Beginning Portable Shell Scripting: From Novice to Professional

Copyright © 2008 by Peter Seebach

ISBN-13 (pbk): 978-1-4302-1043-6

ISBN-10 (pbk): 1-4302-1043-5

ISBN-13 (electronic): 978-1-4302-1044-3

ISBN-10 (electronic): 1-4302-1044-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Frank Pohlmann
Technical Reviewer: Gary V. Vaughan
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Kim Benbow
Associate Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Dan Shaw
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

CHAPTER 1

Introduction to Shell Scripting

The UNIX command-line interface has been criticized for complexity and a steep learning curve, but no one disputes that it is one of the most flexible and programmable user interfaces ever developed. The core of the UNIX command-line interface is the *shell*, a program that interprets and executes user commands. The shell can take commands from a keyboard or stored in files; the syntax and commands are the same either way. A file containing shell commands is called a *shell script*. Many systems offer shells that are arguably programmable; the UNIX shell environment is actually good at it. As a result, thousands upon thousands of programs have been implemented as shell scripts. This book treats the shell as a serious programming language and introduces the practice of *portable* shell scripting—the development of scripts that can be expected to run on a variety of host systems or even different shells on the same system. What systems, you ask? Anything that looks reasonably like UNIX, whether it's Solaris, Linux, NetBSD, OS X, or even environments such as Cygwin, which provides UNIX-like behavior under Windows. Don't mistake this for an exhaustive list; I don't have the space to include one; and furthermore, new systems that are released may well comply with the same standards.

Not everyone thinks highly of portability as a goal. Linus Torvalds once said, "Portability is for people who cannot write new programs." As a great fan of portability, I am inclined to nearly agree. I prefer, "Portability is for people who are too busy to write new programs." Users often imagine that portability is a gigantic nightmare requiring a huge amount of additional work; however, in the vast majority of cases, writing portable code takes little extra time, and pays for itself quickly. Portability usually does not mean writing completely different versions of the same program for every system; rather, it means writing a single version that is correct everywhere.

About This Book

This book is about programming in the Bourne shell and its derivatives; the Korn shell, the Bourne-again shell, and the POSIX shell are among the most obvious relatives. This book does not cover the many other UNIX and UNIX-like shells, such as Plan 9's `rc`, or the Berkeley `csh`. It does cover a number of common UNIX commands, as well as a few somewhat less common commands, and briefly looks into some common utilities, such as `sed` and `awk`, which have historically been used heavily in shell scripting. While even further divergent things, such as AppleScript, the Tcl shell, or even graphical shells like the Mac OS X Finder, are technically shells, this book ignores them entirely, and hereafter uses the term *shell* to refer to the UNIX Bourne shell family.

The audience for this book is UNIX users who wish to write shell scripts. The focus on portability offers a better investment in future utility, but the material should be useful even to users who only plan to work on a single system. Portability is not restricted to running on different variants of UNIX; it also applies to running on future releases of the same variant or on different shells. This book is not intended as a complete or comprehensive reference to the shell or to UNIX commands. Instead, it offers information on the shell language, with a focus on areas where shells or utilities vary. Since the shell is not just a single language but a family of related languages, this book also talks about how you decide what shells to use and how to adapt to other shells. Furthermore, this book covers the basics of effective shell programming and effective use of other peoples' shell programs. You may be asked to run a nonportable script on a new system; this book helps you do that. It is never too late to begin thinking about portability.

It is assumed that the reader has some familiarity with UNIX shell conventions and syntax, as well as common UNIX commands. For instance, I do not explain what `echo` or `rm` does. However, a novice user should be able to understand the material with a little extra time spent playing with the examples.

Conventions

This book uses a number of typographical conventions for clarity. As you have probably already noticed, code fragments, such as the names of programs like `grep` in the main text, are represented in monospaced text. Longer code fragments are illustrated as follows:

```
#!/bin/sh
# hello.sh, version 6.23.7: greet the reader
echo "hello, world!"
```

When the results of a code fragment are displayed, they are shown in a separate listing, like this:

```
hello, world!
```

References to UNIX manual pages use the conventional *`program(section)`* usage; for instance, ls(1) refers to the `ls` program in section 1 of the UNIX manual. In cases where user input is included in a code listing, user input is in **`bold`**. Italicized text is used to indicate key definitions of technical terms; a *technical term* is a word you already know being used to refer to a specific technical feature of the shell (or of this book). Italics are also used to identify placeholder names. The `rm` *`file`* command removes *`file`*.

Shell scripts are rich with punctuation, and conventions are adopted for this as well. When first referring to punctuation, I generally use the most common name (and occasionally common alternatives, especially when they are more common for a particular usage) and illustrate the symbol in parentheses. Fonts and displays vary widely, so the symbols you see on your computer may look a bit different. For instance, the vertical bar character (`|`) is a solid bar on some systems, and on other systems may look nearly the same as a colon (`:`). The character with the most names is probably the symbol called variously octothorpe, sharp, hash, or pound (`#`). The name octothorpe is my favorite because people are completely consistent in not having any idea what I'm talking about, but in this book I use sharp, which is originally musical terminology for the symbol. If you have never studied music, this is a great excuse to start now. Further references to a symbol may use a common name or just the symbol inline in the code format for brevity.

What Shell Scripting Is

For purposes of this book, the term *shell script*, or just *script*, is used to refer to any program written in the Bourne shell language (Bourne shell is the traditional UNIX `/bin/sh`) or its derivatives. The UNIX shell is not just the primary user interface of a traditional UNIX system; it is also a full-featured programming language. Shell scripts are simply sets of shell commands, just like those entered on the command line. Indeed, experienced shell programmers often type simple scripts directly on the command line, rather than storing them in a file.

Although any shell can be used interactively or for scripting, there are often differences between a good scripting shell and a good interactive shell. An interactive shell should nearly always have rich command editing and history facilities; these are useless in scripting. A good scripting shell should be fast and ideally small for performance reasons; this weighs against it being the best choice for a command-line shell. Some users do prefer the more advanced feature sets of the bigger shells, even for scripting, but those features make scripts less portable.

The word "scripting" reflects a historical assumption about shell programming; the majority of shell programs are automations of tasks humans can, or used to, perform. The computing world is full of horror stories about some huge task that someone spent hours (or days) performing manually. The horror in these stories comes not just from the large amount of time spent, but rather, from the understanding that the time was *wasted*. However, in modern usage, scripting usually refers more to a language implementation choice; a scripting language is one in which code is parsed at runtime, rather than a compiled language (such as C). In this book, the term *script* is usually used to refer to a shell program.

Scripting is not restricted in scope to major defined projects with written requirements. Shell programs tend to be small, quickly written, and astonishingly powerful for their size. Common, daily tasks are frequently automated by shell users. Even users who do not consider themselves programmers often take advantage of some of the more familiar "cookbook" shell scripting features. Experienced users will write simple programs "on the fly," typing them directly at the prompt without bothering to create a file to hold the code. Larger shell programs are somewhat rarer, but they have their place as well. Most noticeably, the `configure` scripts shipped with thousands of free software packages are actually shell scripts, and very complicated (and surprisingly portable) ones at that.

Used well, the shell can perform in a variety of roles. In general, the shell is used to manipulate things. Three things shell programs often manipulate are

- Data
- Files
- Other programs

Manipulating Data

A huge number of shell scripts revolve around the large variety of line-oriented utilities for manipulating textual data that have been provided with UNIX systems. Not every data stream is easily or naturally represented as a series of lines of text. However, an astounding variety of data can be represented this way, and doing so provides access to a wealth of flexible utilities to solve any number of complicated problems.

The exceptionally rich selection of data-manipulation utilities provided by UNIX-like systems owes a great deal to the huge problem space that line-oriented data lends itself well to. Furthermore, line-oriented data are often exceptionally friendly to human users, who can immediately see whether results are roughly as expected; this has led to using a lot of simple line-oriented data streams in prototyping and development phases.

A number of UNIX utilities build on this by providing translations from other data sources to line-oriented data, which can then be massaged easily through simple shell scripts to produce detailed reports. The existence of a convenient "glue" language to let utilities cooperate encourages the development of small, specialized programs that work together effectively to accomplish virtually any computing task. Such programs may be slower than custom-coded applications for a particular job, but they take a fraction of the development effort to produce.

For the cases where line-oriented textual representations of data are inappropriate, the shell can still provide an excellent glue language. Archive utilities, graphics utilities, and others have been built around the flexibility inherent in the shell. Users coming from a non-UNIX background are often shocked that the most common native archive utilities on UNIX do not perform any compression; their archives contain the included files unmodified. The wisdom of this is revealed when you look at the space savings of replacing the classic `compress` program with the newer `gzip`, and later of replacing `gzip` with `bzip2` or even the newer `lzma`. In each case, the archive programs (most notably `tar`) could be used with a new compression or decompression algorithm without any modifications at all. Likewise, a change from one archiver to another can be handled nearly transparently.

Manipulating Files

Sometimes, what matters isn't the content of files, but manipulating the files themselves or the directories containing them. Thus far, the shell is simply unequaled in this field; indeed, the most technically impressive graphical applications are so far from the power and flexibility of the shell that it is hard to think of them as competition at all. On some desktop systems, a program to tell you how much space each directory on your system takes up, and perhaps help you clean it, might be successfully sold as commercial software. In the UNIX shell, a fairly detailed report—even coupled with automatic archiving of rarely used large files—is a simple matter of typing a few lines of code.

However, file manipulation code is easy to get wrong, especially with files with unusual names, such as those using special characters or spaces. There are many shell idioms that simply cannot be made to function when interacting with file names containing new lines; avoid these like the plague. Other special characters can be less painful, but spaces in file names trigger an astounding variety of bugs, even in long-used scripts written by experienced programmers. Unfortunately, you cannot always control the names files are given by users.

A great deal of shell programming revolves around file manipulation, and this has led to an astonishingly rich variety of file-related tools on most UNIX systems. Some of these tools are a little opaque to users; in many cases, this is because they are designed first as building blocks to use in shell scripts and only secondly as programs to be used directly by users.

Installation utilities can be written as shell scripts, taking advantage of the portability and flexibility of the shell, as well as its ability to embed other data. A once-common example was "shell archive" (or `shar`) files, which consisted of simple shell programs to reproduce a series of files portably on remote systems. These fell into relative disuse due to the security problems implicit in using execution of code generated on another system as an archive format and the widespread availability of substantially more robust archive utilities. On the other hand, the GNU `tar` utility source code is available as a shell script because users who have no archive utilities have to start somewhere. (Similarly, GNU `make` has a shell script build procedure available.)

Manipulating Programs

Scripts are used heavily in nearly every phase of a UNIX system's life. With the exception of Mac OS X's `launchd`, virtually every UNIX system's service startup is handled through a maze of shell scripts. From the System V `init` scripts (reproduced, loosely, in many Linux systems) to NetBSD's very different `rc.d` subsystem, shell programs are excellently suited to the task of identifying configuration state and running programs appropriately. I have met many programmers who are not system administrators, but I've never met a successful UNIX system administrator who couldn't program in shell.

Shell programs have been heavily used in bootstrapping the builds of programs in other languages, such as C. The GNU `autoconf` suite builds huge (and impressively robust) shell scripts that run reliably across hundreds of platforms. The portability of carefully written shell code often makes it easier to develop robust tests and automate common build processes; anyone who has had to manually edit header files to configure for a local UNIX-like system's quirks will be familiar with the disadvantages of the manual process. Shell scripts are often more portable than features of other programs; for instance, using a shell script to build a makefile for use with the `make` utility may be easier than writing a makefile that works with all the common variants.

This kind of usage is often called "glue" code; it is the code that holds things together. UNIX utilities and programs are often designed with the assumption that, if you really need to automate a function, you will write a program to do so. The shell is one of the primary languages used to do this.

What Shell Scripting Isn't

Shell scripts are not a complete replacement for programs in other languages. One issue is that the shell is almost always slower than a compiled language, or even than many interpreted languages, but there are additional limitations. Many of the utility programs scripts are ill-equipped to handle arbitrary binary data, or data where the primary unit of operation isn't a single line of text. The shell itself is virtually useless as a language; the real power and flexibility come from the huge variety of utility programs the shell can use to perform common tasks. Paying attention to portability can restrict your options further, even as it gives you the chance of running on a wider variety of targets.

Scripts are not usually high-performance. Programs that are computationally intensive are unlikely to be written in shell in the first place; a ray-tracer in shell would be more of an installation artwork than a programming project.

Shell scripts are usually portable only between UNIX-like environments. With the introduction of Mac OS X, that has come to include the Macintosh desktop environment, as well as the traditional UNIX-like servers (and desktops). Windows users with the Cygwin environment, or commercial products like the MKS Toolkit, may also take advantage of shell scripting, but shell techniques usually don't translate well to an unmodified Windows system.

With specialized exceptions, shell scripts are essentially free of graphical interfaces. There are a few add-ons or specialized utilities to allow some graphical work, but they are not portable. Furthermore, they don't seem to fit well with the underlying philosophy of the shell; you're usually better off switching to a more general language at that point.

Performance Issues

Shell scripts generally don't perform very quickly. Many of the most fundamental actions in a script involve the spawning of a completely separate program to actually do the work. In fact, the amount of work going on behind the scenes in shell scripts is huge; it is a major influence on the very low tolerance that UNIX-like systems have for high-process startup costs. Shell scripts are at their best when the computational or I/O requirements of their tasks exceed the cost of spawning additional processes. Writers expecting their scripts to run on emulated UNIX-like environments on systems like Windows have to exercise caution.

In many cases, a script with performance issues becomes an invitation to develop a well-considered utility program. Sometimes writing even a simple small C program can dramatically improve the performance of a script, but the development time cost tends to reserve this for the rare case when a script's performance is primarily shell-bound.

Expansion Options

The shell has either the most astoundingly flexible plug-in architecture ever seen in a programming language, or no plug-in architecture at all. I lean toward the latter view. You can implement anything you want as a tool usable in shell scripts, as long as it maps well onto line-oriented textual data. Support for manipulation of binary streams is decent but much more limited; you cannot expect to use `grep` effectively on them, for instance. The language simply doesn't provide for the sorts of plug-ins and extensions that you see in languages like C, Perl, or Ruby. While some shells offer frameworks to allow new features to be plugged in, the language specification doesn't provide for this at all. This can be a disadvantage. Well-designed glue utilities can help some, but there are limits to what you can express cleanly enough to justify the effort.

Why Shell?

After this discussion of the limitations and weaknesses of the shell, you may wonder why shell scripting matters. The answer is simple; while there are things the shell isn't good at, for the things shell is good at, there is very little better. Very few languages are as widely available. There are hundreds of thousands of devices running UNIX-like systems these days, and they may not come with a compiler, but they very often include a shell and some core functional commands. Even if you do not intend to write many shell scripts, the commands in a UNIX makefile are shell commands; and many things that are too complicated for `make` to handle are trivial in the shell. Of course, this makes it possible that your makefile, rather than your C code, will be the limiting factor on where your code is portable.

The shell is a powerful and expressive language. The huge library of existing filters and tools is a great starting point. Furthermore, shell programming makes it easy to build new filters from existing filters. Familiarity with the shell can also make it easier to solve programming tasks that require work in other languages; often, the shell's amazing flexibility as glue code to combine things allows a couple of small, simple programs to give you a very powerful program.

Furthermore, shell programming pays dividends in daily use if you work on a UNIX-like system. Being able to write small scripts to handle common tasks (or even one-off tasks that would take hours to do by hand) justifies a little time spent learning a new language.

In the end, the UNIX shell is one of the most durable programming languages in use today. While there have been extensions and developments in shell programming over the past 30 years, many shell programs written in the 1970s are still usable today. While competing languages have found some traction, they primarily target tasks the shell isn't good at, leaving the shell's primary domain largely unchallenged.

The Bourne Shell Family

This book focuses on shells derived from the classic Bourne shell distributed with early UNIX, and, in particular, on the standard POSIX shell. There are a number of shells in this family, and this book covers some of the common (or unique, but interesting anyway) extensions they offer over the base shell language. The shells covered are the following:

- *Bourne shell (old* `sh`*)*: The shell that started it all. The Bourne shell was the standard system shell in early AT&T UNIX, and most modern shells are moderately compatible with it. While this is the baseline, early Bourne shells lacked some features now universally provided. For most users, this shell is of marginal relevance, as nearly all major systems now provide a POSIX shell; only a few do not provide a POSIX shell as `/bin/sh`. More information on getting into a more modern shell is provided in Chapter 7.
- *POSIX shell (*`sh`*)*: This is the baseline shell. Users familiar with older UNIX systems will note that many of the features described for the POSIX shell are innovations (many of them inherited or acquired from variant shells). While an occasional reference is made to some of the limitations of earlier shells, the modern landscape is consistent in providing reasonably stable and full-featured POSIX shells. Furthermore, even on systems lacking such a shell, it is usually practical to acquire one of the other variants. In most cases, *portable* in this book means "running on the POSIX shell without modification."
- *Almquist shell (*`ash`*)*: The Almquist shell was developed as a reasonably compatible independent re-implementation of the POSIX shell included with SVR4 UNIX. It was distributed originally primarily with BSD variants. This shell is much smaller than some of the other variants, but maintains (in its original form) POSIX compatibility. The Almquist shell is familiar to many users as the Busybox shell.
- *Bourne-again shell (*`bash`*)*: The GNU Bourne-again shell is the largest, and arguably most complete, shell. It has a history of aggressive feature adoption, and can run nearly anything, although early versions had some compatibility quirks. Many Linux systems ship with `bash` as the default `/bin/sh`, as does current Mac OS X. Scripts developed for `bash`, and using its extensions, may not run on other shells. Many users dislike the performance costs of `bash`. A number of scripts for Linux systems (such as the `quilt` patch manager) assume that the shell is `bash`, which has caused portability problems. Don't do that.

- *Debian Almquist shell (*`dash`*)*: This is a derivative of the Almquist shell used in Debian and derived systems, such as Ubuntu. It has been installed as the default shell on some Debian variants for a while now, exposing a number of scripts that erroneously depended on `bash`-only extensions. This shell exists as a small, fast implementation of the basic portable shell.
- *Korn shell (*`ksh`*)*: Developed by David G. Korn at AT&T, `ksh` was one of the first Bourne shell derivatives to add many of the features now adopted elsewhere as standard. There are multiple versions: historic `ksh`, the 1988 revision (`ksh88`), and the 1993 revision (`ksh93`). The current versions are available as source from AT&T.
- *Public-domain Korn shell (*`pdksh`*)*: Before the Korn shell became free software, a public domain clone of it was written. While there are a few noticeable compatibility differences, for the most part, `pdksh` and `ksh88` are compatible implementations. A number of systems have used `pdksh` as a shortcut to getting a reasonably full-featured POSIX shell. More modern systems often replace `pdksh` with `ksh93`.
- *Z shell (*`zsh`*)*: The Z shell is probably by far the most divergent of those listed here from the historical Bourne shell. However, `zsh` can be configured to perform as a fairly solid POSIX shell, and on some systems it may be the only shell available that can be made to execute POSIX shell code at all. (For more information on encouraging `zsh` to behave like a POSIX shell, see Chapter 7.)

Nearly every example in this book (except those used to illustrate differences between these shells) will run identically on all of these except the pre-POSIX Bourne shell. This diversity of options is certainly one of the reasons to favor shell-derived languages for programming.

Why Portable?

Portable code is more useful. If your scripts are portable, they will survive changes in your platform. This offers two key benefits. First, you can freely switch platforms whenever you want. Second, you can use a broader range of platforms.

There is no perfect system. Every system you might use has flaws. You will want to change systems from time to time. You may find that your best choices for different systems are different operating systems, running on different hardware. Portability lets you share code between things. People, and companies, have been known to get trapped on a platform because they wrote unportable code that makes it too expensive to migrate. In the long run, writing portable code saves you work.

Furthermore, the cost of portability is often greatly overestimated. People look at the pages and pages of output of a typical `configure` script and assume that there are dozens or hundreds of things they need to check for and write alternative code for. In general, this is not the case. Writing unportable code for two systems, as well as code to distinguish between them, is not generally the first strategy to take when pursuing portability. Programs taking that approach, whether in shell or any other language, in general become quickly unmaintainable.

The best way to write portable code is to understand the language and tools you are working with. A lot of unportable code results from people who don't understand a chunk of code, copying it and modifying it until it seems to work. Don't do that! It is fine to copy a chunk of code you do not understand; however, study it and experiment until you understand it before you use it. Often, much of the code in a large and poorly maintained project is irrelevant and unneeded, and it is present only because someone saw it done and didn't understand why.

As developers throughout the open source world have learned, and documented, in many cases there is no real difference between writing code well and writing it portably. Well-organized code that isolates its assumptions (see the discussion on this in Chapters 7 and 8) tends to be easy to maintain on any platform, as well as easy to port. Furthermore, as soon as you give some attention to the assumptions you have made about your environment, you are likely to realize that you do not need to make those assumptions. Fixing the assumptions creates code that is not only more portable, but also simpler and thus easier to maintain.

Many programming books encourage you to try experiments to see what happens. This often leads to horribly unportable code. Portable code behaves predictably on multiple systems. Unportable code doesn't necessarily behave badly; it may, in fact, do exactly what you were hoping for on the machine you were on when you tried it. Knowing that something "works" on a single machine is no substitute for knowing how it works and being able to predict what it will do on another machine.

Does this mean you shouldn't perform experiments? Of course not! It means you should perform your experiments several times on a variety of computers and shells. Don't be too quick to trust vague memories, either of what is or what is not portable. (For a concrete example, see the sidebar "Checking Your Assumptions" later in this chapter.)

Why Not?

This book focuses on the Bourne shell, with a little discussion of the major derivatives. I don't talk about the C shell (`csh`) much, even though this is a book on shell scripting. There are several reasons for this, but they all come down to the C shell being a pretty decent interactive shell, though not nearly as good as a programming shell. Tom Christiansen's seminal article "Csh Programming Considered Harmful" has stood the test of time, and this representative quote is as true today as it was when it was first written:

> *While some vendors have fixed some of the csh's bugs (the tcsh also does much better here), many have added new ones. Most of its problems can never be solved because they're not actually bugs per se, but rather the direct consequences of braindead design decisions. It's inherently flawed.*
>
> —Tom Christiansen, `www.faqs.org/faqs/unix-faq/shell/csh-whynot/`

The C shell is not a compatible relative of the Bourne shell; past the simplest one-liner scripts, the two are simply incompatible. The C shell and its variants (most noticeably `tcsh`) are popular for interactive use but have little to offer for portable shell programming. Their most significant features are related to elaborate (and extremely powerful) command history. The job control features that motivated many users to switch to the C shell are now widely available in Bourne shell derivatives. Perhaps more importantly for the purposes of this book, these features have essentially no impact on noninteractive use in scripts.

Other UNIX-like shells, such as Plan 9's excellent `rc`, are omitted, not because they are ill-suited to development, but because they are not as consistently installed on as broad a base of systems as the Bourne shell family and offer very different language features. Any UNIX-like machine you use will have a shell that is similar enough to the Bourne shell for programming purposes; many will not have the other shells installed. While it is often easy to install these shells, you may not always have the permissions needed to install them globally, and needing to do so is one more thing that can go wrong. More generally, non-shell scripting languages, such as Tcl or Perl, are too far afield to mix well, even though some of them provide shell-like interfaces; Tcl's `wish` and Ruby's `irb`, while interesting and useful, are better approached in the context of those languages, rather than as though they were "shell variants."

The only languages other than shell discussed at any length are `sed` and `awk`, which are often used for stream editing and processing within shell scripts. Note, though, that the shell is a very friendly language and loves to cooperate. You can generally use other scripting languages easily from shell.

Beyond Portability: Cleanliness and Good Living

This book talks a lot about portability concerns. It also talks about how to write clean code. Cleanliness in code is a somewhat fuzzy concept, and programmers argue over it for hours. You might wonder why this book discusses style, defensive programming, and common conventions, which have no functional impact on the shell itself. There are good reasons.

If you are interested in portable code, it is because you want your code to work on multiple platforms. Code that is broken identically on ten platforms is of no use to you. Established shell conventions make it easier for other people to read your code, make it easier for you to read other peoples' code, and usually have sound engineering principles behind them. Similarly, learning to program defensively is essential to making good use of the shell. Because the shell runs with no sandbox, and with all of the caller's privileges and access to the file system, it is extremely dangerous to run badly written shell code.

Writing clear, clean code makes it easier for you to see what is happening and why. Clean code is more likely to be portable, easier to port when needed, and more likely to be useful enough to be worth the bother of porting. In short, to borrow the words of C.A.R. Hoare:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> —C.A.R. Hoare, 1980 Turing Award Lecture

Writing clean code saves you time. It saves you time porting, it saves you time debugging, and it generally saves you time even during initial development. If you are sharing source with other users or developers, clean code will get more contributions and more useful feedback. If you are prototyping and revising your design as you go, clean code will be easier to maintain and update. Of the code I wrote ten and 15 years ago, the clean code has stayed with me and adapted quickly to new systems; the badly written code has not survived and has often been too much work to update.

Do not write carelessly or poorly, even for small one-off scripts. Sloppy work is habit-forming.

What's in This Book

The next section offers a very quick overview of the shell, without going into great detail on the formal syntax or semantics of shell scripting. If you've used the shell before, you may be able to skip it and get into the more detailed material in the following chapters. The next chapter gives a detailed look at the various ways in which the shell performs pattern-matching. Following this are four chapters of detailed discussion of shell features, explaining their specifications more precisely, and showing how to make effective use of them. After this are chapters on portability of shell language constructs and utilities commonly used in shell programming, and then on shell script design and interactions with other languages. If you encounter unfamiliar terminology, look in related sections; I have tried to define terms when I first use them.

Introducing the Shell

This section gives a quick tour of shell usage, starting with basic usage and display conventions, and then moving on to the basics of quoting and variables. There are some generalizations to which you will later learn exceptions, but it gives a quick basic grounding in what the shell does. This overview should make it easier to see where each of the following chapters fits in. Throughout this, you may find yourself asking questions that start out "but what happens if . . . ?" which are not answered in this chapter. As mentioned in the previous discussion on portability, go ahead and try them, but be aware that the results may sometimes vary between shells.

Whether being used interactively or from a script, the shell's basic operation is the same. It reads lines of input, which it breaks into words (usually around spaces), performs substitutions and expansions, and finally executes commands. Much of the shell's power comes from the fact that the shell has rules for modifying the words it is given to generate commands. This section gives a brief overview of these rules, and the ways to keep the shell from performing these modifications inappropriately.

Most of the material in this section is covered in more detail (indeed, with a particular attention to fiddly little details) later in the book.

Interactive and Noninteractive Usage

In interactive usage, the shell indicates readiness for input by displaying a string called a prompt Generally, the default prompt is a dollar sign ($). If the shell is expecting a continuation of previous input, the prompt changes to a greater-than sign (>). In interactive usage, the shell usually shows the output of each command before printing the next prompt. For instance, the following interactive session shows both of these prompts:

```
$ echo 'hello
> there'
hello
there
```

In the preceding example, the shell gives the first prompt ($) and waits for input. The user enters the text **echo 'hello**. The apostrophe, or single quote, begins a quoted string; in a quoted string, the shell does not break words around spaces or new lines, and the string is not complete until the other quote is seen (this is explained a bit more in the section "Introducing Quoting" later in this chapter). The shell cannot execute this command yet because the string isn't complete. The shell knows there must be more input; it displays the secondary prompt (`>`), and waits for more input. The user enters the text **`there'`**. The second apostrophe ends the string. Unlike some languages, the shell uses pairs of identical apostrophes for strings, rather than using left and right quotes. With the string complete, the shell now returns to looking for the ends of words or commands. It sees a new line (from the user hitting return), and this ends the command.

The shell runs the command, passing the quoted string to the `echo` command, which displays its arguments. Note that the new line within the quoted string becomes part of the argument to `echo`, and the result has a new line in the same place. The same continuation prompt is used when a shell syntax structure (such as `if-then`) is incomplete.

When the shell is running a command provided to it from a noninteractive source, no prompts are displayed. To run these examples noninteractively, save them in a file, then run your shell of choice on the file; for instance, if you have saved an example as `hello`, you can invoke it with the command `sh hello`. Another option is to create an executable shell script. To do this, add a line to the beginning of the file indicating that it is a shell script:

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

This line is often called a *shebang* (short for *sharp-bang*, the nicknames of the first two characters on the line.) A file starting with this, and marked as executable, is treated as a script for the named program. Some users prefer to put a space after the exclamation mark (`!`), but it is not needed (the notion that it might be on some systems is a very persistent portability myth). To mark a script executable, change its mode:

```
$ chmod +x hello
```

Once you have done this, you no longer need to specify the shell interpreter, although you will usually need to specify the path to a program in the current directory to use it:

```
$ ./hello
```

Prompts vary from shell to shell, and many systems change the default shell prompt. While not all shells support this, some can interpolate things (such as the current directory) into the shell prompt. Some shells, on hardware that supports it, will even colorize the prompt.

The biggest difference between interactive and script usage is in the interleaving of output and input. When you work interactively, each command's output is displayed before the shell offers you a new prompt. When you run a script, commands are run in sequence without pauses. While this book typically uses chunks of shell code without prompts to illustrate points, the same code entered at a prompt would generally have the same effect, despite the formatting differences.

Simple Commands

A *simple command* is just a command (such as `echo` or `ls`) and its arguments. In the absence of special characters (called *metacharacters*) or words that have special meaning to the shell (called *keywords*), a series of words followed by a new line are a simple command. Control flow constructs (such as `if` statements, discussed in Chapter 3) are not simple commands.

The shell breaks each line of input into sequences of characters called *words*, usually around spaces and tabs, although there are other ways to separate words. The process of splitting text into words is called *word splitting*. It does not matter how many spaces (or tabs) you place between the words. A line beginning with a sharp (#) is a comment and is ignored by the shell. The first word on a line (which may be the only word) is the command (usually an external program) to run; the following words are passed to it as parameters, called *arguments*. The `echo` command displays its arguments, separated by spaces if there's more than one argument, and followed by a new line. (This is usually the case; however, the `echo` command is full of nonportable special cases discussed in Chapter 8.) Each of the following commands produces the same output:

```
$ echo hello, world
hello, world
$ echo hello,     world
hello, world
$ echo       hello,    world
hello, world
```

As you can see, the shell modifies input text before executing it. In this case, for instance, the spaces between words are not counted or recorded; rather, the shell just uses them to detect word boundaries. A quick way to get some insight into how your commands are being modified is to put the shell into trace mode, by issuing the command `set -x`. This will cause the shell to show you each simple command before executing it. To turn this off, issue the command `set +x`. Each simple command, as finally executed, is echoed back with a plus sign (+) in front of it before the shell executes that command. For more information on trace mode, including the circumstances where it displays something other than the plus sign, see Chapter 6. Be aware that the exact format of the message may vary from one shell to another. Regardless, tracing is usually quite helpful in understanding the shell. For instance, the following transcript shows that the commands executed by the shell have had strings of spaces replaced by single spaces:

```
$ set -x
$ echo hello, world
+ echo hello, world
hello, world
$ echo hello,     world
+ echo hello, world
hello, world
$ echo       hello,    world
+ echo hello, world
hello, world
$ set +x
+ set +x
```

If you are experimenting with the trace feature on the command line, do not forget to turn it off with a final `set +x`. However, in the pursuit of brevity, the `set +x` is omitted in future examples.

You can enter several simple commands on a line by separating them with semicolons (;). Semicolons are a good example of a *metacharacter*, a character that has special meaning to the shell even when it is not separated from other text by spaces:

```
$ echo hello;echo world
hello
world
```

The semicolon breaks this into two commands. The second command is executed immediately after the first, and the user is not prompted between them.

Introducing Variables

Variables are named storage that can hold values. The shell expands a variable when the variable's name occurs after a dollar sign ($), replacing the dollar sign and variable name with the contents of the variable. This is called variable *expansion*, or *substitution*, or occasionally *replacement* or even *interpolation*. Of these terms, expansion is used most often in documentation, but substitution is probably the clearest. Variables are assigned using an equals sign (=) with no spaces around it:

```
$ name=John
$ echo hello, $name
hello, John
```

Unlike most other programming languages, the shell uses different syntax when referring to a variable than when assigning a value to it. Variable names start with a letter or underscore, followed by zero or more letters, numbers, and underscores. Some shell programmers use all capitalized names for variables by convention, but in this book, I use all capitalized names only for environment variables or special predefined shell variables (such as $IFS, which is explained in Chapter 4). Do not use mixed case; it works, but it is not idiomatic for the shell.

If a variable has not been set, it expands to an empty string; there is no warning (usually) for trying to use an unset variable, which can make it hard to detect simple typos. Variables in shell are always strings; the shell does not distinguish between strings, integers, or other data types. The shell is generally considered a typeless language.

If you want to obtain a value from the user, you can do so using the read command to read a line of input and store it in a variable, as in the following example:

```
$ echo Please enter your name. ; read name
Please enter your name.
Dave
$ echo Hello, $name.
Hello, Dave.
```

If you try to assign a value including spaces to a variable, you will discover that the shell splits the line into words before trying to assign variables. Thus, this doesn't work:

```
$ name=John Smith
sh: Smith: command not found
$ echo hello, $name
hello,
```

A brief explanation of what went wrong follows in the next section; a full explanation of what went wrong is found in Chapter 3. For now, the key lesson is that the assignment doesn't work, and you need a way to prevent the shell from splitting words.

Introducing Quoting

The separation of input into words is generally very useful, but it is occasionally desirable to prevent it. For instance, if someone created a file named `hey you`, trying to remove it might prove frustrating:

```
$ rm hey you
rm: hey: No such file or directory
rm: you: No such file or directory
```

To overcome this, you must tell the shell that, rather than being a special character that separates words, the space is just a literal character with no special meaning. This is called *quoting*, and the most common way to do it is by enclosing material in quotes. Quotes can be single quotes or double quotes; in both cases, the shell does not use distinct left and right quotes, but uses the same quotes on both sides. [On a slightly related note, text surrounded by back quotes (`` ` ``) is not being quoted; that is one of the syntaxes used for embedding the output of shell commands, much as variables are substituted. Command substitution is explained in Chapter 5.] Most commonly, you simply enclose a string in single quotes (`'`) to prevent the shell from modifying it. Here's a review of the hello world example, using quoting:

```
$ set -x
$ echo 'hello, world'
+ echo hello, world
hello, world
$ echo 'hello,     world'
+ echo hello,     world
hello,     world
$ echo '      hello,    world'
+ echo       hello,    world
      hello,    world
```

Single quotes prevent the shell from modifying input, including word splitting. The quote marks themselves are removed. While this is useful for arguments, it is important not to quote things that you do want the shell to split. For instance, the following script doesn't do what the user probably wanted:

```
$ set -x
$ 'echo hello, world'
+ echo hello, world
sh: echo hello, world: command not found
```

With all the spaces quoted, the shell has no way of knowing the user meant to invoke the `echo` command; instead, it obligingly looks for a command named `echo hello, world`. Since there isn't one, the shell prints an error message.

■Note Error messages may vary between shells. Do not worry if you try an example and get an error message in a slightly different format. Also, the display of an error message may depend on whether the shell was executing a script. For a syntax error or other shell error, the shell usually gives the name of the script and the line number it was executing. This does not mean you should not run examples and try them out for yourself; it does mean that it is not always a good idea to depend on the exact contents of an error message.

You can now pursue the previous example of trying to assign a full name, including a space, to a variable:

```
$ name='John Smith'
$ echo hello, $name
hello, John Smith
```

The quotes allow you to assign a value containing spaces to a variable. When the variable is substituted, the space is included. Try to figure out the next example before you run it:

```
$ command='echo hello, world'
$ $command
```

There are two ways you might reasonably expect this to play out. One is that the shell will respond with `hello, world`. The other is that it will respond with an error message, such as `sh: echo hello, world: command not found`. Since word splitting happens before variable expansion, you might reasonably expect the error message, but in fact the shell greets you. The reason for this is that the outputs of variable substitution are usually subject to word splitting again. (The results are not then subject to variable substitution.) In this case, that's very useful. But consider what happens if you are trying to preserve spaces, not just include them:

```
$ name='Smith,     John'
$ echo $name
Smith, John
```

No problem; you know how to protect spaces, right?

```
$ name='Smith,     John'
$ echo '$name'
$name
```

You need a way to ask the shell to expand variables but not perform word splitting. Conveniently, the shell has multiple quoting mechanisms. If you want some special characters, but you still want to quote spaces, you can use double quotes. The shell substitutes variables in double-quoted strings:

```
$ set -x
$ name='Smith,     John'
+ name=Smith,     John
$ echo $name
+ echo Smith, John
Smith, John
```

```
$ echo '$name'
+ echo $name
$name
$ echo "$name"
+ echo Smith,     John
hello, Smith,     John
```

The text `$name` is substituted both when unquoted and when in double quotes; it is not substituted when in single quotes. When unquoted, the substituted value is subject to field splitting (much like word splitting, but see the in-depth discussion in Chapter 4); inside double quotes, it is not. If you have used other scripting languages, you may have seen this distinction between single and double quotes before; it is very useful to be able to distinguish between a purely literal string and one in which you want variables to be substituted, and the shell syntax is familiar to many users. Single quotes are the easiest to understand; a single-quoted string lasts until the next single quote. No other characters have any special meaning within single quotes. Of course, this makes it hard to get a literal string including a single quote; to do that, use double quotes. This allows an expansion on an earlier example:

```
echo 'What is your name? '
read name
echo "I'm sorry, $name, but I can't let you do that."
```

```
What is your name?
Dave
I'm sorry, Dave, but I can't let you do that.
```

Note the use of double quotes to protect the spaces and other punctuation, including single quotes. Try to guess what the following code will produce before running it:

```
echo "What is your name?"
read name
echo I'm sorry, $name, I can't let you do that.
```

In this version, because the single quotes were not themselves quoted, they defined a quoted string. This has two effects: The first is to prevent `$name` from being expanded, and the second is the removal of the apostrophes, which the shell interprets as single quotes. For this reason, shell programmers often use double quotes around strings passed to `echo` even when no obvious need to is in evidence. It is easy to forget that a harmless apostrophe is actually a sinister and dire single quote, plotting ambush and mayhem from its lair between “n” and “t.” Having the quotes there is a good example of defensive programming. The use of double quotes to get literal single quotes (and single quotes to get literal double quotes) is easy enough once you are used to it, but it can be a bit surprising at first.

As you have probably noticed, the quote characters themselves are not included in the strings they quote. Quoted strings are considered to be part of the same word as anything they are adjacent to. It is a common misconception that the quoted material is itself a “word” to the shell, and that a pair of adjacent quotes are treated as separate words. This is not so, and this leads to the way to get single quotes into a string that is single-quoted when you want an apostrophe but do not want any expansion:

```
$ echo '$name is a variable, now, isn'"'"'t it?'
$name is a variable, now, isn't it?
```

The two single-quoted strings are adjacent to a double-quoted string containing only an apostrophe; when the quotes are removed, these become a single shell word. The `echo` command receives only one argument. As this example illustrates, there are a great number of subtleties to the interactions of these features, but this quick tour should prepare you to follow along with code even if you haven't used any of these features before.

The printf Command

The `printf` command was introduced some years back, but many users are unaware of it. It mostly emulates the behavior of a C function by the same name, used to format output. The first argument to `printf` is called a *format string*, and describes an output format. Certain special characters in the format string need to be filled in with data; these data are taken from the other arguments, in order. This is easier to show than to describe:

```
$ name="John"
$ printf "Hello, %s!\n" "$NAME"
Hello, John!
```

CHECKING YOUR ASSUMPTIONS

When I originally drafted this text, I used `echo` in all of the examples because it was the only portable command for displaying text. It is problematic in many ways (it gets its own section in Chapter 8), but there's nothing else. The wonderful and expressive `printf` utility is unfortunately not portable. After all, it's only found on BSD systems and Linux systems and built-in to `bash` and `ksh93`. Actually, it looks like Solaris has it. In fact, I searched among something around 30 systems, and the only system I could find that anyone had still running in which `printf` did not work in `/bin/sh` was a SunOS box (not Solaris) a friend of mine had still running, even though it was officially unsupported due to unfixed Y2K bugs.

I did some informal polling. Every experienced shell programmer I talked to "knew" that the shell command `printf` was a new feature (or had never even heard of it). No one thought it was portable, but it turns out to be substantially more portable than many features I have been taking for granted for ten years. While it is true that it is a new change, it is a change specified by the current UNIX standards, and one that appears to have become essentially universal. So, even if you are pretty sure you know that something isn't portable (or that it is), check your assumptions!

The special sequence `%s` indicates that a string should be displayed; the next argument is interpreted as a string and replaces the `%s`. The character determining what kind of object to print (such as a string or a number) is called a *format character*, and the whole character sequence is called a *format specification*. Other common formats are `%` (print a percent sign), `d` (print a number), `o` (print a number in octal), `x` (print a number in hexadecimal), and `f` (print a floating-point value). The other thing `printf` does is interpret backslashes followed by special characters; these combinations are called *escape sequences*. The most important one to know about is `\n`, which represents a new line. Unlike `echo`, `printf` does not automatically finish its output with a new line character:

```
$ echo "Hello!" ; echo "Goodbye!"
Hello!
Goodbye!
$ printf "Hello!" ; printf "Goodbye!"
Hello!Goodbye!$
```

In fact, even the shell's prompt can end up on the same line as the output from a `printf` command. This can be a bit of a surprise to new users, and even experienced users will get it wrong occasionally. However, it is also extremely useful in some cases. If you wish to display a prompt and then request input from the user, being able to omit the new line is quite handy. (There is no portable way to do this with `echo`, although there are several nonportable ways that may work on individual systems.)

The other thing `printf` is good at is formatting—not just displaying output, but displaying it according to particular rules. There are three key concepts in displaying fields. The first is *width*, or how many characters to display at a minimum. Width, given as a number between the `%` and the format character, is used to format output so it lines up nicely:

```
$ printf "%3d: lined\n%3d: up\n" 1 100
  1: lined
100: up
```

The width of 3 causes the `printf` command to display at least three characters, even if it does not need that many. But what if there are more? There is also a way to limit the number of characters printed; this is called *precision*, and is written as a number following a period, once again between the `%` and the format character. Precision limits the total number of characters printed for strings; for floating-point numbers, it limits the number of characters printed after the decimal point.

```
$ printf "%.5s\n%.5s\n" John Samantha
John
Saman
```

You can specify both width and precision; if you do this, the precision is used to determine what to print, and the width then influences whether the shell pads the output. Padding can be controlled a little. The two most common ways to control padding are to specify flags, which are put before the width. (If there is no width, there is no padding, and there is no point to specifying flags.) The two common flags are left justification (`-`), and zero padding (`0`). Zero padding applies only to numeric values. Idiomatically, the pattern `%02x` is used to express byte values in hexadecimal:

```
$ printf "%02x\n" 197 198
c5
c6
```

This example also illustrates another feature: If you provide additional arguments, `printf` recycles its format string, starting over at the beginning. Format specifications without arguments are treated as though the argument was a `0` (for numeric formats) or an empty string (for string formats).

There is only one significant flaw in the `printf` command, which is that you cannot easily use it to display arbitrary characters. The C language `printf` function has a `%c` format specifier, which prints a numeric value as a raw character; for instance, on an ASCII-based machine, the C code `printf("%c", 64);` prints an at sign (`@`). In the shell command, `%c` is equivalent to `%.1s`; it prints the first character of its argument, which is treated as a string. So, for instance, `printf '%c\n' 64` prints `6`. However, you can print characters using an escape sequence; a backslash followed by three octal digits is printed as the character in question, so `printf '\100\n'` prints `@`. In later chapters, you'll see how you could use `printf` to create a format string containing arbitrary characters as octal escape sequences.

Some implementations of `printf` take additional options (starting with hyphens) before the format string. In portable code, never start a format string with a hyphen. If you want to display a hyphen, use `%s`:

```
$ printf '%sv' -
-v
```

Be very careful with parameter substitution in `printf` format strings. The translation of format specifiers into arguments happens after parameter substitution; if you substitute a parameter containing `%` characters into a format string, those `%` characters may become format specifiers. If you wish to include the value of a parameter in your displayed output, always use a suitable format (usually `%s`) and provide the parameter as a double-quoted argument:

```
$ password="xfzy%dNo"
$ printf "Your password is $password.\n"
Your password is xfzy0No.
$ printf "Your password is %s.\n" "$password"
Your password is xfzy%dNo.
```

In general, I recommend using single-quoted strings as `printf` format strings; this ensures that the calling shell will not do something unexpected with any backslash escape sequences you used (behavior of backslashes inside double-quoted strings is not always 100% portable), and that no parameter substitution occurs; this allows you to be sure you know what your format string is.

Some implementations of `printf` have difficulties with particularly long formats; for example, the Solaris `printf` aborts when given the format string `%05000d`. Exercise caution with large formats.

What's Next?

This whirlwind introduction covers enough so that, even if you've never tried to use the shell before, you can follow along with the examples used to illustrate various points of shell architecture and design. The next chapter talks about patterns and regular expressions; if you're familiar with those already, you can skip ahead, but you might like the quick refresher. Now, on to the fiddly little details!