O'REILLY®

Efficient Linux at the Command Line

Boost Your Command-Line Skills

Free Chapter

Daniel J. Barrett
Author of *Linux Pocket Guide*

Efficient Linux at the Command Line

Boost Your Command-Line Skills

This excerpt contains Chapter 4. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Daniel J. Barrett

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Efficient Linux at the Command Line
by Daniel J. Barrett

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Gregory Hyman
Copyeditor: Kim Wimpsett
Proofreader: Piper Editorial Consulting, LLC
Indexer: Daniel J. Barrett
Index Editor: Sue Klefstad
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

February 2022: First Edition

Revision History for the First Edition
2022-02-16: First Release
2022-04-01: Second Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098113407* for release details.

978-1-098-11340-7

[LSI]

Table of Contents

CHAPTER 4

Cruising the Filesystem

In the movie *The Adventures of Buckaroo Banzai Across the 8th Dimension*, a classic cult comedy from 1984, the swashbuckling title character offers the following Zen-like words of wisdom: "Remember, no matter where you go…there you are." Buckaroo could very well have been talking about the Linux filesystem:

```
$ cd /usr/share/lib/etc/bin            No matter where you go...
$ pwd
/usr/share/lib/etc/bin                 ...there you are.
```

It's also the case that wherever you are in the Linux filesystem—your current directory—you will eventually go somewhere else (to another directory). The faster and more efficiently you can perform this navigation, the more productive you can be.

The techniques in this chapter will help you navigate the filesystem more quickly with less typing. They look deceptively simple but have *enormous* bang for the buck, with small learning curves and big payoffs. These techniques fall into two broad categories:

- Moving quickly to a specific directory
- Returning rapidly to a directory you've visited before

For a quick refresher on Linux directories, see Appendix A. If you use a shell other than `bash`, see Appendix B for additional notes.

Visiting Specific Directories Efficiently

If you ask 10 Linux experts what is the most tedious aspect of the command line, seven of them will say, "Typing long directory paths."[1] After all, if your work files are in */home/smith/Work/Projects/Apps/Neutron-Star/src/include*, your financial documents are in */home/smith/Finances/Bank/Checking/Statements*, and your videos are in */data/Arts/Video/Collection*, it's no fun to retype these paths over and over. In this section, you'll learn techniques to navigate to a given directory efficiently.

Jump to Your Home Directory

Let's begin with the basics. No matter where you go in the filesystem, you can return to your home directory by running `cd` with no arguments:

```
$ pwd
/etc                               Start somewhere else
$ cd                               Run cd with no arguments...
$ pwd
/home/smith                        ...and you're home again
```

To jump to subdirectories within your home directory from anywhere in the filesystem, refer to your home directory with a shorthand rather than an absolute path such as */home/smith*. One shorthand is the shell variable `HOME`:

```
$ cd $HOME/Work
```

Another is a tilde:

```
$ cd ~/Work
```

Both `$HOME` and ~ are expressions expanded by the shell, a fact that you can verify by echoing them to stdout:

```
$ echo $HOME ~
/home/smith /home/smith
```

The tilde can also refer to another user's home directory if you place it immediately in front of their username:

```
$ echo ~jones
/home/jones
```

1 I made this up, but it's surely true.

Move Faster with Tab Completion

When you're entering cd commands, save typing by pressing the Tab key to produce directory names automatically. As a demonstration, visit a directory that contains subdirectories, such as */usr*:

```
$ cd /usr
$ ls
bin  games  include  lib  local  sbin  share  src
```

Suppose you want to visit the subdirectory *share*. Type sha and press the Tab key once:

```
$ cd sha<Tab>
```

The shell completes the directory name for you:

```
$ cd share/
```

This handy shortcut is called *tab completion*. It works immediately when the text that you've typed matches a single directory name. When the text matches multiple directory names, your shell needs more information to complete the desired name. Suppose you had typed only s and pressed Tab:

```
$ cd s<Tab>
```

The shell cannot complete the name *share* (yet) because other directory names begin with s too: *sbin* and *src*. Press Tab a second time and the shell prints all possible completions to guide you:

```
$ cd s<Tab><Tab>
sbin/  share/  src/
```

and waits for your next action. To resolve the ambiguity, type another character, h, and press Tab once:

```
$ cd sh<Tab>
```

The shell completes the name of the directory for you, from *sh* to *share*:

```
$ cd share/
```

In general, press Tab once to perform as much completion as possible, or press twice to print all possible completions. The more characters you type, the less ambiguity and the better the match.

Tab completion is great for speeding up navigation. Instead of typing a lengthy path like */home/smith/Projects/Web/src/include*, type as little as you want and keep pressing the Tab key. You'll get the hang of it quickly with practice.

Tab Completion Varies by Program

Tab completion isn't just for `cd` commands. It works for most commands, though its behavior may differ. When the command is `cd`, the Tab key completes directory names. For other commands that operate on files, such as `cat`, `grep`, and `sort`, tab completion expands filenames too. If the command is `ssh` (secure shell), it completes hostnames. If the command is `chown` (change the owner of a file), it completes usernames. You can even create your own completion rules for speed, as we'll see in Example 4-1. Also see `man bash` and read its topic "programmable completion."

Hop to Frequently Visited Directories Using Aliases or Variables

If you visit a faraway directory frequently, such as */home/smith/Work/Projects/Web/src/include*, create an alias that performs the `cd` operation:

```
# In a shell configuration file:
alias work="cd $HOME/Work/Projects/Web/src/include"
```

Simply run the alias anytime to reach your destination:

```
$ work
$ pwd
/home/smith/Work/Projects/Web/src/include
```

Alternatively, create a variable to hold the directory path:

```
$ work=$HOME/Work/Projects/Web/src/include
$ cd $work
$ pwd
/home/smith/Work/Projects/Web/src/include
$ ls $work/css                          Use the variable in other ways
main.css  mobile.css
```

Edit Frequently Edited Files with an Alias

Sometimes, the reason for visiting a directory frequently is to edit a particular file. If that's the case, consider defining an alias to edit that file by absolute path without changing directory. The following alias definition lets you edit *$HOME/.bashrc*, no matter where you are in the filesystem, by running `rcedit`. No `cd` is required:

```
# Place in a shell configuration file and source it:
alias rcedit='$EDITOR $HOME/.bashrc'
```

If you regularly visit lots of directories with long paths, you can create aliases or variables for each of them. This approach has some disadvantages, however:

- It's hard to remember all those aliases/variables.
- You might accidentally create an alias with the same name as an existing command, causing a conflict.

An alternative is to create a shell function like the one in Example 4-1, which I've named `qcd` ("quick cd"). This function accepts a string key as an argument, such as `work` or `recipes`, and runs `cd` to a selected directory path.

Example 4-1. A function for `cd`-ing to faraway directories

```
# Define the qcd function
qcd () {
  # Accept 1 argument that's a string key, and perform a different
  # "cd" operation for each key.
  case "$1" in
    work)
      cd $HOME/Work/Projects/Web/src/include
      ;;
    recipes)
      cd $HOME/Family/Cooking/Recipes
      ;;
    video)
      cd /data/Arts/Video/Collection
      ;;
    beatles)
      cd $HOME/Music/mp3/Artists/B/Beatles
      ;;
    *)
      # The supplied argument was not one of the supported keys
      echo "qcd: unknown key '$1'"
      return 1
      ;;
  esac
  # Helpfully print the current directory name to indicate where you are
  pwd
}
# Set up tab completion
complete -W "work recipes video beatles" qcd
```

Store the function in a shell configuration file such as *$HOME/.bashrc*, source it, and it's ready to run. Type `qcd` followed by one of the supported keys to quickly visit the associated directory:

```
$ qcd beatles
/home/smith/Music/mp3/Artists/B/Beatles
```

As a bonus, the script's final line runs the command `complete`, a shell builtin that sets up customized tab completion for `qcd`, so it completes the four supported keys. Now you don't have to remember `qcd`'s arguments! Just type `qcd` followed by a space and press the Tab key twice, and the shell will print all the keys for your reference, and you can complete any of them in the usual way:

```
$ qcd <Tab><Tab>
beatles  recipes  video    work
$ qcd v<Tab><Enter>                       Completes 'v' to 'video'
/data/Arts/Video/Collection
```

Make a Big Filesystem Feel Smaller with CDPATH

The `qcd` function handles only the directories that you specify. The shell provides a more general `cd`-ing solution without this shortcoming, called a *cd search path*. This shell feature transformed how I navigate the Linux filesystem.

Suppose you have an important subdirectory that you visit often, named *Photos*. It's located at */home/smith/Family/Memories/Photos*. As you cruise around the filesystem, anytime you want to get to the *Photos* directory, you may have to type a long path, such as:

```
$ cd ~/Family/Memories/Photos
```

Wouldn't it be great if you could shorten this path to just *Photos*, no matter where you are in the filesystem, and reach your subdirectory?

```
$ cd Photos
```

Normally, this command would fail:

```
bash: cd: Photos: No such file or directory
```

unless you happen to be in the correct parent directory (*~/Family/Memories*) or some other directory with a *Photos* subdirectory by coincidence. Well, with a little setup, you can instruct `cd` to search for your *Photos* subdirectory in locations other than your current directory. The search is lightning fast and looks only in parent directories that you specify. For example, you could instruct `cd` to search *$HOME/Family/Memories* in addition to the current directory. Then, when you type `cd Photos` from elsewhere in the filesystem, `cd` will succeed:

```
$ pwd
/etc
$ cd Photos
/home/smith/Family/Memories/Photos
```

A cd search path works like your command search path, `$PATH`, but instead of finding commands, it finds subdirectories. Configure it with the shell variable `CDPATH`, which has the same format as `PATH`: a list of directories separated by colons. If your `CDPATH` consists of these four directories, for example:

```
$HOME:$HOME/Projects:$HOME/Family/Memories:/usr/local
```

and you type:

```
$ cd Photos
```

then `cd` will check the existence of the following directories in order, until it finds one or it fails entirely:

1. *Photos* in the current directory
2. *$HOME/Photos*
3. *$HOME/Projects/Photos*
4. *$HOME/Family/Memories/Photos*
5. */usr/local/Photos*

In this case, `cd` succeeds on its fourth try and changes directory to *$HOME/Family/Memories/Photos*. If two directories in `$CDPATH` have a subdirectory named *Photos*, the earlier parent wins.

Ordinarily, a successful `cd` prints no output. When `cd` locates a directory using your `CDPATH`, however, it prints the absolute path on stdout to inform you of your new current directory:

```
$ CDPATH=/usr       Set a CDPATH
$ cd /tmp           No output: CDPATH wasn't consulted
$ cd bin            cd consults CDPATH...
/usr/bin            ...and prints the new working directory
```

Fill `CDPATH` with your most important or frequently used parent directories, and you can `cd` into any of their subdirectories from anywhere in the filesystem, no matter how deep they are, without typing most of the path. Trust me, this is *awesome*, and the following case study should prove it.

Organize Your Home Directory for Fast Navigation

Let's use `CDPATH` to simplify the way you navigate your home directory. With a little configuration, you can make many directories within your home directory easily accessible with minimal typing, no matter where you are in the filesystem. This technique works best if your home directory is well organized with at least two levels of subdirectories. Figure 4-1 shows an example of a well-organized directory layout.

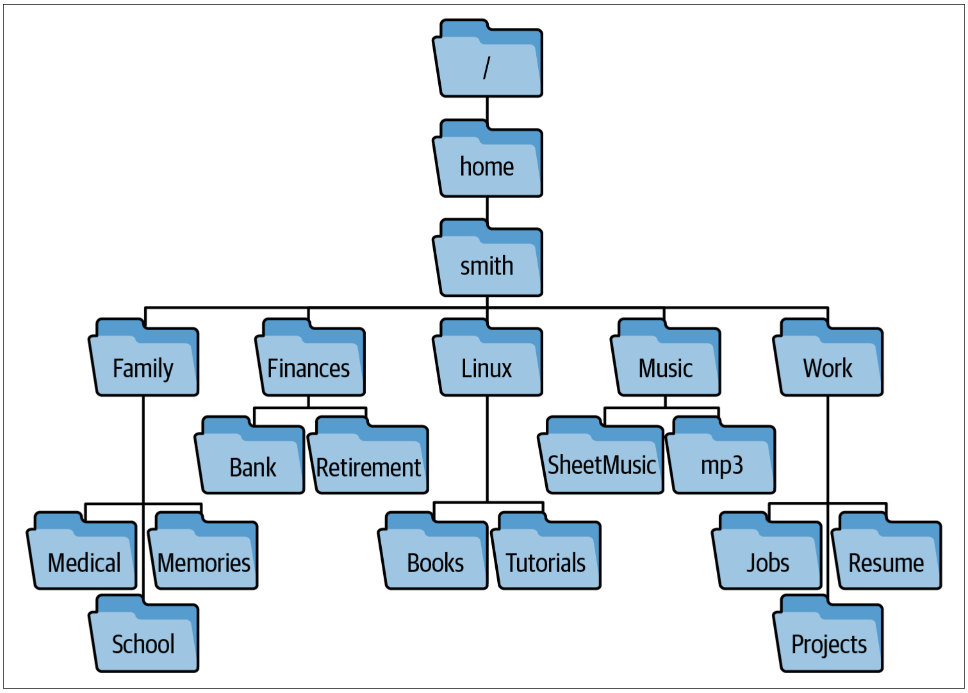

Figure 4-1. Two levels of subdirectories in the directory /home/

smith The trick is to set up your `CDPATH` to include, in order:

1. `$HOME`
2. Your choice of subdirectories of `$HOME`
3. The relative path for a parent directory, indicated by two dots (`..`)

By including `$HOME`, you can jump immediately to any of its subdirectories (*Family*, *Finances*, *Linux*, *Music*, and *Work*) from anywhere else in the filesystem without typing a leading path:

```
$ pwd
/etc                                    Begin outside your home directory
$ cd Work
/home/smith/Work
$ cd Family/School                      You jumped 1 level below $HOME
/home/smith/Family/School
```

By including subdirectories of `$HOME` in your `CDPATH`, you can jump into *their* subdirectories in one shot:

```
$ pwd
/etc                                    Anywhere outside your home directory
```

```
$ cd School
/home/smith/Family/School            You jumped 2 levels below $HOME
```

All the directories in your CDPATH so far are absolute paths in $HOME and its subdirectories. By including the relative path `..` however, you empower new `cd` behavior in *every* directory. No matter where you are in the filesystem, you can jump to any *sibling* directory (*../sibling*) by name without typing the two dots, because `cd` will search your current parent. For example, if you're in */usr/bin* and want to move to */usr/lib*, all you need is `cd lib`:

```
$ pwd
/usr/bin                            Your current directory
$ ls ..
bin   include   lib   src           Your siblings
$ cd lib
/usr/lib                            You jumped to a sibling
```

Or, if you're a programmer working on code that has subdirectories *src*, *include*, and *docs*:

```
$ pwd
/usr/src/myproject
$ ls
docs   include   src
```

you can jump between the subdirectories concisely:

```
$ cd docs                         Change your current directory
$ cd include
/usr/src/myproject/include        You jumped to a sibling
$ cd src
/usr/src/myproject/src            Again
```

A CDPATH for the tree in Figure 4-1 might contain six items: your home directory, four of its subdirectories, and the relative path for a parent directory:

```
# Place in a shell configuration file and source it:
export CDPATH=$HOME:$HOME/Work:$HOME/Family:$HOME/Linux:$HOME/Music:..
```

After sourcing the configuration file, you can `cd` to a large number of important directories without typing long directory paths, just short directory names. Hooray!

This technique works best if all subdirectories beneath the CDPATH directories have unique names. If you have duplicate names, such as *$HOME/Music* and *$HOME/Linux/Music*, you might not get the behavior you want. The command `cd Music` will always check *$HOME* before *$HOME/Linux* and consequently will not locate *$HOME/Linux/Music* by search.

To check for duplicate subdirectory names in the first two levels of $HOME, try this brash one-liner. It lists all subdirectories and sub-subdirectories of $HOME, isolates the sub-subdirectory names with `cut`, sorts the list, and counts occurrences with `uniq`:

```
$ cd
$ ls -d */ && (ls -d */*/ | cut -d/ -f2-) | sort | uniq -c | sort -nr | less
```

You may recognize this duplicate-checking technique from Chapter 1. If the output displays any counts greater than 1, you have duplicates. I realize this command includes a few features I haven't covered yet. You'll learn double ampersand (`&&`) and the parentheses in Chapter 7.

Returning to Directories Efficiently

You've just seen how to visit a directory efficiently. Now I'll show you how to revisit a directory quickly when you need to go back.

Toggle Between Two Directories with "cd -"

Suppose you're working in a deep directory and you run `cd` to go somewhere else:

```
$ pwd
/home/smith/Finances/Bank/Checking/Statements
$ cd /etc
```

and then think, "No, wait, I want to go back to the *Statements* directory where I just was." Don't retype the long directory path. Just run `cd` with a dash as an argument:

```
$ cd -
/home/smith/Finances/Bank/Checking/Statements
```

This command returns your shell to its previous directory and helpfully prints its absolute path so you know where you are.

To jump back and forth between a pair of directories, run `cd -` repeatedly. This is a time-saver when you're doing focused work in two directories in a single shell. There's a catch, however: the shell remembers just one previous directory at a time. For example, if you are toggling between */usr/local/bin* and */etc*:

```
$ pwd
/usr/local/bin
$ cd /etc                 The shell remembers /usr/local/bin
$ cd -                    The shell remembers /etc
/usr/local/bin
$ cd -                    The shell remembers /usr/local/bin
/etc
```

and you run `cd` without arguments to jump to your home directory:

```
$ cd                      The shell remembers /etc
```

the shell has now forgotten */usr/local/bin* as a previous directory:

```
$ cd -                    The shell remembers your home directory
/etc
$ cd -                    The shell remembers /etc
/home/smith
```

The next technique overcomes this limitation.

Toggle Among Many Directories with pushd and popd

The `cd -` command toggles between two directories, but what if you have three or more to keep track of? Suppose you're creating a local website on your Linux computer. This task often involves four or more directories:

- The location of live, deployed web pages, such as */var/www/html*
- The web-server configuration directory, often */etc/apache2*
- The location of SSL certificates, often */etc/ssl/certs*
- Your work directory, such as *~/Work/Projects/Web/src*

Believe me, it's tedious to keep typing:

```
$ cd ~/Work/Projects/Web/src
$ cd /var/www/html
$ cd /etc/apache2
$ cd ~/Work/Projects/Web/src
$ cd /etc/ssl/certs
```

If you have a large, windowing display, you can ease the burden by opening a separate shell window for each directory. But if you're working in a single shell (say, over an SSH connection), take advantage of a shell feature called a *directory stack*. It lets you quickly travel among multiple directories with ease, using the built-in shell commands `pushd`, `popd`, and `dirs`. The learning curve is maybe 15 minutes, and the huge payoff in speed lasts a lifetime.[2]

A *directory stack* is a list of directories that you've visited in the current shell and decided to keep track of. You manipulate the stack by performing two operations called *pushing* and *popping*. Pushing a directory adds it to the beginning of the list, which is traditionally called the *top* of the stack. Popping removes the topmost directory from the stack.[3] Initially, the stack contains only your current directory, but you can add (push) and remove (pop) directories and rapidly `cd` among them.

2 An alternative is to open multiple virtual displays using command-line programs like `screen` and `tmux`, which are called *terminal multiplexers*. They're more effort to learn than directory stacks but worth a look.

3 If you know stacks from computer science, a directory stack is precisely a stack of directory names.

Every running shell maintains its own directory stack.

I'll begin with the basic operations (pushing, popping, viewing) and then get to the good stuff.

Push a directory onto the stack

The command `pushd` (short for "push directory") does all of the following:

1. Adds a given directory to the top of the stack
2. Performs a `cd` to that directory
3. Prints the stack from top to bottom for your reference

I'll build a directory stack of four directories, pushing them onto the stack one at a time:

```
$ pwd
/home/smith/Work/Projects/Web/src
$ pushd /var/www/html
/var/www/html ~/Work/Projects/Web/src
$ pushd /etc/apache2
/etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ pushd /etc/ssl/certs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ pwd
/etc/ssl/certs
```

The shell prints the stack after each `pushd` operation. The current directory is the left-most (top) directory.

View a directory stack

Print a shell's directory stack with the `dirs` command. It does not modify the stack:

```
$ dirs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
```

If you prefer to print the stack from top to bottom, use the `-p` option:

```
$ dirs -p
/etc/ssl/certs
/etc/apache2
/var/www/html
~/Work/Projects/Web/src
```

and even pipe the output to the command `nl` to number the lines from zero onward:

```
$ dirs -p | nl -v0
     0  /etc/ssl/certs
     1  /etc/apache2
     2  /var/www/html
     3  ~/Work/Projects/Web/src
```

Even simpler, run `dirs -v` to print the stack with numbered lines:

```
$ dirs -v
 0  /etc/ssl/certs
 1  /etc/apache2
 2  /var/www/html
 3  ~/Work/Projects/Web/src
```

If you prefer this top-down format, consider making an alias:

```
# Place in a shell configuration file and source it:
alias dirs='dirs -v'
```

Pop a directory from the stack

The `popd` command ("pop directory") is the reverse of `pushd`. It does all of the following:

1. Removes one directory from the top of the stack
2. Performs a `cd` to the new top directory
3. Prints the stack from top to bottom for your reference

For example, if your stack has four directories:

```
$ dirs
/etc/ssl/certs /etc/apache2 /var/www/html ~/Work/Projects/Web/src
```

then repeatedly running `popd` will traverse these directories from top to bottom:

```
$ popd
/etc/apache2 /var/www/html ~/Work/Projects/Web/src
$ popd
/var/www/html ~/Work/Projects/Web/src
$ popd
~/Work/Projects/Web/src
$ popd
bash: popd: directory stack empty
$ pwd
~/Work/Projects/Web/src
```

The `pushd` and `popd` commands are such time-savers that I recommend creating two-character aliases that are as quick to type as `cd`:

```
# Place in a shell configuration file and source it:
alias gd=pushd
alias pd=popd
```

Swap directories on the stack

Now that you can build and empty the directory stack, let's focus on practical use cases. pushd with no arguments swaps the top two directories in the stack and navigates to the new top directory. Let's jump between */etc/apache2* and your work directory several times by simply running pushd. See how the third directory */var/www/html* remains in the stack as the first two directories swap positions:

```
$ dirs
/etc/apache2 ~/Work/Projects/Web/src /var/www/html
$ pushd
~/Work/Projects/Web/src /etc/apache2 /var/www/html
$ pushd
/etc/apache2 ~/Work/Projects/Web/src /var/www/html
$ pushd
~/Work/Projects/Web/src /etc/apache2 /var/www/html
```

Notice that pushd behaves similarly to the `cd -` command, toggling between two directories, but it does not have the limitation of remembering just one directory.

Turn a mistaken cd into a pushd

Suppose you are jumping among several directories with pushd and you accidentally run cd instead and lose a directory:

```
$ dirs
~/Work/Projects/Web/src /var/www/html /etc/apache2
$ cd /etc/ssl/certs
$ dirs
/etc/ssl/certs /var/www/html /etc/apache2
```

Oops, the accidental cd command replaced *~/Work/Projects/Web/src* in the stack with */etc/ssl/certs*. But don't worry. You can add the missing directory back to the stack without typing its long path. Just run pushd twice, once with a dash argument and once without:

```
$ pushd -
~/Work/Projects/Web/src /etc/ssl/certs /var/www/html /etc/apache2
$ pushd
/etc/ssl/certs ~/Work/Projects/Web/src /var/www/html /etc/apache2
```

Let's dissect why this works:

- The first `pushd` returns to your shell's previous directory, *~/Work/Projects/Web/src*, and pushes it onto the stack. `pushd`, like `cd`, accepts a dash as an argument to mean "go back to my previous directory."
- The second `pushd` command swaps the top two directories, bringing you back to */etc/ssl/certs*. The end result is that you've restored *~/Work/Projects/Web/src* to the second position in the stack, exactly where it would have been if you hadn't made your mistake.

This "oops, I forgot a pushd" command is useful enough that it's worth an alias. I call it `slurp` because in my mind, it "slurps back" a directory that I lost by mistake:

```
# Place in a shell configuration file and source it:
alias slurp='pushd - && pushd'
```

Go deeper into the stack

What if you want to `cd` between directories in the stack other than the top two? `pushd` and `popd` accept a positive or negative integer argument to operate further into the stack. The command:

```
$ pushd +N
```

shifts *N* directories from the top of the stack to the bottom and then performs a `cd` to the new top directory. A negative argument (`-N`) shifts directories in the opposite direction, from the bottom to the top, before performing the `cd`.[4]

```
$ dirs
/etc/ssl/certs ~/Work/Projects/Web/src /var/www/html /etc/apache2
$ pushd +1
~/Work/Projects/Web/src /var/www/html /etc/apache2 /etc/ssl/certs
$ pushd +2
/etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src /var/www/html
```

In this manner, you can jump to any other directory in the stack with a simple command. If your stack is long, however, it may be difficult to judge a directory's numeric position by eye. So, print the numeric position of each directory with `dirs -v`, as you did in "View a directory stack" on page 12:

```
$ dirs -v
 0  /etc/apache2
 1  /etc/ssl/certs
 2  ~/Work/Projects/Web/src
 3  /var/www/html
```

4 Programmers may recognize these operations as rotating the stack.

To shift */var/www/html* to the top of the stack (and make it your current directory), run `pushd +3`.

To jump to the directory at the bottom of the stack, run `pushd -0` (dash zero):

```
$ dirs
/etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src /var/www/html
$ pushd -0
/var/www/html /etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src
```

You also can remove directories from the stack beyond the top directory, using `popd` with a numeric argument. The command:

```
$ popd +N
```

removes the directory in position *N* from the stack, counting down from the top. A negative argument (`-N`) counts up from the bottom of the stack instead. Counting begins at zero, so `popd +1` removes the second directory from the top:

```
$ dirs
/var/www/html /etc/apache2 /etc/ssl/certs ~/Work/Projects/Web/src
$ popd +1
/var/www/html /etc/ssl/certs ~/Work/Projects/Web/src
$ popd +2
/var/www/html /etc/ssl/certs
```

Summary

All of the techniques in this chapter are easy to grasp with a bit of practice and will save you lots of time and typing. The techniques I've found particularly life changing are:

- `CDPATH` for rapid navigation
- `pushd` and `popd` for rapid returns
- The occasional `cd -` command

About the Author

Daniel J. Barrett has been teaching and writing about Linux and related technologies for more than 30 years. He is an author of numerous O'Reilly books such as *Linux Pocket Guide*, *Linux Security Cookbook*, *SSH, The Secure Shell: The Definitive Guide*, *Macintosh Terminal Pocket Guide*, and *MediaWiki*. Dan has also been a software engineer, heavy metal singer, system administrator, university lecturer, web designer, and humorist. He works at Google. Visit DanielJBarrett.com to learn more.

Colophon

The animal on the cover of *Efficient Linux at the Command Line* is a saker falcon (*Falco cherrug*).

Swift, powerful, and aggressive, these large raptors have been prized by falconers for thousands of years. Today, they are the celebrated national bird of several countries, including Hungary, Mongolia, and the United Arab Emirates.

Adult sakers often reach 45–57 cm (18–22 in) in size, with broad wings spanning 97–126 cm (38–50 in). Females of the species are significantly larger than males, weighing 970–1,300 g (34–46 oz) as compared to 730–990 g (26–35 oz) for the latter. The plumage of both sexes is highly variable, ranging from deep brown to pale tan, or even white, with brown streaks or bars.

In the wild, sakers hunt primarily birds and rodents, achieving flight speeds of up to 120–150 km/h (75–93 mph) before swooping on their prey. Typical habitats include grasslands, cliffsides, and gallery forests, where the falcons occupy nests abandoned by other birds. With some exceptions in their ranges' southernmost areas, sakers are migratory birds, traveling annually from eastern Europe and central Asia to northern parts of Africa and southern Asia for the winter.

Other than humans, sakers have no known predators in the wild. Yet, due to rapid population decline, the saker falcon is now classified as endangered, as are many of the animals on O'Reilly covers. All of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Lydekker's *The Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.