Using the Bash Command Line

Michael Marsh

Using the Bash Command Line

Michael Marsh

How to Read This Book

There's a good chance you don't really know how to read a reference book like this. That's OK! It's a skill you need to learn, and reading this foreword is a great place to start.

First, while you *can* read this book from start to finish, and that's not an unreasonable thing to do, you don't necessarily *need* to. You should definitely start by reading the Introduction, though. This will establish the notation, including formatting, that we use, so the rest of the book makes sense to you. It will also introduce you to *manual pages*, which are typically your best source of information on various commands, functions, and file formats.

Once you understand the notation, feel free to look through the Table of Contents for topics of relevance to you. At the very least, read through this to see what *is* and *is not* covered in this book. Refer to this frequently as issues come up.

Contents

1 Introduction **1**

1.1 Notation 2

1.2 Manual Pages 3

2 File and Directory Structure **9**

2.1 Filesystems 9

2.2 Moving Through the Filesystem 10

2.3 Disk Usage 12

3 Files **13**

3.1 Viewing File Contents 14

3.2 Moving, Copying, and Extracting Pieces of Files 16

3.3 File Permissions 18

3.4 Wildcards 20

3.5 Standard File Descriptors 20

4 Shortcuts 23

4.1 Editing a Line 23

4.2 Auto-Filling 24

4.3 Aliasing 26

5 Programs 27

5.1 Running Programs 27

5.2 Processes 30

5.2.1 Process Information 30

5.2.2 Background Processes 31

5.2.3 Terminating Processes 32

5.2.4 Setting Priorities 33

5.3 Input and Output Redirection 34

5.4 Pipelines 36

6 Shell Configuration 37

7 How to Find Things 39

7.1 locate 39

7.2 grep and ack 40

7.3 find 41

8 Variables **43**

8.1 Basic Variable Usage 43

8.2 Parameter Expansion 45

8.3 Quoting 46

9 Scripting **49**

9.1 Running Scripts 49

9.2 Working with Positional Parameters 51

9.3 Mathematical Expressions 52

9.4 Control Flow 53

9.5 Functions 55

9.6 Utility Programs 56

9.6.1 true and false 56

9.6.2 yes 57

9.6.3 seq 57

9.6.4 cut 58

9.6.5 awk 58

Chapter 1

Introduction

This is intended as a short reference for working with the bash shell from the command line. Many Computer Science courses rely on command-line usage, but students often have a substantial gap between when they were first introduced to the topic, and when it next comes up in their education.

In addition to acting as a refresher (or even initial reference), we also present some more advanced topics. These include things like parameter expansion, control flow, and writing complex scripts for bash. It's well-worth familiarizing yourself with these concepts, because automating tasks with bash scripts can save you a lot of time. A set of principles worth keeping in mind:

1. Anything you have to do more than once is worth scripting.
2. Anything non-trivial that you have to do, you will probably have to do again.

While this is focused on the bash, shell, most of what is presented here should be applicable to other modern shells that share the same lineage as bash. For example, everything here should be applicable to zsh as well, though zsh has some additional functionality we do not cover, and some situations might behave slightly differently. It's important to always test your scripts!

1.1 Notation

We will make heavy use of font type throughout this book, so it's important to introduce this as our principle method of notation. Much of this follows the convention used in official documentation. Things that you will type into the shell will be given in **`typewriter font`**. Arguments to commands will be written in multiple ways, depending on the context. Let's explain by example.

Consider the **`ls`** command. We might specify the calling "signature" of this as (though there are many other options we can provide)

```
ls <dir>
```

In this case, we use the notation **`<dir>`** to indicate an argument that you would provide. For example,

```
ls /bin
```

In general, an argument name enclosed in angle brackets is something that you will replace when calling the command. We could, in text, refer to this argument in *italics*, without the angle brackets. This means **`<dir>`** and *dir* would be equivalent. Very rarely, we might write something like **`ls`** *dir*. You will also see underlines instead of italics or angle brackets, so that **`<dir>`**, *dir*, and dir all mean the same thing, but we will not use underlines.

Commands often take optional arguments. These are enclosed in square braces. Since **`ls`** does not require a directory argument, we might change the above signature to

```
ls [<dir>]
```

to indicate that *dir* is optional.

Sometimes an argument (typically the last one) can be replicated multiple times. To indicate this, we use an ellipsis after the argument. For example, **`ls`** can take multiple files or directories to list, so we could further amend our signature to

```
ls [<dir> ...]
```

Note that this is *not* the full command signature for **`ls`**. Here is a more typical signature:

```
ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1%] [<file> ...]
```

Sometimes, we will show a command that you would type along with its output. When running as a normal user, we will indicate this with a **`$`** indicating a shell prompt; for a privileged (root) user, we will use **`#`** as the prompt. For example,

```
$ ls /
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

1.2 Manual Pages

Manual pages ("manpages", for short) are a particular type of documentation common on Posix-compliant systems, like Linux or MacOS. Much of what's in this book can, in fact, be found in the various manpages for the commands we're presenting. They have a particular format (actually, several formats), so it's worth spending some time learning how to read these. The last few chapters will build towards understanding and writing your own *shell scripts*, which are a powerful automation tool, at which point being able to read manpages will be pretty much essential. They're extremely useful for interactive shell use, as well.

The program to run is called **`man`**, and it has it's own manpage! You can view a manpage with

```
$ man ls
```

This will show you the manpage for the **`ls`** command. The manpage for **`man`** is

```
$ man man
```

Manpages are separated into *sections*. Each section holds a different sort of documentation. Section 1 contains almost all of the commands you'll run from the command line. Sections 2 and 3 contain function documentation; section 2 for system functions (like **`open`**) and section 3 for "normal" functions (like **`printf`**). Sections 4 and 5 contain file documentation, while 6 and 7 contain miscellaneous documentation. Finally, section 8 contains documentation for system administration tools and daemons (background services).

If you run **`man printf`**, you will probably get the manpage for the *shell* command, rather than the function. This is because **`man`** searches for pages starting in section 1, and progressing through the sections. We can work around this by *specifying* the section in which to look, by running

```
$ man 3 printf
```

You can also search for manpages using either **`man -k`** or **`apropos`**. For example, to find manpages about timeouts, you could run

```
$ man -k timeout
aio_suspend (3)      - wait for asynchronous I/O operation or timeout
timeout (1)          - run a command with a time limit
XtAddInput (3)       - register input, timeout, and workprocs
XtAddTimeOut (3)     - register input, timeout, and workprocs
XtAddWorkProc (3)    - register input, timeout, and workprocs
XtAppAddTimeOut (3)  - register and remove timeouts
XtAppGetSelectionTimeout (3) - set and obtain selection timeout values
XtAppSetSelectionTimeout (3) - set and obtain selection timeout values
XtGetSelectionTimeout (3) - set and obtain selection timeout values
XtRemoveTimeOut (3)  - register and remove timeouts
XtSetSelectionTimeout (3) - set and obtain selection timeout values
```

The number in parentheses is the particular manpage's section. We often refer to a manpage with this section, particularly when there are multiple manpages of that name. So **`printf(1)`** is the **`printf`** manpage in section 1, and **`printf(3)`** is the version in section 3.

Our notation is based on the notation used in manpages. That is, optional arguments are surrounded by square braces, and vertical bars separate alternatives. Here's an example for a command, from the manpage for **`man`**:

```
MAN(1)                        Manual pager utils                        MAN(1)

NAME
       man - an interface to the system reference manuals

SYNOPSIS
       man [man options] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [man options] [section] term ...
       man -f [whatis options] page ...
       man -l [man options] file ...
       man -w|-W [man options] page ...
```

The NAME section tells us the name and a brief description, and is typically followed by a SYNOPSIS, showing the calling options. This is followed by a potentially-lengthy DESCRIPTION section, which gives details about the command. Other common sections are OPTIONS, EXAMPLES, EXIT STATUS, and SEE ALSO, though there are often other sections. The OPTIONS and EXAMPLES are often the most useful places to look, once you're somewhat familiar with a command. SEE ALSO shows you other related manpages (which may not actually be installed).

For a function, the format of the manpage looks different. For example, here's part of the **`printf(3)`** manpage:

```
PRINTF(3)                 Linux Programmer's Manual                PRINTF(3)

NAME
       printf,  fprintf,  dprintf,  sprintf,  snprintf, vprintf, vfprintf, vd-
       printf, vsprintf, vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int dprintf(int fd, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vdprintf(int fd, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

Often, these manpages will collect the documentation for multiple related functions (command manpages sometimes do this, as well). As before, we have NAME and SYNOPSIS sections. The latter includes (for C) the file includes needed to call the functions. Since (again, in C) there is no concept of optional or alternate arguments, it shows all of the various ways in which one of the functions can be called individually. The DESCRIPTION section tends to be the longest, and often there will also be an EXAMPLES section. Instead of

EXIT STATUS, you'll see a RETURN VALUE section, telling you what to expect the function to return, including for errors.

File format manpages tend to be less structured. If you want to see some examples, try **`sudoers(5)`** and **`crontab(5)`**.

Chapter 2

File and Directory Structure

2.1 Filesystems

Everything on your computer is stored in a *filesystem*, which is divided into a (large) number of *directories*. While modern operating systems and applications tend to hide this from you, when working with the command line it's important to understand the general structure. Your typical view is of some folder, with all your files in it, possibly organized into subfolders. The filesystem actually begins a few layers "above" this, in a directory called **`/`** (called *slash* or *root*). How do folders and directories differ? They actually don't, but for the sake of exposition we use "folder" to refer to a GUI-focused organization of data, and "directory" to refer to the underlying filesystem structure. They're really the same thing, though.

Under the root directory are a bunch of other directories. The folder with all of your stuff is a subdirectory of **`/home`**. If your username is **`mike`**, for example, this would be **`/home/mike`**. Note the presence of slashes—this is how we separate directory names in a *path*. **`mike`** is a subdirectory of **`home`**, which is a subdirectory of the root. This is an *absolute path*.

Command	Result
`ls`	List the directory contents
`ls -l`	Long listing; lots of details
`ls -a`	Include files/directories beginning with a dot
`ls -A`	Like `-a`, omitting `.` and `..`
`ls -lt`	Long listing, sorted by modified time (latest first)
`ls -ltr`	`-lt` with reversed order
`ls -1`	List in a single column
`ls -m`	List as a single comma-delimited line
`ls -F`	Add type indicators (`/`=directory,`@`=link,etc.)
`ls -lh`	Sizes shown in friendly units
`ls -ln`	Display owner and group as numbers, not names
`ls -d`	Do not expand directories

All optionally take one or more files/directories.

You can view the contents of a directory with the command `ls`. This hides the names of files and directories beginning with a dot by default, but you can show these with `ls -a`. If you run this, you'll see every directory has two entries: `.` and `..` that refer to special directories. A single dot refers to the current directory, which is very handy in some situations. A double dot refers to the *parent* directory, with the root directory's parent being itself. These special directories allow us to construct *relative paths*, like `../foo` for a subdirectory `foo` under our parent directory.

2.2 Moving Through the Filesystem

`cd` Change the working directory to the user's home directory

`cd <dir>` Change the working directory to *dir*

The principal command you'll use to change directories is the `cd` command. It's usage is rather simple, typically taking one argument that is either an absolute path (eg, `cd /var/run/netns`) or a relative path (eg, `cd ..`). There's one

Command	**Result**
`cd`	Change the working directory to the user's home directory
`cd <dir>`	Change the working directory to *dir*
`cd ~<user>`	Change the working directory to *user*'s home directory
`cd -`	Change the working directory to the previous working directory
`pushd <dir>`	Like `cd <dir>`, but adds *dir* to bash's directory stack
`popd`	Pop the top of the directory stack, cd'ing to the new top

other *extremely* useful way to call `cd` that most people are unware of, though: `cd -` This moves you to the *previous* directory that you were in, and is a great way to either work in two different directories from the same shell or to quickly change to another directory for one or two commands, and then return to where you were.

There are two special cases for `cd`: When run without arguments, it takes you to your home directory (eg, `/home/mike`); and when you run `cd ~other`, it will take you to the home directory for the user `other`. Technically, the latter is not a special case, because Linux recognizes `~other` as "the home directory for `other`".

There are other commands that move *files and directories* through the filesystem: `cp` and `mv`. `cp` copies a file (or, with `-r`, a directory) to another location, while `mv` moves it (no additional argument required for a directory). Both commands take the thing you're copying/moving followed by where to place them. You can specify this either as a file name, or just the parent directory, so

```
cp foo /etc/foo
```

and

```
cp foo /etc/
```

are equivalent.

Command	**Result**
`df`	Display statistics for all mounted filesystems
`df <dir>`	Display statistics for the filesystem on which *dir* is mounted
`df -h`	Use friendly units for sizes
`du <dir>`	Count the disk usage for the specified directory and subdirs
`du -s`	Only show the total usage, not the subdir breakdown
`du -h`	Use friendly units for sizes

2.3 Disk Usage

As you create or download files (including installing new libraries or applications), your disk will begin to fill. When it fills too much, you'll need to figure out where the space is being used, so you know what you need to delete. There are two useful commands for this:

`df` Tells you how much space is free or in use for every filesystem

`du` Tells you the usage of directories or files

These will let you figure out how much space is used/available, and where that used space is.

Chapter 3

Files

Files store the actual stuff on your computer. This may seem obvious, but a lot of operating systems try to hide this fact from you. In Linux, pretty much *everything is a file*. That includes your keyboard!

One important thing to remember is that you don't have to "be" in the same directory as a file you want to work with. All of the file commands we will look at take files as *absolute* or *relative* paths.

Command	**Result**
`cat`	Print one or more files to the terminal
`head`	Print the first 10 lines of a file
`tail`	Print the last 10 lines of a file
`more`	Show one screen's worth of a file at a time
`less`	Like `more`, but with additional features
`xxd`	Display a file as hexadecimal bytes

3.1 Viewing File Contents

We'll begin with text files. There are a number of ways to view a text file, and all of these take optional arguments to control their behavior. In section 1.2, you'll learn more about how to discover these options and what they do.

The simplest program is **`cat`**, which *concatenates* files (or standard input), and writes the result to standard output. A simple example of this is

```
cat ~/.bashrc
```

which will dump the contents of your shell configuration file to the terminal. The programs **`head`** and **`tail`** do something similar, but only the first or last lines (10 by default) of the file.

Except for very short files, you generally don't want to write the entire contents of a file to the terminal all at once. Instead, you want to use a program called a *pager*. That is, a program that shows you one "page" of the file at a time (defined as however many lines your terminal can display at once). The most common one that people use is called **`less`**, which is admittedly a strange name for what it does. The name is basically a pun based on the maxim "Less is more." **`more`** is the name of one of the earlier pagers, and is in fact still available on most systems.

Both **`more`** and **`less`** have some similar functionality:

- You can scroll forward a page at a time with the space key.
- The "q" key quits the program.

That's about all that **`more`** does, while **`less`** does more. In particular, **`less`** allows you to scroll both forward and backward, both by page and by line. While **`more`** generally exits when you reach the end of the file, **`less`** does not, and also has commands to jump to the first ("g") or last ("G") line of the file. Another nice feature of **less** is that if you type "-N", it will toggle the display of line numbers.

Not all files are ASCII (that is, "normal" text). These display poorly with **cat** and **less** (though it might know how to display useful information about the file). In these cases, it's useful to be able to view a *hex dump* of the file. There are a number of utilities that can do this, but we'll take a look at a fairly common one, called **xxd**.

The first thing to note is that **xxd** behaves like **cat**, in that it writes everything to the terminal all at once. For very short files, this is fine, but for longer ones, you will want to *pipe* it to another program, like **less**:

```
xxd my_binary_file | less
```

We'll look at *pipelines* in more detail in a later chapter.

The output of **xxd** typically begins with a byte offset for the line, in hex, followed by a colon. After that, it will show 16 bytes of data, grouped in pairs, again in hex. Finally, for any "printable" characters, it will display them as ASCII, with a "." for any non-printable characters.

Let's make this clearer with an example. Consider the following file, which we'll call **hello**:

```
Hello, world!
```

If we run **xxd hello**, we will see:

```
00000000: 4865 6c6c 6f2c 2077 6f72 6c64 210a       Hello, world!.
```

Command	Result
`cp <a> <b>`	Create a copy of file `<a>` named `<b>`
`mv <a> <b>`	Rename a file `<a>` to `<b>`
`dd if=<a> of=<b>`	Dump the contents of input file `<a>` to output file `<b>`

If we look up a hexadecimal ASCII table, we see that character `0x48` is “H”, `0x65` is “e”, and so on. Character `0x0a` is the newline character.

You can omit the byte offsets and ASCII display with the **`-p`** option, so **`xxd -p hello`** gives us

```
48656c6c6f2c20776f726c64210a
```

You can also run **`xxd`** in reverse with the **`-r`** option:

```
$ cat 48656c6c6f2c20776f726c64210a | xxd -r -p
Hello, world!
```

3.2 Moving, Copying, and Extracting Pieces of Files

Some programs will automatically place files in certain places, like **`~/Downloads`**, but we might want them elsewhere. We also sometimes amass enough files in one place that we want to reorganize things. For this, we can use the **`mv`** command, which *moves* a file. The first argument is the file to move, and the second is where to move this. If the second argument is an existing directory, the file *name* will be preserved in the new directory; if not the name of the file will be changed. Note that, by default, **`mv`** will overwrite (or *clobber*) an existing file at the destination. You can disable this with the **`-i`** option, which makes the overwrite behavior interactive. Many people will *alias* the **`mv`** command to always provide this argument.

As an example, let's say we downloaded a file **`test_file`**, and we want to place it in the **`tests`** subdirectory of the current directory. We can do this with

```
mv ~/Downloads/test_file ./tests/
```

The **`cp`** command works much like **`mv`**, except that it makes a *copy* of the file instead of moving it.

`dd` is an extremely powerful tool. Its basic syntax is

```
dd if=foo of=bar
```

This will copy the contents of *input file* **`foo`** to *output file* **`bar`**. **`if`** and **`of`** are only the beginning of its options. Either can be omitted, and use STDIN or STDOUT as the default. Here's another simple example:

```
dd if=/dev/zero of=foo bs=1024 count=5
```

This will create a file **`foo`** with the first 5kB of **`/dev/zero`**, which will provide you with as many NULL bytes as you request. Give this a try, and then run

```
xxd foo
```

See the documentation for **`dd`** for all options. It's well worth your time to learn more about this command! One thing worth noting is that it will take much longer to copy a large number of small blocks than a small number of large blocks, even if the total number of bytes copied is the same.

3.3 File Permissions

Every file or directory has a user (owner) and group, and a set of permission bits (the first column of `ls -l`). On most systems, your group will be the same as your username, though other groups are likely to exist, and you may be a member of some of them. The `groups` command will show you what groups your account belongs to.

Here are some examples:

```
$ ls -ld utilities
drwxr-xr-x 15 mmarsh mmarsh 480 Dec 14 18:57 utilities
$ ls -ld .ssh
drwx------ 11 mmarsh mmarsh 352 Sep 10 13:32 .ssh
$ ls -l .ssh
total 52
-rw-r--r-- 1 mmarsh mmarsh   391 Jun 29  2017 authorized_keys
-rw-r--r-- 1 mmarsh mmarsh    40 Dec 29  2017 config
-rw------- 1 mmarsh mmarsh  3326 Jan 23  2017 id_rsa
-rw-r--r-- 1 mmarsh mmarsh   743 Jan 23  2017 id_rsa.pub
-rw-r--r-- 1 mmarsh mmarsh 18349 Oct 29 13:19 known_hosts
```

In all of these, `mmarsh` is the owner, and all files and directories are also assigned to the group `mmarsh`. The first column is 10-characters wide:

Character	**Meaning**
0	File type: `d`=directory, `l`=symlink, `c`=char device
1	User (`u`) read (`r`) permission
2	User (`u`) write (`w`) permission
3	User (`u`) execute (`x`) permission
4	Group (`g`) read (`r`) permission
5	Group (`g`) write (`w`) permission
6	Group (`g`) execute (`x`) permission
7	Other (`o`) read (`r`) permission
8	Other (`o`) write (`w`) permission
9	Other (`o`) execute (`x`) permission

Anything not set is indicated with a `-`, which for character 0 means a normal file. We see that **`utilities`** is a directory, readable and executable by everyone (user, group, and other), but writable only by user and group. For directories, "executable" means a user with matching credentials can **`cd`** into that directory. **`~/.ssh/id_rsa`**, a *private key*, has full permissions for the user, but no permissions for anyone else. **`~/.ssh/id_rsa.pub`** is readable by everyone, but only writable by the user.

We can change the permissions on a file (a directory is just a type of file) using **`chmod`**. Here are some options

Option	Meaning
`u+rwx`	Add read, write, and execute permissions for the user
`g+rwx`	The same, for the group
`o+rwx`	The same, for others
`o-w`	Remove write permissions for others
`go-rwx`	Remove all permissions for the group and others
`ugo+x`	Add execute permissions for all users
`a+x`	The same as the previous
`700`	Set the permissions to `-rwx------`
`655`	Set the permissions to `-rwxr-xr-x`
`-R`	Apply the permissions recursively, when given a directory

The numeric versions set permissions exactly, and use octal to specify the bits (1=x, 2=w, 4=r) in the order (user, group, other). After writing a lot of scripts, **`chmod a+x <file>`** will become part of your muscle memory.

You can also change the ownership of files, using **`chown`**. The syntax is

```
chown <user>:<group> <file>
```

When you run things as root, you often have to run this (using **`sudo`**) to fix the file ownership. As with **`chmod`**, you can provide **`-R`** to change ownership recursively.

3.4 Wildcards

Many commands that take files or directories accept multiple files. For example, you can move files **`foo`** and **`bar`** to directory **`baz`** with

```
mv foo bar baz/
```

In this case, if the final argument is a directory, you can move as many files as you like.

To make this easier, you can use *wildcards*, which are expanded to all matching filenames. The common wildcards are **`*`** and **`?`**. These behave similarly to their common regular expression forms, but with an implied "any character." That is, the regular expression **`file.*\.txt`** would, in the shell, be written **`file*.txt`** The wildcard **`*`** matches zero or more of any character, while **`?`** matches any *single* character.

When writing scripts, you might find yourself doing the following frequently:

```
chmod a+x *.sh
```

3.5 Standard File Descriptors

Programs interact through open files using *file descriptors*, which are integers. Every process is assigned three automatically:

0 Standard input (`stdin` or `STDIN`)

1 Standard output (`stdout` or `STDOUT`)

2 Standard error (`stderr` or `STDERR`)

Typically, the first file that a program opens will be assigned file descriptor 3. A process can only *read* from standard input, and can only *write* to standard output and standard error.

We will look at these again when we discuss how to run programs and pipelining.

Chapter 4

Shortcuts

4.1 Editing a Line

Many users new to the command line tend to type out their commands, and if there's a mistake, they use backspace to erase everything and re-type it. This is time-consuming, and can introduce other mistakes. Fortunately, the shell provides a number of keyboard bindings that make this process easier.

The first is to simply note that the left and right arrow keys allow you to move back and forth through the line, so you can add or delete characters only where they are needed. There's a way to skip over entire words, though! If you hold down the Alt or Option key, the arrows will take you through the line much faster (unless you have a lot of non-alphanumeric characters).

You can also use a lot of common Emacs key bindings, like Ctrl-A and Ctrl-E to move to the start or end of a line. Deleting characters and words also uses Emacs key bindings, like Ctrl-W, Ctrl-D, and Ctrl-K.

Key	**Moves the cursor...**
Left	left one character
Right	right one character
Alt-Left	left one word
Alt-Right	right one word
Ctrl-A	to the beginning of the line
Ctrl-E	to the end of the line

Key	**Deletes...**
Backspace	the previous character
Ctrl-D	the current character
Ctrl-W	the previous word
Ctrl-K	to the end of the line

4.2 Auto-Filling

The bash shell provides you with a couple of different mechanisms to reduce the amount of typing you have to do (in addition to wildcards). The first is tab-completion, and the second is a history of prior commands.

Bash defines a variable called **`PATH`**, which is a colon-separated list of directories. This is where it looks for executables that are not preceded by a path (for example, **`chmod`** rather than **`/bin/chmod`**). If you start typing a command, and then hit the tab key, bash will expand this to the full command name, as long as it's unique. If it's *not* unique, hit tab again, and it will display all of the potential matches. For example, on my machine, if I type **`ch<TAB>`**, the terminal bell rings and no expansion is done. Hitting tab a second time results in

```
$ ch
chardetect     checkcites     chflags        chktex         chpass
chat           checkgid       chfn           chkweb         chroot
chcon          checklistings  chg            chmod          chsh
checkUpstream  checknr        chgrp          chown
```

Now, I can type the few characters needed to expand fully. For **`chmod`**, this would just be adding **`m<TAB>`**. For **`checkgid`**, this would be adding **`e<TAB>g<TAB>`**.

Bash will also provide tab-completion for filenames, and in some cases options (this is less common). The filename completion behaves similarly to command completion, but only for files in the currently-specified directory. That is, you can find all files beginning with **`l`** in **`/etc`** with **`ls /etc/l<TAB>`**. (Note that this example could also be accomplished with **`ls /etc/l*`**.)

If you hit the up arrow, you will notice that the last command you ran appears. This is because bash maintains a *command history*. If you exit the shell gracefully (by running **`exit`** instead of closing the window), it will typically even save this in a file called **`~/.bash_history`**, though this tends to get overwritten fairly frequently. This history can save you a lot of time when you have to run the same command multiple times.

You can modify the command line from the history, which is very helpful when you have to do essentially the same thing with slightly different options. Another useful way to do this is to test what a command *would* look like by preceding it with **`echo`**. If you're expanding variables or using subshell output (we'll discuss both of these later), then you can verify that you're seeing what you expect. Then you simply hit the up arrow until the command appears again (down arrow will go in the opposite direction through the history), remove the **`echo`**, and hit **`<RETURN>`**. It doesn't matter where your cursor is in the line when you hit **`<RETURN>`**.

If you know a command is probably in your history, but you don't know where, or you know that you'd have to hit the up arrow many times, there's another shortcut to help you. Ctrl-R will start a reverse search through the history; start typing *any part of the command*, including arguments, and it will find the most recent matching entry. You can continue to hit Ctrl-R to find older matches.

4.3 Aliasing

There will often be fairly simple things that you do often enough you'd rather not type all of the options each time. It's also possible that you'll have a preferred set of options you always want to provide for a specific command. For both of these cases, bash allows you do define *aliases*.

The syntax is simple:

```
alias ls='ls -FC'
```

With this defined, any time I type **`ls`**, it will automatically add the options **`-FC`**, which decorates file names to indicate things like whether they are directories, symbolic links, or executables, as well as ensuring a multi-column layout for the listing.

Another useful alias is:

```
alias ltr='ls -ltr'
```

This alias lets me quickly get a *long* listing (with details), sorted by modification time, in reverse order (oldest to newest). When trying to find the recently-modified files in a directory, this is extremely handy.

These aliases are *literal replacements*. That is, if I type **`ls foo`**, it will be executed as **`ls -FC foo`**, and if I type **`ls -l foo`**, it will be executed as **`ls -FC -l foo`**.

Obviously, you don't want to have to type these alias commands all the time, so bash lets you put them in a special file **`~/.bashrc`**, which is executed every time a new shell is created. We'll look at shell configuration in more detail in Chapter 6.

Chapter 5

Programs

In this chapter, we're going to take a deeper look into how to run and manage processes. We'll also look at how we can capture or otherwise redirect input and output.

5.1 Running Programs

We briefly discussed the **PATH** variable in Section 4.2. We'll look at variables more generally Chapter 8. For now, all you need to know is that you can set a variable with

```
PATH=some_value
```

and you can get its value with either of the following:

```
$PATH
${PATH}
```

The version with curled braces makes your statements clearer, and we'll see how to use this in building more complex statements.

The **`PATH`** variable defines a search path for the shell to find executables. The first matching entry "wins," since there's no reason for the shell to keep looking once it's found something. That means the order of directories in your path is very important. Typically, you will install new programs somewhere like **`/usr/local/bin`**, while system programs will be in **`/usr/bin`** or **`/bin`**.

Here's an example of a path:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This shows us that bash will first look for an executable in **`/usr/local/sbin`**, which holds third-party system administration executables. It will then continue looking until it reaches the last path entry, **`/bin`**. If it still hasn't found an executable with a matching name, it will fail with **`command not found`**. The directories in the path are separated with a colon, which cannot normally be used in a file or directory name.

Bash has built-in commands, as well. These are always selected before anything on the path. If you want a specific version of a program (other than a built-in), you can always call it with the path to the program included. This will also override aliases:

```
$ alias ls='ls -FC'
$ ls
bin@    dev/   home/   lib32@   libx32@   mnt/   proc/   run/    srv/   tmp/   var/
boot/   etc/   lib@    lib64@   media/    opt/   root/   sbin@   sys/   usr/
$ /bin/ls
bin    dev   home   lib32   libx32   mnt   proc   run    srv   tmp   var
boot   etc   lib    lib64   media    opt   root   sbin   sys   usr
```

The current directory is always **`.`**, and some people like to add this to their **`PATH`** variable. The advantage of this is that when you start writing scripts, you can just call them if they're in the current directory. There are some potentially serious security implications for this, though (consider a local program named **`ls`**). If you *want* to put **`.`** in your **`PATH`**, you should make it the *last* entry in the list. It's better not to include it, and just call a program **`my_app`** in the current directory as **`./my_app`**.

When a program exits, it returns some value. By convention, successful execution should exit with the value 0. A non-0 value generally indicates some error condition, and depends on the specific program. You can view the exit status of the last program to run with the variable **`?`**, typically with something like

```
$ echo $?
```

There are multiple ways to run a program, and which you use will depend on what you're trying to do. Consider an executable **`foo`**:

Invocation	Effect
`foo`	foo is run as a subprocess normally
`foo &`	foo is run in the background, as execution continues
`. foo`	foo is run *in the current shell*
`(foo)`	a subshell is started, and foo is run as a subprocess of it
`` `foo` ``	foo is run as a subprocess, and its STDOUT is returned
`$(foo)`	Same as the above

The last two are equivalent, but the **`$()`** form is preferable, because it is clearer and can be nested.

The dot-execution is useful for snippets of bash code which set environment variables or define functions. Think of it like an "include" statement.

The advantage of running in a subshell is that it doesn't impact the current shell. Here's an example:

```
for d in *  # "*" expands to the contents of the current directory
do
    if [ -d $d ]  # Test that $d is a directory
    then
        (cd $d; git pull origin master)
    fi
done
```

If we ran in the parent shell, we would have to make this longer:

```
cwd=$(pwd)
for d in *  # "*" expands to the contents of the current directory
do
    if [ -d $d ]  # Test that $d is a directory
    then
        cd $d
        git pull origin master
        cd $cwd
    fi
done
```

We could use **`..`** instead of **`$cwd`**, but then we'd have to worry about **`cd $d`** failing. With a subshell, we don't need to worry about this at all.

5.2 Processes

5.2.1 Process Information

We can get a list of running processes with the **`ps`** command. By default, it only lists processes "of interest," which typically means subprocesses of the current shell. Often, you want to see more than that. There are actually two common sets of options, one from the BSD family of Posix systems, and the other from the UNIX family. What does this mean for you? It means the documentation is frequently confusing. Consult the manpage for **`ps`** for details, but here are some useful invocations for the UNIX-style options (result descriptions are approximate in some cases):

Invocation	Displays
`ps -a`	all processes attached to a terminal
`ps -ae`	all processes
`ps -f`	more information on selected processes
`ps -aef`	more information on all processes
`ps -u root`	all processes owned by the user `root`

These can be combined in any order with a single dash.

Another way to view processes is with `top`. This is a full-terminal program that updates its display every 5 seconds, with the processes sorted by CPU usage. It's possible to change the sorting used, as well, but CPU is often what you're most interested in. It will also report on memory usage for each process and the system as a whole. If all you want is the total CPU load, you can use the `uptime` command. The load is the number of effective CPU cores being used by every process in the system. If all you want is the memory usage, you can use the `free` command, on some systems (like Linux).

5.2.2 Background Processes

A single shell can run multiple subprocesses in parallel, as long as at most one is in the *foreground*. A foreground process has access to STDIN. A process in the *background* does not have access to STDIN, but can still write to STDOUT and STDERR. You can run a process in the background either by appending `&` to the end (as shown above), or by starting it in the foreground, then hitting Ctrl-Z to pause it, and then running `bg`. A process in the background can be brought to the foreground with `fg`.

With multiple background processes, how do we know which one to bring to the foreground? For this, we use the `jobs` command. Consider the following:

```
$ sleep 20 &
[1] 292
$ sleep 30 &
[2] 293
$ jobs
[1]-  Running                  sleep 20 &
[2]+  Running                  sleep 30 &
```

We have two process running **`sleep`**, for different amounts of time. When we start them, bash gives us a number, **`[1]`** and **`[2]`**. These are *job handles*, and can be used as a shorthand for the process (prepended with a **`%`**). The process with a **`+`** in the first column is the one that will be passed by default to a command like **`fg`**. If we want to bring the other process to the foreground, we would run **`fg %1`**.

Sometimes we want to start a process in the background, and then exit the shell. We might do this if we've logged into a remote system, and we want to start a long-running program. By default, when the shell exits it will terminate all of the running subprocesses, even if they're in the background. We can prevent this by telling the shell *not to hang-up* on a process by preceding it with the **`nohup`** command. The new process will then ignore the **`HUP`** signal. The output from the process will be written to a file called **`nohup.out`** in the directory from which you started it, assuming you have write permission.

5.2.3 Terminating Processes

When a program is out of control, or if it's running in the background, you will probably need to fall back on the **`kill`** command to terminate it. This takes one or more process IDs (PIDs) as arguments, and optionally the signal to use. **`SIGTERM`** is the default (this is what Ctrl-C sends), and is usually what you want, though sometimes you want **`SIGKILL`** (which is sent with Ctrl-\):

```
$ kill 1234     # kill PID 1234 with TERM signal
$ kill -9 1234  # kill PID 1234 with KILL signal
```

Note that the **`TERM`** signal can be caught by the process being killed, allowing it to clean up after itself. The **`KILL`** signal cannot be caught, and causes the process to terminate immediately.

The **`killall`** program matches command names, rather than PIDs. It is potentially error-prone, but sometimes very useful.

5.2.4 Setting Priorities

Modern operating systems are almost universally multiprocessing systems. That means multiple programs can be running at the same time, often more than the number of physical processors available on the machine. The kernel has to determine what gets to run when, but we don't necessarily want everything to be *scheduled* equally. For example, the kernel itself should have a higher *priority* than anything else, while a process that does some cleanup of unused resources in the background might only need to have a low priority.

We control process priorities with the **`nice`** and **`renice`** commands. We use **`nice`** when starting a process to set a priority, which ranges from -20 (highest) to 19 (lowest). Without this, a process will have priority 0. If we call

```
$ nice my_program
```

the resulting process will have a priority of 10 instead. If we want to set the priority explicitly, we can instead use

```
$ nice -n 15 my_program
```

to set the priority to 15. Note that only the superuser (**`root`** on a Posix system, like Linux or MacOS) can set negative priorities.

If we want to change the priority of a running process, we use the **`renice`** command. The exact behavior of this depends on your specific version of **`renice`**, so you'll want to consult the manpage. Here are some example usages, for changing the priority of process 1234:

```
$ renice 10 1234    # sets the priority of 1234 to 10
$ renice -n 10 1234 # either sets the priority to 10 or increases it by 10
```

One important thing to note about **`renice`** is that you can only run it on processes that you own, unless you are running as the superuser.

5.3 Input and Output Redirection

We introduced the standard file descriptors **`STDIN`**, **`STDOUT`**, and **`STDERR`** in Section 3.5. The normal thing for a program to do is read from **`STDIN`** and write to **`STDOUT`**. Many programs will also write to **`STDERR`**, but if all you have is the terminal, it's hard to tell **`STDOUT`** from **`STDERR`**. However, the shell still knows, and lets us treat these differently. Here's an example. First, try (we'll look at **`grep`** in Chapter 7):

```
$ grep bash /etc/*
```

Now, try this:

```
$ grep bash /etc/* 2>/dev/null
```

The second form told the shell that **`STDERR`** (2) should be redirected to (**`>`**) the special file **`/dev/null`**. We could also do:

```
$ grep bash /etc/* 2>/dev/null >bash_in_etc
```

You shouldn't see any output now, but take a look at the new file **`bash_in_etc`**, which will contain the matches. If unspecified, output redirection applies to **`STDOUT`**.

We can also merge **`STDOUT`** and **`STDERR`**:

```
$ grep bash /etc/* 2>&1
```

Here, we've specified the redirection target as **`&1`**, which means "whatever file descriptor 1 points to".

To append, instead of overwriting, we can use **`>>`** instead of **`>`**.

We can also redirect **`STDIN`**, by using **`<`**:

```
$ wc -l <bash_in_etc # This counts the number of lines in the file
```

In this particular case, **`wc`** can take a filename instead of reading from **`STDIN`**, so this is not generally the way we'd accomplish this particular task.

There's a special form of input redirection, called a *HERE document*, using **`<<`**:

```
$ wc <<EOFWC
this
is
a
test
EOFWC
```

The string **`EOFWC`** is arbitrary, but **`EOF<command>`** is fairly common. We can also pass a single string as **`STDIN`** with **`<<<`**:

```
$ wc <<<"this is a test"
```

5.4 Pipelines

The Unix philosophy is that a given tool should do one thing, and if you have to do multiple things, you should compose different tools. Pipelines are the shell's way to do this. In short:

```
$ foo | bar | baz
```

takes **`STDOUT`** from **`foo`**, redirects that to the **`STDIN`** of **`bar`**, and redirects **`bar`**'s **`STDOUT`** to **`baz`**'s **`STDIN`**.

You should become comfortable with this pattern, because it is one of the keys to creating powerful scripts. We can also combine with other things we've seen:

```
$ echo "grep produced $(grep bash /etc/* 2>&1 >/dev/null|wc -l) errors"
```

Chapter 6

Shell Configuration

We took a look at aliases and search paths in Chapter 4. These are things we want to configure once, rather than having to do them every time we start a new shell. Bash gives us a couple of ways to do this.

The first way is to run some configuration when a new *login* is created. This typically occurs when we first log into a system or open a new terminal. This is when we want to set things like **`PATH`**. We can also run more complex programs, like **`ssh-agent`**. The critical thing to note is that these are things we don't want to do every time we create a new shell, which happens more often than you might realize.

Bash will look for a file in your home directory called **`.bash_profile`**. If it doesn't find this, it will look for **`.profile`**. If this is also not found, it will execute a system-wide profile, probably **`/etc/profile`**. It's a good idea to execute the system profile in your user profile, with

```
. /etc/profile
```

Note the dot and space before the filename. As we saw in Section 5.1, this will execute the file in the current shell, rather than creating a new shell.

For things we want to run for *every* shell we create, we put our configuration in the **`.bashrc`** file in our home directory. This is where you set aliases and some variables like your Java search path (this could also be done in the profile). As with the profile, there's a system-wide configuration file, so you should add

```
. /etc/bashrc
```

often at the start of your own **`.bashrc`**. These configuration files are bash scripts, so you can technically do anything in them that you'd do in any other script. See Chapter 9 for more details on this.

There is typically a system-defined path, which you will obtain from **`/etc/profile`**. Let's say we have our own directory called **`~/bin`** that we want to add to the start of our path. We could do this in our **`.bash_profile`** with

```
PATH="~/bin:$PATH"
```

There's one other wrinkle we must consider when setting our path. By default, shell variables are only defined for the shell in which you set them, and aren't passed to subprocesses. We can tell the shell that we want to pass these to subprocesses using the **`export`** command. You should always have the following after you finish defining your **`PATH`** variable:

```
export PATH
```

You can always re-execute a configuration file with

```
$ . ~/.bash_profile
$ . ~/.bashrc
```

Chapter 7

How to Find Things

Being able to find something specific is extremely useful. That might mean a particular file, or some text within a file. You might even have more complex search criteria.

7.1 locate

The **`locate`** command finds files by name, including as substrings. It requires an index database, so it's not always installed and configured, but it's very useful when it is. Let's say you run a command, either from the command line or in something like an IDE. You know it created a file called **`C7B16B6C-DB04-41C1-83CE-D622BCB93ABA.log`**, but you have no idea where. You would then run

```
$ locate C7B16B6C-DB04-41C1-83CE-D622BCB93ABA.log
```

If it's in the database, you'll see the full path to the file.

7.2 grep and ack

One of the most useful commands is **`grep`**, to the extent that it has become a common verb. The basic usage is:

```
$ grep <pattern> <file>...
```

The pattern might be a simple string, or it can be a *regular expression* (the command is short for "get regular expression"). You can provide multiple files, or even directories with the **`-d`** option. Here are some of the options:

Option	Meaning
`-E`	Interpret **`<pattern>`** as an extended regular expression
`-r`	Recursively grep directories
`-A <n>`	Include **`<n>`** lines of context after a matching line
`-B <n>`	Include **`<n>`** lines of context before a matching line
`-C <n>`	Include **`<n>`** lines of context before and after a matching line
`-H`	Prepend matching lines with the name of the file
`-i`	Ignore case in matches
`-l`	Only print the names of files with matches
`-L`	Only print the names of files without matches
`-n`	Prepend the matching line number
`-q`	Don't print matches, just return 0 (match) or -1 (no match)
`-v`	Match lines *not* including **`<pattern*>`**

`ack` isn't as commonly installed, but it's very good for searching directories recursively. The arguments are similar to **`grep`** (it expects directories, not regular files), but you should check the documentation.

```
$ ack <pattern> [<directory>...]
```

7.3 find

The `find` command recursively searches a directory for files/directories matching a set of criteria, and can optionally perform actions on those files. If `locate` isn't installed, or you need to search by more than just the filename, then `find` is a great option. Once you gain some familiarity with it, you will likely use it more often than you might expect.

The first argument is the directory in which to begin the search. After that, you specify what you want to find, and what to do with things that you find. `find` has a *lot* of options, far too many to go into detail here. Some of the more useful ones:

Option	Meaning
`-name <n>`	Match files containing `<n>`
`-iname <n>`	Case-insensitive version of `-name`
`-type <t>`	Match files of type `<t>` (`f`=normal file, `d`=directory, etc.)
`-depth <d>`	Limit the depth of the search
`-size <s>`	Match files with size matching `<s>`, like `10`, `20k`, `32M`, etc.
`-size -<s>`	Match files smaller than `<s>`
`-size +<s>`	Match files larger than `<s>`
`-newer <f>`	Match files modified more recently than file `<f>`
`-mtime <t>`	Match files modified within time *t*, default unit days
	Also, `-<t>` or `+<t>`
	`-ctime` and `-atime` do same thing for file creation and access
`-print`	Print the name of a matched file (default)
`-ls`	Print `ls -l`-like lines for matching files
`-exec ...`	Execute a command on matches (see below)
`-delete`	Removes files and directories - **USE WITH EXTREME CAUTION**

For matches, the order can matter, especially for performance. You want to run `-exec` as late in the filtering process as possible, for example, since it runs an external program for each file.

-exec is very powerful, because it allows you to extend **find**'s already-considerably functionality. Here's an illustrative example:

```
$ find . -name \*.txt -exec grep -H foo {} \;
```

This will start from the current directory, match all files ending in "**.txt**", and run **grep** on them. The string "**{}**" is replaced with the name of the current match. The **-exec** command must be terminated with "**\;**", regardless of whether any other commands are provided. This is essentially the same as:

```
$ grep --include \*.txt -Hr foo .
```

Chapter 8

Variables

8.1 Basic Variable Usage

You set a variable with

```
my_var=123
```

Referencing variables is done with a dollar sign, but there is more than one way to do this. These are equivalent:

```
$HOME
${HOME}
```

Why would we use the longer version? Try the following:

```
for f in *; do echo $f0; done
```

Now try:

```
for f in *; do echo ${f}0; done
```

The braces give us considerable more control, and some extra features, as we'll see.

The following table contains many useful shell variables, both interactively and in scripts.

Variable	Contents
`$$`	Current process (shell) PID
`$!`	PID of last subprocess started in the background
`$?`	Return value of the last completed subprocess
`$0`	Name with which the shell/script was invoked
`$1`, `$2`, ...	Positional parameters to the shell/script
`$#`	Number of positional parameters to shell/script
`$`	Positional parameters, expanded as separate words
`$*`	Positional parameters, expanded as a single word
`$HOME`	The current user's home directory
`$OLDPWD`	The previous working directory (see `cd -`)
`$PATH`	The directories searched for commands
`$PPID`	The PID of the shell/script's parent process
`$PWD`	The current working directory
`$RANDOM`	A random int in the range [0,32767]
`$EUID`	The current effective user numeric ID
`$UID`	The current user numeric ID
`$USER`	The current username

It is possible to set variables for a single command. There are two ways to do this:

```
/usr/bin/env a=foo my_command
a=foo my_command
```

Both of these set the variable **`a`** to the value **`foo`**, but *only* for the environment seen by **`my_command`**. This is used frequently to override default variables without changing them:

```
JAVA_HOME=${HOME}/my_java some_java_program
LD_LIBRARY_PATH=${HOME}/build/lib my_c_program
```

By convention, script-local variables are lowercase, and more global variables (like **`HOME`**) are uppercase.

Subprocesses typically don't get passed the variables you define. To change this, you need to export the variable:

```
export PATH
```

8.2 Parameter Expansion

See the ”Parameter Expansion” section of the bash manpage for more details and other expansion options.

Expansion	Effect
`${#var}`	The length of `var`
`${var:-def}`	`var`, if set, otherwise `def`
`${var:=def}`	As above, but `var` will be set to `def` if not set
`${var:off}`	Substring of `var`, beginning with character `off`
`${var:o:l}`	As above, but at most kodel characters
`${var/p/s}`	Expand `var`, replace pattern `p` with string `s`
	If `s` not provided, remove the pattern
	`#` and `%` perform prefix and suffix matches

We can use this in a script along with variable overriding for handling script inputs symbolically, rather than positionally:

```
${target:=foo.txt}
grep foobar ${target}
```

We would call this (assuming it's called `myscript`):

```
$ target=/etc/hosts myscript
```

8.3 Quoting

How a language handles single- vs double-quotes varies quite a bit. Python treats them equivalently, while C only allows a single character between single-quotes. Bash works a bit differently: single-quoted strings do not have parameters expanded, while double-quoted strings do. For example, compare:

```
echo '${HOME}'
echo "${HOME}"
```

You will most often want double-quotes, but single-quotes are very useful when preparing input to another program. For example:

```
find . -name '*.txt'
```

which is equivalent to

```
find . -name \*.txt
```

Chapter 9

Scripting

9.1 Running Scripts

Most shells, bash included, treat executable text files specially. Given no other information, they run them as scripts for the current shell. *Do not assume that file extensions mean anything.* At the very least, bash does not care about the name of your file, it only cares about the content. Naming a file **`foo.py`** does not mean bash will treat it as a python file, for instance.

To run the script correctly, there's an easy way and a hard way. The hard way is to call the appropriate interpreter explicitly:

```
$ bash my_shell_script.sh
$ python my_python_script.py
```

Note for Windows users: In PowerShell, this way will work correctly for you. The following "easy way" will not!

The easy way is to use a convention called *shebang* (short for "hash-bang"). The shell will look at the first line of an executable ASCII file. If that line begins with a shebang, the arguments to that provide the program with which to run the script:

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

You can even provide options:

```
#! /bin/bash -x
```

For python, there's a better way to call it:

```
#! /usr/bin/env python
```

What does this do? We're not actually invoking python directly. Instead, we invoke **`env`**, which passes the parent environment. In particular, this means it's also using the parent shell's **`$PATH`** variable to determine how to find python. This has a couple of advantages:

- The location of **`python`** may vary from installation to installation, but **`env`**'s location is always predictable.
- If you use python virtual environments, this will pick up your virtualenv **`python`**.

As a final note, make sure your scripts are executable! See Section 3.3 for details, but 99 times out of 100, you will want to run **`chmod a+x`** on your script. **`git`** keeps track of file permissions in addition to the contents, so see a reference on **`git`** for information about keeping these in sync with **`git update-index`** and **`git ls-files`**.

9.2 Working with Positional Parameters

Sometimes, **`$*`** or **`$`** are good enough:

```
foo $*               # Pass all parameters to this other command
for a in $*; do ... # Loop over the parameters
```

If the parameters have different meanings, we can do the following:

```
a=$1
b=$2
```

We can make this more robust, with defaults:

```
a=${1:-foo}
b=${2:-bar}
```

We can also use **`shift`**, which pops the first positional parameter:

```
a=$1
shift
b=$1
shift
```

or, more compactly:

```
a=$1; shift
b=$1; shift
```

The advantage of the **`$1; shift`** form is that we can add more positional parameters without having to keep count. We'll see other uses later.

9.3 Mathematical Expressions

Many mathematical operations can be put in `$(( ))`. This will only perform integer math, however. Here's an example:

```
total=0
for thing in $*
do
    total=$(( ${total} + ${#thing} ))
done
echo ${total}
```

This will sum the lengths of the positional parameters, and print the result to STDOUT.

Bash supports a number of mathematical operators:

`+ -` Addition and subtraction

`++ --` Pre- and post-increment and decrement

`!` Binary and bitwise negation

`* / %` Multiplication, division, and modulo

`**` Exponentiation

`<< >>` Bit shifting

`== != < > <= >=` Standard comparators

`& ^ | && ||` Bitwise and logical operators

`:?` Ternary operator

`= *=` ... Assignment, including compounds

Note again that these are all integer operations. For floating-point math, we have to use other options (such as **`awk`**).

9.4 Control Flow

The simplest form of control flow uses boolean operators to combine commands

Combination	Execution
`foo ; bar`	Execute **`foo`**, then execute **`bar`**
`foo && bar`	Execute **`foo`**; if successful execute **`bar`**
`foo || bar`	Execute **`foo`**; if *not* successful execute **`bar`**

The return status is always the status of the last command executed.

Bash has an **`if`**/**`then`**/**`elif`**/**`else`**/**`fi`** construction. The minimal version is **`if`**/**`then`**/**`fi`**, as in:

```
if $foo
then
    echo "foo"
fi
```

The full form would be:

```
if $foo
then
    echo "foo"
elif $bar
then
    echo "bar"
else
    echo "baz"
fi
```

The **`if`** statement uses command return codes, so you can put a command in the test, or use the **`test`** command (usually written **`[`**):

```
grep foo /etc/hosts
have_foo=$?
grep localhost /etc/hosts
have_local=$?
if [ 0 -eq ${have_foo} ]
then
    echo "We have foo"
elif [ 0 -eq ${have_local} ]
then
    echo "We have localhost"
fi
```

See the **`test`** manpage for details; there are many tests you can perform, and the manpage is fairly compact.

We can also construct loops in bash, as we've already seen briefly. There are **`for`** loops and **`while`** loops, and they behave as you'd expect. Both have the format:

```
<for or while>
do
    # ...
done
```

A **`for`** statement looks like

```
for loop_var in <sequence>
```

The sequence can be something like **`$*`**, or **`$a $b $c`**, or **`$(ls /etc)`**. If you want to iterate over numbers, you can do something like

```
for loop_var in $(seq 10)
do
    echo "foo${loop_var}"
done
```

The **`while`** statement takes a conditional, much like **`if`**. We can loop indefinitely with it:

```
while true
do
    # ...
    if $condition
    then
        break
    fi
done
```

Finally, bash has a **`case`** statement:

```
case ${switch_var} in
    foo) echo "foo";;
    bar|baz) echo "bar"; echo "baz";;
    *) echo "default";;
esac
```

Let's combine this for command-line argument parsing:

```
a="foo"
b="bar"
c=""
while [ $# -gt 0 ]
do
    case $1 in
      -a) shift; a=$1; shift;;
      -b) shift; b=$1; shift;;
      *) c="$c $1"; shift;;
    esac
done
```

9.5 Functions

Defining a function is fairly simple:

```
function my_func {
    local a=$1
    echo $a
}
```

Functions begin with the `function` keyword, then a name, and then the body of the function, in curled braces. The `local` keyword defines a variable in the function's scope. If not used, the variable will be defined in the global scope, and hence visible outside of the function. Positional parameters are redefined for the function's scope.

Once defined, the function behaves like any other command:

```
my_func "hello"
```

You can define particularly useful functions in your `~/.bashrc`, which will be executed (using `.`) whenever you start a new shell.

9.6 Utility Programs

9.6.1 true and false

There are two commands, `true` and `false`, which are very useful in scripts. `true` exits with status 0, and does nothing else. `false` exits with a non-0 status (often -1), and does nothing else. These can be used as nops, or to create infinite loops:

```
while true
do
    # ...
```

```
done

until false
do
    # ...
done
```

9.6.2 yes

This program is similar to the file **/dev/zero**, in that it will keep providing output as long as you read it. Rather than producing nulls, it produces an infinite stream of lines containing the character **y**. This can be useful for scripting with tools that require confirmation.

9.6.3 seq

This produces a sequence of numbers, optionally with a starting point and increment. Compare the following:

```
seq 5
seq 1 5
seq 1 2 5
seq 5 1
seq 5 -2 1
```

See the manpage for other options, including more complex formatting.

This is useful in scripts to provide a loop over indices:

```
for a in $(seq 0 5)
do
    echo $a
done
```

9.6.4 cut

This is a workhorse for splitting lines of text.

```
cut -d, -f2 foo.csv      # get column 2 from a comma-separated list
cut -d, -f2,4-7 foo.csv  # get columns 2, 4, 5, 6, and 7
ifconfig | grep flags | cut -d< -f2 | cut -d> -f1
```

9.6.5 awk

`cut` is somewhat limited, so a more powerful tool is frequently useful. **`awk`** has a full programming language, but you'll typically only need a few pieces of it.

By default, **`awk`** splits on whitespace, but you can change this with the **`-F`** option, which takes a regex, rather than a single character. A typical invocation would look like:

```
awk '{ print $1,$3 }' foo.txt
```

to print columns 1 and 3 from **`foo.txt`**.

You can also do math in **`awk`**, which makes it a useful supplement to bash's integer math (Section 9.3). For example:

```
total=$(echo ${total} ${s} | awk '{ print $1 + $2 }')
```

This allows us to sum potentially floating-point numbers. We could also do this by assigning values to variables:

```
total=$(echo | awk -v a=${total} b=${s} '{print a + b }')
```

We still have to pass a file to **awk**, because it's expecting to operate on a file. Fortunately, **echo** is fairly light-weight.

Here's an example from a script that updates a single column in a CSV, re-sums the values, and dumps the results. It also strips off a trailing comma, using another utility called **sed** (see the manpage).

```
echo $LINE | awk -v s=${score} -F, '{
        $5=s
        for (i=3; i<=7; i++) SUM+=$i;
        for (i=1; i<=NF; i++){
                if(i == 2) $i=SUM
                printf "%s,",$i
        }
        print ""
}' | sed 's/,$//g'
```

This overwrites one of the input fields in the line

```
        $5=s
```

The first time we add to the variable **SUM**, it's initialized to 0. The **printf** command works pretty much the same as in C.