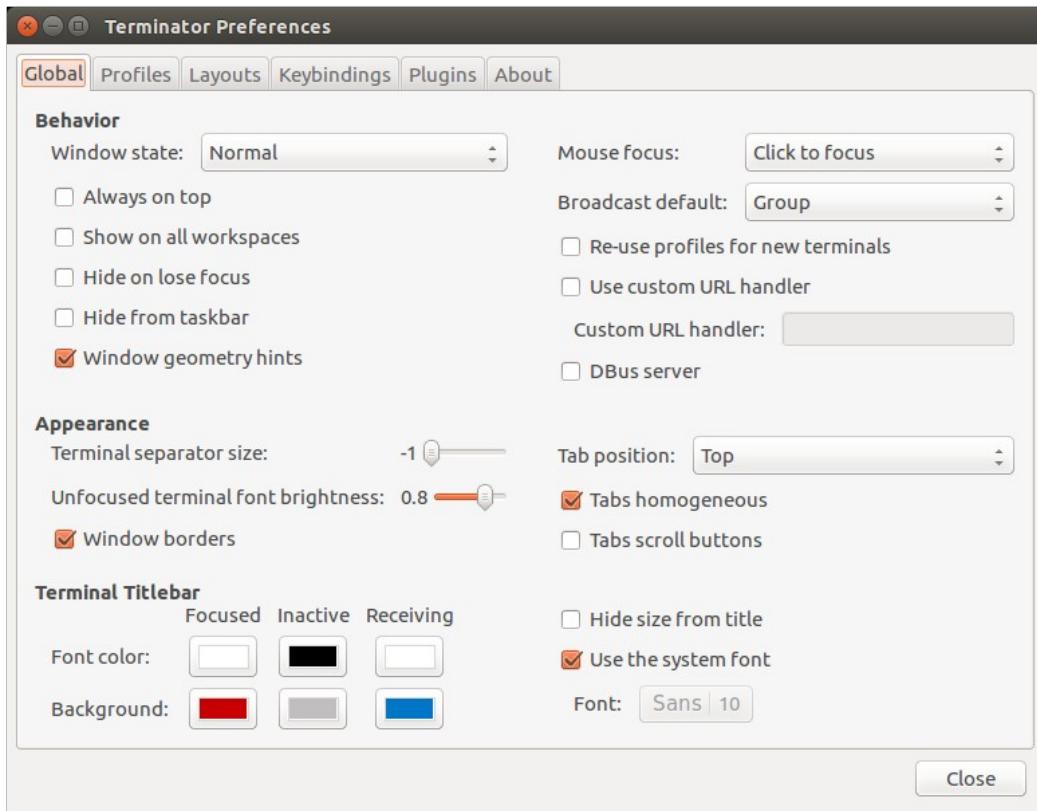


The terminator menu

Once split, each terminal pane can be dragged and dropped. Panes can also be resized with either the mouse or a keyboard shortcut. Another nice feature of `terminator` is the

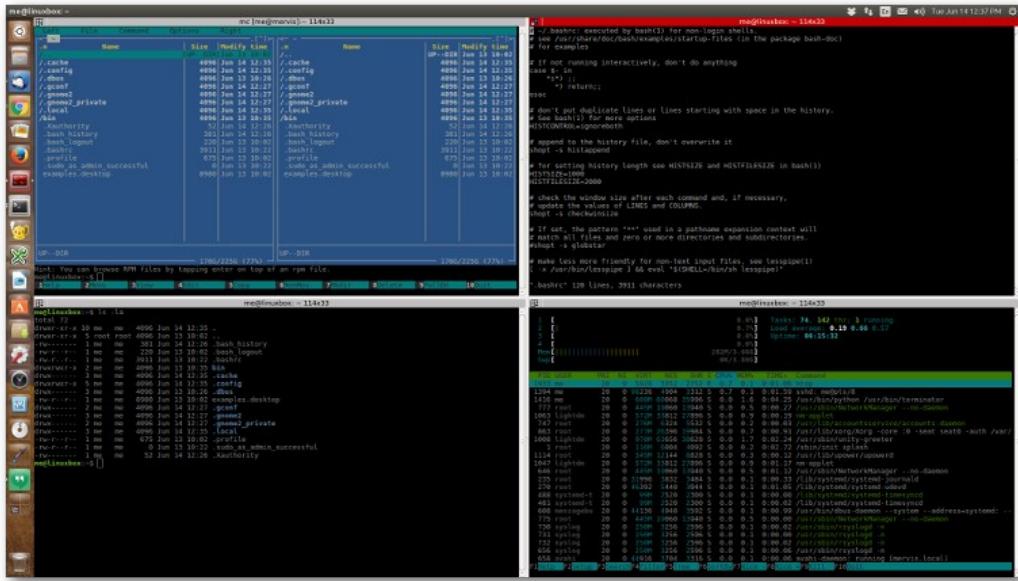
ability to set the focus policy to “focus follows mouse” so that we can change the active pane by simply hovering the mouse over the desired pane without have to perform an extra click to make the pane active.

The preferences dialog supports many of the same configuration features as that of `gnome-terminal`, including profiles with custom commands:



The terminator preferences dialog

A good way to use `terminator` is to expand its window to full screen and then split it into multiple panes:



Full screen terminator window with multiple panes

We can even automate this by going into Preferences/Layouts and storing our full screen layout (let's call it "2x2") then, by invoking terminator this way:

```
terminator --maximise --layout=2x2
```

to get our layout instantly.

Terminals for Other Platforms

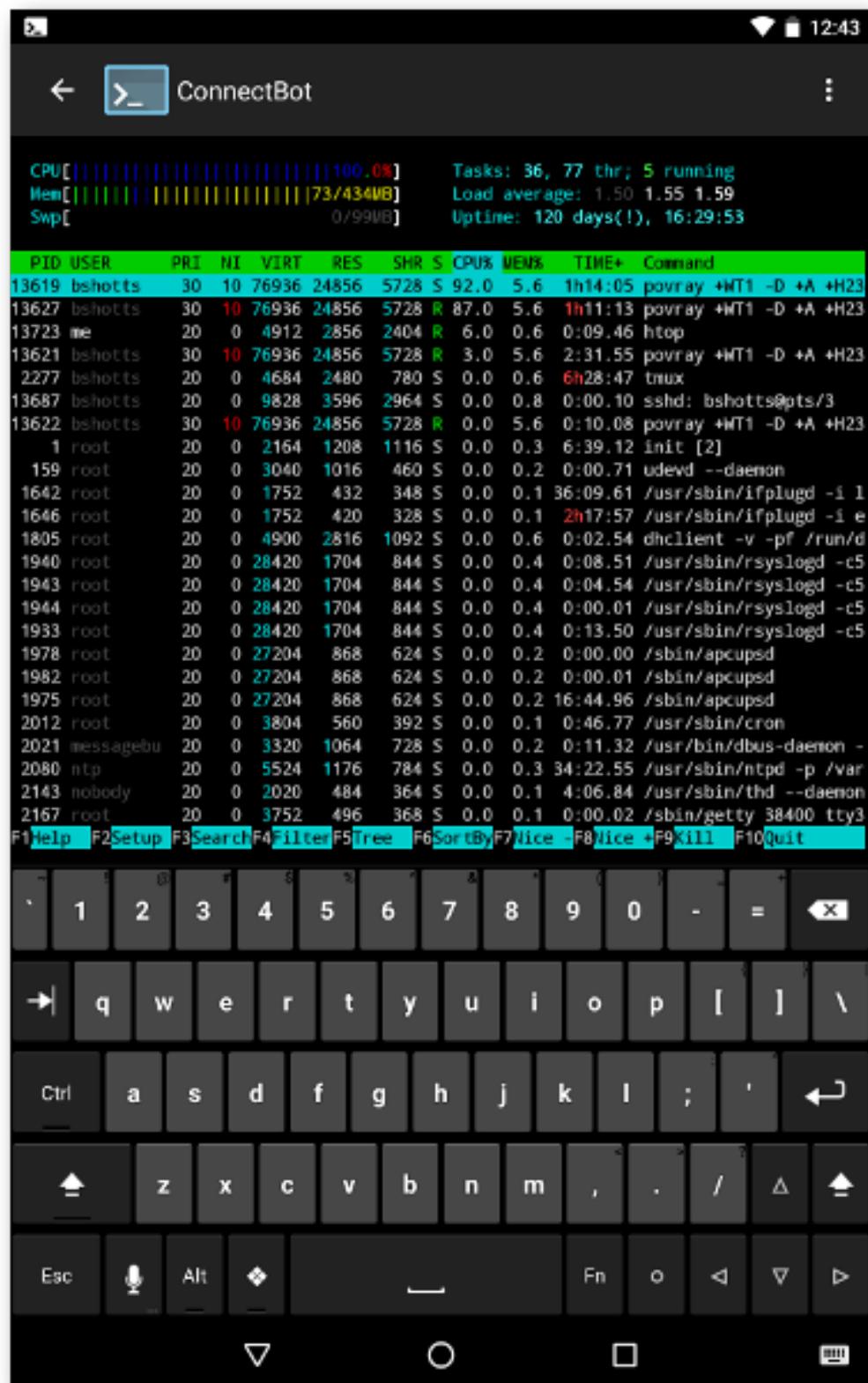
Android

While we might not think of an Android phone or tablet as a Linux computer, it actually is, and we can get terminal apps for it which are useful for administering remote systems.

Connectbot

Connectbot is a secure shell client for Android. With it, we can log into any system running an SSH server. To the remote system, Connectbot looks like a terminal using the GNU Screen terminal type.

One problem with using a terminal emulator on Android is the limitations of the native Google keyboard. It does not have all the keys required to make full use of a terminal session. Fortunately, there are alternate keyboards that we can use on Android. A really good one is Hacker's Keyboard by Klaus Weidner. It supports all the normal keys, **Ctrl**, **Alt**, **F1-F10**, arrows, **PgUp**, **PgDn**, etc. Very handy when working with **vi** on a phone.



Connectbot with Hacker's Keyboard on Android

Termux

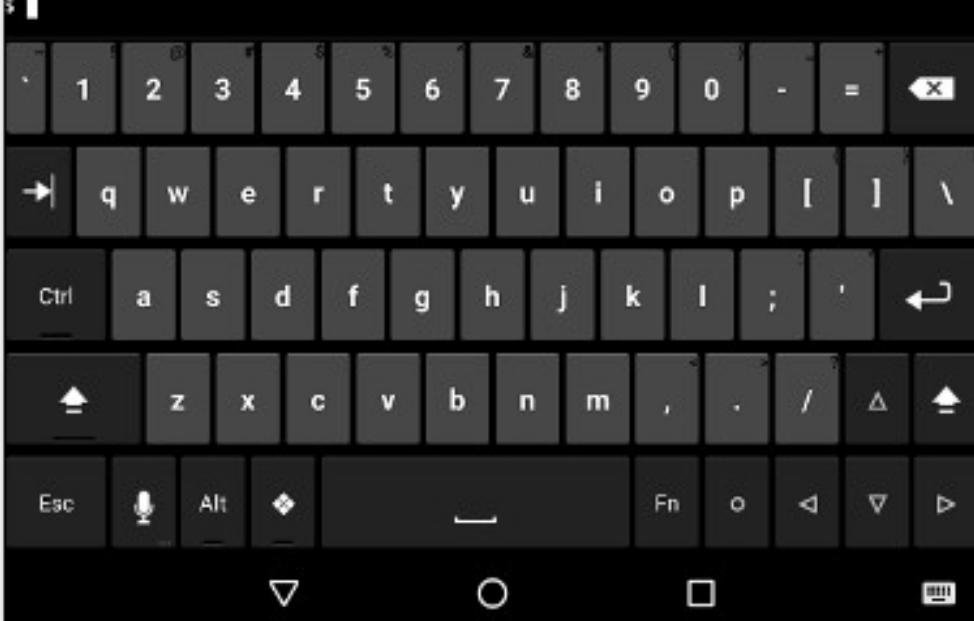
The Termux app for Android is unexpectedly amazing. It goes beyond being merely an SSH client; it provides a full shell environment on Android without having to root the device.

After installation, there is a minimal base system with a shell (`bash`) and many of the most common utilities. Initially, these utilities are the ones built into `busybox` (a compact set of utilities joined into a single program that is often used in embedded systems to save space), but the `apt` package management program (like on Debian/Ubuntu) is provided to allow installation of a wide variety of Linux programs.

```

job_spec (&)
(( expression ))
filename [arguments]
:
[ arg... ]
[[ expression ]]
alias [-p] [name[=value] ... ]
bg [job_spec ...]
bind [-lpsvPSVX] [-m keymap] [-f filename]
break [n]
builtin [shell-builtin [arg ...]]
caller [expr]
case WORD in [PATTERN [| PATTERN]...) C>
cd [-L|-P [-e]] [-@] [dir]
command [-pVv] command [arg ...]
compgen [-abcdefgjksuv] [-o option] [->]
complete [-abcdefgjksuv] [-pr] [-DE] [->]
compopt [-o]+o option] [-DE] [name ...]>
continue [n]
coproc [NAME] command [redirections]
declare [-aAfGgilrtux] [-p] [name[=val]>
dirs [-clpv] [+N] [-N]
disown [-h] [-ar] [jobspec ...]
echo [-neE] [arg ...]
enable [-a] [-dnpS] [-f filename] [name]>
eval [arg ...]
exec [-cl] [-a name] [command [argument]>
exit [n]
export [-fn] [name[=value] ...] or expo>
false
fc [-e ename] [-lnr] [first] [last] or >
fg [job_spec]
for NAME [in WORDS ...] ; do COMMANDS;>
for (( exp1; exp2; exp3 )); do COMMANDS>
function name { COMMANDS ; } or name ()>
getopts optstring name [arg]
hash [-lr] [-p pathname] [-dt] [name ...]>
help [-dns] [pattern ...]
history (-c) (-d offset) [n] or hist>
if COMMANDS; then COMMANDS; [ elif COM>
jobs [-lprS] [jobspec ...] or jobs -x>
kill [-s sigspec | -n signum | -sigspe>
let arg [arg ...]
local [option] name[=value] ...
logout [n]
mapfile [-n count] [-0 origin] [-s cou>
popd [-n] [+N | -N]
printf [-v var] format [arguments]
pushd [-n] [+N | -N | dir]
pwd [-LP]
read [-ers] [-a array] [-d delim] [-i >
readarray [-n count] [-0 origin] [-s c>
readonly [-aAf] [name[=value] ...] or >
return [n]
select NAME [in WORDS ... ;] do COMMAND>
set [-abefhkmnptuvxBCHP] [-o option-na>
shift [n]
shopt [-pqsu] [-o] [optname ...]
source filename [arguments]
suspend [-t]
test [expr]
time [-p] pipeline
times
trap [-lp] [[arg] signal_spec ...]
true
type [-afptP] name [name ...]
typeset [-aAfGgilrtux] [-p] name[=valu>
ulimit (-SHabcdefilmnoprstuvwxyz) [limit]>
umask [-p] [-S] [mode]
unalias [-a] name [name ...]
unset [-f] [-v] [-n] [name ...]
until COMMANDS; do COMMANDS; done
variables - Names and meanings of some>
wait [-n] [id ...]
while COMMANDS; do COMMANDS; done
{ COMMANDS ; }

```



Termux displaying builtin shell commands

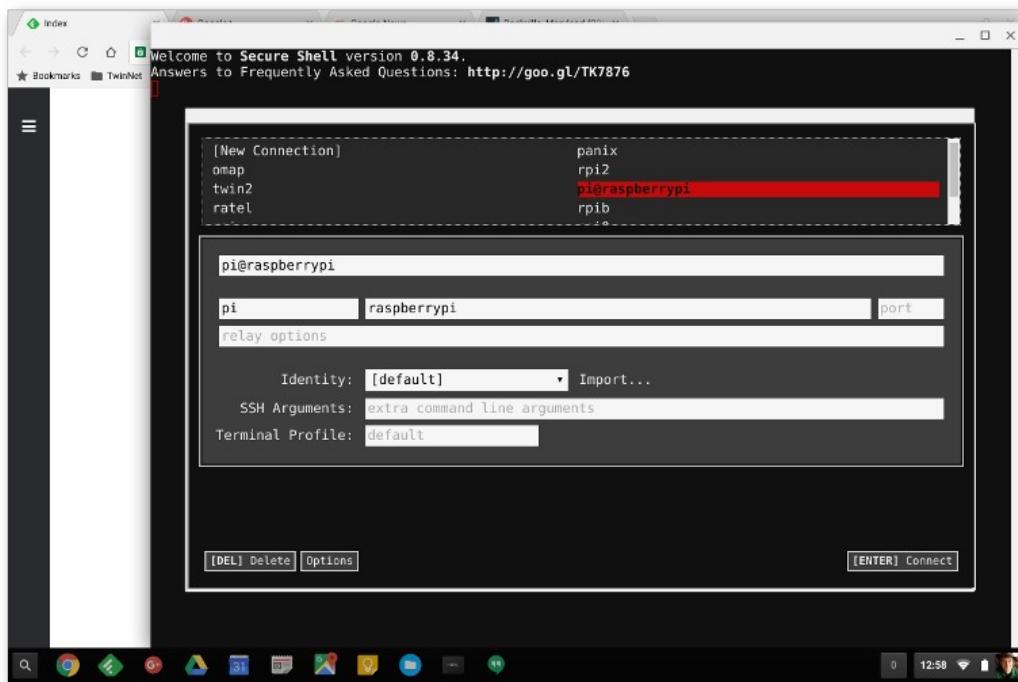
We can have dot files (like `.bashrc`) and even write shell scripts and compile and debug programs in Termux. Pretty neat.

When executing `ssh`, Termux looks like an “xterm-256color” terminal to remote systems.

Chrome/Chrome OS

Google makes a decent SSH client for Chrome and Chrome OS (which is Linux, too, after all) that allows logging on to remote systems. Called Secure Shell, it uses hterm (HTML Terminal, a terminal emulator written in JavaScript) combined with an SSH client. To remote systems, it looks like a “xterm-256color” terminal. It works pretty well, but lacks some features that advanced SSH users may need.

Secure Shell is available at the Chrome Web Store.



Secure Shell running on Chrome OS

Summing Up

Given that our terminal emulators are among our most vital tools, they should command more of our attention. There are many different terminal programs with potentially interesting and helpful features, many of which, most users rarely, if ever, use. This is a shame since many of these features are truly *useful* to the busy command line user. We have looked at a few of the ways these features can be applied to our daily routine, but there are certainly many more.

Further Reading

- “The Grumpy Editor’s guide to terminal emulators” by Jonathan Corbet:
<https://lwn.net/Articles/88161/>

xterm:

- xterm on Wikipedia: <https://en.wikipedia.org/wiki/Xterm>
- Homepage for the current maintainer of xterm, Thomas Dickey: <https://invisible-island.net/xterm/>

Tektronix 4014:

- Tektronix 4014 on Wikipedia: https://en.wikipedia.org/wiki/Tektronix_4010
- Some background on the 4014 at Chilton Computing: <http://www.chilton-computing.org.uk/acd/icf/terminals/p005.htm>

rxvt:

- Home page for rxvt: <http://rxvt.sourceforge.net/>

urxvt (rxvt-Unicode):

- Home page for the rxvt-Unicode project: <http://software.schmorp.de/pkg/rxvt-unicode.html>

gnome-terminal:

- Help pages for gnome-terminal:
<https://help.gnome.org/users/gnome-terminal/stable/>

konsole:

- The Konsole Manual at the KDE Project:
<https://docs.kde.org/stable5/en/applications/konsole/index.html>

quake:

- The home page for the quake project: <http://quake-project.org/>
- The Arch Wiki entry for quake (contains a lot of useful information but some is Arch Linux specific): <https://wiki.archlinux.org/index.php/Guake>

terminator:

- The home page for the terminator project:
<https://gnometerminator.blogspot.com/p/introduction.html>

Connectbot:

- Connectbot at the Google Play Store: <https://play.google.com/store/apps/details?id=org.connectbot&hl=en>

Hacker's Keyboard:

- Hacker's Keyboard at the Google Play Store:
<https://play.google.com/store/apps/details?id=org.pocketworkstation.pckeyboard&hl=en>

Termux:

- Termux at the Google Play Store: <https://play.google.com/store/apps/details?id=com.termux&hl=en>

Secure Shell

- Secure Shell at the Chrome Web Store:
<https://chrome.google.com/webstore/detail/secure-shell-app/pnhechapfaindjhompbnflcldabbghjo>
- Secure Shell FAQ:
<https://chromium.googlesource.com/apps/libapps/+/master/nassh/doc/FAQ.md>

9 Other Shells

While we have spent a great deal of time learning the bash shell, it's not the only "game in town." Unix has had several popular shells and almost all are available for Linux, too. In this adventure, we will look at some of these, mostly for their historical significance. With a couple of possible exceptions, there is very little reason to switch, as bash is a pretty good shell. Some of these alternate shells are still popular on other Unix and Unix-like systems, but are rarely used in Linux except when compatibility with other systems is required.

The Evolution of Shells

The first Unix shell was developed in 1971 by Ken Thompson who, along with Dennis Richie, created Unix at AT&T Bell Telephone Laboratories. The *Thompson shell* introduced many of the core ideas that we see in shells today. These include I/O redirection, pipelines, and the ability to place processes in the background. This early shell was intended only for interactive use, not for use as a programming language.

The Thompson shell was followed in 1975 by the *Mashey shell*, written by John Mashey. This shell extended the Thompson shell to support shell scripting by including variables, a built-in if/then/else, and other rudimentary flow control constructs.

At this point we come to a big split in shell design philosophies. In 1978 Steve Bourne created the *Bourne shell*. The following year, Bill Joy (the original author of *vi*) released the *C shell*.

The Bourne shell added a lot of features that greatly improved shell scripting. These included flow control structures, better variables, command substitutions, and here scripts. The Bourne shell contains much of the functionality that we see in the bash shell today.

On the other hand, the C shell was designed to improve interactive use by adding command history and job control. The C shell, as its name would imply, uses a syntax that mimics the C programming language. C language programmers abounded in the Unix community, so many preferred this style. Ironically, the C shell is not very good at scripting. For example, it lacks user defined functions and the shell's parser (the portion of the shell that reads and figures out what the script is saying) suffers from serious limitations.

In 1983, in an effort to improve the Bourne shell, David Korn released the *Korn shell*. Command history, job control, associative arrays, vi and Emacs style command editing are among the features that were added. In the 1993 release (known as *ksh93*), floating point arithmetic was added. The Korn shell was good for both interactive use and

scripting. Unfortunately, the Korn shell was proprietary software distributed under license from AT&T. This changed in 2000 when it was released under an open source license.

When POSIX standardized the shell for use on Unix systems, it specified a subset of the Korn shell that would be largely compatible with the earlier Bourne shell. As a result, most Bourne-type shells now conform to the POSIX standard, but include various extensions.

Partially in response to the proprietary licensing of the Korn shell, the GNU project developed `bash`, which includes many Korn shell features. The first version, written by Brian Fox was released in 1989 and is today maintained by Chet Ramey. Bash is best known as the default shell in most Linux distributions. It is also the default shell in versions of macOS; however, due to Apple's obsession with secrecy and lock-down, they refuse to update `bash` to version 4 because of provisions in the GNU GPLv3.

Since the development of `bash`, one new shell has emerged that is gaining traction among Linux and MacOS users. It's the *Z shell* (`zsh`). Sometimes described as "the Emacs of shells" because of its large feature set, `zsh` adds a number of features to enhance interactive use.

Modern Implementations

Modern Linux users have a variety of shell programs from which to choose. Of course, the overwhelming favorite is `bash`, since it is the default shell supplied with most Linux distributions. That said, users migrating from other Unix and Unix-like systems may be more comfortable with other shells. There is also the issue of portability. If a script is required to run on multiple Unix-like systems, then care must be taken to either: 1) make sure that all the systems are running the same shell program, or 2) write a script that conforms to the POSIX standard, since most modern Bourne shell derivatives are POSIX complaint.

A Reference Script

In order to compare the various shell dialects, we'll start with this `bash` script taken from Chapter 33 of TLCL:

```
#!/bin/bash

# longest-word : find longest string in a file

for i; do
    if [[ -r "$i" ]]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
            fi
        done
    fi
done
```

```

        max_word=$j
    fi
done
echo "$i: '$max_word' ($max_len characters)"
fi
done

```

dash - Debian Almquist Shell

The Debian Almquist shell is Debian's adaptation of the *Almquist shell* (ash) originally written in the 1980s by Kenneth Almquist. The ash shell is the default shell on several of the BSD flavors of Unix. `dash`, like its ancestor `ash`, has the advantage of being small and fast; however, it achieves this by forgoing conveniences intended for interactive use such as command history and editing. It also lacks some builtin commands, relying instead on external programs. Its main use is the execution of shell scripts, particularly during system startup. On Debian and related distributions such as Ubuntu, `dash` is linked to `/bin/sh`, the shell used to run the system initialization scripts.

`dash` is a POSIX compliant shell, so it supports Bourne shell syntax with a few additional Korn shell features:

```

#!/bin/dash

# longest-word.dash : find longest string in a file

for i; do
    if [ -r "$i" ]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len=${#j}
            if [ $len -gt $max_len ]; then
                max_len=$len
                max_word=$j
            fi
        done
        echo "$i: '$max_word' ($max_len characters)"
    fi
done

```

Here we see that the `dash` script is mostly the same as the `bash` reference script, but we do see some differences. For one thing, `dash` does not support the `[[`] syntax for conditional tests; it uses the older Bourne shell syntax. The POSIX specification is also missing the `((expression))` syntax for arithmetic expansion, nor does it support brace expansion. `dash` and the POSIX specification do support the `$(cmd)` syntax for command substitution in addition to the older ``cmd`` syntax.

tcsh - TENEX C Shell

The `tcsh` program was developed in the early 1980s by Ken Greer as an enhanced replacement for the original `csh` program. The name TENEX comes from the operating system of the same name, which was influential in the design of the interactive features in

`tcsh`. Compared to `csh`, `tcsh` added additional command history features, Emacs and vi-style command line editing, spelling correction, and other improvements intended for interactive use. Early versions of Apple's OS X used `tcsh` as the default shell. It is still the default root shell on several BSD distributions.

`tcsh`, like the C shell, is not POSIX compliant as we can see here:

```
#!/usr/bin/tcsh

# longest-word.tcsh : find longest string in a file

foreach i ($argv)
    set max_word=""
    set max_len=0
    foreach j (`strings $i`)
        set len=$%j
        if ($len > $max_len) then
            set max_word=$j
            set max_len=$len
        endif
    end
    echo "$1 : $max_word ($max_len characters)"
end
```

Our `tcsh` version of the script demonstrates many differences from Bourne style syntax. In C shell, most of the flow control statements are different. We see for example, that the outer loop starts with a `foreach` statement incrementing the variable `i` with succeeding values from the word list `$argv`. `argv`, taken from the C programming language, refers to an array containing the list of command line arguments.

While this simple script works, `tcsh` is not very capable when things get more complicated. It has two major weaknesses. First, it does not support user-defined functions. As a workaround, separate scripts can be called from the main script to carry out the individual functions. Second, many complex constructs easily accomplished with the POSIX shell, such as:

```
{ if [[ "$a" ]]; then
    grep "string1"
else
    grep "string2"
fi
} < file.txt
```

are not possible because the C shell parser cannot handle redirection with flow control statements. The parser also makes quoting very troublesome.

ksh - Korn Shell

The Korn shell comes in several different flavors. Basically, there are two groups, `ksh88` and `ksh93`, reflecting the year of their release. There is a public domain version of `ksh88` called `pdksh`, and more official versions of both `ksh88` and `ksh93`. All three are available for Linux. `ksh93` would be the preferred version for most users, as it is the version found

on most modern commercial Unix systems. During installation is it often symlinked to `ksh`.

```
#!/usr/bin/ksh

# longest-word.ksh : find longest string in a file

for i; do
    if [ -r "$i" ]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        print "$i: '$max_word' ($max_len characters)"
    fi
done
```

As we can see in this example, `ksh` syntax is very close to `bash`. The one visible difference is the `print` command used in place of `echo`. Korn shell has `echo` too, but `print` is the preferred Korn shell command for outputting text. Another subtle difference is the way that pipelines work in `ksh`. As we learned in Chapter 28 of TLCL, a construct such as:

```
#!/bin/bash
str=""
echo "foo" | read str
echo $str
```

always produces an empty result because, in `bash` pipelines, each command in a pipeline is executed in a subshell, so its data is destroyed when the subshell exits. In this example, the final command (`read`) is in a subshell, and thus `str` remains empty in the parent process.

In `ksh`, the internal organization of pipelines is different. When we do this in `ksh`:

```
#!/usr/bin/ksh
str=""
echo "foo" | read str
echo $str
```

The output is “foo” because in the `ksh` pipeline, the `echo` is in the subshell rather than the `read`.

zsh - Z Shell

At first glance, the Z shell does not differ very much from `bash` when it comes to scripting:

```
#!/bin/zsh

# longest-word.zsh : find longest string in a file
```

```

for i; do
    if [[ -r "$i" ]]; then
        max_word=
        max_len=0
        for j in $(strings "$i"); do
            len=${#j}
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
        done
        print "$i: '$max_word' ($max_len characters)"
    fi
done

```

It runs scripts the same way that `bash` does. This is to be expected, as `zsh` is intended to be a drop-in replacement for `bash` in most cases. A couple of things to note however. First, `zsh` handles pipelines like the Korn shell does; the last command in a pipeline is executed in the current shell. Second, in `zsh`, the first element of an array is index 1, not 0 as it is in `bash` and `ksh`.

Where `zsh` does differ significantly is in the number of bells and whistles it provides for interactive use (some of which can be applied to scripting as well). Let's take a look at a few:

Tab Completion

Many kinds of tab completion are supported by `zsh`. These include command names, command options, and arguments.

When using the `cd` command, repeatedly pressing the tab key first displays a list of the available directories, then begins to cycle through them. For example:

```

me@linuxbox ~ $ cd <tab>
me@linuxbox ~ $ cd <tab>
Desktop/ Documents/ Downloads/ Music/ Pictures/ Public/
Templates/ Videos/

me@linuxbox ~ $ cd Desktop/<tab>
Desktop/ Documents/ Downloads/ Music/ Pictures/ Public/
Templates/ Videos/

me@linuxbox ~ $ cd Documents/
Desktop/ Documents/ Downloads/ Music/ Pictures/ Public/
Templates/ Videos/

```

`zsh` can be configured to display a highlighted selector on the list of directories, and we can use the arrow keys to directly move the highlight to the desired entry in the list to select it.

We can also switch directories by replacing one part of a path name with another:

```
me@linuxbox ~ $ cd /usr/local/share
```

```
me@linuxbox share $ cd share bin  
me@linuxbox bin $ pwd  
/usr/local/bin
```

Pathnames can be abbreviated as long as they are unambiguous. If we type:

```
me@linuxbox ~ $ ls /u/l/share<tab>
```

zsh will expand it into:

```
me@linuxbox ~ $ ls /usr/local/share/
```

That can save a lot of typing!

Help for options and arguments is provided for many commands. To invoke this feature, we type the command and the leading dash for an option, then hit the tab key:

```
me@linuxbox ~ $ rm -<tab>  
--force           -f      -- ignore nonexistent files, never prompt  
--help            --      -- display help message and exit  
-i                --      -- prompt before every removal  
-I                --      -- prompt when removing many files  
--interactive     --      -- prompt under given condition  
                      (defaulting to always)  
--no-preserve-root --      -- do not treat / specially  
--one-file-system --      -- stay within file systems of files given  
                      as arguments  
--preserve-root   --      -- do not remove / (default)  
--recursive       -R    -r  -- remove directories and their contents  
                      recursively  
--verbose         -v      -- explain what is being done  
--version         --      -- output version information and exit
```

This displays a list of options for the command, and like the `cd` command, repeatedly pressing `tab` causes zsh to cycle through the available options.

Pathname Expansion

The Z shell provides several powerful additions to pathname expansion that can save steps when specifying files as command arguments.

We can use "`**`" to cause recursive expansion. For example, if we wanted to list every file name ending with `.txt` in our home directory and its subdirectories, we would have to do this in bash:

```
me@linuxbox ~ $ find . -name "*.txt" | sort
```

In zsh, we could do this:

```
me@linuxbox ~ $ ls **/*.txt
```

and get the same result.

And if that weren't cool enough, we can also add *qualifiers* to the wildcard to perform many of the same tests as the `find` command. For example:

```
me@linuxbox ~ $ **/*.*txt(@)
```

will only display the files whose names end in `.txt` and are symbolic links.

There are many supported qualifiers and they may be combined to perform very fine grained file selection. Here are some examples:

Qualifier	Description	Example
.	Regular files	<code>ls *.txt(..)</code>
/	Directories	<code>ls *.txt(/)</code>
@	Symbolic links	<code>ls *.txt(@)</code>
*	Executable files	<code>ls *(*)</code>
F	Non-empty (“full”) directories	<code>ls *(F)</code>
/^F	Empty directories	<code>ls *(/^F)</code>
mn	Modified exactly n days ago	<code>ls *(m5)</code>
m-n	Modified less than n days ago	<code>ls *(m-5)</code>
m+n	Modified more than n days ago	<code>ls *(m+5)</code>
L0	Empty (zero length) file	<code>ls *(L0)</code>
LM+n	File larger than n megabytes	<code>ls *(LM+5)</code>
LK-n	File smaller than n kilobytes	<code>ls *(LK-100)</code>

Z shell pathname expansions

Global Aliases

Z shell provides more powerful aliases. With `zsh` we can define an alias in the usual way, such as:

```
me@linuxbox ~ $ alias vi='/usr/bin/vim'
```

and it will behave just as it would in `bash`. But we can also define a *global alias* that can be used at any position on the command line, not just at the beginning. For example, we can define a commonly used file name as an alias:

```
me@linuxbox ~ $ alias -g LOG='/var/log/syslog'
```

and then use it anywhere on a command line:

```
me@linuxbox ~ $ less LOG
```

The use of an uppercase alias name is not a requirement, it's just a custom to make its use easier to see. We can also use global aliases to define common redirections:

```
me@linuxbox ~ $ alias -g L='| less"
```

or

```
me@linuxbox ~ $ alias -g W='| wc -l'
```

Then we can do things like this:

```
me@linuxbox ~ $ cat LOG W
```

to display the number of lines in `/var/log/syslog`.

Suffix Aliases

What's more, we can define aliases to act like an "open with..." by defining a *suffix alias*. For example, we can define an alias that says all files that end with ".txt" should be viewed with less:

```
me@linuxbox ~ $ alias -s txt='less'
```

Then we can just type the name of a text file, and it will be opened by the application specified by the alias:

```
me@linuxbox ~ $ dir-list.txt
```

How cool is that?

Improved History Search

`zsh` adds a neat trick to history searching. In `bash` (and `zsh` too) we can perform a reverse incremental history search by typing `Ctrl-r`, and each subsequent keystroke will refine the search. `zsh` goes one better by allowing us to simply type a few letters of the desired search string on the command line and then press up-arrow. It moves back through the history to find the first match, and each time we press the up-arrow, the next match is displayed.

Environment Variable Editing

`zsh` provides a shell builtin called `vared` for editing shell variables. For example, if we wanted to make a quick change to our `PATH` variable we can do this:

```
me@linuxbox ~ $ vared PATH
```

and the contents of the `PATH` variable appear in the command editor, so we can make a change and press Enter and the change takes effect.

Frameworks

We have only touched on a few of the features available in `zsh`. It has a lot. But with a large feature set comes complexity, and configuring `zsh` to take advantage of its full potential can be daunting. Heck, its man page is a only a table of contents to the other 10+ man pages that cover various topics. Fortunately, communities have sprung up to provide *frameworks* that supply ready-to-use configurations and add-ons for `zsh`. By far, the most popular of these is Oh-My-Zsh, a project led by Robby Russell.

Oh-My-Zsh is a large collection of configuration files, plugins, aliases, and themes. It offers support for tailoring `zsh` for many types of common tasks, particularly software development and system administration.

Changing to Another Shell

Now that we have learned a little about the different shells available for Linux, how can we experiment with them? First, we can simply enter the name of the shell from our `bash` prompt. This will launch the second shell as a child process of `bash`:

```
me@linuxbox ~ $ tcsh  
%
```

Here we have launched `tcsh` from the `bash` prompt and are presented with the default `tcsh` prompt, a percent sign. Since we have not yet created any startup files for the new shell, we get a very bare-bones environment. Each shell has its own configuration file(s) for interactive use just as `bash` has the `.bashrc` file to configure its interactive sessions.

Here is a table that lists the configuration files for each of the shells when used as an interactive (i.e., not a login) shell:

Shell	Configuration File(s)
dash	User-defined by setting the ENV variable in <code>~/.profile</code>
bash	<code>~/.bashrc</code>
ksh	<code>~/.kshrc</code>
tcsh	<code>~/.tchrcc</code>
zsh	<code>~/.zshrc</code>

Interactive shell configuration files

We'll need to consult the respective shell's man page (always a fun exercise!) to see the complete list of shell features. Most shells also include additional documentation and example configuration files in the `/usr/share/doc` directory.

To exit our temporary shell, we simply enter the `exit` command:

```
% exit  
me@linuxbox ~ $
```

Once we are done with our experimentation and configuration, we can change our default shell from `bash` to our new shell by using the `chsh` command. For example, to change from `bash` to `zsh`, we could do this:

```
me@linuxbox ~ $ chsh  
password:  
Changing the login shell for me  
Enter the new value, or press ENTER for the default  
    Login Shell [/bin/bash]: /usr/bin/zsh  
  
~ 23:30:40  
$
```

We are prompted for our password and then prompted for the name of the new shell whose name must appear in the `/etc/shells` file. This is a safety precaution to prevent an invalid name from being specified and thus preventing us from logging in again. That would be bad.

Summing Up

Because of the growing popularity of Linux among Unix-like operating systems, `bash` has become the world's predominant shell program. It has many of the best features of earlier shells and a few tricks of its own. However, if light weight and quick script execution is needed (for example, in embedded systems), `dash` is a good choice. Likewise, if working with other Unix systems is required, `ksh` or `tcsh` will provide the necessary compatibility. For the adventuresome among us, the advanced interactive features of `zsh` can enhance our day-to-day shell experience.

Further Reading

Shells and their history:

- A history of Unix shells from IBM Developer Works:
<https://developer.ibm.com/tutorials/l-linux-shells/>

C shell:

- A comparison of bash and tcsh syntax by Joe Linoff: http://joelinoff.com/blog/?page_id=235
- Tom Christiansen's famous "Csh Programming Considered Harmful" explains the many ways that csh bugs out when scripting:
<https://www-uxsup.csx.cam.ac.uk/misc/csh.html>
- And on a related note, here are the "Top Ten Reasons not to use the C shell" by Bruce Barnett: <https://www.grymoire.com/unix/CshTop10.txt>

Korn shell:

- Korn shell documentation: <http://www.kornshell.com/doc/>
- The on-line version of "Learning the Korn Shell" from O'Reilly:
<http://web.deu.edu.tr/doc/oreilly/unix/ksh/index.htm>

Z shell:

- Brendon Rapp's slide presentation on "Why zsh Is Cooler Than Your Shell":
<https://www.slideshare.net/jaguardesignstudio/why-zsh-is-cooler-than-your-shell-16194692>
- Joe Wright's list of favorite zsh features: <https://code.joejag.com/2014/why-zsh.html>
- David Fendrich's "No, Really. Use Zsh.": <http://fendrich.se/blog/2012/09/28/no/>

- Nacho Caballero’s “Master Your Z Shell with These Outrageously Useful Tips”: <http://reasoniamhere.com/2014/01/11/outrageously-useful-tips-to-master-your-z-shell/>
- Home page for Oh-My-Zsh: <https://ohmyz.sh/>

10 Vim, with Vigor

TLCL Chapter 12 taught us the basic skills necessary to use the vim text editor. However, we barely scratched the surface of its capabilities. Vim is a very powerful program. In fact, it's safe to say that vim can do anything. It's just a question of figuring out how. In this adventure, we will acquire an intermediate level of skill in this popular tool. In particular, we will look at ways to improve our productivity writing shell programs, configuration files, and documentation. Even better, after we get the hang of some of these additional features, using vim is actually fun.

In this adventure, we will look at some of the features that make vim so popular among developers and administrators. The community supporting vim is large and vigorous. Because vim is extremely rich in features and scriptable, there are many plugins and add-ons available. However, we are going to restrict ourselves to stock vim and the plugins that normally ship with it.

A note about nomenclature: in TLCL we used the terms “command”, “insert”, and “ex” to identify the three primary modes of vim. We did this to match the traditional modes of vim’s ancestor, vi. Since this is an all-vim adventure, we will switch to the names used in the vim documentation which are *normal*, *insert*, and *command*.

Let’s Get Started

First, we need to be sure we are running the full version of vim. Many distributions only ship with an abbreviated version. To get the full version, install the “vim” package if it’s not already installed. This is also be a good time to add an alias to the `.bashrc` file to make “vi” run vim (some distributions symbolically link ‘vi’ to vim, so this step might not be needed).

```
alias vi='vim'
```

Next, let’s create a minimal `.vimrc`, its main configuration file.

```
[me@linuxbox ~]$ vi ~/.vimrc
```

Edit the file so it contains these two lines:

```
set nocompatible  
filetype plugin on
```

This will ensure that vim is not restricted to the vi feature set, and load a standard plugin that lets vim recognize different file types. After inserting the two lines of text, return to normal mode and (just for fun) type lowercase ‘m’ followed by uppercase ‘V’.

```
mV
```

Nothing will appear to happen, and that’s OK. We’ll come back to that later. Save the file and exit vim.

```
:wq
```

Getting Help

Vim has an extensive built-in help system. If we start vim:

```
[me@linuxbox ~]$ vi
```

and enter the command:

```
:help
```

It will appear at the top of the display.

```
*help.txt*      For Vim version 7.3.  Last change: 2010 Jul 20

          VIM - main help file

Move around: Use the cursor keys, or "h" to go left,           k
              "j" to go down, "k" to go up, "l" to go right.   h   l
Close this window: Use ":q<Enter>".
Get out of Vim: Use ":qa!<Enter>" (careful, all changes are lost!).

Jump to a subject: Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
With the mouse:   ":set mouse=a" to enable the mouse (in xterm or GUI).
                  Double-click the left mouse button on a tag, e.g. |bars|.
Jump back:        Type CTRL-T or CTRL-O (repeat to go further back).

Get specific help: It is possible to go directly to whatever you want help
on, by giving an argument to the |:help| command.
It is possible to further specify the context:
          *help-context*
          WHAT          PREPEND     EXAMPLE
          Normal mode command    (nothing)  :help x
help.txt [Help][R0]          1,1        Top
[No Name]                   0,0-1      All
"help.txt" [readonly] 221L, 8239C
```

Vim help window

Though help is extensive and very useful, it immediately presents a problem because it creates a *split* in the display. This is a rather advanced feature that needs some explanation.

Vim can divide the display into multiple panes, which in vim parlance are called *windows*. These are very useful when working with multiple files and other vim features such as help. When the display is divided this way, we can toggle between the windows by typing `Ctrl-w` twice. We can manipulate vim windows with the following commands:

```
:split      Create a new window
Ctrl-w Ctrl-w  Toggle between windows
Ctrl-w _      Enlarge the active window
Ctrl-w =      Make windows the same size
:close       Close active window
:only        Close all other windows
```

When working with files, it's important to note that "closing" a window (with either `:q` or `:close`) does not remove the buffer containing the window's content; we can recall it at any time. However, when we close the final window, vim terminates.

To exit help, make sure the cursor is in the help window and enter the quit command.

```
:q
```

But enough about windows, let's get back to help. If we scroll around the initial help file, we see it is a hypertext document full of links to various topics and it begins with the commands we need to navigate the help system. This is all well and good, but it's not the most interesting way to use it.

The best way is to type `:h` followed by the topic we are interested in. The fact we don't have to type out "`:help`" reveals that most vim commands can be abbreviated. This saves a lot of work. In general, commands can be shortened to their smallest non-ambiguous form. Frequently used commands, like `help`, are often shortened to a single character but the system of abbreviations isn't predictable, so we have to use `help` to find them. For the remainder of this adventure, we will try to use the shortest available form.

There is an important table near the beginning of the initial help file:

WHAT	PREPEND	EXAMPLE
Normal mode command	(nothing)	<code>:help x</code>
Visual mode command	<code>v_</code>	<code>:help v_u</code>
Insert mode command	<code>i_</code>	<code>:help i_<Esc></code>
Command-line command	<code>:</code>	<code>:help :quit</code>
Command-line editing	<code>c_</code>	<code>:help c_</code>
Vim command argument	<code>-</code>	<code>:help -r</code>
Option	<code>'</code>	<code>:help 'textwidth'</code>

Search for help: Type "`:help word`", then hit CTRL-D to see matching help entries for "word".

This table describes how we should ask for help in particular contexts. We're familiar with the normal mode command '`i`' which invokes insert mode. In the case of such a normal mode command, we simply type:

```
:h i
```

to display its help page. For command mode commands, we precede the command with a '`:`', for example:

```
:h :q
```

gets help with the `:quit` command.

There are other contexts for modes we have yet to cover. We'll get to those in a little bit.

As we go along, feel free to use help to learn more about the commands we discuss. As this adventure goes on, the text will include suggested help topics to explore.

Oh, and while we're on the subject of command mode, now is a good time to point out that command mode has command line history similar to the shell. After typing ':' we can use the up and down arrows to scroll through past commands.

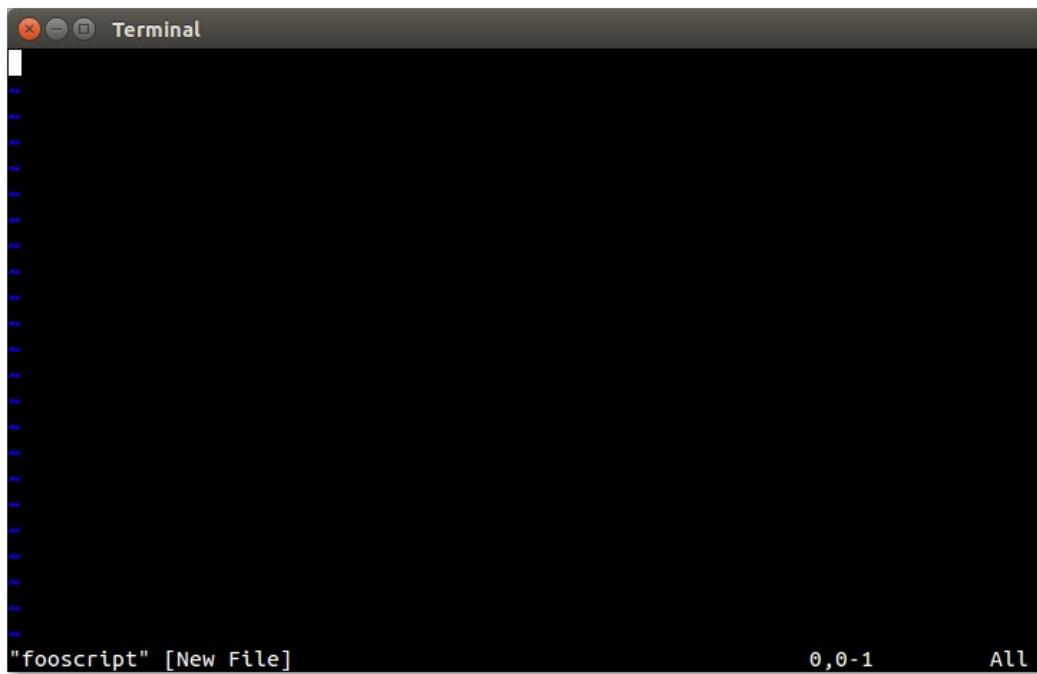
Help topics: :split :close :only ^w

Starting a Script

In order to demonstrate features in vim, we're going to write a shell script. What it does is not important, in fact, it won't do anything at all except to show how we can edit scripts. To begin, let's start vim with the name of an non-existent script file:

```
[me@linuxbox ~]$ vi fooscript
```

and we will get our familiar "new file" window:



New file

Setting the Filetype

At this point vim has no idea what kind of file we are creating. If we had named the file `fooscript.sh` the filetype plugin would have determined that we were editing a shell script. We can verify this by asking vim what the current filetype is:

```
:set ft?
```

When we use the `set` command this way, it displays the current value of an option— in this case the `ft` (short for `filetype`) option. It should respond with the following indicating that the `ft` option is unset:

```
filetype=
```

For the curious, we can ask for help like this to get more information:

```
:h :set  
:h 'ft'
```

To see all the current option settings, we can do this and the entire list will appear:.

```
:set
```

Since we want our new file to be treated as a shell script, we can set the filetype manually:

```
:set ft=sh
```

Next, let's enter insert mode and type the first couple of lines in our script:

```
#!/bin/bash  
  
# Script to test editing with vim
```

Exit insert mode by pressing the `Esc` key and save the file:

```
:w
```

Now that our file contains the shebang on the first line, the filetype plugin will recognize the file as a shell script whenever it is loaded.

Using the Shell

One thing we can do with filetypes is create a configuration file for each of the supported types. Normally, these are placed in the `~/.vim/ftplugin` directory. To do this, we need to create the directory.

We don't have leave vim to do this; we can launch a shell from within vim. This is easily done by entering the command:

```
:sh
```

After doing this, a shell prompt will appear and we can enter our shell command:

```
[me@linuxbox ~]$ mkdir -p ~/.vim/ftplugin
```

When we're done with the shell, we return to vim by exiting the shell:

```
[me@linuxbox ~]$ exit
```

Now that we have a place for our configuration file to live, let's create it. We'll open a new file:

```
:e ~/.vim/ftplugin/sh.vim
```

The filename `sh.vim` is required.

Help topics: `:sh`

Buffers

Before we start editing our new file, let's look at what vim is doing. Each file that we edit is stored in a *buffer*. We can look the current list of buffers this way:

```
:ls
```

This will display the list. There are several ways that we can switch buffers. The first way is to cycle between them:

```
:bn
```

This command (short for :bnext) cycles through the buffer list, wrapping around at the end. Likewise, there is a :bp (:bprevious) command which cycles through the buffer list backwards. We can also select a buffer by number:

```
:b 2
```

We can even refer to a buffer by using a portion of the file name:

```
:b fooscript
```

Let's cycle back to our new buffer and add this line to our configuration file:

```
setlocal number
```

This will turn on line numbering each time we load a shell script. Notice that we use the setlocal command rather than set. This is because set will apply an option globally, whereas the setlocal command only applies the option to the current buffer. This will prevent settings conflicts when we edit multiple files of different types.

We can also control syntax highlighting while we're here. We can turn it on with:

```
syntax on
```

Or turn it off with:

```
syntax off
```

We'll save this file now, but before we do that, let's type `mS` (lowercase m uppercase S), similar to what we did when we saved our initial `.vimrc`.

Help topics: :ls :buffers :bnext :bprevious :setlocal 'number' :syntax

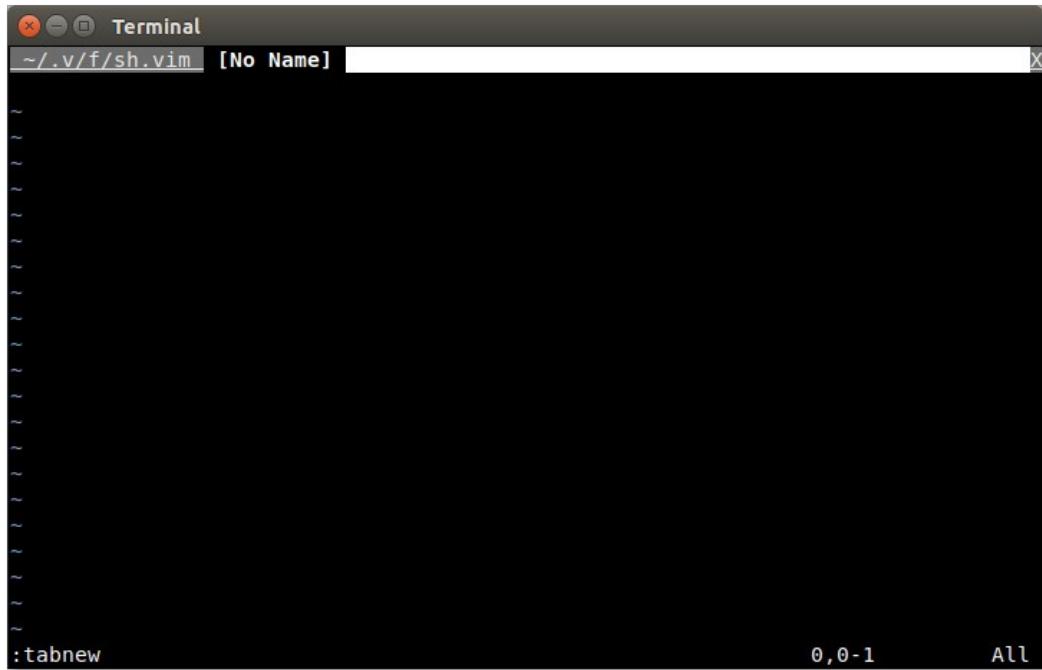
Tabs

Before we leave the subject of buffers, let's take a look a possible way of using them. We have already discussed splits and windows, but recent versions of vim include a useful alternative called *tabs*. As the name suggests, this feature allows each buffer to appear in its own tab.

To create a new tab, we type the following command:

```
:tabnew
```

This will open a new tab. Since we haven't associated the tab with a buffer yet, the tab will be labeled "[No Name]".



New tab

While we are in the newly created tab, we can switch to one of the existing buffers as before by typing:

:bn

```
#!/bin/bash
#
# Script to test editing with vim

```

Displaying a buffer in a tab

We can open files in tabs, too. It works much like the `:e` command. To open a file in a tab, we type `:tabe` followed by the name of the file.

Switching tabs is easy. To cycle through the tabs forward, we type `gt`. To cycle backwards, we type `gT`. If mouse support is enabled, tabs can be selected by clicking on them and new tabs can be opened by double clicking on the tab bar at the top of the screen.

It's also possible to start vim with multiple files loaded in tabs by adding the `-p` option to the command line. For example:

```
[me@linuxbox ~]$ vim -p file1 file2
```

To close tabs, we use `:q` command just like closing a vim window. When only one tab remains, vim leaves tabbed mode and the display returns to its usual state.

There are a lot of tab-related features in vim. See the help topic for full details.

Help topics: `tabpage`

Color Schemes

If we return to the buffer containing our shell script, we should see the effects of our `sh.vim` file. When syntax highlighting is turned on (`:syn on` will do the trick) it assumes the current color scheme. Vim ships with a bunch of different ones. To see the name of the current scheme, type this command:

```
:colo
```

and it will display the name. To see the entire set of available color schemes, type :colo followed by a space, then the tab key. This will trigger vim's autocomplete and we should see the first name in the list. Subsequent use of the tab key will cycle through the list and we can try each one.

The ‘desert’ color scheme looks pretty good with shell scripts, so let’s add this to our sh.vim file. To do this, switch to the buffer containing that file and add the following line:

```
colorscheme desert
```

Notice that we used the long form of the colorscheme command. We could have used the abbreviated form colo but it’s a common custom to use the long names in configuration files for clarity.

There are many additional color schemes for vim on the Internet. To use one, first create a ~/.vim/colors directory and then download the new scheme into it. The new scheme will appear when we cycle through the list.

Now, save the file and return to our shell script.

Help topics: :colorscheme

Marks and File Marks

We know there are various ways of moving around within document in vim. For example, to get to the top, we can type:

```
gg
```

To go to the bottom we can type:

```
G
```

Vim (and real vi for that matter) also allows us to *mark* an arbitrary location within a document that we can recall at will. To demonstrate this, go to the top of the script and type:

```
ma
```

Next, go to the bottom of the document and type:

```
mb
```

We have just set two marks, the first called “a” and the second called “b”. To recall a mark, we precede the name of the mark with the ‘ character, like so and we are taken to the top of the file again:

```
' a
```

We can use any lowercase letter to name a mark. Now, the clever among us will remember that we set marks in both the `.vimrc` file, and the `sh.vim` file but we used uppercase letters.

Yes we did, because they're special. They're called *file marks* and they let us set a mark in a file that vim will remember between sessions. Since we set the V mark in the `.vimrc` file and the S mark in `sh.vim` file, if we ever type:

```
'V
```

vim will immediately take us to that mark even if vim has to load the file to do it. By doing this to `.vimrc` and `sh.vim`, we're set up to edit our configuration files anytime we get another bright idea about customizing vim.

Help topics: `m` '

Visual Mode

Among the best features that vim adds to ordinary vi is *visual mode*. This mode allows us to visually select text in our document. If we type:

```
v
```

An indicator will appear at the bottom of the screen showing that we have entered this mode. While in visual mode, when we move the cursor (using any of the available movement commands), the text is both visually highlighted and selected. Once this is done we can apply the normal editing commands on the selected text such as `c` (change), `d` (delete), and `y` (yank). Typing `v` a second time will exit visual mode. If we type:

```
V
```

we again enter visual mode, but this time selection is done on a line-by-line basis rather than by individual characters. This is handy when cutting and copying blocks of code.

There is a third way of using visual mode. If we type:

```
Ctrl-v
```

we are able to select rectangular blocks of text by columns. For example, we could select a column from a table.

Help topics: `v` `V` `^v`

Indentation

We're going to continue working on our shell script, but first we need to talk a little about *indentation*. As we know, indentation is used in programming to help communicate program structure. The shell does not require any particular style of indentation; it's purely for the benefit of the humans trying to read the code. However, some other computer languages, such as Python, require indentation to express program structure.

Indentation is accomplished in one of two ways; either by inserting tab characters or by inserting a sequence of spaces. To understand the difference, we have to go way back in time to typewriters and teletype machines.

In the beginning, there were typewriters. On a typewriter, in order to make indenting the first line of a paragraph easier, someone invented a mechanical device that would move the carriage over a set amount of space. Over time, these devices became more sophisticated and allowed multiple tab stops to be set. When teletype machines came about, they implemented tabs with a specific ASCII character called HT (horizontal tab, code 9) which, by default, was rendered by moving the cursor to the next character position evenly divisible by 8.

In the early days of computing, when memory was precious, it made sense to conserve space in text files by using tab characters to avoid having to pad the text file with spaces.

Using tab characters creates a problem, though. Since a tab character has no intrinsic width (it only signifies the desire to move to the next tab stop), it's up to the receiving program to render the tab with some defined width. This means that a file containing tabs could be rendered in different ways in different programs and in different contexts.

Since memory is no longer expensive, and using tabs creates this rendering confusion, modern practice calls for spaces instead of tabs to perform indentation (though this remains somewhat controversial). Vim provides a number of options for setting tabs and indentation. An excerpt from the help file for the `tabstop` option explains the ways vim can treat tabs:

There are four main ways to use tabs in Vim:

1. Always keep '`tabstop`' at 8, set '`softtabstop`' and '`shiftwidth`' to 4 (or 3 or whatever you prefer) and use '`noexpandtab`'. Then Vim will use a mix of tabs and spaces, but typing `<Tab>` and `<BS>` will behave like a tab appears every 4 (or 3) characters.
2. Set '`tabstop`' and '`shiftwidth`' to whatever you prefer and use '`expandtab`'. This way you will always insert spaces. The formatting will never be messed up when '`tabstop`' is changed.
3. Set '`tabstop`' and '`shiftwidth`' to whatever you prefer and use a `|modeline|` to set these values when editing the file again. Only works when using Vim to edit the file.
4. Always set '`tabstop`' and '`shiftwidth`' to the same value, and '`noexpandtab`'. This should then work (for initial indents only) for any tabstop setting that people use. It might be nice to have tabs after the first non-blank inserted as spaces if you do this though. Otherwise, aligned comments will be wrong when '`tabstop`' is changed.

Indentation Settings For Scripts

For our purposes, we will use method 2 and add the following lines to our `sh.vim` file to set tabs to indent 2 spaces. This is a popular setting specified in some shell script coding standards.

```
setlocal tabstop=2
setlocal shiftwidth=2
setlocal expandtab
setlocal softtabstop=2
setlocal autoindent
setlocal smartindent
```

In addition to the tab settings, we also included the `autoindent` and `smartindent` settings, which will automate indentation when we write blocks of code.

After adding the indentation settings to our `sh.vim` file, we'll add some more lines to our shell script (type this in to see how it behaves):

```
1  #! /bin/bash
2
3  # This is a shell script to demonstrate features in vim.
4  # It doesn't really do anything, it just shows what we can do.
5
6  # Constants
7  A=1
8  B=2
9
10 if [[ "$A" == "$B" ]]; then
11     echo "This shows how smartindent works."
12     echo "This shows how autoindent works."
13     echo "A and B match."
14 else
15     echo "A and B do not match."
16 fi
17
18 afunction() {
19     cmd1
20     cmd2
21 }
22
23 if [[ -e file ]]; then
24     cmd1
25     cmd2
26 fi
```

As we type these additional lines into our script, we notice that vim can now automatically provide indentation as needed. The `autoindent` option causes vim to repeat the previous line's indentation while the `smartindent` option provides indentation for certain program structures such as the function and `if` statements. This saves a lot of time while coding and ensures that our code stays nice and neat.

If we find ourselves editing an existing script with a indentation scheme differing from our current settings, vim can convert the file. This is done by typing:

```
:retab
```

The file will have its tabs adjusted to match our current indentation style.

Help topics: 'tabstop' 'shiftwidth' 'expandtab' 'softtabstop' 'autoindent'
'smartindent'

Power Moves

As we learned in TLCL, vim has lots of *movement commands* we can use to quickly navigate around our documents. These commands can be employed in many useful ways.

Here is a list of the common movement commands. Some of this is review, some is new.

```
h      Move left (also left-arrow)
l      Move right (also right-arrow)
j      Move down (also down-arrow)
k      Move up (also up-arrow)
0      First character on the line (also the Home key)
^      First non-whitespace character on the line
$      Last character on the line (also the End key)
f{char} Move right to the next occurrence of char on the current
          line
t{char} Move right till (i.e., just before) the next occurrence of
          char on the current line
;      Repeat last f or t command
gg     Go to first line
G      Go to last line. If a count is specified, go to that line.
w      Move forward (right) to beginning of next word
b      Move backward (left) to beginning of previous word
e      Move forward to end of word
)      Move forward to beginning of next sentence
(      Move backward to beginning previous sentence
}      Move forward to beginning of next paragraph
{      Move backward to beginning of previous paragraph
```

Remember, each of these commands can be preceded with a count of how many times the command is to be performed.

Operators

Movement commands are often used in conjunction with *operators*. The movement command determines how much of the text the operator affects. Here is a list of the most commonly used operators:

```
c      Change (i.e., delete then insert)
d      Delete/cut
y      Yank (i.e., copy)
~      Toggle case
gu     Make lowercase
gU     Make uppercase
qq     Format text (a topic we'll get to shortly)
g?     ROT13 encoding (for obfuscating text)
>      Shift (i.e., indent) right
<      Shift left
```

We can use visual mode to easily demonstrate the movement commands. Move the cursor to the beginning of line 3 of our script and type:

```
vf.
```

This will select the text from the beginning of the line to the end of the first sentence. Press `v` again to cancel visual mode. Next, return to the beginning line 3 and type:

```
v)
```

to select the first sentence. Cancel visual mode again and type:

```
v}
```

to select the entire paragraph (any block of text delimited by a blank line). Pressing `}` again extends the selection to the next paragraph.

Text Object Selection

In addition to the traditional vi movement commands, vim adds a related feature called *text object selection*. These commands only work in conjunction with operators. These commands are:

```
a   Select entire (all) text object.  
i   Select interior (in) of text object.
```

The text objects are:

```
w   Word  
s   Sentence  
p   Paragraph  
t   Tag block (such as <aaa>...</aaa> used in HTML)  
[   [ enclosed block  
(   ( enclosed block (b can also be used)  
{   { enclosed block (B can also be used)  
"   " quoted string  
'   ' quoted string
```

The way these work is very interesting. If we place our cursor on a word for example, and type:

```
caw
```

(short for “change all word”), vim selects the entire word, deletes it, and switches to insert mode. Text objects work with visual mode too. Try this: move to line 11 and place the cursor inside the quoted string and type:

```
vi"
```

The interior of the quoted string will be selected. If we instead type:

```
va"
```

the entire string including the quotes is selected.

Help topics: motion.txt text-objects

Text Formatting

Let's say we wanted to add a license header to the beginning of our script. This would consist of a comment block near the top of the file that includes the text of the copyright notice.

We'll move to line 3 of our script and add the text, but before we start, let's tell vim how long we want the lines of text to be. First we'll ask vim what the current setting is:

```
:set tw?
```

Vim should respond:

```
textwidth=0
```

"tw" is short for `textwidth`, the length of lines setting. A value of zero means that vim is not enforcing a limit on line length. Let's set `textwidth` to another value:

```
:set tw=75
```

Vim will now wrap lines (at word boundaries) when the length of a line exceeds this value.

Formatting Paragraphs

Normally, we wouldn't want to set a text width while writing code (though keeping line length below 80 characters is a good practice), but for this task it will be useful.

So let's add our text. Type this in:

```
# This program is free software: you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by the
# Free Software Foundation, either version 3 of the License, or (at your
# option) any later version.

# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
# Public License at <http://www.gnu.org/licenses/> for more details.
```

Notice the magic of vim as we type. Each time the length of the line reaches the text width, vim automatically starts a new line including, the comment symbol. While the filetype is set for shell scripting, vim understands certain things about shell syntax and tries to help. Very handy.

Now let's say we were not happy with the length of these lines, or that we have edited the text in such a way that some of the lines are either too long or too short to maintain our well-formatted text. Wouldn't be great if we could reformat our comment block? Well, we can. Very easily, in fact.

To demonstrate, let's change the text width to 65 characters:

```
:set tw=65
```

Now place the cursor inside the comment block and type:

```
qqip
```

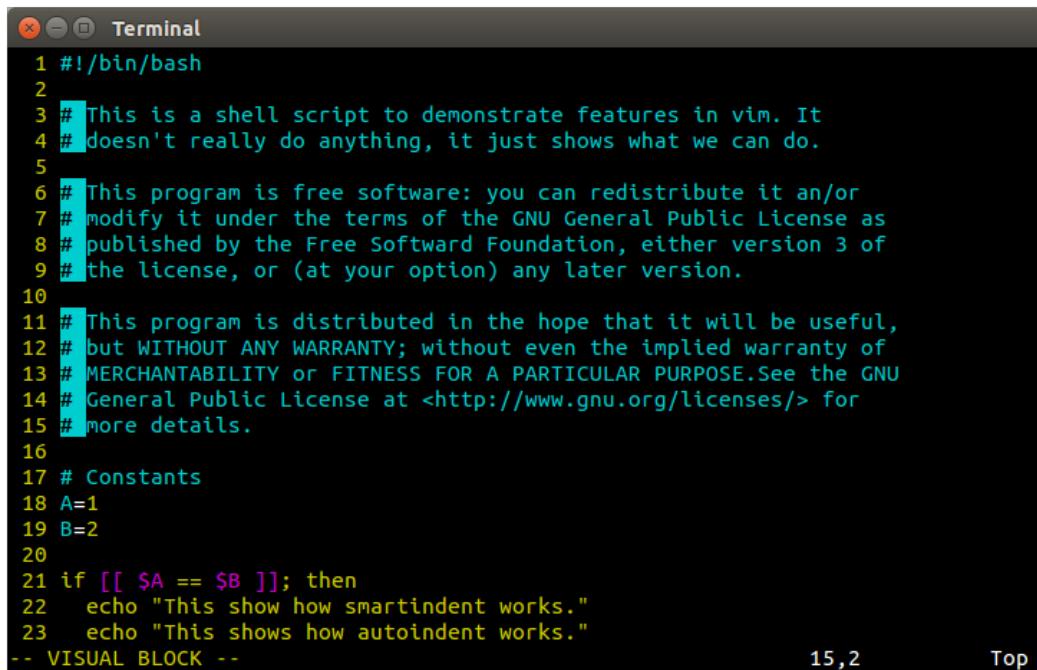
(meaning “format in paragraph”) and watch what happens. Presto, the block is reformatted to the new text width! A little later, we will show how to reduce this four key sequence down to a single key.

Comment Blocks

There is a fun trick we can perform on this comment block. When we write code, we frequently perform testing and debugging by commenting out sections. Vim makes this process pretty easy. To try this out, let’s first remove the commenting from our block. We will do this by using visual mode to select a block. Place the cursor on the first column of the first line of the comment block, then enter visual mode:

```
Ctrl-v
```

Then, move the cursor right one column and then down to the bottom of the block.



The screenshot shows a terminal window titled "Terminal" running a bash script. The script contains several multi-line comments. The first comment spans lines 3 to 5. The second spans lines 6 to 9. The third spans lines 11 to 15. The fourth spans lines 17 to 20. The fifth spans lines 21 to 23. The cursor is positioned at the start of the first line of the first comment block. The status bar at the bottom right shows "15,2" and "Top".

```
1#!/bin/bash
2
3# This is a shell script to demonstrate features in vim. It
4# doesn't really do anything, it just shows what we can do.
5
6# This program is free software: you can redistribute it and/or
7# modify it under the terms of the GNU General Public License as
8# published by the Free Software Foundation, either version 3 of
9# the license, or (at your option) any later version.
10
11# This program is distributed in the hope that it will be useful,
12# but WITHOUT ANY WARRANTY; without even the implied warranty of
13# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
14# General Public License at <http://www.gnu.org/licenses/> for
15# more details.
16
17# Constants
18A=1
19B=2
20
21if [[ $A == $B ]]; then
22    echo "This shows how smartindent works."
23    echo "This shows how autoindent works."
-- VISUAL BLOCK --
```

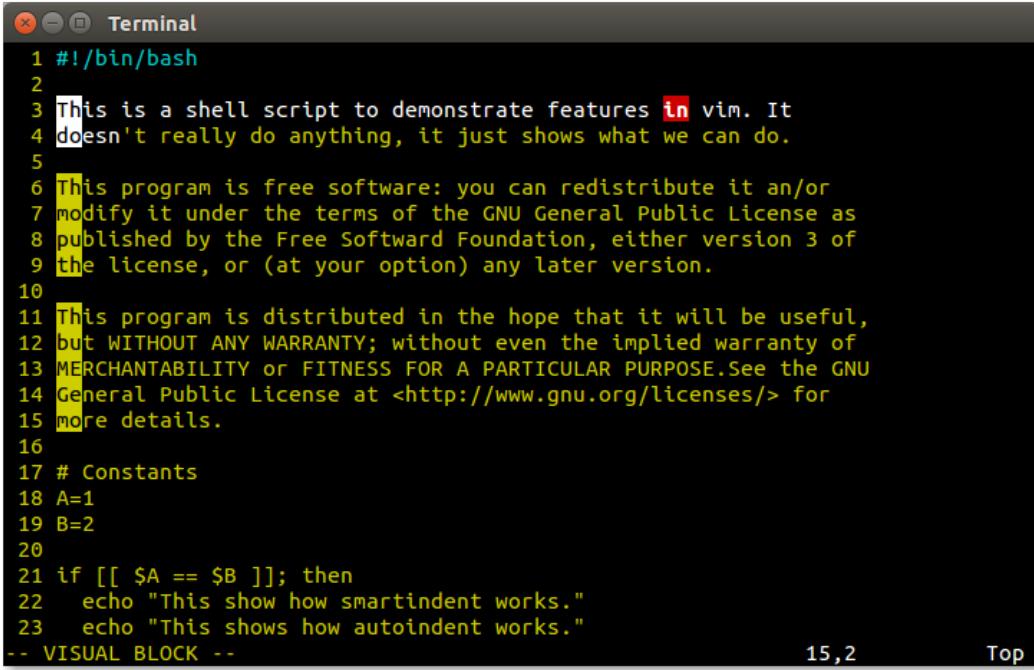
Visual block select

Next, type:

```
d
```

This will delete the contents of the selected area. Now our block is uncommented.

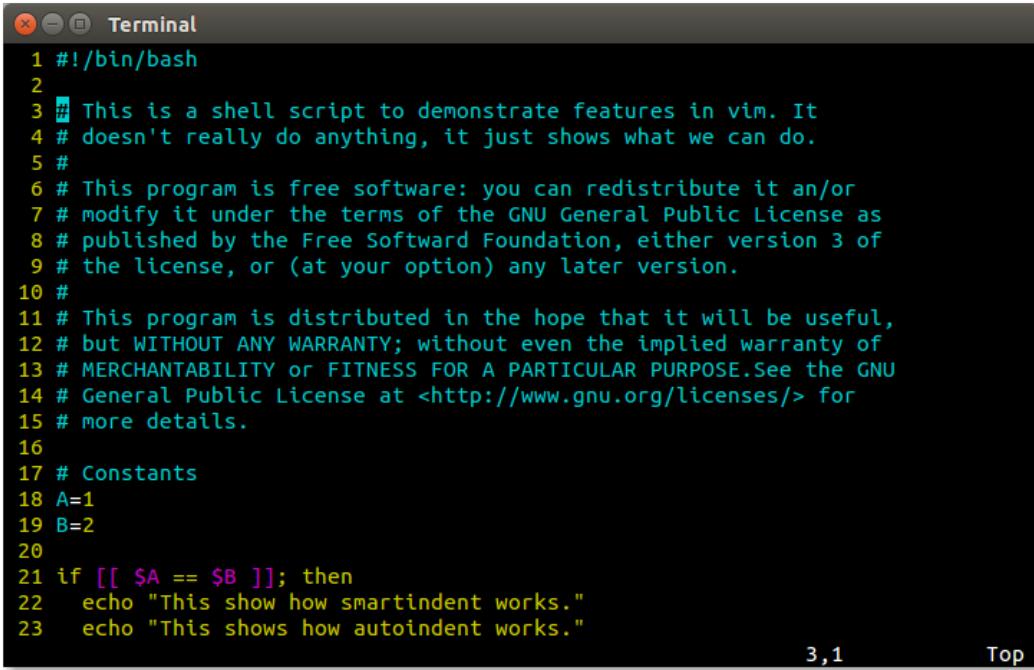
To comment the block again, move the cursor to the first character of the block and, using visual block selection, select the first 2 columns of the block.



```
1 #!/bin/bash
2
3 # This is a shell script to demonstrate features in vim. It
4 doesn't really do anything, it just shows what we can do.
5
6 # This program is free software: you can redistribute it an/or
7 modify it under the terms of the GNU General Public License as
8 published by the Free Software Foundation, either version 3 of
9 the license, or (at your option) any later version.
10
11 # This program is distributed in the hope that it will be useful,
12 but WITHOUT ANY WARRANTY; without even the implied warranty of
13 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
14 General Public License at <http://www.gnu.org/licenses/> for
15 more details.
16
17 # Constants
18 A=1
19 B=2
20
21 if [[ $A == $B ]]; then
22   echo "This shows how smartindent works."
23   echo "This shows how autoindent works."
-- VISUAL BLOCK --                                15,2          Top
```

Column selection

Next, enter insert mode using `Shift-i` (command to insert at the beginning of the line), then type the `#` symbol followed by a space. Finally, press the `Esc` key twice. Vim will insert the `#` symbol and space into each line of the block.



```
1 #!/bin/bash
2
3 # This is a shell script to demonstrate features in vim. It
4 # doesn't really do anything, it just shows what we can do.
5 #
6 # This program is free software: you can redistribute it an/or
7 # modify it under the terms of the GNU General Public License as
8 # published by the Free Software Foundation, either version 3 of
9 # the license, or (at your option) any later version.
10 #
11 # This program is distributed in the hope that it will be useful,
12 # but WITHOUT ANY WARRANTY; without even the implied warranty of
13 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
14 # General Public License at <http://www.gnu.org/licenses/> for
15 # more details.
16
17 # Constants
18 A=1
19 B=2
20
21 if [[ $A == $B ]]; then
22   echo "This shows how smartindent works."
23   echo "This shows how autoindent works."
3,1          Top
```

Completed block

Case Conversion

Sometimes, we need to change text from upper to lower case and vice versa. vim has the following case conversion commands:

```
~      Toggle the case of the current character
gU     Convert text to upper case
gu     Convert text to lower case
```

Both the `gU` and `gu` commands can be applied to text selected in visual mode or used in conjunction with either movement commands or text object selections. For example:

```
gUi$   Convert the current sentence to upper case
guf:   Convert text from the cursor position to the next ':'
           character on the current line
```

File Format Conversion

Once in a while, we are inflicted with a text file that was created on a DOS/Windows system. These files will contain an extra carriage return at the end of each line. Vim will indicate this after loading the file by displaying a “DOS” message at the bottom of the editing window. To correct this annoying condition, do the following:

```
:set fileformat=unix
:w
```

The file will be rewritten in the correct format.

Help topics: 'textwidth' gq 'fileformat' ~ gu gU

Macros

Text editing sometimes means we get stuck with a tedious repetitive editing task where we do the same set of operations over and over again. This is the bane of every computer user. Fortunately, vim provides us a way to record a sequence of operations we can later playback as needed. These recordings are called *macros*.

To create a macro, we begin recording by typing `q` followed by a single letter. The character typed after the `q` becomes the name of the macro. After we start recording, everything we type gets stored in the macro. To conclude recording, we type `q` again.

To demonstrate, let's consider our comment block again. To create a macro that will remove a comment symbol from the beginning of the line, we would do this: move to the first line in the comment block and type the following command:

```
qa^xxjq
```

Let's break down what this sequence does:

```
qa      Start recording macro "a"
^      Move to the first non-whitespace character in the line
xx     Delete the first two characters under the cursor
j      Move down one line
```

```
q      End recording
```

Now that we have removed the comment symbol from the first line and our cursor is on the second line, we can replay our macro by typing:

```
@a
```

The recorded sequence will be performed. To repeat the macro on succeeding lines, we can use the repeat last macro command which is:

```
@@
```

Or we could precede the macro invocation with a count as with other commands. For example, if we type:

```
5@a
```

the macro will be repeated 5 times.

We can undo the effect of the macro by repeatedly typing:

```
u
```

One nice thing about macros is that vim remembers them. Each time we exit vim, the current macro definitions are stored and ready for reuse the next time we start another editing session.

Help topics: q @

Registers

We are no doubt familiar with the idea of copying and pasting in text editors. With vim, we know `y` performs a yank (copy) of the selected text, while `p` and `P` each paste text at the current cursor location. The way vim does this involves the use of *registers*.

Registers are named areas of memory where vim stores text. We can think of them as a series of string variables. Vim uses one particular set to store text that we delete, but there are others that we can use to store text and restore it as we desire. It's like having a multi-element clipboard.

To refer to a register, we type "`"` followed by a lowercase letter or a digit (though these have a special use), for example:

```
"a
```

refers to the register named "a". To place something in the register, we follow the register with an operation like "yank to end of the line":

```
"ay$
```

To recall the contents of a register, we follow the name of the register with a paste operation like so:

```
"ap
```

Using registers enables us to place many chunks of text into our clipboard at the same time. But even without consciously trying to use registers, vim is using them while we perform deletes and yanks.

As we mentioned earlier, the registers named 0-9 have a special use. When we perform ordinary yanks and deletes, vim places our latest yank in register 0 and our last nine deletes in registers 1-9. As we continue to make deletions, vim moves the previous deletion to the next number, so register 1 will contain our most recent deletion and register 9 the oldest.

Knowing this allows us to overcome the problem of performing a yank and then a delete and losing the text we yanked (a common hazard when using vim). We can always recall the latest yank by referencing register 0.

To see the current contents of the registers we can use the command:

```
:reg
```

Help topics: " :registers

Insert Sub-Modes

While it's not obvious, vim has a set of commands inside of insert mode. Most of these commands invoke some form of automatic completion to make our typing faster. They're a little clumsy, but might be worth a try.

Automatically Complete Word Ctrl-n

Let's go to the bottom of our script file and enter insert mode to add a new line at the bottom. We want the line to read:

```
afunction && echo "It worked."
```

We start to type the first few characters ("afun") and press `Ctrl-n`. Vim should automatically complete the function name "afunction" after we press it. In those cases where vim presents us with more than one choice, use `Ctrl-n` and `Ctrl-p` to move up and down the list. Typing any another character, such as a space, to continue our typing will accept our selection and end the automatic completion. `Ctrl-e` can be used to exit the sub-mode immediately.

Insert Register Contents - Ctrl-r

Typing `Ctrl-r` followed by a single character register name will insert the contents of that register. Unlike doing an ordinary paste using `p` or `P`, a register insert honors text formatting and indentation settings such as `textwidth` and `autoindent`.

Automatically Complete Line - Ctrl-x Ctrl-l

Typing `Ctrl-x` while in insert mode launches a sub-mode of automatic completion features. A small menu will appear at the bottom of the display with a list of keys we can type to perform different completions.

If we have typed the first few letters of a line found in this or any other file that vim has open, typing `Ctrl-x Ctrl-l` will attempt to automatically complete the line, copying the line to the current location.

Automatically Complete Filename Ctrl-x Ctrl-f

This will perform filename completion. If we start the name of an existing path/file, we can type `Ctrl-x Ctrl-f` and vim will attempt to complete the name.

Dictionary Lookup - Ctrl-x Ctrl-k

If we define a dictionary (i.e., a sorted list of words), by adding this line to our configuration file:

```
setlocal dictionary=/usr/share/dict/words
```

which is the default dictionary on most Linux systems, we can begin typing a word, type `Ctrl-x Ctrl-k`, and vim will attempt to automatically complete the word using the dictionary. We will be presented with a list of words from which we can choose the desired entry.

Help topics: `i_n i_p i_x^l i_x^r i_x^f i_x^k 'dictionary'`

Mapping

Like many interactive command line programs, vim allows users to remap keys to customize vim's behavior. It has a specific command for this, `map`, that allows a key to be assigned the function of another key or a sequence of keys. Further, vim allows us to say that a key is to be remapped only in a certain mode, for example only in normal mode but not in insert nor command modes.

Before we go on, we should point out that use of the `map` command is discouraged. It can create nasty side effects in some situations. Vim provides another set of mapping commands that are safer to use.

Earlier, we looked at the paragraph reformatting command sequence `gqip`, which means "format in paragraph." To demonstrate a useful remapping, we will map the `Q` key to generate this sequence. We can do this by entering:

```
:nnoremap Q gqip
```

After executing this command, pressing the `Q` key in normal mode will cause the normal mode sequence `gqip` to be performed.

The `nnoremap` command is one of the `noremap` commands, the safe version of `map` command. The members of this family include:

```
noremap      Map key regardless of mode  
nnoremap    Map normal mode key  
inoremap    Map insert mode key  
cnoremap    Map command mode key
```

Most of the time we will be remapping normal mode keys, so the `nnoremap` command will be the used most often. Here is another example:

```
:nnoremap S :split<Return>
```

This command maps the `S` key to enter command mode, type the `split` command and a carriage return. The “`<Return>`” is called a *key notation*. For non-printable characters, vim has a representation that can be used to indicate the key when we specifying mapping. To see the entire list of possible codes, enter:

```
:h key-notation
```

So how do we know which keys are available for remapping assignment? As vim uses almost every key for something, we have to make a judgment call as to what native functionality we are willing to give up to get the mapping we want. In the case of the `Q` key, which we used in our first example, it is normally used to invoke ex mode, a very rarely used feature. There are many such cases in vim; we just have to be selective. It is best to check the key first by doing something like:

```
:h Q
```

to see how a key is being used before we apply our own mapping.

To make mappings permanent, we can add these mapping commands to our `.vimrc` file:

```
nnoremap Q gqip  
nnoremap S :split<Return>
```

Help topics: `:map key-notation`

Snippets

Mapping is not restricted to single characters. We can use sequences too. This is often helpful when we want to create a number of easily remembered, related commands of our own design. Take for example, inserting boilerplate text into a document. If we had a collection of these snippets, we might want to uniquely name them but have a common structure to the name for easily recollection.

We added the GPL notice to the comment block at the beginning of our script. As this is rather tedious to type, and we might to use it again, it makes a good candidate for being a snippet.

To do this, we'll first go out to the shell and create a directory to store our snippet text files. It doesn't matter where we put the snippet files, but in the interest of keeping all the vim stuff together, we'll put them with our other vim-related files.

```
:sh  
[me@linuxbox ~]$ mkdir ~/.vim/snippets  
[me@linuxbox ~]$ exit
```

Next, we'll copy the license by highlighting the text in visual mode and yanking it. To create the snippet file, we'll open a new buffer:

```
:e ~/.vim/snippets/gpl.sh
```

Thus creating a new file called gpl.sh. Finally, we'll paste the copied text into our new file and save it:

```
p  
:w
```

Now that we have our snippet file in place, we are ready to define our mapping:

```
:nnoremap ,GPL :r ~/.vim/snippets/gpl.sh<Return>
```

We map “,GPL” to a command that will cause vim to read the snippet file into the current buffer. The leading comma is used as a *leader character*. The comma is a rarely used command that is usually safe to remap. Using a leader character will reduce the number of actual vim commands we have to remap if we create a lot of snippets.

As we add mappings, it's useful to know what they all are. To display a list of mappings, we use the :map command followed by no arguments:

```
:map
```

Once we are satisfied with our remapping, we can add it to one of our vim configuration files. If we want it to be global (that is, it applies to all types of files), we could put it in our .vimrc file like this:

```
nnoremap ,GPL :r ~/.vim/snippets/gpl.sh<Return>
```

If, on the other hand, we want it to be specific to a particular file type, we would put it in the appropriate file such as ~/.vim/ftplugin/sh.vim like this:

```
nnoremap <buffer> ,GPL :r ~/.vim/snippets/gpl.sh<Return>
```

In this case, we add the special argument <buffer> to make the mapping local to the current buffer containing the particular file type.

Help topics: :map <buffer>

Finishing Our Script

With all that we have learned so far, it should be pretty easy to go ahead and finish our script:

```
#!/bin/bash
```

```

# -----
# This is a shell script to demonstrate features in vim. It
# doesn't really do anything, it just shows what we can do.
#
# This program is free software: you can redistribute it an/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of
# the license, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License at <http://www.gnu.org/licenses/> for
# more details.
# -----
#
# -----
# Constants
# -----
A=1
B=2

# -----
# Functions
# -----
afunction() {
    cmd1
    cmd2
}

# -----
# Main Logic
# -----
if [[ $A == $B ]]; then
    echo "This shows how smartindent works."
    echo "This shows how autoindent works."
    echo "A and B match."
else
    echo "A and B do not match."
fi

if [[ -e file ]]; then
    cmd1
    cmd2
fi

```

Using External Commands

Vim is able to execute external commands and add the result to the current buffer or to filter a text selection using an external command.

Loading Output From a Command Into the Buffer

Let's edit an old friend. If we don't have a copy to edit, we can make one. First we'll open a buffer:

```
:e dir-list.txt
```

Next, we'll load the buffer with some appropriate text:

```
:r ! ls -l /usr/bin
```

This will read the results of the specified external command into our buffer.

Running an External Command on the Current File

Let's save our file and then run an external command on it:

```
:w  
:! wc -l %
```

Here we tell vim to execute the `wc` command on the current file `dir-list.txt`. This does not affect the current buffer, just the file when we specify it with the `%` symbol.

Using an External Command to Filter the Current Buffer

Let's apply a filter to the text. To do this, we need to select some text. The easiest way to do this is with visual mode:

```
ggVG
```

This will move the cursor to the beginning of the file and enter visual mode then move to the end of the file, thus selecting the entire buffer.

We'll filter the selection to remove everything except lines containing the string "zip". When we start entering a command after performing a visual selection, the presence of the selection will be indicated this way:

```
: '<, >
```

This actually signifies a range. We could just as easily specify a pair of line numbers such as 1, 100 instead. To complete our command, we add our filter:

```
: '<, > ! grep zip
```

We are not limited to a single command. We can also specify pipelines, for example:

```
: '<, > ! grep zip | sort
```

After running this command, our buffer contains a small selection of files, each containing the letters "zip" in the name.

Help topics: : ! filter

File System Management and Navigation

We know that we can load files into vim by specifying them on the command line when we initially invoke vim, and that we can load files from within vim with the :edit and :read commands. But vim also provides more advanced ways of working with the file system.

netrw

When we load the filetype plugin (as we have set up our .vimrc file to do), vim also loads another plugin called *netrw*. This plugin can, as its name suggests, read and write files from the local file system and from remote systems over the network. In this adventure, we're going concern ourselves with using netrw as a local file browser.

To start the browser in the current window, we use the :Ex (short for :Explore) command. To start the browser in a split window, we use the amusingly named :Sex (short for :Sexplore) command. The browser looks like this:

```
" =====
" Netrw Directory Listing                               (netrw v145)
" /home/me
"   Sorted by      name
"   Sort sequence: [\/]$,\<core\%(\.\d\+)\)=\>,\.h$,\.c$,\.cpp$,\~\=\\*$,*,\.o$,\.
"   Quick Help: <F1>:help  - :go up dir  D:delete  R:rename  s:sort-by  x:exec
" =====
[./
/byobu/
.cache/
.config/
.local/
.vim/
Documents/
Pictures/
bin/
.bash_history
.bash_logout
.bashrc
.lessht
.profile
.screenrc
.viminfo
```

File browser

At the top, we have the banner which provides some clues to the browser's operation, followed by a vertical list of directories and files. We can toggle the banner on and off with Shift-i and cycle through available listing views by pressing the i key. The sort order (name, time, size) may be changed with s key.

Using the browser is easy. To select a file or directory, we can use the up and down arrows (or `Ctrl-p` and `Ctrl-n`) to move the cursor. Pressing Enter will open the selected file or directory.

:find

The `:find` command loads a file by searching a path variable maintained by vim. With `:find` we can specify a partial file name, and vim will attempt to locate the file and automatically complete the name when Tab key is pressed.

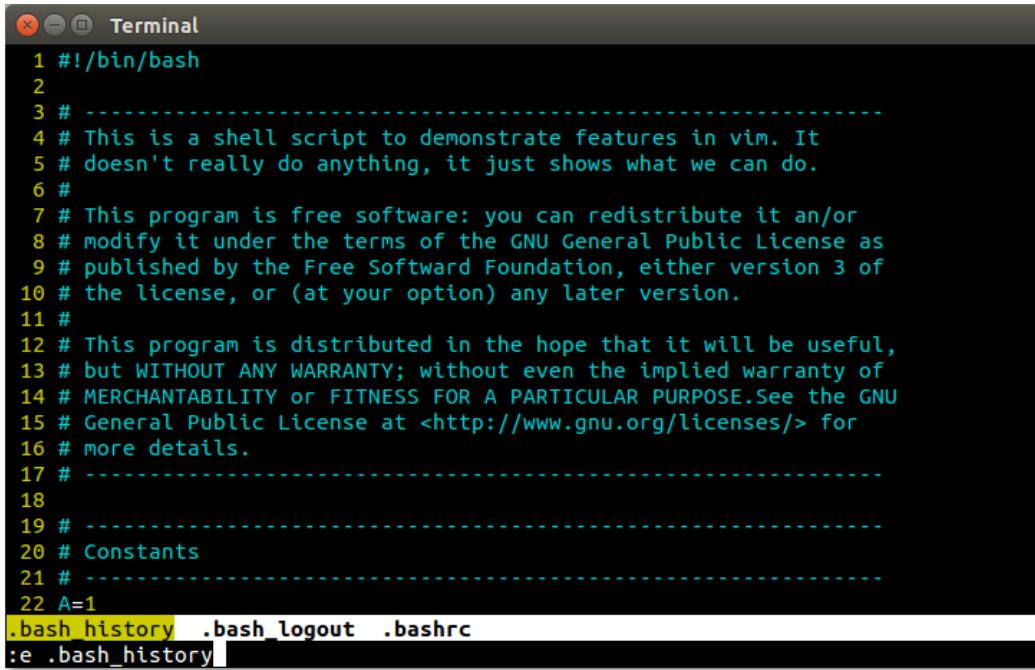
The action of the `:find` command can be enhanced if the characters “`**`” are appended to the end of the path. The best way to do this is:

```
:set path+=**
```

Adding this to the path allows `:find` to search directories recursively. For example, we could change the current working directory to the top of a project’s source file tree and use `:find` to load any file in the entire tree.

wildmenu

Another cool enhancement we can apply is the *wildmenu*. This is a highlighted bar that will appear above the command line when we are entering file names. The word “wild” in the name refers to use of the “wild” key, by default the Tab key. When the wild key is pressed, automatic completion is attempted with the list of possible matches displayed in the wildmenu. Using the left and right arrow keys (or `Ctrl-p` and `Ctrl-n`) allows us to choose one of the displayed items.



A screenshot of a terminal window titled "Terminal". The window displays a shell script with comments about its license and purpose. The script includes lines for constants and a variable assignment. At the bottom of the script, there are three file names: ".bash_history", ".bash_logout", and ".bashrc". Below the script, the command ":e .bash_history" is entered and highlighted in yellow.

The wildmenu

We can turn on the wildmenu with this command:

```
:set wildmenu
```

Opening Files Named in a Document

If the document we are editing contains a file name, we can open that file by placing the cursor on the file name and typing either of these commands:

```
gf      Open file name under cursor  
^w^f    Open file name under cursor in new window
```

Help topics: netrw :find 'path' 'wildmenu' gf ^w^f

One Does Not Live by Code Alone

While vim is most often associated with writing code of all sorts, it's good at writing ordinary prose as well. Need proof? All of the adventures were written using vim running on a Raspberry Pi!

We can configure vim to work well with text by creating a file for the text file type in the `~/.vim/ftplugin` directory:

```
"### ~/.vim/ftplugin/text.vim  
setlocal textwidth=75  
setlocal tabstop=4  
setlocal shiftwidth=4  
setlocal expandtab
```

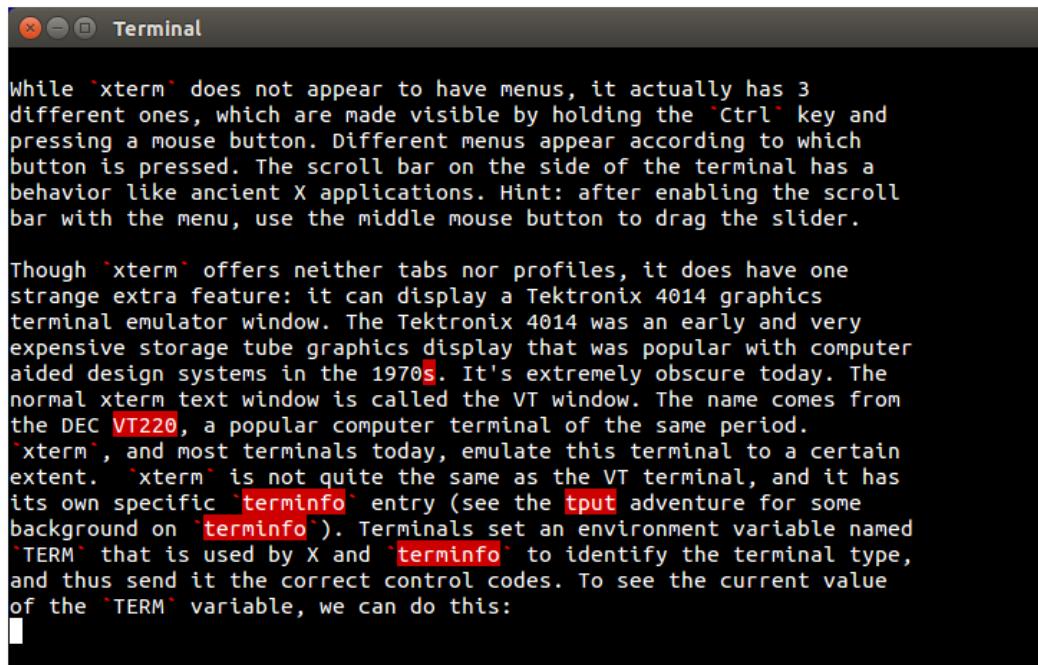
This configuration will automatically wrap lines at word boundaries once the line length exceeds 75 characters, and will set tabs to 4 spaces wide. Remember that when `textwidth` is non-zero, vim will automatically constrain line length, and we can use the `gqip` command to reformat paragraphs to the specified width.

Spell Checking

When we write text, it's handy to perform spell checking while we type. Fortunately, vim can do this, too. If we add the following lines to our `text.vim` file, vim will help fix those pesky spelling mistakes:

```
setlocal spelllang=en_us  
setlocal dictionary=/usr/share/dict/words  
setlocal spell
```

The first line defines the language for spell checking, in this case US English. Next, we specify the dictionary file to use. Most Linux distributions include this list of words, but other dictionary files can be used. The final line turns on the spell checker. When active, the spell checker highlights misspelled words (that is, any word not found in the dictionary) as we type.



```
While `xterm` does not appear to have menus, it actually has 3 different ones, which are made visible by holding the 'Ctrl' key and pressing a mouse button. Different menus appear according to which button is pressed. The scroll bar on the side of the terminal has a behavior like ancient X applications. Hint: after enabling the scroll bar with the menu, use the middle mouse button to drag the slider.

Though `xterm` offers neither tabs nor profiles, it does have one strange extra feature: it can display a Tektronix 4014 graphics terminal emulator window. The Tektronix 4014 was an early and very expensive storage tube graphics display that was popular with computer aided design systems in the 1970s. It's extremely obscure today. The normal xterm text window is called the VT window. The name comes from the DEC VT220, a popular computer terminal of the same period.
`xterm`, and most terminals today, emulate this terminal to a certain extent. `xterm` is not quite the same as the VT terminal, and it has its own specific `terminfo` entry (see the tput adventure for some background on `terminfo`). Terminals set an environment variable named `TERM` that is used by X and `terminfo` to identify the terminal type, and thus send it the correct control codes. To see the current value of the `TERM` variable, we can do this:
```

Highlighted misspellings

Correcting misspelled words is pretty easy. Vim provides the following commands:

<code>]s</code>	Next misspelled word
<code>[s</code>	Previous misspelled word
<code>z=</code>	Display suggested corrections
<code>zg</code>	Add word to personal dictionary

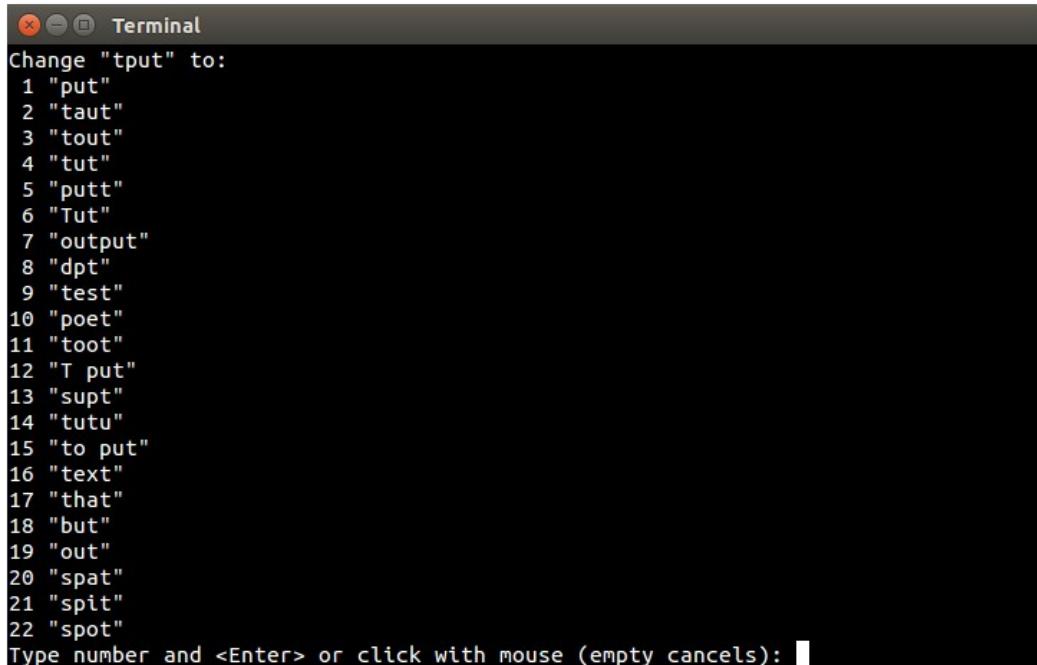
To correct a misspelling, we place the cursor on the highlighted word and type:

```
z=
```

Vim will display a list of suggested corrections and we choose from the list. It is also possible to maintain a personal dictionary of words not found in the main dictionary, for example specialized technical terms. Vim creates the personal dictionary automatically (in `~/.vim/spell`) and words are added to it when we place the cursor on the highlighted word and type:

```
zg
```

Once the word is added to our personal dictionary it will no longer be marked as misspelled by the spelling checker.



A screenshot of a terminal window titled "Terminal". The window displays a list of suggested corrections for the word "tput". The suggestions are numbered from 1 to 22. The list includes: 1 "put", 2 "taut", 3 "tout", 4 "tut", 5 "putt", 6 "Tut", 7 "output", 8 "dpt", 9 "test", 10 "poet", 11 "toot", 12 "T put", 13 "supt", 14 "tutu", 15 "to put", 16 "text", 17 "that", 18 "but", 19 "out", 20 "spat", 21 "spit", 22 "spot". At the bottom of the list, there is a prompt: "Type number and <Enter> or click with mouse (empty cancels): █".

Suggested corrections

Help topics: 'spelllang' 'spell'

More .vimrc Tricks

Before we go, there are a few more features we can add to our `.vimrc` file to juice things up a bit. The first one:

```
set laststatus=2
```

This will cause vim to display a status bar near the bottom of the display. It will normally appear when more than one window is open (`laststatus=1`), but changing this value to 2 causes it to always be displayed regardless of the number of windows. Next, we have:

```
set ruler
```

will display the cursor position (row, column, relative %) in the window status bar. Handy for knowing where we are within a file.

Finally, we'll add mouse support (not that we should ever use a mouse ;-):

```
if has('mouse')
  set mouse=a
endif
```

This will activate mouse support if vim detects a mouse. Mouse support allows us to position the cursor, switching windows if needed. It works in visual mode too.

Help topics: 'laststatus' 'ruler' 'mouse'

Summing Up

We can sometimes think of vim as being a metaphor for the command line itself. Both are arcane, vast, and capable of many amazing feats. Despite its ancient ancestry, vim remains a vital and popular tool for developers and administrators.

Here are the final versions of our 3 configuration files:

```
"### ~/.vimrc
set nocompatible
filetype plugin on
nnoremap Q ggip
nnoremap S :split<Return>
set path+=**
set wildmenu
set spelllang=en_us
if has('mouse')
  set mouse=a
endif
set laststatus=2
set ruler

"### ~/.vim/ftplugin/sh.vim
setlocal number
colorscheme desert
syntax off
setlocal tabstop=2
setlocal shiftwidth=2
setlocal expandtab
setlocal softtabstop=2
setlocal autoindent
setlocal smartindent

"### ~/.vim/ftplugin/text.vim
colorscheme desert
setlocal textwidth=75
setlocal tabstop=4
setlocal shiftwidth=4
setlocal expandtab
setlocal complete=.,w,b,u,t,i
setlocal dictionary=/usr/share/dict/words
setlocal spell
```

We covered a lot of ground in this adventure and it will take some time for it to all sink in. The best advice was given back in TLCL. The only way to become a vim master is to “practice, practice, practice!”

Further Reading

Vim has a large and enthusiastic user community. As a result, there are many online help and training resources. Here are some that I found useful during my research for this adventure.

- The eternal struggle between tabs and spaces in indentation:
<https://www.jwz.org/doc/tabs-vs-spaces.html>
- List of key notations used when remapping keys:
<http://vimdoc.sourceforge.net/htmldoc/intro.html#key-notation>
- A concise tutorial on vim registers: <https://www.brianstorti.com/vim-registers/>
- Learn Vimscript the Hard Way is a detailed tutorial of the vim scripting language useful for customizing vim and even writing your own plugins:
<https://learnvimscriptthehardway.stevelosh.com>
- From the same source, a discussion of the leader key:
<https://learnvimscriptthehardway.stevelosh.com/chapters/06.html>
- Using external commands and the shell while inside vim:
<https://www.linux.com/training-tutorials/vim-tips-working-external-commands>
- Vim: you don't need NERDtree or (maybe) netrw <https://shapeshed.com/vim-netrw/#removing-the-banner>
- A tutorial on using the vim spell checker: <https://www.linux.com/training-tutorials/using-spell-checking-vim/>

Videos

There are also a lot of video tutorials for vim. Here are a few:

- How to Do 90% of What Plugins Do (With Just Vim):
<https://youtu.be/XA2WjJbmmoM>
- Let Vim do the Typing: <https://youtu.be/3TX3kV3TICU>
- Damian Conway, “More Instantly Better Vim” - OSCON 2013:
<https://youtu.be/aHm36-na4-4>
- vim + tmux - OMG!Code: <https://youtu.be/5r6yzFEXajQ>

11 source

Most programming languages permit programmers to specify external files to be included within their programs. This is often used to add “boilerplate” code to programs for such things as defining standard constants and referencing external library function definitions.

Bash (along with `ksh` and `zsh`) has a builtin command, `source`, that implements this feature. We looked at `source` briefly when we worked with the `.profile` and `.bashrc` files used to establish the shell environment.

In this adventure, we will look at `source` again and discover the ways it can make our scripts more powerful and easier to maintain.

To recap, `source` reads a specified file and executes the commands within it using the current shell. It works both with the interactive command line and within a script. Using the command line for example, we can reload the `.bashrc` file by executing the following command:

```
me@linuxbox: ~$ source ~/.bashrc
```

Note that the `source` command can be abbreviated by a single dot character like so:

```
me@linuxbox: ~$ . ~/.bashrc
```

When `source` is used on the command line, the commands in the file are treated as if they are being typed directly at the keyboard. In a shell script, the commands are treated as though they are part of the script.

Configuration Files

During our exploration of the Linux ecosystem, we have seen that many programs rely on configuration files. Most of these are simple text files just like our bash shell scripts. By using `source`, we can easily create configuration files for our shell scripts as well.

Consider this example. Let’s imagine that we have several computers on our network that need to get backed up on a regular basis and that a central backup server is used to store the files from these various systems. On each of the backup client systems we have a script called `back_me_up` that copies the files over the network. Let’s further imagine that each client system needs to back up a different set directories.

To implement this, we might define a constant in the `back_me_up` script like this:

```
BACKUP_DIRS="/etc /usr/local /home"
```

However, doing it this way will require that we maintain a separate version of the script for each client. This will make maintaining the script much more laborious, as any future improvement to the script will have to be applied to each copy of the script individually.

What's more, this list of directories might be useful to other programs, too. For example, we could have a file restoration script called `restore_me` that restores files from the backup server to the backup client system. If this were the case, we would then have twice as many scripts to maintain. A much better way handle this issue would be to create a configuration file to define the `BACKUP_DIR` constant and source it into our scripts at run time.

Here's how we could do it.

First, we will create a configuration file named `back_me_up.cfg` and place it somewhere sensible. Since the `back_me_up` and `restore_me` scripts are used on a system-wide basis (as would most backup programs), we will treat them like locally installed system resources. Thus, we would put them in the `/usr/local/sbin` directory and the configuration file in `/usr/local/etc`. The configuration file would contain the following:

```
# Configuration file for the back_me_up program  
BACKUP_DIRS="/etc /usr/local /home"
```

While our configuration file must contain valid shell syntax, since its contents are executed by the shell, it differs from a real shell script in two regards. First, it does not require a shebang to indicate which shell executes it, and second, the file does not need executable permissions. It only needs to be readable by the shell.

Next, we would add the following code to the `back_me_up` and `restore_me` scripts to source our configuration file:

```
CONFIG_FILE=/usr/local/etc/back_me_up.cfg  
  
if [[ -r "$CONFIG_FILE" ]]; then  
    source "$CONFIG_FILE"  
else  
    echo "Cannot read configuration file!" >&2  
    exit 1  
fi
```

Function Libraries

In addition to the configuration shared by both the `back_me_up` and `restore_me` scripts, there could be code shared between the two programs. For example, it makes sense to have a shared function to display error messages:

```
# -----  
# Send message to std error  
# Options: none  
# Arguments: 1 error_message  
# Notes: Use this function to report errors to standard error. Does  
# not generate an error status.  
# -----  
error_msg() {
```

```

printf "%s\n" "$1" >&2
}

```

How about a function that detects if the backup server is available on the network:

```

# -----
# Detect if the specified host is available
#   Options:    none
#   Arguments:  1 hostname
#   Notes:
# -----
ping_host() {

    local MSG="Usage: ${FUNCNAME[0]} host"
    local MSG2="${FUNCNAME[0]}: host $1 unreachable"

    [[ $# -eq 1 ]] || { error_msg "$MSG" ;return 1; }
    ping -c1 "$1" &> /dev/null || { error_msg "$MSG2"; return 1; }
    return 0
}

```

Another function both scripts could use checks that external programs we need for the scripts to run (like rsync) are actually installed:

```

# -----
# Check if function/application is installed
#   Options:    none
#   Arguments:  application...
#   Notes:      Exit status equals the number of missing functions/apps.
# -----
app_avail() {

    local MSG1="Usage: ${FUNCNAME[0]} app..."
    local MSG2
    local exit_status=0

    [[ $# -gt 0 ]] || { error_msg "$MSG1"; return 1; }
    while [[ -n "$1" ]]; do
        MSG2="Required program '$1' not available - please install"
        type -t "$1" > /dev/null || \
            { error_msg "$MSG2"; exit_status=$((exit_status + 1)); }
        shift
    done
    return "$exit_status"
}

```

To share these functions between the `back_me_up` and `restore_me` scripts, we could build a library of functions and source that library. As an example, we could put all the common code in a file called `/usr/local/lib/bmulib.sh` and add the following code to both scripts to source that file:

```

FUNCLIB=/usr/local/lib/bmulib.sh

if [[ -r "$FUNCLIB" ]]; then
    source "$FUNCLIB"
else
    echo "Cannot read function library!" >&2
    exit 1
fi

```

General Purpose Libraries

Since we hope to become prolific script writers, it makes sense over time, to build a library of common code that our future scripts could potentially use. When undertaking such a project, it's wise to write high quality functions for the library, as the code may get heavy use. It's important to test carefully, include a lot of error handling, and fully document the functions. After all, the goal here is to save time writing good scripts, so invest the time up front to save time later.

Let's Not Forget .bashrc

`source` can be a powerful tool for coordinating the configuration of small sets of machines. For large sets, there are more powerful tools, but `source` works fine if the job is not too big.

We've worked with the `.bashrc` file before and added things like aliases and a few shell functions. However, when we work with multiple systems (for example, a small network), it might be a good idea to create a common configuration file to align all of the systems. To demonstrate, let's create a file called `.mynetworkrc.sh` and place all of the common aliases and shell function we would expect on every machine. To use this file, we would add this one line of code to `.bashrc`:

```
[ [ -r ~/.mynetworkrc.sh ] ] && source ~/.mynetworkrc.sh
```

The advantage of doing it this way is that we won't have to cut and paste large sections of code every time we configure a new machine or perform an operating system upgrade. We just copy the `.mynetworkrc.sh` file to the new machine and add one line to `.bashrc`.

We can even go further and create a host-specific configuration file that the `.mynetworkrc.sh` file will `source`. This would be handy if we need to override something in `.mynetworkrc.sh` on a particular host. We can do this by creating a configuration file with a file name based on the system's host name. For example, if our system's host name is `linuxbox1` we could create a configuration file named `.linuxbox1rc.sh` and add this line of code to the `.mynetworkrc.sh` file:

```
[ [ -r ~/.$(hostname)rc.sh ] ] && source ~/.$(hostname)rc.sh
```

By using the `hostname` command we are able to build a file name that is specific to a particular host.

So, what could we put in our `.mynetworkrc.sh` file? Here are some ideas:

```
### Aliases ###

# Reload the .mynetworkrc.sh file. Handy after editing.
alias reload='source ~/.mynetworkrc.sh'

# Get a root shell
```

```

alias root='sudo -i'

# Print the size of a terminal window in rows and columns
alias term_size='echo "Rows=$(tput lines) Cols=$(tput cols)"'

### Functions ###

# Check to see if a specified host is alive on the network

ping_host() {

    local target

    if [[ -z "$1" ]]; then
        echo "Usage: ping_host host" >&2
        return 1
    fi
    target="$1"
    ping -c1 "$target" &> /dev/null || \
        { echo "Host '$target' unreachable." >&2; return 1; }
    return 0
}

# Display a summary of system health

status() {
    { # Display system uptime
        echo -e "\nuptime:"
        uptime

        # Display disk resources skipping snap's pseudo disks
        echo -e "\ndisk space:"
        df -h 2> /dev/null | grep -v snap
        echo -e "\nnodes:"
        df -i 2> /dev/null | grep -v snap
        echo -e "\nblock devices:"
        /bin/lsblk | grep -v snap

        # Display memory resources
        echo -e "\nmemory:"
        free -m

        # Display latest log file entries
        if [[ -r /var/log/syslog ]]; then # Ubuntu
            echo -e "\nsyslog:"
            tail /var/log/syslog
        fi
        if [[ -r /var/log/messages ]]; then # Debian, et al.
            echo -e "\nmessages:"
            tail /var/log/messages
        fi
        if [[ -r /var/log/journal ]]; then # Arch, others using systemd
            echo -e "\njournal:"
            journalctl | tail
        fi
    } | less
}

# Install a package from a distro repository
# Supports Ubuntu, Debian, Fedora, CentOS

```

```

install() {
    if [[ -z "$1" ]]; then
        echo "Usage: install package..." >&2
        return 1
    elif [[ "$1" == "-h" || "$1" == "--help" ]]; then
        echo "Usage: install package..."
        return
    elif [[ -x /usr/bin/apt ]]; then
        sudo apt update && sudo apt install "$@"
        return
    elif [[ -x /usr/bin/apt-get ]]; then
        sudo apt-get update && sudo apt-get install "$@"
        return
    elif [[ -x /usr/bin/yum ]]; then
        sudo yum install -y "$@"
    fi
}

# Perform a system update
# Supports Debian, Ubuntu, Fedora, CentOS, Arch

update() {
    if [[ -x /usr/bin/apt ]]; then # Debian, et al
        sudo apt update && sudo apt upgrade
        return
    elif [[ -x /usr/bin/apt-get ]]; then # Old Debian, et al
        sudo apt-get update && sudo apt-get upgrade
        return
    elif [[ -x /usr/bin/yum ]]; then # CentOS/Fedora
        # su -c "yum update"
        sudo yum update
        return
    elif [[ -x /usr/bin/pacman ]]; then # Arch
        sudo pacman -Syu
    fi
}

# Display distro release info (prints OS name and version)

version() {

    local s

    for s in os-release \
            lsb-release \
            debian_version \
            centos-release \
            fedora-release; do
        [[ -r "/etc/$s" ]] && cat "/etc/$s"
    done
}

```

Ever Wonder Why it's Called .bashrc?

In our various wanderings around the Linux file system, we have encountered files with names that end with the mysterious suffix “rc” like `.bashrc`, `.vimrc`, etc. Heck, many distributions have a bunch of directories in `/etc` named `rc`. Why is that? It’s a holdover from ancient Unix. Its original meaning was “run commands,” but it later became “run-

control.” A run-control file is generally some kind of script or configuration file that prepares an environment for a program to use. In the case of `.bashrc` for example, it’s a script that prepares a user’s bash shell environment.

Security Considerations and Other Subtleties

Even though sourced files are not directly executable, they do contain code that will be executed by anything that sources them. It is important, therefore, that permissions be set to allow writing only by their owners.

```
me@linuxbox:~$ sudo chmod 644 /usr/local/etc/back_me_up.cfg
```

If a sourced file contains confidential information (as a backup program might), set the permissions to 600.

While `bash`, `ksh`, and `zsh` all have the `source` builtin, `dash` and all other strictly POSIX compatible shells support only the single dot (.)

If the file name argument given to `source` does not contain a / character, the directories listed in the `PATH` variable are searched for the specified file. For security reasons, it’s probably not a good idea to rely on this. Always specify a explicit path name.

Another subtlety has to do with positional parameters. As `source` executes its commands in the current shell environment, this includes the positional parameters a script was given as well. This is fine in most cases; however, if `source` is used within a shell function and that shell function has its own positional parameters, `source` will ignore them and use the shell’s environment instead. To overcome this, positional parameters may be specified after the file name. Consider the following script:

```
#!/bin/bash

# foo_script

source ~/foo.sh

foo() {
    source ~/foo.sh
}

foo "string2"
```

`foo.sh` contains this one line of code:

```
echo $1
```

We expect to see the following output:

```
me@linuxbox: ~$ foo_script string1
string1
string2
```

But, what we actually get is this:

```
me@linuxbox: ~$ foo_script string1
string1
string1
```

This is because `source` uses the shell environment the script was given, not the one that exists when the function called. To correct this, we need to write our script this way:

```
#!/bin/bash

# foo_script

source ~/foo.sh

foo() {
    source ~/foo.sh "$1"
}

foo "string2"
```

By adding the desired parameter to the `source` command within the function `foo`, we are able to get the desired behavior. Yes, it's subtle.

Summing Up

By using `source`, we can greatly reduce the effort needed to maintain our bash scripts particularly when we are deploying them across multiple machines. It also allows us to effectively reuse code with function libraries that all of our scripts can share. Finally, we can use `source` to build much more capable shell environments for our day to day command line use.

Further Reading

- A Wikipedia article on the dot command from which the `source` builtin is derived: [https://en.wikipedia.org/wiki/Dot_\(command\)](https://en.wikipedia.org/wiki/Dot_(command))
- Another article about run-commands: https://en.wikipedia.org/wiki/Run_commands

12 Coding Standards Part 1: Our Own

Most computer programming is done by organizations and teams. Some programs are developed by lone individuals within a team and others by collaborative groups. In order to promote program quality, many organizations develop formal programming guidelines called *coding standards* that define specific technical and stylistic coding practices to help ensure code quality and consistency.

In this adventure, we’re going to develop our own shell script coding standard to be known henceforth as the *LinuxCommand Bash Script Coding Style Guide*. The Source adventure is a suggested prerequisite for this adventure.

Roaming around the Internet, we can find lots of articles about “proper” shell script standards and practices. Some are listed in the “Further Reading” section at the end of this adventure. While the scripts presented in TLCL do not follow any particular standard (instead, they present common practice from different historical perspectives), their design promotes several important ideas:

1. **Write cleanly and simply.** Look for the simplest and most easily understood solutions to problems.
2. **Use modern idioms, but be aware of the old ones.** It’s important that scripts fit within common practice so that experienced people can easily understand them.
3. **Be careful and defensive.** Follow Murphy’s Law: anything that can go wrong eventually will.
4. **Document your work.**
5. **There are lots of ways to do things, but some ways are better than others.**

Coding standards generally support the same goals, but with more specificity.

In reviewing the Internet’s take on shell scripting standards, one might notice a certain undercurrent of hostility towards using the shell as a programming medium at all. Why is this?

Most programmers don’t learn the shell at the beginning of their programming careers; instead, they learn it (haphazardly) after they have learned one or more traditional programming languages. Compared to most programming languages, the shell is an odd beast that seems, to many, a chaotic mess. This is largely due to the shell’s unique role as both a command line interpreter and a scripting language.

As with all programming languages, the shell has its particular strengths and weaknesses. The shell is good at solving certain kinds of problems, and not so good at others. Like all good artists, we need to understand the bounds of our medium.

What the Shell is Good At

- The shell is a powerful glue for connecting thousands of command line programs together into solutions to a variety of data processing and system administration problems.
- The shell is adept at *batch processing*. In the early days of computing, programs were not interactive; that is, they started, they carried out their tasks, and they ended. This style of programming dominated computing until the early 1960s when disk storage and virtual memory made timesharing and interactive programs possible. Those of us who remember MS-DOS will recall that it had a limp substitute for shell scripts called batch files.

What the Shell is Not So Good At

- The shell does not excel with programs requiring a lot of user interaction. Yes, the shell does have the `read` command and we could use `dialog`, but let's face it, the shell is not very good at this.
- The shell is not suitable for implementing algorithms requiring complex data structures. While the shell does have integers, strings, and one dimensional arrays (which can be associative), it doesn't support anything beyond that. For example, it doesn't have structures, enumerated or Boolean types, or even floating point numbers.

A Coding Standard of Our Own

Keeping the points above in mind, let's make our own shell script coding standard. It will be an amalgam of various published standards, along with a dash of the author's own humble wisdom on the subject. As the name implies, the *LinuxCommand Bash Script Coding Style Guide* coding standard will be very `bash` specific and as such, it will use some features that are not found in strictly POSIX compliant shells.

Script Naming, Location, and Permissions

Like other executables, shell script file names should not include an extension. Shared libraries of shell code which are not standalone executables should have the extension `.sh` if the code is portable across multiple shells (for example `bash` and `zsh`) or `.bash` if the code is `bash`-specific.

For ease of execution, scripts should be placed in a directory listed in the user's `PATH`. The `~/bin` directory is a good location for personal scripts as most Linux distributions support this out of the box. Scripts intended for use by all users should be located in the `/usr/local/bin` directory. System administration scripts (ones intended for the

superuser) should be placed in `/usr/local/sbin`. Shared code can be located in any subdirectory of the user's home directory. Shared code intended for use system wide should be placed in `/usr/local/lib` unless the shared code specifies only configuration settings, in which case it should be located in `/usr/local/etc`.

Executable code must be both readable and executable to be used, thus the permission setting for shell scripts should be 755, 750 or 700 depending on security requirements. Shared code need only be readable, thus the permissions for shared code should be 644, 640, or 600.

Structure

A shell script is divided into five sections. They are:

1. The shebang
2. The comment block
3. Constants
4. Functions
5. Program body

The Shebang

The first line of a script should be a shebang in either of the following forms:

```
#!/bin/bash
```

Or

```
#!/usr/bin/env bash
```

The second form is used in cases where the script is intended for use on a non-Linux system (such as macOS). The `env` command will search the user's `PATH` for a `bash` executable and launch the first instance it finds. Of the two forms, the first is preferred, as its results are less ambiguous.

The Comment Block

The first bit of documentation to appear in the script is the comment block. This set of comments should include the name of the script and its author, any necessary copyright and licensing information, a brief description of the script's purpose, and its command line options and arguments.

It is also useful to include version information and a revision history. Here is an example of a fully populated comment block. The exact visual style of the block is undefined and is left to the programmer's discretion.

```
# -----
# new_script - Bash shell script template generator
```

```

# Copyright 2012-2021, William Shotts <bshotts@users.sourceforge.net>

# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License at <http://www.gnu.org/licenses/> for
# more details.

# Usage: new_script [[-h|--help]
#                   new_script [-q|--quiet] [-s|--root] [script]]

# Revision history:
# 2021-04-02 Updated to new coding standard (3.5)
# 2019-05-09 Added support for shell scripting libraries (3.4)
# 2015-09-14 Minor cleanups suggested by Shellcheck (3.3)
# 2014-01-21 Minor formatting corrections (3.2)
# 2014-01-12 Various cleanups (3.1)
# 2012-05-14 Created
# -----

```

Constants

After the comment block, we define the constants used by the script. As we recall, constants are variables that have a fixed value. They are used to define values that are used in various locations in the script. Constants simplify script maintenance by eliminating hard coded values scattered throughout the code. By placing this section at the top, changes to the script can be made more easily.

There are two constants that every script should include. First, a constant that contains the name of the program, for example:

```
PROGNAME=${0##*/}
```

This value is useful for such things as help and error messages. In the example above, the name of the program is calculated from the first word on the command line that invoked the script (`$0`) with any leading path name stripped off. By calculating it this way, if the name of the script file ever changes, the constant will automatically contain the new name.

The second constant that should be included contains the script's version number. Like the `PROGNAME` constant, this value is useful in help and error messages.

```
VERSION="3.5"
```

Constants should also be defined for numerical values such as maximum or minimum limits used in calculations, as well as for the names of files or directories used by, or acted upon, by the script.

Functions

Function definitions should appear next in the script. They should follow an order that indicates their hierarchy; that is, functions should be ordered so they are defined before they are called. For example, if `func_b` calls `func_a`, function `func_a` should be defined ahead of `func_b`.

Functions should have the following form:

```
func_b {  
    local v1="$1"  
    local v2="$2"  
  
    command1  
    command2  
    return  
}
```

Positional parameters passed to a function should be assigned to local variables with descriptive names to clarify how the parameters are used. Unless there is a good reason, all functions should end with a `return` statement with an exit status as required.

Each function defined in a code library should be preceded by a short comment block that lists the function's name, purpose, and positional parameters.

Program Body

The final section of a script is the program body, where we get to do what we came here to do. In this section we handle such things as positional parameter processing, acquisition and validation of required input, processing, and output.

A common practice is to write the program body as a very simple abstraction of the program, so that the basic program flow is easy to understand and most of the fussy bits are placed in functions defined above the program body.

The end of the program body should include some means of housekeeping, to do such things as removing temporary files. All scripts should return a useful exit status.

Formatting and Visual Style

Good visual layout makes scripts easier to read and understand. It's important to make formatting consistent so it creates a reliable visual language for the code.

Line Length

In order to improve readability and to display properly in small terminal windows, line length should be limited to 80 characters. Line continuation characters should be used to break long lines, with each subsequent line indented one level from the first. For example:

```
printf "A really long string follows here: %s\n" \
      "Some really, really, really long string."
```

Indentation

Indentation should be done with spaces and never with tab characters. Most text editors can be set to use spaces for tabbing. In order for the maximum number of characters to be included on each line, each level of indentation should be 2 spaces.

Constant, Variable and Function Names

Constant names should be written in all caps. Variable and function names should be written in all lowercase. When a constant or variable name consists of multiple words, underscore characters should be used as the separator. Camel case (“camelCase”) should be avoided as it makes people think we’re one of those snooty Java programmers just learning to write shell scripts ;-)

Long Command Option Names

When using less common commands with multiple options (or more common commands with less common options), it is sometimes best to use the long form option names and split the option list across multiple lines.

```
# Use long option names to improve readability

rsync \
  --archive \
  --delete-excluded \
  --rsh=ssh \
  --one-file-system \
  --relative \
  --include-from="$INCLUDE_FILE" \
  --exclude-from="$EXCLUDE_FILE" \
  "$SOURCE" "$DESTINATION"
```

Pipelines

Pipelines should be formatted for maximum clarity. If a pipeline will fit on one line cleanly, it should be written that way. Otherwise, pipelines should be broken into multiple lines with one pipeline element per line.

```
# Short pipeline on one line

command1 | command2

# Long pipeline on multiple lines

command1 \
  | command2 \
  | command3 \
  | command4
```

Compound Commands

Here are the recommended formatting styles for compound commands;

```
# 'then' should appear on the same line as 'if'

if [[ -r ~/.bashrc ]]; then
    echo ".bashrc is readable."
else
    echo ".bashrc is not readable." >&2
    exit 1
fi

# Likewise, 'do' should appear on the same line as
# the 'while', 'until', and 'for' commands

while [[ -z "$str" ]]; do
    command1
    command2
done

for i in 1 2 3 4 5; do
    command1
    command2
done

# In a case statement, simple one-line commands can be
# formatted this way:

case s in
    1|one)
        command1 ;;
    2|two)
        command2 ;;
    3|three)
        command3 ;;
    *)
        command4 ;;
esac

# Multiple commands should be formatted this way

case s in
    1|one)
        command1
        command2
        ;;
    2|two)
        command3
        command4
        ;;
    3|three)
        command5
        command6
        ;;
    *)
        command7
        ;;
esac

# Logical compound commands using && and ||
```

```
command1 && short_command  
command2 \  
|| long_command "$param1" "$param2" "$param3"
```

Coding Practices

In order to achieve our goal of writing robust, easily maintained scripts, our coding standard recommends the following coding practices.

Commenting

Good code commenting is vital for script maintenance. The purpose of commenting is to explain vital facts about a program. If a script is to have any durability, we must anticipate that someone (even if it's just the author) will revisit the script at a later date and will need a refresher on how the script works. Do not comment code that is obvious and easily understood; rather, explain the difficult to understand parts. Rule of thumb: the more time a chunk of code takes to design and write, the more commenting it will likely need to explain it.

Function libraries are a special case. Each function in a library should be preceded by a comment block that documents its purpose and its positional parameters.

A common type of comment points to where future additions and changes are needed. These are called "todo" comments and are typically written like this:

```
# TODO Fix this routine so it can do this better
```

These comments begin with the string `TODO` so they can be found easily using a text editor's search feature.

Shell Builtins vs. External Programs

Where possible, use bash builtins rather than external commands. For example, the `basename` and `dirname` programs can be replaced by parameter expansions to strip leading or trailing strings from pathnames. Compared to external programs, shell builtins use fewer resources and execute much faster.

Variable Expansion and Quoting

Double quotes must be used to manage word splitting during parameter expansion. This is particularly important when working with filenames. Assume that every variable, parameter expansion, and command substitution may contain embedded spaces, newlines, etc. There are situations where quoting is not needed (for example, within `[[...]]`) but we use double quotes anyway, because it doesn't hurt anything and it's easier to always quote variables than remembering all the special cases where it is not required.

```
a="$var"
```

```
b="$1"
c=$(command1)
command2 "$file1" "$file2"
[[ -z "$str" ]] || exit 1
```

Contrary to some other coding standards, brace delimiting variables is required only when needed to prevent ambiguity during expansion:

```
a="Compat"
port="babily condit"
echo "${a}bility is proportional to desire."
```

Pathname Expansion and Wildcards

Since pathnames in Unix-like systems can contain any character except / and NULL, we need to take special precautions during expansion.

```
# To prevent filenames beginning with '-' from being interpreted
# as command options, always do this:
command1 ./*.txt

# Not this:
command1 *.txt
```

Here is a snippet of code that will prepend ./ to a pathname when needed.

```
# This will sanitize '$pathname'
[[ "$pathname" =~ ^[./].*\$ ]] || pathname="./$pathname"
```

[[...]] vs. [...]

Unless a script must run in a POSIX-compatible environment, use [[...]] rather than [...] when performing conditional tests. Unlike the [and test bash builtins, [[...]] is part of shell syntax, not a command. This means it can handle its internal elements (test conditions) in a more robust fashion, as pathname expansion and word splitting do not occur. Also, [[...]] adds some additional capabilities such as =~ to perform regular expression tests.

Use ((...)) for Integer Arithmetic

Use ((...)) in place of let or expr when performing integer arithmetic. The bash let builtin works in a similar way as ((...)) but its arguments often require careful quoting. expr is an external program and many times slower than the shell.

```
# Use (( ... )) rather than [[ ... ]] when evaluating integers
if (( i > 1 )); then
...
fi

while (( y == 5 )); do
...
done

# Perform arithmetic assignment
(( y = x * 2 ))
```

```
# Perform expansion on an arithmetic calculation
echo $(( i * 7 ))
```

printf vs. echo

In some cases, it is preferable to use `printf` over `echo` when parameter expansions are being output. This is particularly true when expanding pathnames. Since pathnames can contain nearly any character, expansions could result in command options, command sequences, etc.

Error Handling

The most important thing for a script to do, besides getting its work done, is making sure it's getting its work done successfully. To do this, we have to handle errors.

1. **Anticipate errors.** When designing a script, it is important to consider possible points of failure. Before the script starts, are all the necessary external programs actually installed on the system? Do the expected files and directories exist and have the required permissions for the script to operate? What happens the first time a script runs versus later invocations? The beginning of the program should include tests to ensure that all necessary resources are available.
2. **Do no harm.** If the script must do anything destructive, for example, deleting files, we must make sure that the script does only the things it is supposed to do. Test for all required conditions prior to doing anything destructive.
3. **Report errors and provide some clues.** When an error is detected, we must report the error and terminate the script if necessary. All error messages must be sent to standard error and should include useful information to aid debugging. A good way to do this is to use an error message function such as the one below:

```
error_exit() {  
    local error_message="$1"  
  
    printf "%s\n" "${PROGNAME}: ${error_message:-"Unknown Error"}" >&2  
    exit 1  
}
```

We can call the error message function to report an error like this:

```
command1 || error_exit "command1 failed in line $LINENO"
```

The shell variable `LINENO` is included in the error message passed to the function. This will contain the line number where the error occurred.

4. **Clean up the mess.** When we handle an error we need to make sure that we leave the system in good shape. If the script creates temporary files or performs some

operation that could leave the system in an undesirable state, provide a way to return the system to useful condition before the script exits.

Bash offers a setting that will attempt handle errors automatically, which simply means that with this setting enabled, a script will terminate if any command (with some necessary exceptions) returns a non-zero exit status. To invoke this setting, we place the command `set -e` near the beginning of the script. Several bash coding standards insist on using this feature along with a couple of related settings, `set -u` which terminates a script if there is an uninitialized variable, and `set -o PIPEFAIL` which causes script termination if any element in a pipeline fails.

Using these features is not recommended. It is better to design proper error handling and not rely on `set -e` as a substitute for good coding practices.

The *Bash FAQ #105* provides the following opinion on this:

`set -e` was an attempt to add “automatic error detection” to the shell. Its goal was to cause the shell to abort any time an error occurred, so you don’t have to put `|| exit 1` after each important command.

That goal is non-trivial, because many commands intentionally return non-zero. For example,

```
if [ -d /foo ]; then ...; else ...; fi
```

Clearly we don’t want to abort when the `[-d /foo]` command returns non-zero (because the directory does not exist) – our script wants to handle that in the `else` part. So the implementors decided to make a bunch of special rules, like “commands that are part of an `if` test are immune”, or “commands in a pipeline, other than the last one, are immune.”

These rules are extremely convoluted, and they still fail to catch even some remarkably simple cases. Even worse, the rules change from one Bash version to another, as bash attempts to track the extremely slippery POSIX definition of this “feature.” When a subshell is involved, it gets worse still – the behavior changes depending on whether bash is invoked in POSIX mode.

Command Line Options and Arguments

When possible, scripts should support both short and long option names. For example, a “help” feature should be supported by both the `-h` and `--help` options. Dual options can be implemented with code such as this:

```
# Parse command line
while [[ -n "$1" ]]; do
  case $1 in
    -h | --help)
```

```

    help_message
    graceful_exit
    ;;
-q | --quiet)
    quiet_mode=yes
    ;;
-s | --root)
    root_mode=yes
    ;;
--* | -*)
    usage > &2; error_exit "Unknown option $1"
    ;;
*)
    tmp_script="$1"
    break
    ;;
esac
shift
done

```

Assist the User

Speaking of “help” options, all scripts should include one, even if the script supports no other options or arguments. A help message should include the script name and version number, a brief description of the script’s purpose (as it might appear on a man page), and a usage message that describes the supported options and arguments. A separate usage function can be used for both the help message and as part of an error message when the script is invoked incorrectly. Here are some example usage and help functions:

```

# Usage message - separate lines for mutually exclusive options
# the way many man pages do it.
usage() {
    printf "%s\n" \
        "Usage: ${PROGNAME} [-h|--help]"
    printf "%s\n" \
        "          ${PROGNAME} [-q|--quiet] [-s|--root] [script]"
}

help_message() {
cat <<- _EOF_
${PROGNAME} ${VERSION}
Bash shell script template generator.

$(usage)

Options:

-h, --help      Display this help message and exit.
-q, --quiet     Quiet mode. No prompting. Outputs default script.
-s, --root      Output script requires root privileges to run.

_EOF_
}

```

Traps

In addition to a normal exit and an error exit, a script can also terminate when it receives a signal. For some scripts, this is an important issue because if they exit in an unexpected manner, they may leave the system in an undesirable state. To avoid this problem, we include traps to intercept signals and perform cleanup procedures before the scripts exits. The three signals of greatest importance are SIGINT (which occurs when Ctrl-c is typed) and SIGTERM (which occurs when the system is shut down or rebooted) and SIGHUP (when a terminal connection is terminated). Below is a set of traps to manage the SIGINT, SIGTERM, and SIGHUP signals.

```
# Trap signals
trap "signal_exit TERM" TERM HUP
trap "signal_exit INT" INT
```

Due to the syntactic limitations of the `trap` builtin, it is best to use a separate function to act on the trapped signal. Below is a function that handles the signal exit.

```
signal_exit() { # Handle trapped signals

    local signal="$1"

    case "$signal" in
        INT)
            error_exit "Program interrupted by user"
            ;;
        TERM)
            printf "\n%s\n" "$PROGNAME: Program terminated" >&2
            graceful_exit
            ;;
        *)
            error_exit "$PROGNAME: Terminating on unknown signal"
            ;;
    esac
}
```

We use a `case` statement to provide different outcomes depending on the signal received. In this example, we also see a call to a `graceful_exit` function that could provide needed cleanup before the script terminates.

Temporary Files

Wherever possible, temporary files should be avoided. In many cases, process substitution can be used instead. Doing it this way will reduce file clutter, run faster, and in some cases be more secure.

```
# Rather than this:
command1 > "$TEMPFILE"
.
.
.
command2 < "$TEMPFILE"

# Consider this:
command2 < <(command1)
```

If temporary files cannot be avoided, care must be taken to create them safely. We must consider, for example, what happens if there is more than one instance of the script running at the same time. For security reasons, if a temporary file is placed in a world-writable directory (such as `/tmp`) we must ensure the file name is unpredictable. A good way to create temporary file is by using the `mktemp` command as follows:

```
TEMPFILE=$(mktemp /tmp/"$PROGNAME".$$XXXXXXXXXX")
```

In this example, a temporary file will be created in the `/tmp` directory with the name consisting of the script's name followed by its process ID (PID) and 10 random characters.

For temporary files belonging to a regular user, the `/tmp` directory should be avoided in favor of the user's home directory.

ShellCheck is Your Friend

There is a program available in most distribution repositories called `shellcheck` that performs analysis of shell scripts and will detect many kinds of faults and poor scripting practices. It is well worth using it to check the quality of scripts. To use it with a script that has a shebang, we simply do this:

```
shellcheck my_script
```

ShellCheck will automatically detect which shell dialect to use based on the shebang. For shell script code that does not contain a shebang, such as function libraries, we use ShellCheck this way:

```
shellcheck -s bash my_library
```

Use the `-s` option to specify the desired shell dialect. More information about ShellCheck can be found at its website <http://www.shellcheck.net>.

Summing Up

We covered a lot of ground in this adventure, specifying a complete set of technical and stylistic features. Using this coding standard, we can now write some serious production-quality scripts. However, the complexity of this standard does impose some cost in terms of development time and effort.

In Part 2, we will examine a program from LinuxCommand.org called `new_script`, a bash script template generator that will greatly facilitate writing scripts that conform to our new coding standard.

Further Reading

Here are some links to shell scripting coding standards. They range from the lax to the obsessive. Reading them all is a good idea in order to get a sense of the community's

collective wisdom. Many are not bash-specific and some emphasize multi-shell portability, not necessarily a useful goal.

- *The Google Shell Style Guide* The coding standard for scripts developed at Google. It's among the most sensible standards.
<https://google.github.io/styleguide/shellguide.html>
- *Anyone Can Write Good Bash (with a little effort)*
<https://blog.yossarian.net/2020/01/23/Anybody-can-write-good-bash-with-a-little-effort>
- *Some Bash coding conventions and good practices* <https://github.com/icy/bash-coding-style>
- *Bash Style Guide and Coding Standard*
<https://lug.fh-swf.de/vim/vim-bash/StyleGuideShell.en.pdf>
- *Shell Script Standards* <https://engineering.vokal.io/Systems/sh.md.html>
- *Unix/Linux Shell Script Programming Conventions and Style*
http://teaching.idallen.com/cst8177/13w/notes/000_script_style.html
- *Scripting Standards* <http://ronaldbradford.com/blog/scripting-standards/>

Pages with advice on coding practices. Some have conflicting advice so *caveat emptor*.

- *Make Linux/Unix Script Portable With #!/usr/bin/env As a Shebang*
<https://www.cyberciti.biz/tips/finding-bash-perl-python-portably-using-env.html>
- *Good practices for writing shell scripts* <http://www.yoone.eu/articles/2-good-practices-for-writing-shell-scripts.html>
- *Why is printf better than echo?*
<https://unix.stackexchange.com/questions/65803/why-is-printf-better-than-echo>
- *Why doesn't set -e (or set -o errexit, or trap ERR) do what I expected?*
<http://mywiki.wooleedge.org/BashFAQ/105>
- *Bash: Error handling* https://fvue.nl/wiki/Bash:_Error_handling
- *Writing Robust Bash Shell Scripts* <https://www.davidpashley.com/articles/writing-robust-shell-scripts/>
- *Filenames and Pathnames in Shell: How to do it Correctly* A good page full of cautionary tales and advice on dealing with “funny” file and path names. A serious problem in Unix-like systems. Highly recommended.
<https://dwheeler.com/essays/filenames-in-shell.html>

- *1963 Timesharing: A Solution to Computer Bottlenecks* This YouTube video from MIT provides some historical perspective on the invention of interactive systems and the transition away from batch processing. The concepts presented here are still the basis of all modern computing. <https://youtu.be/Q07PhW5sCEk>

13 Coding Standards Part 2: new_script

In Part 1, we created a coding standard that will assist us when writing serious, production-quality scripts. The only problem is the standard is rather complicated, and writing a script that conforms to it can get a bit tedious. Any time we want to write a “good” script, we have to do a lot of rote, mechanical work to include such things as error handlers, traps, help message functions, etc.

To overcome this, many programmers rely on script templates that contain much of this routine coding. In this adventure, we’re going to look at a program called `new_script` from LinuxCommand.org that creates templates for bash scripts. Unlike static templates, `new_script` custom generates templates that include usage and help messages, as well as a parser for the script’s desired command line options and arguments. Using `new_script` saves a lot of time and effort and helps us make even the most casual script a well-crafted and robust program.

Installing new_script

To install `new_script`, we download it from LinuxCommand.org, move it to a directory in our PATH, and set it to be executable.

```
me@linuxbox:~$ curl -O http://linuxcommand.org/new_script.bash  
me@linuxbox:~$ mv new_script.bash ~/bin/new_script  
me@linuxbox:~$ chmod +x ~/bin/new_script
```

After installing it, we can test it this way:

```
me@linuxbox:~$ new_script --help
```

If the installation was successful, we will see the help message:

```
new_script 3.5.3  
Bash shell script template generator.  
  
Usage: new_script [-h|--help ]  
                  new_script [-q|--quiet] [-s|--root] [script]  
  
Options:  
  
-h, --help      Display this help message and exit.  
-q, --quiet     Quiet mode. No prompting. Outputs default script.  
-s, --root      Output script requires root privileges to run.
```

Options and Arguments

Normally, `new_script` is run without options. It will prompt the user for a variety of information that it will use to construct the script template. If an output script file name is not specified, the user will be prompted for one. For some special use cases, the following options are supported:

- **-h, --help** The help option displays the help message we saw above. The help option is mutually exclusive with the other `new_script` options and after the help message is displayed, `new_script` exits.
- **-q, --quiet** The quiet option causes `new_script` to become non-interactive and to output a base template without customization. In this mode, `new_script` will output its script template to standard output if no output script file is specified.
- **-s, --root** The superuser option adds a routine to the template that requires the script to be run by the superuser. If a non-privileged user attempts to run the resulting script, it will display an error message and terminate.

Creating Our First Template

Let's make a template to demonstrate how `new_script` works and what it can do. First, we'll launch `new_script` and give it the name of a script we want to create.

```
me@linuxbox:~$ new_script new_script-demo
-----
** Welcome to new_script version 3.5.3 **
-----
File 'new_script-demo' exists. Overwrite [y/n] > y
```

We'll be greeted with a welcome message. If the script already exists, we are prompted to overwrite. If we had not specified a script file name, we would be prompted for one.

```
-----
** Comment Block **

The purpose is a one line description of what the script does.
-----
The purpose of the script is to: > demonstrate the new_script template
-----
The script may be licensed in one of two ways:
1. All rights reserved (default) or
2. GNU GPL version 3 (preferred).
-----
Include GPL license header [y/n]? > y
```

The first information `new_script` asks for are the purpose of the script and how it is licensed. Later, when we examine the finished template below, we'll see that `new_script` figures out the author's name and email address, as well as the copyright date.

```
-----
** Privileges **

The template may optionally include code that will prevent it from
running if the user does not have superuser (root) privileges.
```

```
Does this script require superuser privileges [y/n]? > n
```

If we need to make this script usable only by the superuser, we set that next.

```
-----  
** Command Line Options **
```

The generated template supports both short name (1 character), and long name (1 word) options. All options must have a short name. Long names are optional. The options 'h' and 'help' are provided automatically.

Further, each option may have a single argument. Argument names must be valid variable names.

Descriptions for options and option arguments should be short (less than 1 line) and will appear in the template's comment block and help_message.

```
-----
```

```
Does this script support command-line options [y/n]? > y
```

Now we get to the fun part; defining the command line options. If we answer no to this question, new_script will write the template and exit.

As we respond to the next set of prompts, remember that we are building a help message (and a parser) that will resemble the new_script help message, so use that as a guide for context. Keep responses clear and concise.

Option 1:

```
Enter short option name [a-z] (Enter to end) -> a  
Description of option -----> the first option named 'a'  
Enter long option name (optional) -----> option_a  
Enter option argument (if any) ----->
```

Option 2:

```
Enter short option name [a-z] (Enter to end) -> b  
Description of option -----> the second option named 'b'  
Enter long option name (optional) -----> option_b  
Enter option argument (if any) -----> b_argument  
Description of argument (if any)-----> argument for option 'b'
```

Option 3:

```
Enter short option name [a-z] (Enter to end) ->
```

By entering nothing at the short option prompt, new_script ends the input of the command options and writes the template. We're done!

A note about short option names: new_script will accept any value, not just lowercase letters. This includes uppercase letters, numerals, etc. Use good judgment.

A note about long option names and option arguments: long option names and option arguments must be valid bash variable names. If they are not, new_script will attempt to correct them. If there are embedded spaces, they will be replaced with underscores.

Anything else will cause no_script to replace the name with a calculated default value based on the short option name.

Looking at the Template

Here we see a numbered listing of the finished template.

```
1 #!/usr/bin/env bash
2 # -----
3 # new_script-demo - Demonstrate the new_script template
4 # Copyright 2021, Linux User <me@linuxbox.example.com>
5
6 # This program is free software: you can redistribute it and/or modify
7 # it under the terms of the GNU General Public License as published by
8 # the Free Software Foundation, either version 3 of the License, or
9 # (at your option) any later version.
10 # This program is distributed in the hope that it will be useful,
11 # but WITHOUT ANY WARRANTY; without even the implied warranty of
12 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13 # GNU General Public License at <http://www.gnu.org/licenses/> for
14 # more details.
15 # Usage: new_script-demo [-h|--help]
16 #           new_script-demo [-a|--option_a] [-b|--option_b b_argument]
17 # Revision history:
18 # 2021-05-05 Created by new_script ver. 3.5.3
19 # -----
```

The comment block is complete with license, usage, and revision history. Notice how the first letter of the purpose has been capitalized and the author's name and email address have been calculated. `new_script` gets the author's name from the `/etc/password` file. If the `REPLYTO` environment variable is set, it supplies the email address (this was common with old-timey email programs); otherwise the email address will be expanded from `$USER@$hostname`. To define the `REPLYTO` variable, we just add it to our `~/.bashrc` file. For example:

```
export REPLYTO=me@linuxbox.example.com
```

Our script template continues with the constants and functions:

```
20 PROGNAME=${0##*/}
21 VERSION="0.1"
22 LIBS=      # Insert pathnames of required external shell libraries here
```

The global constants appear next, with the program name (derived from `$0`) and the version number. The `LIBS` constant should be set to contain a space-delimited list (in double quotes of course) of any files to be sourced. Note: the way the template implements this feature requires that library pathnames do not contain spaces. Besides the template not working, including embedded spaces in a library name would be in extremely poor taste.

```
23 clean_up() { # Perform pre-exit housekeeping
24     return
25 }
```

```

26 error_exit() {
27     local error_message="$1"
28     printf "%s: %s\n" "$PROGNAME" "${error_message:-"Unknown Error"}" >&2
29     clean_up
30     exit 1
31 }
32 graceful_exit() {
33     clean_up
34     exit
35 }
36 signal_exit() { # Handle trapped signals
37     local signal="$1"
38     case "$signal" in
39         INT)
40             error_exit "Program interrupted by user" ;;
41         TERM)
42             error_exit "Program terminated" ;;
43         *)
44             error_exit "Terminating on unknown signal" ;;
45     esac
46 }

```

The first group of functions handles program termination. The `clean_up` function should include the code for any housekeeping tasks needed before the script exits. This function is called by all the other exit functions to ensure an orderly termination.

```

47 load_libraries() { # Load external shell libraries
48     local i
49     for i in $LIBS; do
50         if [[ -r "$i" ]]; then
51             source "$i" || error_exit "Library '$i' contains errors."
52         else
53             error_exit "Required library '$i' not found."
54         fi
55     done
56 }

```

The `load_libraries` function loops through the contents of the `LIBS` constant and sources each file. If any file is missing or contains errors, this function will terminate the script with an error.

```

57 usage() {
58     printf "%s\n" "Usage: ${PROGNAME} [-h|--help]"
59     printf "%s\n" \
60             "${PROGNAME} [-a|--option_a] [-b|--option_b b_argument]"
61 }
62 help_message() {
63     cat <<- _EOF_
64     $PROGNAME ver. $VERSION
65     Demonstrate the new_script template

```

```

65 $(usage)

66 Options:
67 -h, --help          Display this help message and exit.
68 -a, --option_a      The first option named 'a'
69 -b, --option_b b_argument  The second option named 'b'
70   Where 'b_argument' is the argument for option 'b'.

71 _EOF_
72 return
73 }

```

The `usage` and `help_message` functions are based on the information we supplied. Notice how the help message is neatly formatted and the option descriptions are capitalized as needed.

```

74 # Trap signals
75 trap "signal_exit TERM" TERM HUP
76 trap "signal_exit INT" INT

77 load_libraries

```

The last tasks involved with set up are the signal traps and calling the function to source the external libraries, if there are any.

Next comes the parser, again based on our command options.

```

78 # Parse command-line
79 while [[ -n "$1" ]]; do
80   case "$1" in
81     -h | --help)
82       help_message
83       graceful_exit
84     ;;
85     -a | --option_a)
86       echo "the first option named 'a'"
87     ;;
88     -b | --option_b)
89       echo "the second option named 'b'"
90       shift; b_argument="$1"
91       echo "b_argument == $b_argument"
92     ;;
93     -* | -*)
94       usage >&2
95       error_exit "Unknown option $1"
96     ;;
97   *)
98     printf "Processing argument %s...\n" "$1"
99   ;;
100 esac
101 shift
102 done

```

The parser detects each of our specified options and provides a simple stub for our actual code. One feature of the parser is that positional parameters that appear after the options are assumed to be arguments to the script so this template is ready to handle them even if the script has no options.

```
103 # Main logic  
104 graceful_exit
```

We come to the end of the template where the main logic is located. Since this script doesn't do anything yet, we simply call the `graceful_exit` function so that we, well, exit gracefully.

Testing the Template

The finished template is a functional (and correct!) script. We can test it. First the help function:

```
me@linuxbox:~$ ./new_script-demo --help  
new_script-demo ver. 0.1  
Demonstrate the new_script template  
  
Usage: new_script-demo [-h|--help]  
                  new_script-demo [-a|--option_a] [-b|--option_b b_argument]  
  
Options:  
-h, --help           Display this help message and exit.  
-a, --option_a       The first option named 'a'  
-b, --option_b b_argument   The second option named 'b'  
                           Where 'b_argument' is the argument for option 'b'.  
  
me@linuxbox:~$
```

With no options or arguments, the template produces no output.

```
me@linuxbox:~$ ./new_script-demo  
me@linuxbox:~$
```

The template displays informative messages as it processes the options and arguments.

```
me@linuxbox:~$ ./new_script-domo -a  
the first option named 'a'  
me@linuxbox:~$ ./new_script-demo -b test  
the second option named 'b'  
b_argument == test  
me@linuxbox:~$ ./new_script-demo /*  
Processing argument ./bin...  
Processing argument ./Desktop...  
Processing argument ./Disk_Images...  
Processing argument ./Documents...  
Processing argument ./Downloads...  
.  
.  
.
```

Summing Up

Using `new_script` saves a lot of time and effort. It's easy to use and it produces high quality script templates. Once a programmer decides on a script's options and arguments, they can use `new_script` to quickly produce a working script and add feature after feature until everything is fully implemented.

Feel free to examine the `new_script` code. Parts of it are exciting.

Further Reading

There are many bash shell script “templates” available on the Internet. A Google search for “bash script template” will locate some. Many are just small code snippets or suggestions on coding standards. Here are a few interesting ones worth reading:

- *Boilerpalte Shell Script Template* <https://natelandau.com/boilerplate-shell-script-template/>
- *Another Bash Script Template* <https://jonlabelle.com/snippets/view/shell/another-bash-script-template>
- *Basic script template for every bash script* <https://coderwall.com/p/koixia/logging-mini-framework-snippet-for-every-shell-script>
- *Argbash documentation* A template generator that works from a configuration file rather than interactively. <https://argbash.readthedocs.io/en/latest/>

14 SQL

Okay kids, gird your grid for a big one.

The world as we know it is powered by data. Data, in turn, is stored in databases. Most everything we use computers for involves using data stored in some kind of database. When we talk about storing large amounts of data, we often mean *relational database management systems* (RDBMS). Banks, insurance companies, most every accounting system, and many, many websites use relational databases to store and organize their data.

The idea of relational data storage is generally credited to English computer scientist and IBM researcher E. F. Cobb, who proposed it in a paper in 1970. In the years that followed, a number of software companies built relational database systems with varying degrees of success. Around 1973, IBM developed a simplified and flexible method of managing relational databases called *Structured Query Language* (SQL, often pronounced “sequel”). Today the combination of RDBMS and SQL is a huge industry, generating many billions of dollars every year.

Relational databases are important to the history of Linux as well. It was the availability of open source database programs (such as MySQL) and web servers (most notably, Apache) that led to an explosion of Linux adoption in the early days of the world wide web.

In this adventure, we’re going to study the fundamentals of relational databases and use a popular command line program to create and manage a database of our own. The AWK adventure is a suggested prerequisite.

A Little Theory: Tables, Schemas, and Keys

Before we can delve into SQL we have to look at what relational databases are and how they work.

Tables

Simply put, a relational database is one or more *tables* containing columns and rows. Technically, the columns are known as *attributes* and the rows as *tuples*, but most often they are simply called columns and rows. In many ways, a table resembles the familiar spreadsheet. In fact, spreadsheet programs are often used to prepare and edit database tables. Each column contains data of a consistent type, for example, one column might consist of integers and another column strings. A table can contain any number of rows.

Schemas

The design of a database is called its *schema* and it can be simple, containing just a single table or two, or it can be complex, containing many tables with complex interrelationships between them.

Let's imagine a database for a bookstore consisting of three tables. The first is called `Customers`, the second is called `Items`, and the third is called `Sales`. The `Customers` table will have multiple rows with each row containing information about one customer. The columns include a customer number, first and last names, and the customer's address. Here is such a table with just some made-up names:

Cust	First	Last	Street	City	ST
0001	Richard	Stollman	1 Outonthe Street	Boston	MA
0002	Eric	Roymond	2 Amendment Road	Bunker Hill	PA
0003	Bruce	Porens	420 Middleville Drive	Anytown	US

The `Items` table lists our books and contains the item number, title, author, and price.

Item	Title	Author	Price
1001	Winning Friends and Influencing People	Dale Carnegie	14.95
1002	The Communist Manifesto	Marx & Engels	00.00
1003	Atlas Shrugged	Ayn Rand	99.99

As we go about selling items in our imaginary bookstore, we generate rows in the `Sales` table. Each sale generates a row containing the customer number, date and time of the sale, the item number, the quantity sold, and the total amount of the sale.

Cust	Date_Time	Item	Quan	Total
0002	202006150931	1003	1	99.99
0001	202006151108	1002	1	0.00
0003	202006151820	1001	10	149.50

Keys

Now we might be wondering what the `Cust` and `Item` columns are for. They serve as *keys*. Keys are values that serve to uniquely identify a table row and to facilitate interrelationships between tables. Keys have some special properties. They must be both unique (that is, they can appear only once in a table and specifically identify a row) and they must also be immutable (they can never change). If they can't meet these requirements, they cannot be keys. Some database implementations have methods of enforcing these requirements during table creation and keys can be formally specified. In the case of our bookstore database, the `Cust` column contains the keys for the `Customers` table and the `Item` column contains the keys for the `Items` table.

Knowing this about keys, we can now understand why the `Sales` table works the way it does. We can see for example that row 1 of the `Sales` table tells us that customer 0002 purchased 1 copy of item 1003 for \$99.99. So why do we need special values for the

keys? Why not, for instance, just use the customer's name as the key? It's because we can't guarantee that the name won't change, or that two people might have the same name. We can guarantee that an arbitrarily assigned value like our customer number is unique and immutable.

Database Engines/Servers

There are a number of database servers available for Linux. The two most prominent are MySQL (and its fork MariaDB) and PostgreSQL. These database servers implement *client/server architecture*, a concept that became a hot topic in the 1990s. Database servers of this type run as server processes and clients connect to them over a network connection and send SQL statements for the server to carry out. The results of the database operations are returned to the client program for further processing and presentation. Many web sites use this architecture with a web server sending SQL statements to a separate database server to dynamically create web pages as needed. The famous *LAMP stack* consisting of Linux, Apache web server, MySQL, and PHP powered much of the early web.

For purposes of this adventure, setting up database servers such as MySQL and PostgreSQL is too complicated to cover here since, among other things, they support multiple concurrent users and their attendant security controls. It's more than we need for just learning SQL.

sqlite3

The database server we will be using is SQLite. SQLite is a library that can be used with applications to provide a high-level of SQL functionality. It's very popular with the embedded systems crowd. It saves application developers the trouble of writing custom solutions to their data storage and management tasks. In addition to the library, SQLite provides a command line tool for directly interacting with SQLite databases. Also, since it accepts SQL from standard input (as well as its own command line interface) and sends its results to standard output, it's ideal for use in our shell scripts.

SQLite is available from most Linux distribution repositories. Installation is easy, for example:

```
me@linuxbox:~$ sudo apt install sqlite3
```

Building a Playground

Let's build a playground and play with some real data. On the LinuxCommand.org site there is a archive we can download that will do the trick.

```
me@linuxbox:~$ cd  
me@linuxbox:~$ curl -c http://linuxcommand.org/adventure-sql.tgz  
me@linuxbox:~$ tar xzf adventure-sql.tgz  
me@linuxbox:~$ cd adventure-sql
```

Extracting the `.tgz` file will produce the playground directory containing the data sets, some demonstration code, and some helpful tools. The data sets we will be working with contain the listings of installed packages on an Ubuntu 18.04 system. This will include the name of packages, a short description of each one, a list of files contained in each package, and their sizes.

```
me@linuxbox:~/adventure-sql$ ls
```

All the files are human-readable text, so feel free to give them a look. The data sets in the archive are the `.tsv` files. These are tab-separated value files. The first one is the `package_descriptions.tsv` file. This contains two columns of data; a package name and a package description. The second file, named `package_files.txt`, has three columns: a package name, the name of a file installed by the package and the size of the file.

Starting sqlite3

To launch SQLite, we simply issue the command `sqlite3` followed optionally by the name of a file to hold our database of tables. If we do not supply a file name, SQLite will create a temporary database in memory.

```
me@linuxbox:~/adventure-sql$ sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

When loading is complete, SQLite will present a prompt where we can enter commands. Commands can be either SQL statements or *dot commands* that are used to control SQLite itself. To see a list of the available dot commands, we enter `.help` at the prompt.

```
sqlite> .help
```

There are only a few of the dot commands that will be of interest to us and they deal mainly with how output is formatted. To exit `sqlite3`, we enter the dot command `.quit` at the prompt.

```
sqlite> .quit
```

Though we can interact directly with the `sqlite3` program using the `sqlite>` prompt, `sqlite3` can also accept streams of dot commands and SQL statements through standard input. This is how SQLite is most often used.

Creating a Table and Inserting Our Data

To get started with our database, we need to first convert our `.tsv` files into a stream of SQL statements. Our database will initially consist of two tables. The first is named `Package_Descriptions` and the second is named `Package_Files`. To create the SQL stream for the `Package_Descriptions` table we will use the `insert_Package_Descriptions.awk` program supplied in the playground archive.

```
me@linuxbox:~/adventure-sql$ ./insert_Package_Descriptions.awk \
< package_descriptions.tsv > insert_Package_Descriptions.sql
```

Let's take a look at the resulting SQL stream. We'll use the `head` command to display the first few lines of the stream.

```
me@linuxbox:~/adventure-sql$ head insert_Package_Descriptions.sql
DROP TABLE IF EXISTS Package_Descriptions;
CREATE TABLE Package_Descriptions (
    package_name VARCHAR(60),
    description   VARCHAR(120)
);
BEGIN TRANSACTION;
INSERT INTO Package_Descriptions
    VALUES ( 'a2ps',
              'GNU a2ps - ''Anything to PostScript'' converter and pretty-printer');
INSERT INTO Package_Descriptions
    VALUES ( 'accounts-service',
              'query and manipulate user account information');
```

And the last few lines using the `tail` command.

```
me@linuxbox:~/adventure-sql$ tail insert_Package_Descriptions.sql
    VALUES ( 'zlib1g:amd64',
              'compression library - runtime');
INSERT INTO Package_Descriptions
    VALUES ( 'zlib1g:i386',
              'compression library - runtime');
INSERT INTO Package_Descriptions
    VALUES ( 'zlib1g-dev:amd64',
              'compression library - development');
INSERT INTO Package_Descriptions
    VALUES ( 'zsh',
              'shell with lots of features');
INSERT INTO Package_Descriptions
    VALUES ( 'zsh-common',
              'architecture independent files for Zsh');
COMMIT;
```

As we can see, SQL is verbose and somewhat English-like. Convention dictates that language keywords be in uppercase; however it is not required. SQL is case insensitive. White space is not important, but is often used to make the SQL statements easier to read. Statements can span multiple lines but don't have to. Statements are terminated by a semicolon character. The SQL in this adventure is generally formatted in accordance with the style guide written by Simon Holywell linked in the "Further Reading" section below. Since some SQL can get quite complicated, visual neatness counts when writing code.

SQL supports two forms of commenting.

```
-- Single line comments are preceded by 2 dashes
/*
 * And multi-line comments are done in the
 * style of the C programming language */
```

Before we go on, we need to digress for a moment to discuss SQL as a standard. While there are ANSI standards for SQL, every database server implements SQL differently. Each one has a slightly different dialect. The reason for this is partly historical; in the

early days there weren't any firm standards, and partly commercial. Each database vendor wanted to make it hard for customers to migrate to competing products so each vendor added unique extensions and features to the language to promote the dreaded "vendor lock-in" for their product. SQLite supports most of standard SQL (but not all of it) and it adds a few unique features.

Creating and Deleting Tables

The first 2 lines of our SQL stream deletes any existing table named `Package_Descriptions` and creates a new table with that name. The `DROP TABLE` statement deletes a table. The optional `IF EXISTS` clause is used to prevent errors if the table does not already exist. There are a lot of optional clauses in SQL. The `CREATE TABLE` statement defines a new table. As we can see, this table will have 2 columns. The first column, `package_name` is defined to be a variable length string up to 60 characters long. `VARCHAR` is one of the available data types we can define. Here are some of the common data types supported by SQL databases:

Data Type	Description
<code>INTEGER</code>	Integer
<code>CHAR(n)</code>	Fixed length string
<code>VARCHAR(n)</code>	Variable length string
<code>NUMERIC</code>	Decimal numbers
<code>REAL</code>	Floating point numbers
<code>DATETIME</code>	Date and time values

Common SQL data types

Data Types

SQL databases support many types of data. Unfortunately, this varies by vendor. Even in cases where two databases share the same data type name, the actual meaning of the data type can differ. Data types in SQLite, unlike other SQL databases, are not rigidly fixed. Values in SQLite are *dynamically typed*. While SQLite allows many of the common data types found in other databases to be specified, it actually only supports 4 general types of data storage.

Data Type	Description
<code>INTEGER</code>	Signed integers using 1, 2, 3, 4, 6, or 8 bytes as needed
<code>REAL</code>	8-byte IEEE floating point numbers
<code>TEXT</code>	Text strings
<code>BLOB</code>	Binary large objects (for example JPEG, or MP3 files)

SQLite data types

In our example above, we specified `VARCHAR` as the data type for our columns. SQLite is perfectly happy with this, but it actually stores the values in these columns as just `TEXT`. It ignores the length restrictions set in the data type specification. SQLite is extremely

lenient about data types. In fact, it allows any kind of data to be stored in any specified data type, even allowing a mixture of data types in a single column. This is completely incompatible with all other databases, and relying on this would be very bad practice. In the remainder of this adventure we will be sticking to conventional data types and behavior.

Inserting Data

Moving on with our SQL stream, we see that the majority of the stream consists of `INSERT` statements. This is how rows are added to a table. Insert is sort of a misnomer as `INSERT` statements append rows to a table.

We surround the `INSERT` statements with `BEGIN TRANSACTION` and `COMMIT`. This is done for performance reasons. If we leave these out, the rows will still be appended to the table but each `INSERT` will be treated as a separate transaction, vastly increasing the amount of time it takes to append a large number of rows. Treating a transaction this way also has another important benefit. SQL does not apply the transaction to the database until it receives the `COMMIT` statement, thus it is possible to write SQL code that will abandon a transaction if there is a problem and the change will be rolled back leaving the database unchanged.

Let's go ahead and create our first table and add the package names and descriptions.

```
me@linuxbox:~/adventure-sql$ sqlite3 adv-sql.sqlite \
< insert_Package_Descriptions.sql
```

We execute the `sqlite3` program specifying `adv-sql.sqlite` as the file used to store our tables. The choice of file name is arbitrary. We read our SQL stream into standard input and `sqlite3` carries out the statements.

Doing Some Queries

Now that we have a database (albeit a small one), let's take a look at it. To do this, we will start up `sqlite3` and interact with it at the prompt.

```
me@linuxbox:~/adventure-sql$ sqlite3 adv-sql.sqlite
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
sqlite>
```

We'll first use some SQLite dot commands to examine the structure of the database.

```
sqlite> .tables
Package_Descriptions
sqlite> .schema Package_Descriptions
CREATE TABLE Package_Descriptions (
    package_name VARCHAR(60),
    description   VARCHAR(120)
);
```

The `.tables` dot command displays a list of tables in the database while the `.schema` dot command lists the statements used to create the specified table.

Next, we'll get into some real SQL using `SELECT`, probably the most frequently used SQL statement.

```
sqlite> SELECT * FROM Package_Descriptions;
a2ps|GNU a2ps - 'Anything to PostScript' converter and pretty-printer
accountsservice|query and manipulate user account information
acl|Access control list utilities
acpi-support|scripts for handling many ACPI events
acpid|Advanced Configuration and Power Interface event daemon
adduser|add and remove users and groups
adium-theme-ubuntu|Adium message style for Ubuntu
adwaita-icon-theme|default icon theme of GNOME (small subset)
aisleriot|GNOME solitaire card game collection
alsa-base|ALSA driver configuration files
.
.
.
```

This is the simplest form of the `SELECT` statement. The syntax is the word `SELECT` followed by a list of columns or calculated values we want, and a `FROM` clause specifying the source of the data. This example uses `*` which means every column. Alternately, we could explicitly name the columns, like so:

```
sqlite> SELECT package_name, description FROM Package_Descriptions;
```

And achieve the same result.

Controlling the Output

As we can see from the output above, the default format is fine for further processing by tools such as `awk`, but it leaves a lot to be desired when it comes to being read by humans. We can adjust the output format with some dot commands. We'll also add the `LIMIT` clause to the end of our query to output just 10 rows.

```
sqlite> .headers on
sqlite> .mode column
sqlite> SELECT * FROM Package_Descriptions LIMIT 10;
package_name    description
-----  -----
a2ps          GNU a2ps - 'Anything to PostScript' converter and pretty-printer
accountsserv  query and manipulate user account information
acl           Access control list utilities
acpi-support   scripts for handling many ACPI events
acpid          Advanced Configuration and Power Interface event daemon
adduser        add and remove users and groups
adium-theme-  Adium message style for Ubuntu
adwaita-icon   default icon theme of GNOME (small subset)
aisleriot      GNOME solitaire card game collection
alsa-base      ALSA driver configuration files
```

By using the `.headers on` and `.mode column` dot commands, we add the column headings and change the output to column format. These settings stay in effect until we change them. The `.mode` dot command has a number of interesting possible settings.

Mode	Description
csv	Comma-separated values
column	Left-aligned columns. Use .width n1 n2... to set column widths.
html	HTML <table> code
insert	SQL insert statements for TABLE
line	One value per line
list	Values delimited by .separator string. This is the default.
tabs	Tab-separated values
tcl	TCL (Tool Control Language) list elements

SQLite output modes

Here we will set the mode and column widths for our table.

```
sqlite> .mode column
sqlite> .width 20 60
sqlite> SELECT * FROM Package_Descriptions LIMIT 10;
package_name          description
-----
a2ps                 GNU a2ps - 'Anything to PostScript' converter and pretty
accountsservice       query and manipulate user account information
acl                  Access control list utilities
acpi-support         scripts for handling many ACPI events
acpid                Advanced Configuration and Power Interface event daemon
adduser              add and remove users and groups
adium-theme-ubuntu   Adium message style for Ubuntu
adwaita-icon-theme   default icon theme of GNOME (small subset)
aisleriot            GNOME solitaire card game collection
alsa-base            ALSA driver configuration files
```

In addition to listing columns, `SELECT` can be used to perform various output tasks. For example, we can perform calculations such as counting the number of rows in the `Package_Descriptions` table.

```
sqlite> SELECT COUNT(package_name) FROM Package_Descriptions;
count(package_name)
-----
1972
```

Being Selective

We can make `SELECT` output rows based on some selection criteria. Either an exact match:

```
sqlite> SELECT * FROM Package_Descriptions WHERE package_name = 'bash';
package_name          description
-----
bash                 GNU Bourne Again SHell
```

A partial match using SQL wildcard characters:

```
sqlite> SELECT * FROM Package_Descriptions WHERE description LIKE '%bash%';
package_name          description
-----
bash-completion       programmable completion for the bash shell
command-not-found    Suggest installation of packages in interactive bash ses
```

SQL supports two wildcard characters. The underscore (_), which matches any single character, and the percent sign (%), which matches zero or more instances of any character.

Notice too that strings are surrounded with single quotes. If a value is quoted this way, SQL treats it as a string. For example, the value '123' would be treated as a string rather than a number.

Sorting Output

Unless we tell `SELECT` to sort our data, it will be listed in the order it was inserted into the table. Our data appears in alphabetical order by package name because it was inserted that way, not because of anything SQLite is doing. The `ORDER BY` clause can be added to determine which column is used for sorting. To demonstrate, let's sort the output by the description,

```
sqlite> SELECT * FROM Package_Descriptions ORDER BY description LIMIT 10;
package_name      description
-----          -----
network-manager-conf
fonts-noto-cjk    "No Tofu" font families with large Unicode coverage (CJ)
fonts-noto-mono   "No Tofu" monospaced font family with large Unicode cov
udev             /dev/ and hotplug management daemon
procps            /proc file system utilities
alsa-base         ALSA driver configuration files
libasound2-plugins:a  ALSA library additional plugins
libhyphen0:amd64  ALTLinux hyphenation library - shared library
apcupsd           APC UPS Power Management (daemon)
apcupsd-doc       APC UPS Power Management (documentation/examples)
```

The default sorting order is ascending, but we can also sort in descending order by including `DESC` after the column name. Multiple columns can be named and `ASC` can be used to specify ascending order.

```
sqlite> SELECT * FROM Package_Descriptions ORDER BY description DESC LIMIT 10;
package_name      description
-----          -----
xsane-common     xsane architecture independent files
libx264-152:    x264 video coding library
libevdev2:am     wrapper library for evdev devices
wireless-reg    wireless regulatory database
crda            wireless Central Regulatory Domain Agent
libmutter-2-     window manager library from the Mutter window mana
libwayland-s    wayland compositor infrastructure - server library
libwayland-c    wayland compositor infrastructure - cursor library
libwayland-c    wayland compositor infrastructure - client library
libwayland-e    wayland compositor infrastructure - EGL library
```

Adding Another Table

To demonstrate more of what we can do with `SELECT`, we're going to need a bigger database. We have a second `.tsv` file that we can add. To save a step, we'll filter the file into SQL and pipe it directly into `sqlite3`.

```
me@linuxbox:~/adventure-sql$ ./insert_package_files.awk \
    < package_files-deb.tsv \
    | sqlite3 adv-sql.sqlite
```

The second table is named `Package_Files`. Here is its schema:

```
sqlite> .schema Package_Files
CREATE TABLE Package_Files (
    package_name VARCHAR(60),
    file          VARCHAR(120),
    size_bytes    INTEGER
);
```

As we can see, this table has 3 columns; the package name, the name of a file installed by the package, and the size of the installed file in bytes. Let's do a `SELECT` to see how this table works.

```
sqlite> .headers on
sqlite> .mode column
sqlite> .width 15 50 -10
sqlite> SELECT * FROM Package_Files WHERE package_name = 'bash';
package_name      file                           size_bytes
-----  -----
bash              /bin/bash                      1113504
bash              /etc/bash.bashrc                 2319
bash              /etc/skel/.bash_logout            220
bash              /etc/skel/.bashrc                  3771
bash              /etc/skel/.profile                807
bash              /usr/bin/bashbug                 7115
bash              /usr/bin/clear_console           10312
bash              /usr/share/doc/bash/COMPAT.gz       7853
bash              /usr/share/doc/bash/INTRO.gz        2921
bash              /usr/share/doc/bash/NEWS.gz         27983
bash              /usr/share/doc/bash/POSIX.gz        3702
bash              /usr/share/doc/bash/RBASH             1693
bash              /usr/share/doc/bash/README             3839
bash              /usr/share/doc/bash/README.Debian.gz   1919
bash              /usr/share/doc/bash/README.abs-guide     1105
bash              /usr/share/doc/bash/README.commands.gz  3021
bash              /usr/share/doc/bash/changelog.Debian.gz 1357
bash              /usr/share/doc/bash/copyright          10231
bash              /usr/share/doc/bash/inputrc.arrows      727
bash              /usr/share/lintian/overrides/bash       156
bash              /usr/share/man/man1/bash.1.gz          86656
bash              /usr/share/man/man1/bashbug.1.gz        804
bash              /usr/share/man/man1/clear_console.1.gz  1194
bash              /usr/share/man/man1/rbash.1.gz          154
bash              /usr/share/man/man7/bash-builtins.7.gz  508
bash              /usr/share/menu/bash                   194
bash              /bin/rbash                         4
```

Notice the `.width` dot command above. A negative value causes the corresponding column to be right-aligned.

Subqueries

The `SELECT` statement can be used to produce many kinds of output. For example, it can be used to simply print literal strings.

```
sqlite> .mode column
sqlite> .header off
sqlite> SELECT 'String 1', 'String 2';
string 1      string 2
```

As we saw before, `SELECT` can produce calculated values.

```
sqlite> .header on
sqlite> SELECT 2 + 2;
2 + 2
-----
4
sqlite> SELECT COUNT(file), AVG(size_bytes) FROM Package_Files;
count(file)  avg(size_bytes)
-----  -----
153506      33370.3488658424
```

To make complicated expressions more readable, we can assign their results to *aliases* by using the `AS` clause.

```
sqlite> SELECT COUNT(file) AS Files,
...>     AVG(size_bytes) AS 'Average Size'
...>     FROM Package_Files;
Files      Average Size
-----  -----
153506    33370.3488658424
```

An important feature of `SELECT` is the ability to produce results by combining data from multiple tables. This process is done by performing *joins* and *subqueries*. We'll talk about joins a little later, but for now let's concentrate on subqueries. `SELECT` allows us to include another `SELECT` statement as an item to output. To demonstrate this, we will produce a table that includes columns for package name, number of files in the package, and the total size of the package. The `SELECT` statement to do this is rather formidable. We'll open our text editor and create a file named `subquery_demo1.sql` with the following content:

```
-- subquery_demo1.sql

-- Query to list top 20 packages with the greatest numbers of files

.mode column
.header on
.width 20 40 10 10
SELECT package_name, description,
       (SELECT COUNT(file)
        FROM Package_Files
        WHERE Package_Descriptions.package_name = Package_Files.package_name)
       AS files,
       (SELECT SUM(size_bytes)
        FROM Package_Files
        WHERE Package_Descriptions.package_name = Package_Files.package_name)
       AS size
    FROM Package_Descriptions ORDER BY files DESC LIMIT 20;
```

We'll next run this query and view the results.

```
me@linuxbox:~/adventure-sql$ sqlite3 adv-sql.sqlite < subquery_demo1.sql
package_name          description          files      size
-----  -----  -----  -----

```

linux-headers-4.15.0	Header files related to Linux kernel	14849	63991787
linux-headers-4.15.0	Header files related to Linux kernel	14849	64001943
humanity-icon-theme	Humanity Icon theme	8014	14213715
linux-headers-4.15.0	Linux kernel headers for version	7861	9015084
linux-headers-4.15.0	Linux kernel headers for version	7860	9025673
linux-modules-extra-	Linux kernel extra modules for v	4173	165921470
linux-modules-extra-	Linux kernel extra modules for v	4172	165884678
libreoffice-common	office productivity suite -- arc	3551	76686149
gnome-accessibility-	High Contrast GTK+ 2 theme and icons	3464	3713621
ubuntu-mono	Ubuntu Mono Icon theme	3025	3755093
ncurses-term	additional terminal type definitions	2727	1987483
manpages-dev	Manual pages about using GNU/Linux	2101	2192620
linux-firmware	Firmware for Linux kernel driver	1938	331497257
tzdata	time zone and daylight-saving time	1834	1210058
vim-runtime	Vi IMproved - Runtime files	1642	27941732
codium	Code editing. Redefined.	1307	271907088
zsh-common	architecture independent files	1256	12261077
perl-modules-5.26	Core Perl modules	1144	18015966
adwaita-icon-theme	default icon theme of GNOME (smash)	1127	4848678
gimp-data	Data files for GIMP	1032	45011675

The query takes some time to run (it has a lot to do) and from the results we see that it produces 4 columns: package name, description, number of files in the package, and total size of the package. Let's take this query apart and see how it works. At the uppermost level we see that the query follows the normal pattern of a SELECT statement.

```
SELECT list_of_items FROM Package_Descriptions
ORDER BY total_files DESC
LIMIT 20;
```

The basic structure is simple. What's interesting is the `list_of_items` part. We know the list of items is a comma-separated list of items to output, so if we follow the commas we can see the list:

1. `package_name`
2. `description`
3. `(SELECT COUNT(file) FROM Package_Files WHERE Package_Descriptions.package_name = Package_Files.package_name) AS files`
4. `(SELECT SUM(size_bytes) FROM Package_Files WHERE Package_Descriptions.package_name = Package_Files.package_name) AS size`

It's also possible to use a subquery in a WHERE clause. Consider this query that we will name `subquery_demo2.sql`:

```
-- subquery_demo2.sql

-- Query to list all packages containing more than 1000 files

.mode column
.header on
.width 20 60
SELECT package_name, description
  FROM Package_Descriptions
 WHERE 1000 < (SELECT COUNT(file)
                FROM Package_Files
```

```

    WHERE Package_Descriptions.package_name = Package_Files.package_name)
ORDER BY package_name;

```

When we execute this, we get the following results:

```

me@linuxbox:~/adventure-sql$ sqlite3 adv-sql.sqlite < subquery_demo2.sql
package_name          description
-----
adwaita-icon-theme   default icon theme of GNOME (small subset)
codium                Code editing. Redefined.
gimp-data             Data files for GIMP
gnome-accessibility- High Contrast GTK+ 2 theme and icons
humanity-icon-theme  Humanity Icon theme
inkscape               vector-based drawing program
libreoffice-common    office productivity suite -- arch-independent files
linux-firmware        Firmware for Linux kernel drivers
linux-headers-4.15.0  Header files related to Linux kernel version 4.15.0
linux-headers-4.15.0  Linux kernel headers for version 4.15.0 on 64 bit x86 SM
linux-headers-4.15.0  Header files related to Linux kernel version 4.15.0
linux-headers-4.15.0  Linux kernel headers for version 4.15.0 on 64 bit x86 SM
linux-modules-4.15.0  Linux kernel extra modules for version 4.15.0 on 64 bit
linux-modules-4.15.0  Linux kernel extra modules for version 4.15.0 on 64 bit
linux-modules-extra- Linux kernel extra modules for version 4.15.0 on 64 bit
linux-modules-extra- Linux kernel extra modules for version 4.15.0 on 64 bit
manpages-dev          Manual pages about using GNU/Linux for development
ncurses-term          additional terminal type definitions
perl-modules-5.26     Core Perl modules
tzdata                 time zone and daylight-saving time data
ubuntu-mono            Ubuntu Mono Icon theme
vim-runtime             Vi IMproved - Runtime files
zsh-common              architecture independent files for Zsh

```

Updating Tables

The **UPDATE** statement is used to change values in one or more existing rows. We will demonstrate this by adding 100 to the size of each file in the `sqlite3` package. First, let's look at the files in the package.

```

sqlite> .mode column
sqlite> .header on
sqlite> .width 50 -10
sqlite> SELECT file, size_bytes FROM Package_Files
...> WHERE package_name = 'sqlite3';
file                           size_bytes
-----
/usr/bin/sqlldiff           1103280
/usr/bin/sqlite3            1260976
/usr/share/doc/sqlite3/copyright 1261
/usr/share/man/man1/sqlite3.1.gz 3596
/usr/share/doc/sqlite3/changelog.Debian.gz 35

```

Next, we'll update the table, adding 100 to the size of each file.

```

sqlite> UPDATE Package_Files SET size_bytes = size_bytes + 100
...> WHERE package_name = 'sqlite3';

```

When we examine the rows now, we see the change.

```

sqlite> SELECT file, size_bytes FROM Package_Files
...> WHERE package_name = 'sqlite3';

```

file	size_bytes
/usr/bin/sqlldiff	1103380
/usr/bin/sqlite3	1261076
/usr/share/doc/sqlite3/copyright	1361
/usr/share/man/man1/sqlite3.1.gz	3696
/usr/share/doc/sqlite3/changelog.Debian.gz	135

Finally, we'll subtract 100 from each row to return the sizes to their original values.

```
sqlite> UPDATE Package_Files SET size_bytes = size_bytes - 100
...> WHERE package_name = 'sqlite3';
```

UPDATE can modify multiple values at once. To demonstrate this, we will create a new table called `Package_Stats` and use UPDATE to fill in the values. Since this one is a little complicated, we will put this in a file named `create_Package_Stats.sql`.

```
-- create_Package_Stats.sql

DROP TABLE IF EXISTS Package_Stats;
CREATE TABLE Package_Stats (
    package_name  VARCHAR(60),
    count         INTEGER,
    tot_size      INTEGER,
    min_size      INTEGER,
    max_size      INTEGER,
    avg_size      REAL
);

INSERT INTO Package_Stats (package_name)
    SELECT package_name
        FROM Package_Descriptions;

UPDATE Package_Stats
    SET count = (SELECT COUNT(file)
                  FROM Package_Files
                  WHERE Package_Files.package_name =
                        Package_Stats.package_name),
    tot_size = (SELECT SUM(size_bytes)
                  FROM Package_Files
                  WHERE Package_Files.package_name =
                        Package_Stats.package_name),
    min_size = (SELECT MIN(size_bytes)
                  FROM Package_Files
                  WHERE Package_Files.package_name =
                        Package_Stats.package_name),
    max_size = (SELECT MAX(size_bytes)
                  FROM Package_Files
                  WHERE Package_Files.package_name =
                        Package_Stats.package_name),
    avg_size = (SELECT AVG(size_bytes)
                  FROM Package_Files
                  WHERE Package_Files.package_name =
                        Package_Stats.package_name);
```

This file consists of four SQL statements. The first two are used to create the new table, as we have seen before. The third statement is an alternate form of the `INSERT` statement. This form is useful, as it copies a value from one table into another. This `INSERT` will create all the rows we need but only fill in the `package_name` column. To fill in the rest,

we will use an `UPDATE` that fills in the remaining five values based on the results of some queries of the `Package_Files` table. Note that without a `WHERE` clause, `UPDATE` applies changes to every row.

Once the table is constructed, we can examine its contents.

```
sqlite> .width 25 -5 -10 -8 -8 -10
sqlite> SELECT * FROM Package_Stats LIMIT 10;


| package_name       | count | tot_size | min_size | max_size | avg_size   |
|--------------------|-------|----------|----------|----------|------------|
| a2ps               | 299   | 3455890  | 117      | 388096   | 11558.1605 |
| accountsservice    | 19    | 261704   | 42       | 182552   | 13773.8947 |
| acl                | 11    | 91106    | 30       | 35512    | 8282.36363 |
| acpi-support       | 18    | 13896    | 67       | 4922     | 772.0      |
| acpid              | 19    | 86126    | 115      | 52064    | 4532.94736 |
| adduser            | 81    | 246658   | 7        | 37322    | 3045.16049 |
| adium-theme-ubuntu | 137   | 126759   | 25       | 12502    | 925.248175 |
| adwaita-icon-theme | 1127  | 4848678  | 30       | 87850    | 4302.28748 |
| aisleriot          | 316   | 1890864  | 47       | 281544   | 5983.74683 |
| alsa-base          | 42    | 195295   | 24       | 34160    | 4649.88095 |


```

We'll come back to this table a little later when we take a look at joins.

Deleting Rows

Deleting rows is pretty easy in SQL. There is a `DELETE` statement with a `WHERE` clause to specify a target. We'll demonstrate that, but first there's a nifty trick that SQLite supports.

We can change the output mode to write out `INSERT` statements. Let's try it out.

```
sqlite> .mode insert Package_Files
```

If we use this `.mode` setting, we tell SQLite that we want `INSERT` statements directed at the specified table, in this case, `Package_Files`. Once we set this output mode, we can see the result.

```
sqlite> SELECT * FROM Package_Files WHERE package_name = 'sqlite3';
INSERT INTO Package_Files VALUES('sqlite3','/usr/bin/sqldiff',1103380);
INSERT INTO Package_Files VALUES('sqlite3','/usr/bin/sqlite3',1261076);
INSERT INTO Package_Files VALUES('sqlite3','/usr/share/doc/sqlite3/copyright',1361);
INSERT INTO Package_Files VALUES('sqlite3','/usr/share/man/man1/sqlite3.1.gz',3696);
INSERT INTO Package_Files VALUES('sqlite3','/usr/share/doc/sqlite3/changelog.Debian.gz',135);
```

We'll repeat this `SELECT`, but first we'll change the output from standard output to a file named `insert_sqlite3.sql`.

```
sqlite> .output insert_sqlite3.sql
sqlite> select * from Package_Files where package_name = 'sqlite3';
```

This will write the stream of `INSERT` statements to the specified file. Next we'll set the output back to standard output by issuing the `.output` dot command without an output file name.

```
sqlite> .output
```

Now let's delete the rows in the `Package_Files` table.

```
sqlite> DELETE FROM Package_Files WHERE package_name = 'sqlite3';
```

We can confirm the deletion by running our query again and we see an empty result.

```
sqlite> .header on
sqlite> .mode column
sqlite> .width 12 50 -10
sqlite> SELECT * FROM Package_Files WHERE package_name = 'sqlite3';
sqlite>
```

Since we saved an SQL stream that can restore the deleted rows, we can now put them back in the table. The `.read` dot command can read the stream and execute it as though it came from standard input.

```
sqlite> .read insert_sqlite3.sql
```

Now when we run our query, we see that the rows have been restored.

```
sqlite> SELECT * FROM Package_Files WHERE package_name = 'sqlite3';
package_name    file                                size_bytes
-----  -----
sqlite3        /usr/bin/sqlldiff                  1103280
sqlite3        /usr/bin/sqlite3                   1260976
sqlite3        /usr/share/doc/sqlite3/copyright   1261
sqlite3        /usr/share/man/man1/sqlite3.1.gz    3596
sqlite3        /usr/share/doc/sqlite3/changelog.Debian.gz 35
```

Adding and Deleting Columns

SQL provides the `ALTER TABLE` statement to modify table's schema. To demonstrate this, we will add a couple of columns to the `Package_Descriptions` table and fill them with values calculated from the `Package_Files` table. We'll place the necessary SQL in the `add_column.sql` file.

```
-- add_column.sql

-- Add and populate columns to Package_Descriptions

ALTER TABLE Package_Descriptions ADD COLUMN files INTEGER;
ALTER TABLE Package_Descriptions ADD COLUMN size INTEGER;

UPDATE Package_Descriptions
    SET files = (SELECT COUNT(file)
                  FROM Package_Files
                 WHERE Package_Files.package_name =
                       Package_Descriptions.package_name),
        size = (SELECT SUM(size_bytes)
                  FROM Package_Files
                 WHERE Package_Files.package_name =
                       Package_Descriptions.package_name);
```

We'll execute the statements and examine resulting schema.

```
sqlite> .read add_column.sql
sqlite> .schema
CREATE TABLE Package_Descriptions (
    package_name VARCHAR(60),
```

```

description  VARCHAR(120),
files        INTEGER,
size         INTEGER);

sqlite> SELECT * FROM Package_Descriptions WHERE package_name = 'sqlite3';
package_name  description                               files      size
-----  -----
sqlite3       Command line interface for SQLite 3      5          2369648

```

SQL provides another `ALTER TABLE` statement for deleting columns from a table. It has the following form:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Unfortunately, SQLite does not support this so we have to do it the hard way. This is accomplished in four steps:

1. Create a new temporary table with the desired schema.
2. Copy the columns we want to keep into the temporary table.
3. Drop the original table.
4. Rename the temporary table.

Here is a file called `drop_column.sql` that does the job.

```
-- drop_column.sql

-- Remove extra columns from Package_Descriptions

BEGIN TRANSACTION;

-- Create new table with temporary name

DROP TABLE IF EXISTS temp;
CREATE TABLE temp (
    package_name VARCHAR(60),
    description  VARCHAR(120));

-- Copy columns we want into new table

INSERT INTO temp
    (package_name, description)
SELECT package_name, description
    FROM Package_Descriptions;

-- Get rid of old table and rename the new replacement table

DROP TABLE Package_Descriptions;

ALTER TABLE temp RENAME TO Package_Descriptions;

COMMIT;
```

We again use the alternate form of the `INSERT` statement for copying values from one table to another. We copy the `package_name` and `description` columns from the `Package_Descriptions` table to the correspondingly named columns in the `temp` table.

Joins

A join is a method by which we perform a query and produce a result that combines the data from two tables. SQLite supports several types of joins but we're going to focus on the most commonly used type called an *inner join*. We can think of an inner join as the intersection of two tables. In the example below, a file called `join_demo.sql`, we will produce a query result that matches our earlier example when we added columns to the `Package_Descriptions` table.

```
-- join_demo.sql

-- Demonstrate join by selecting columns from 2 tables

.header on
.mode column
.width 20 35 -5 -10

SELECT Package_Descriptions.package_name AS Package,
       description AS Description,
       count AS Files,
       tot_size AS Size
  FROM Package_Descriptions
 INNER JOIN Package_Stats
    ON Package_Descriptions.package_name =
       Package_Stats.package_name
LIMIT 10;
```

The results of this query are as follows:

Package	Description	Files	Size
a2ps	GNU a2ps - 'Anything to PostScript'	299	3455890
accountsservice	query and manipulate user account information	19	261704
acl	Access control list utilities	11	91106
acpi-support	scripts for handling many ACPI events	18	13896
acpid	Advanced Configuration and Power Interface	19	86126
adduser	add and remove users and groups	81	246658
adium-theme-ubuntu	Adium message style for Ubuntu	137	126759
adwaita-icon-theme	default icon theme of GNOME (small icons)	1127	4848678
aisleriot	GNOME solitaire card game collection	316	1890864
alsa-base	ALSA driver configuration files	42	195295

If we break down this query, we see that it starts out as we expect, then it is followed by the `INNER JOIN` statement. The `INNER` keyword is optional as an inner join is the default. After the `INNER JOIN` we have to specify the relationship on which the join is based. In this example, we are looking for matching instances of the `package_name` in both tables. Any expression may be used to convey the table relationship, provided that the result of the expression is true or false.

Since `package_name` is a column in both the `Package_Descriptions` and `Package_Stats` tables, we must precede it with the name of the respective table to avoid ambiguity.

Views

The join example above is a pretty useful query for our tables, but due to its complexity it's best executed from a file rather than as an ad hoc query. SQL addresses this issue by providing a feature called *views* that allows a complex query to be stored in the database and used to produce a virtual table that can be used with simple query commands. In the following example we will create a view using our `INNER JOIN` query above to create a virtual table called `Stats` that we can use as the target of subsequent queries.

```
-- view_demo.sql

DROP VIEW IF EXISTS Stats;
CREATE VIEW Stats
AS
    SELECT Package_Descriptions.package_name AS Package,
           description AS Description,
           count AS Files,
           tot_size AS Size
      FROM Package_Descriptions
     INNER JOIN Package_Stats
        ON Package_Descriptions.package_name =
           Package_Stats.package_name
   ORDER BY Package;
```

Once our view is created, we can treat `Stats` as just another table in our database even though it does not really exist as such.

```
sqlite> .header on
sqlite> .mode column
sqlite> .width 20 35 -5 -10
sqlite> SELECT * FROM Stats LIMIT 10;
Package          Description               Files      Size
-----          -----
a2ps            GNU a2ps - 'Anything to PostScript'    299    3455890
accountsservice  query and manipulate user account i    19     261704
acl              Access control list utilities       11      91106
acpi-support     scripts for handling many ACPI even   18      13896
acpid            Advanced Configuration and Power In   19      86126
adduser          add and remove users and groups       81     246658
adium-theme-ubuntu Adium message style for Ubuntu      137    126759
adwaita-icon-theme default icon theme of GNOME (small) 1127   4848678
aisleriot         GNOME solitaire card game collectio  316    1890864
alsa-base         ALSA driver configuration files      42     195295
```

To delete a view we use the `DROP VIEW` statement as follows:

```
sqlite> DROP VIEW Stats;
```

Indexes

It's been said that the three most important features of a database system are "performance, performance, and performance." While this is not exactly true (things like data integrity and reliability are important, too), complex operations on large databases can get really slow, so it's important to make things as fast as we can. One feature we can take advantage of are *indexes*.

An index is a data structure the database maintains that speeds up database searches. It's a sorted list of rows in a table ordered by elements in one or more columns. Without an index, a table is sorted according to values in a hidden column called `rowid`. The values in this column are integers that start with 1 and increment each time a row is added to the table. Here we see a query that selects the 100th row from the `Package_Stats` table.

```
sqlite> .header on
sqlite> .mode column
sqlite> .width 20 -5 -8 -8 -8 -8
sqlite> SELECT * FROM Package_Stats WHERE rowid = 100;
package_name      count    tot_size   min_size   max_size   avg_size
-----  -----  -----  -----  -----  -----
cups-server-common      595     1996400        0     370070   3355.294
```

Our database server can locate this row in the table very quickly because it already knows where to find the 100th row. However, if we want to search for the row that contains package name `cups-server-common`, SQLite must examine every row in the table to locate the matching row. To facilitate performance tuning, SQLite provides a way to see what search strategy is used during a query.

```
sqlite> .width -8 -5 -4 55
sqlite> EXPLAIN QUERY PLAN
...> SELECT * FROM Package_Stats WHERE package_name = 'cups-server-common';
selectid  order  from  detail
-----  -----  -----
0        0        0  SCAN TABLE Package_Stats
```

We can see from the `SCAN TABLE Package_Stats` the SQLite performs a sequential search of the table during this query.

To create an index to allow faster searches of `package_name` we can do the following:

```
sqlite> CREATE INDEX idx_package_name
...> ON Package_Stats (package_name);
```

After doing this, we'll look at the query plan and see the difference.

```
sqlite> EXPLAIN QUERY PLAN
...> SELECT * FROM Package_Stats WHERE package_name = 'cups-server-common';
selectid  order  from  detail
-----  -----  -----
0        0        0  SEARCH TABLE Package_Stats USING INDEX
idx_package_name
```

Hereafter, when we search the table for a package name, SQLite will use the index to directly get to the row rather than looking at every row searching for a match. So why don't we just index everything? The reason we don't is that indexes impose overhead every time a row is inserted, deleted, or updated since the indexes must be kept up to date. Indexes are best used on tables that are read more often than written to.

We probably won't see much of a performance improvement when searching the `Package_Stats` table because it's just not that big, but on a large table the improvement can be substantial.

We can see the index when we examine the table's schema.

```
sqlite> .schema Package_Stats
CREATE TABLE Package_Stats (
    package_name  VARCHAR(60),
    count         INTEGER,
    tot_size      INTEGER,
    min_size      INTEGER,
    max_size      INTEGER,
    avg_size      REAL
);
CREATE INDEX idx_package_name
ON Package_Stats (package_name);
```

SQLite also has a dot command.

```
sqlite> .indexes
idx_package_name
```

Another benefit of using an index is that it's kept in sorted order (that's how it performs searches quickly). The side effect is that when an index is used during a query the results of the query will be sorted as well. To demonstrate, we'll create another index for the `Package_Stats` table, this time using the `tot_size` column. Notice that when we perform a select based on that column, the results are in ascending order.

```
sqlite> CREATE INDEX idx_tot_size
...> ON Package_Stats (tot_size);
sqlite> .width 20 -5 -10 -8 -8 -8
sqlite> SELECT * FROM Package_Stats
...> WHERE tot_size > 100000000;
package_name          count   tot_size   min_size   max_size   avg_size
-----  -----  -----  -----  -----  -----
inkscape              1025   127507308     0  19599216  124397.3
libreoffice-core       119   135106135     26  66158968  1135345.
linux-modules-extra-  4172   165884678   1292  4216105  39761.42
linux-modules-extra-  4173   165921470   1292  4216105  39760.71
thunderbird            69   180861838     27  12163098  2621186.
firefox                74   203393773     23  12408360  2748564.
google-chrome-stable   100   235727530     25  16288078  2357275.
libgl1-mesa-dri:amd6   20   237774005     36  19548840  11888700
codium                 1307  271907088     17  11551467  208039.0
linux-firmware          1938  331497257      6  19922416  171051.2
```

To delete our indexes, we use the `DROP INDEX` statement.

```
sqlite> DROP INDEX idx_package_name;
sqlite> DROP INDEX idx_tot_size;
```

Triggers and Stored Procedures

As we saw earlier during our discussion of views, SQL allows us to store SQL code in the database. Besides views, SQL provides for two other ways of storing code. These two methods are *stored procedures* and *triggers*. Stored procedures, as the name implies, allows a block of SQL statements to be stored and treated as a subroutine available to other SQL programs, or for use during ad hoc interactions with the database. Creating a stored procedure is done with this syntax:

```
CREATE PROCEDURE procedure_name
AS
    [block of SQL code];
```

Parameters can be passed to stored procedures. Here is an example:

```
CREATE PROCEDURE list_pkg_files @package VARCHAR(60)
AS
    SELECT package_name, file
        FROM Package_Files
    WHERE package_name = @package;
```

To call this procedure, we would do this:

```
EXEC list_package_files @package = 'bash';
```

Unfortunately, SQLite does not support stored procedures. It does, however, support the second method of code storage, triggers.

Triggers are stored blocks of code that are automatically called when some event occurs and a specified condition is met. Triggers are typically used to perform certain maintenance tasks to keep the database in good working order.

Triggers can be set to activate before, after, or instead of the execution of `INSERT`, `DELETE`, or `UPDATE` statements. In the example below, we will have a trigger activate before a `DELETE` is performed on the `Package_Files` table.

```
/*
    trigger_demo.sql

    Trigger demo where we create a "trash can" for the
    Package_Files table and set a trigger to copy rows
    to the PF_Backup table just before they are deleted
    from Package_Files.
*/

-- Create backup table with the same schema as Package_Files

CREATE TABLE IF NOT EXISTS PF_Backup (
    package_name VARCHAR(60),
    file         VARCHAR(120),
    size_bytes   INTEGER
);

-- Define trigger to copy rows into PF_Backup as they are
-- deleted from Package_Files

CREATE TRIGGER backup_row_before_delete
BEFORE DELETE ON Package_Files
BEGIN
    INSERT INTO PF_Backup
        VALUES (OLD.package_name, OLD.file, OLD.size_bytes);
END;
```

The first thing we do is create a table to hold our deleted rows. We use a slightly different form of the `CREATE TABLE` statement to create the table only if it does not already exist.

This will ensure that an existing table of saved rows will persist, even if we reload the trigger.

After creating the table, we create a trigger called `backup_row_before_delete` to copy data from the `Package_Files` table to the `PF_Backup` table just before any row in `Package_Files` is deleted.

In order to reference data that might be used by the trigger, SQL provides the `NEW` reference for new data that is inserted or updated, and the `OLD` reference for previous data that is updated or deleted. In our example, we use the `OLD` reference to refer to the data about to be deleted.

Performing Backups

Since SQLite uses an ordinary file to store each database (as opposed to the exotic methods used by some other systems), we can use regular command line tools such as `cp` to perform database backups. There is an interesting SQL method we can use, too. The `.dump` dot command will produce a stream of SQL statements that will fully reproduce the database including tables, views, triggers, etc. To output the database this way, we need only do the following:

```
sqlite> .dump
```

The stream will appear on standard output or we can use the `.output` dot command to direct the stream to the file of our choice.

One interesting application of this technique would be to combine tables from multiple databases into one. For example, let's imagine we had several Raspberry Pi computers each performing data logging of an external sensor. We could collect dumps from each machine and combine all of the tables into a single database for data analysis and reporting.

Generating Your Own Datasets

Below are the programs used to create the datasets used in this adventure. They are included in the archive for those who want to create their own datasets.

For Deb-based Systems (Debian, Ubuntu, Mint, Raspberry Pi OS)

The first program named `mk_package_descriptions-deb`, extracts package information and outputs a `.tsv` file.

```
#!/bin/bash

# mk_package_descriptions-deb - Output tsv list of installed debian/ubuntu
#                               packages on standard output

phase1() { # Replace repeated spaces with a tab
```

```

awk '
{
    gsub(/\[ \] [ ]+/, "\t")
    print $0
}'
return
}

phase2() { # Output field 2 and 5 separated by a tab
awk '
BEGIN {
    FS = "\t"
}

$1 == "ii" {
    print $2 "\t" $5
}'
return
}

dpkg-query -l | phase1 | phase2

```

The second program, `mk_package_files-deb` outputs all the files included in each package.

```

#!/bin/bash

# mk_package_files - make list of files in all packages
# Reads *.list files in LIST_DIR. Outputs stream of tsv to stdout.

LIST_DIR=/var/lib/dpkg/info

mk_list () {

local list_file="$1"
local lf_length="${#list_file}"
local len
local package
local -a files

((len = lf_length - 5))
package="${list_file}:0:$len}" # strip '.list' extension
package="${package##*/}" # strip leading pathname
mapfile files < "$list_file" # load list into array
for i in "${files[@]}"; do
    i="${i//[$'\t\r\n']}" # strip trailing newlines
    if [[ -f "$i" ]]; then # write tsv file
        printf "%s\t%s\t%s\n" \
            "$package" \
            "$i" \
            "$(stat --printf '%s' "$i")" # size of file
    fi
done
return
}

for i in "$LIST_DIR"/*.list; do
    mk_list "$i"
done

```

For RPM-based Systems (RHEL, CentOS, Fedora)

The `mk_package_descriptions-rpm` script:

```
#!/bin/bash

# mk_package_descriptions-rpm - Output tsv list of installed Fedora/CentOS
#                               packages on standard output

while read package; do
    description=$(dnf info "$package" \
        | awk '$1 == "Summary" { gsub(/Summary      : /, ""); print $0; exit }')
    printf "%s\t%s\n" \
        "$package" \
        "$description"
done <<(> dnf list installed | awk 'NF == 3 { print $1 }' )
```

The `mk_package_files-rpm` script:

```
#!/bin/bash

# mk_package_files-rpm - Output tsv list of installed Fedora/CentOS files
#                       on standard output

while read package; do
    while read package_file; do
        if [[ -r "$package_file" ]]; then # not all files are present/readable
            printf "%s\t%s\t%s\n" \
                "$package" \
                "$package_file" \
                "$((stat --printf '%s' "$package_file"))"
        fi
    done <<(> rpm -ql "$package" )
done <<(> dnf list installed | awk 'NF == 3 { print $1 }' )
```

Converting .tsv to SQL

Below are two AWK programs used to convert the `.tsv` files into SQL. First, the `insert_package_descriptions.awk` program:

```
#!/usr/bin/awk -f

# insert_package_descriptions.awk - Insert records from
#                                   package_descriptions.tsv

BEGIN {
    FS="\t"
    print "DROP TABLE IF EXISTS Package_Descriptions;"
    print "CREATE TABLE Package_Descriptions (" 
    print "    package_name VARCHAR(60),"
    print "    description  VARCHAR(120)"
    print ");"
    print "BEGIN TRANSACTION;" # vastly improves performance
}

{
    gsub(/\', \"\""/, '\"') # double up single quotes to escape them
    print "INSERT INTO Package_Descriptions"
    print "    VALUES ( '" $1 "', '" $2 "' );"
}
```

```
END {
    print "COMMIT;"
```

Second, the `insert_package_files.awk` program:

```
#!/usr/bin/awk -f

# insert_package_files.awk - Insert records from
#                             package_files.tsv

BEGIN {
    FS="\t"
    print "DROP TABLE IF EXISTS Package_Files;"
    print "CREATE TABLE Package_Files (" 
    print "    package_name VARCHAR(60),"
    print "    file        VARCHAR(120),"
    print "    size_bytes  INTEGER"
    print ");"
    print "BEGIN TRANSACTION;" # vastly improves performance
}

{
    gsub(/\', \"\"") # double up single quotes to escape them
    print "INSERT INTO Package_Files"
    print "    VALUES ('" $1 "', '" $2 "', '" $3 "');"
}

END {
    print "COMMIT;"
```

Summing Up

SQL is an important and widely used technology. It's kinda fun too. While we looked at the general features and characteristics of SQL, there is much more to learn. For example, there are the more advanced concepts such as *normalization*, *referential integrity*, and *relational algebra*. Though we didn't get to the really heady stuff, we did cover enough to get some real work done whenever we need to integrate data storage into our scripts and projects.

Further Reading

- *SQL Style Guide by Simon Holywell* The style guide that influenced the formatting of the SQL presented in this adventure. <https://www.sqlstyle.guide/>
- *SQL Tutorial* A good general tutorial on SQL. Covers most of the major SQL dialects (though not SQLite). <https://www.w3schools.com/sql/>
- *SQLite project home page* Includes detailed documentation.
<https://www.sqlite.org/index.html>

- *SQLite Tutorial* An excellent detailed tutorial and reference for the SQLite dialect of SQL. Definitely worth checking out if you plan to use SQLite seriously.
<https://www.sqlitetutorial.net/>
- *MySQL home page* A very popular multi-user database system used by many websites. <https://mysql.com/>
- *MariaDB Foundation home page* MariaDB is a fork of the open source version of MySQL. The project came about out of concern over Oracle's purchase of the company behind MySQL. <https://mariadb.org/>
- *PostgreSQL project home page* Another very popular open source multi-user database. <https://www.postgresql.org/>
- Various Wikipedia articles providing background for the topics covered in this adventure:
 - *SQL* <https://en.wikipedia.org/wiki/SQL>
 - *Relational Database* https://en.wikipedia.org/wiki/Relational_database
 - *Relational Model* https://en.wikipedia.org/wiki/Relational_model
 - *LAMP Software Stack*
[https://en.wikipedia.org/wiki/LAMP_\(software_bundle\)](https://en.wikipedia.org/wiki/LAMP_(software_bundle))
 - *MySQL* <https://en.wikipedia.org/wiki/MySQL>
 - *MariaDB* <https://en.wikipedia.org/wiki/MariaDB>
 - *PostgreSQL* <https://en.wikipedia.org/wiki/PostgreSQL>
- Stanford University offers an online course called “Introduction to Databases” taught by Professor Jennifer Widom. I took this course; it’s quite good. The course videos are available in this YouTube playlist: <https://www.youtube.com/playlist?list=PLroEs25KGvwzmvIxYHRhoGTz9w8LeXek0>
- Finally, the phrase “gird your grid for a big one” is taken from the 1971 comedy album *I Think We’re All Bozos on This Bus* by the Firesign Theater. Besides being very funny to retro-futurism fans like myself, the album is notable because it contains one of the first pop culture references to computer hacking. You can read more about it at Wikipedia: https://en.wikipedia.org/wiki/I_Think_We%27re_All_Bozos_on_This_Bus

Index

A

Alfred Aho.....91
aliases.....49, 230
Almquist shell.....145
ALTER TABLE statement.....235
Android.....135
ANSI standards for SQL.....223
Apache.....219
appending to files.....105
arctangent.....108
arithmetic operators.....99
arrays.....99
AS clause.....230
associative arrays.....99, 143
AT&T Bell Telephone Laboratories.....91, 143
atan2 function.....108
attributes.....219
autocomplete.....
 in vim.....163
autoindent.....166
AWK one-liners.....113

B

background color.....68
banner command.....72
bash builtins.....202
batch processing.....196
BEGIN.....93
Bill Joy.....143
blinking text.....67
bold text.....67
Bourne shell.....143
brace delimiting variables.....203
break statement.....103
Brian Fox.....144
Brian Kernighan.....91
buffers.....160
busybox.....137

C

C programming language.....91, 100
C shell.....143
calendar.....77
camelCase.....200
capnames.....64
case conversion.....
 in Midnight Commander.....15

 in vim.....167, 172
checklist.....77
Chet Ramey.....144
chsh command.....152
clear screen.....70
client/server architecture.....221
coding standards.....195
color schemes.....
 in gnome-terminal.....124
 in terminal emulators.....117
 in vim.....162
columnar data.....91
comma separated values.....111
command history.....143
 in Midnight Commander.....5
 in readline.....51
 in vim.....158
command line options.....213
command mode.....155
command prefix.....
 in byobu.....43
 in GNU screen.....30
 in tmux.....37
command substitution.....84, 202
commenting out code.....170
comments.....94, 197, 202
compound commands.....201
configuration files.....187
Connectbot.....135
constant names.....200
constants.....198
continue statement.....103
copying files.....
 in Midnight Commander.....12
copying text.....176
 in byobu.....45
 in gnome-terminal.....122
 in GNU screen.....32
 in readline.....52
 in tmux.....39
 in vim.....164, 167, 173, 175
cos function.....108
cosine.....108
countdown timer.....78
CREATE TABLE statement.....224
creating directories.....

in Midnight Commander.....	10	dot commands.....	222
CSV files.....	111	DROP TABLE statement.....	224
cursor movement.....			
in readline.....	51	E	
in tput.....	66	E. F. Cobb.....	219
in vim.....	163, 167	echo.....	204
		edit box.....	77
D		editing files.....	
dash shell.....	145, 193	in dialog.....	77
David Korn.....	143	in Midnight Commander.....	8
DEC VT102.....	120	egrep.....	95
DEC VT220.....	119	END.....	93
delete arrays.....	99	error message function.....	188
DELETE statement.....	234	ESC key.....	
deleting files.....		in Midnight Commander.....	3
in Midnight Commander.....	17	exec command.....	58, 83
deleting text.....		exit statement.....	103
in readline.....	52	exit status.....	199
Dennis Richie.....	143	exp function.....	109
detaching sessions.....			
in byobu.....	43, 46	F	
in GNU screen.....	34	field separator.....	96 f.
in tmux.....	40	fields.....	92
dialog boxes.....	77	file descriptor duplication.....	58
calendar.....	77	file descriptors.....	57
checklist.....	77	file modes.....	16
directory selection.....	77	file record number.....	99
edit box.....	77	file selector.....	77, 80
file selector.....	77, 80	FILENAME.....	99
form.....	77	filetype plugin.....	158, 180
gauge.....	77	filtering text.....	179
info box.....	77	filters.....	91
input box.....	77	finding files.....	
menu box.....	77, 88	in Midnight Commander.....	19
message box.....	78, 88	in vim.....	181
password box.....	78	Firesign Theater.....	246
pause.....	78	FISH protocol.....	
program box.....	78	in Midnight Commander.....	19
progress box.....	78	flow control statements.....	100
radio list box.....	79	FNR.....	99
range box.....	78	focus follows mouse.....	134
tail box.....	78	for loop.....	102
text box.....	78	foreground color.....	68
time box.....	78	form.....	77
tree view.....	78	FS.....	96 f.
yes/no box.....	78 f.	FTP.....	
directory selection.....	77	in Midnight Commander.....	19
display system health.....	191	function names.....	200
do loop.....	103	function statement.....	110
DOS format files.....	172	FVWM.....	120

G	
gauge.....	77
gawk.....	91
getline.....	106
global alias.....	150
GNOME.....	115
gnome-terminal.....	117, 121
GNU GPLv3.....	144
gpm.....	3
gsub.....	107
guake.....	131
H	
Hacker's Keyboard.....	135
help file.....	157
help message.....	206, 213
hostname command.....	190
I	
I Think We're All Bozos on This Bus.....	246
if/then/else construct.....	101
indentation.....	164, 167, 200
index function.....	107
indexes.....	238
info box.....	77
infocmp command.....	63
inner join.....	237
input box.....	77
input files.....	105
insert mode.....	155
INSERT statement.....	223, 233
inserting boilerplate text.....	176
install a package.....	191
int function.....	109
integer arithmetic.....	203
invisible text.....	67
J	
job control.....	143
joins.....	230
K	
KDE.....	115
Ken Thompson.....	143
keyboard shortcuts.....	175
in gnome-terminal.....	122
keys.....	220
konsole.....	117, 128
Korn shell.....	143
ksh88.....	146
ksh93.....	143, 146
L	
LAMP stack.....	221
leader character.....	177
length function.....	107
LIMIT clause.....	226
line continuation characters.....	199
line length.....	199
in vim.....	169
line wrap.....	183
load function library.....	189
local variables.....	110
log function.....	109
logical operators.....	95, 99
long form option names.....	200
long option names.....	205
LXDE.....	115
M	
macOS.....	144, 197
macros.....	172
MariaDB.....	221
Mashey shell.....	143
match function.....	107
mawk.....	91
mcedit.....	8
menu box.....	77, 88
message box.....	77 f., 88
Midnight Commander.....	80
current panel.....	3
hotlist.....	4
information mode.....	4
Meta-key.....	3
other panel.....	3
mktemp command.....	208
multi-dimensional arrays.....	99
MySQL.....	219, 221
N	
natural logarithm.....	109
nawk.....	91
ncurses.....	63
netrw plugin.....	180
next statement.....	103
NF.....	98
noclobber option.....	61
normal mode.....	155
NR.....	98
number of fields.....	98
O	
OFS.....	98

Oh-My-Zsh.....	151
ORDER BY clause.....	228
ORS.....	98
OS X.....	146
output field separator.....	98
output record separator.....	98
P	
parameter expansion.....	202
passed by reference.....	110
passed by value.....	110
password box.....	78
pasting text.....	
in byobu.....	45
in GNU screen.....	32
in tmux.....	40
PATH variable.....	193, 196
pathname expansion.....	149
pattern negation.....	96
pattern/action pairs.....	92
pause.....	78
pdksh.....	146
Peter Weinberger.....	91
piped command.....	78
pipelines.....	105, 200
positional parameters.....	50, 193, 199
POSIX.....	144, 196, 203
PostgreSQL.....	221
print.....	104
printf.....	104, 204
program box.....	78
progress box.....	78
progress indicator.....	77
Q	
queries.....	225
R	
radio list box.....	79
radio list box.....	78
rand function.....	109
random numbers.....	109
range box.....	78
range pattern.....	96
Raspberry Pi.....	v
rc files.....	192
RDBMS.....	219
readline.....	50
record.....	92
record number.....	98
record separator.....	96, 99
region.....	
in GNU screen.....	30
in tmux.....	37
registers.....	173
regular expressions.....	95, 103
relational database management systems.....	219
relational expressions.....	95
renaming files.....	
in Midnight Commander.....	13
REPLYTO variable.....	214
report generators.....	91
return statement.....	199
reverse incremental history search.....	151
reverse video.....	68
revision history.....	197
root shell.....	190
ROT13 encoding.....	167
RS.....	96, 99
rxvt.....	120
S	
scalar variables.....	110
schema.....	220
script maintenance.....	187
script templates.....	211
scrollback mode.....	
in GNU screen.....	32
in tmux.....	39
Secure Shell for Chrome.....	139
sed.....	91
seed random number generator.....	109
SELECT statement.....	226
session.....	
in GNU screen.....	30
in tmux.....	36
terminal multiplexer.....	30
set -e.....	205
set -o PIPEFAIL.....	205
set -u.....	205
setting time.....	78
shebang.....	94, 188, 197
shell builtins.....	202
shell functions.....	49
shellcheck.....	208
SIGHUP.....	207
SIGINT.....	207
SIGTERM.....	207
sin function.....	109
sine.....	109
single dimension arrays.....	99
SMB/CIFS.....	

in Midnight Commander.....	19
sorting database output.....	228
spell checking.....	183
split function.....	108
sprintf function.....	108
SQL.....	219
SQLite.....	221
SQLite output modes.....	227
sqrt function.....	109
square root.....	109
rand function.....	109
stderr.....	57
stdin.....	57
stdout.....	57
Steve Bourne.....	143
stored procedures.....	240
string concatenation.....	97
Structured Query Language.....	219
sub function.....	108
subqueries.....	229 f.
subshell.....	147
subshells.....	178
in Midnight Commander.....	22
in vim.....	159
substr function.....	108
suffix alias.....	151
symbolic links.....	15
syntax highlighting.....	160
T	
tab characters.....	165
tab completion.....	148
tab separated value.....	112
tables.....	219
tabs.....	
in terminal emulators.....	116
in vim.....	160
tail box.....	78
tcsh.....	145
Tektronix 4014 graphics terminal.....	119
teletype machines.....	165
template generator.....	211
temporary file.....	81
temporary files.....	207
TERM variable.....	63, 120
terminal capability names.....	64
terminal multiplexer.....	29
terminal profiles.....	122
terminal type.....	63
terminal width.....	116
terminator.....	132
terminfo.....	63, 119
Termux.....	137
test if host is available.....	189
test if program is installed.....	189
testing array membership.....	102
text box.....	78
text completion.....	
in readline.....	53
in vim.....	174
text editing operators.....	167
text file viewer.....	78
text object selection.....	168
Thompson shell.....	143
time box.....	78
TODO comments.....	202
tput.....	63
traps.....	207
tree view.....	78
triggers.....	240
TRS-80.....	v
truncating files.....	105
TSV files.....	112
tuples.....	219
typewriters.....	165
U	
underlined text.....	67
Unicode.....	120
update a system.....	192
UPDATE statement.....	232
urxvt.....	120
usage message.....	216
using the mouse.....	
in Midnight Commander.....	5
in vim.....	185
V	
variable names.....	200
variables.....	97
vendor lock-in.....	224
version number.....	198
view terminal full screen.....	122
viewing files.....	
in dialog.....	78
in Midnight Commander.....	7
views.....	238
visual block selection.....	170
visual mode.....	164
W	
WHERE clause.....	227

while loop.....	103	zsh.....	144, 148
window.....		.	
create.....		.bashrc.....	50, 139, 152, 155, 187, 190, 192, 214
in byobu.....	43	.tar files.....	18
in GNU screen.....	30		
in tmux.....	37	/	
in GNU screen.....	30	/bin/sh.....	145
split.....		/dev/tty.....	60
in byobu.....	43	/etc/X11/Xresources.....	118
in GNU screen.....	33	/tmp.....	208
in konsole.....	128, 130	/usr/local/bin.....	196
in terminator.....	132	/usr/local/etc.....	197
in tmux.....	37	/usr/local/lib.....	197
in vim.....	156, 176, 180	/usr/local/sbin.....	197
terminal multiplexer.....	30	/usr/share/dict/words.....	183
word splitting.....	202		
wrap lines.....	183	~	
writing to files.....	105	~/.kshrc.....	152
X		~/.profile.....	152
XFCE.....	115	~/.tchrc.....	152
xterm.....	63, 117	~/.vim/ftplugin.....	159
		~/.vim/ftplugin/sh.vim.....	159, 185
Y		~/.vim/ftplugin/text.vim.....	185
yes/no box.....	78 f.	~/.vimrc.....	155, 185
		~/.Xresources.....	118
Z		~/.zshrc.....	152
Z shell.....	144		

Adventures with the Linux Command Line was written in Markdown using Vim on a Raspberry Pi 2B running Raspbian. It was converted into HTML and Open Document format using Pandoc. The PDF version was created with LibreOffice Writer on a System 76 Ratel workstation running Ubuntu 20.04. The text is set in Liberation Serif and Liberation Sans.