Bash Shell Scripting

Currently this book provides an introduction level knowledge of Bash. Go to External Programs, External links, Using man, info and help for further directions and inspirations.

Introduction

What is Bash?

Bash is a "Unix shell": a command line interface for interacting with the operating system. It is widely available, being the default shell on many GNU/Linux distributions and on Mac OSX, with ports existing for many other systems. It was created in the late 1980 by a programmer named Brian Fox, working for the Free Software Foundation. It was intended as a free software alternative to the Bourne shell (in fact, its name is an acronym for *Bourne Again SHell*), and it incorporates all features of that shell, as well as new features such as integer arithmetic and job control[1].

What is shell scripting?

In addition to the interactive mode, where the user types one command at a time, with immediate execution and feedback, Bash (like many other shells) also has the ability to run an entire script of commands, known as a "Bash shell script" (or "Bash script" or "shell script" or just "script"). A script might contain just a very simple list of commands — or even just a single command — or it might contain functions, loops, conditional constructs, and all the other hallmarks of imperative programming. In effect, a Bash shell script is a computer program written in the Bash programming language.

Shell scripting is the art of creating and maintaining such scripts.

Shell scripts can be called from the interactive command-line described above; or, they can be called from other parts of the system. One script might be set to run when the system boots up; another might be set to run every weekday at 2:30 AM; another might run whenever a user logs into the system.

Shell scripts are commonly used for many system administration tasks, such as performing disk backups, evaluating system logs, and so on. They are also commonly used as installation scripts for complex programs. They are particularly suited to all of these because they *allow* complexity without *requiring* it: if a script just needs to run two external programs, then it can be a two-line script, and if it needs all the power and decision-making ability of a Turing-complete imperative programming language, then it can have that as well.

How to read this book

(1) It's ideal to practice BASH while reading

You will probably find this book most useful if you have a Bash command-line open in another window as you are going through it. You may want to try out some of the examples; but more important is to try things out that *aren't* just copied from the book. If you think something probably works a certain way, test out your expectations! Except where noted otherwise, the examples in this book do not require any special privileges or permissions, and do not require any software aside from Bash itself and very common utilities that are generally found on any system where Bash is. (If you are on a Windows 10 system, you can install Windows Subsystem for Linux. If you are on an older machine and don't have SSH access to a GNU/Linux or Unix system, you will probably find it easiest to make use of this book if you are using something like Cygwin, which has not only Bash but also many other common utilities, than if you are using a Bash port for Windows that includes *only* Bash. If you have Cygwin, just make sure that the directory containing those utilities — likely something like `C:\Cygwin\bin` — is listed in your Windows path.)

(2) Try consulting documentation for unknown commands

When you are experimenting, you will likely find it helpful to consult the documentation for various commands. For commands that are built into Bash, you can use the built-in `help` command; for example, `help echo` will "print" (that is, display) information about the built-in `echo` command. For external programs, it is likely that their manual pages are installed on your system, in which case you can view them via the `man` ("manual") command; for example, for information about the `cp` ("copy") command, you can type `man cp`. Additionally, most programs, when run with the argument `--help`, will print some help information; for example, `cp --help` gives almost as much information as `man cp`. (With many programs, however, the `--help` approach does not give anywhere near as much information as the `man` approach.)

(3) Warning against blindly entering commands

A word of caution: this book is part of a wiki, which means that anyone may have edited it. It is possible — unlikely, but possible — that someone has added an "example" that is actually malicious code, or inserted malicious code into an existing example. Trust your instincts; if an example seems suspicious to you, if the surrounding text doesn't adequately explain what it does, if it doesn't seem to match the description in the surrounding text, then *don't run it*. And even if it's clear that an example is non-malicious, it's a good idea to run all of your experiments using an account that has no special privileges or permissions on the system, so you don't (say) accidentally delete something important. Bash is a very powerful systems administration tool, which sometimes means it's a powerful way to accidentally cripple your system!

If you are completely new to Bash, or if you have some experience with the Bash command-line but are completely new to programming, you will probably find this book most useful if you begin at the beginning, and work your way through in linear order. If the topic of any section is completely familiar to you, you might want to skim it quickly for possible surprises, rather than skipping it completely; after all, you may not know what you may not know.

If you already have some Bash experience and some programming experience, you can probably skim the entire first half of the book, focusing only on examples that show Bash syntax you aren't already familiar with. That done, you should be in a good position to get the most out of the second half of the book.

If you already have a lot of Bash shell-scripting experience, this may not be the book for you; you will probably learn some things, but not as much as you would learn from the Bash Reference Manual on the Free Software Foundation's web-site, or simply from reading the entirety of `man bash`. (But this being a wiki, you are invited to share the fruits of your experience!) This book is not a comprehensive reference manual; it focuses on techniques that are either widely used or widely useful, and mostly assumes a typical setup rather than exhaustively documenting all the unusual features that can affect the behavior of a command.

A few notes on terminology

The language of Bash commands is a Turing-complete programming language, meaning that if it is even theoretically possible for a given computation to be performed, then it is theoretically possible to perform it in Bash. Furthermore, from a practical standpoint, it is possible to do some very sophisticated things in Bash, and people often do. Nonetheless, it is usually described as a "shell-scripting language" (or similar) rather than a "programming language", and the programs written in it are usually referred to as "Bash shell scripts" (or similar) rather than "Bash programs". We will follow that convention here, preferring **script** for a Bash script and **program** only when referring to external programs that might be invoked from a Bash script.

We will also often use the term **utility**. This term is somewhat subjective, and usage varies, but it typically describes an external program that requires no human interaction and is intended to work well with other programs. All utilities are programs, so we will use the terms somewhat interchangeably. The extreme opposite of a "utility" is an **application**; that term is also subjective, with variable usage, but it typically describes an external program that has a relatively complex graphical user interface (GUI) and is not intended to work well with other programs. Since applications, by their nature, are not generally called from Bash scripts, we will have less occasion to use that term. Many programs are neither "utilities" nor "applications", and some programs have elements both of utilities and of applications.

Of the various elements of a Bash script, some are typically called **commands**, while others are typically called **statements**. We will observe this distinction here, but you need not overly concern yourself with it. Normally "command" is used in reference to invocations of external programs, of Bash built-ins such as `echo` that resemble external programs, of shell functions (which we will see below), and so on. Normally "statement" is used in reference to instances of Bash programming constructs such as `if ... then ... else ... fi` or variable assignment (which we will see shortly). Perhaps confusingly, **command** is also used in reference to any Bash built-in, even if a use of that built-in would result in a "statement" rather than a "command"; for example, "an `if` statement" is a statement that uses "the `if` command". If this distinction seems too fine, do not worry; in context, it should always be sufficiently clear what is meant by one term or the other.

Some notes on your shell

Shell version

This textbook provides information for those commonly-available versions of Bash, as of July 2014, this should be Bash version 4. Using an earlier version of bash will make you fail to try some of the features out.

The bash, version 3.2, included in OS X is quite important to mention. A really noticeable change for some users may be the missing `-e` option which enables interpretation of backslash escapes, an important feature for color printing. Although this can be done using a workaround (e.g. using `printf` instead), this can be quite inconvenient.

Consider getting the source from http://www.gnu.org/software/bash/ and compile one.

- Bash 4.3.18 for OS X 10.8+, built by Arthur200000: Baidu Download Link Read *KNOWN_ISSUES* before you download. Just look for '下载', it means 'Download'.

bashrc

The `~/.bashrc` file determines the behavior of interactive shells, and having it properly configured can make life easier.

Most GNU/Linux distros have the `~/.bashrc` file. Use your favourite text editor to explore it.

In most cases, there is also a global `bashrc` file, usually located at `/etc/bashrc`. Try changing it, if you are root. If you don't know what to do, take a look at those `bashrc`s:

- Advanced Bash Scripting:Sample Bashrc
- A Modified version of The BLFS /etc/bashrc

Some introductory examples

Let's start with a simple "hello world" program:

```
echo 'Hello, world!'
```

We can either type this directly at the Bash prompt, or else save this as a file (say, `hello_world.sh`) and run it by typing `bash hello_world.sh` at the Bash prompt. (Later we will see some more sophisticated ways to create and run a Bash script file.) In either case, it will print `Hello, world!`:

```
$ echo 'Hello, world!'
Hello, world!
```

Here we have used the `$` symbol to indicate the Bash prompt: after `$`, the rest of the line shows the command that we typed, and the following line shows the output of the command.

Here is a slightly more complex script:

```
if [[ -e readme.txt ]] ; then
  echo 'The file "readme.txt" exists.'
else
  echo 'The file "readme.txt" does not exist.'
fi
```

This script tests whether there exists a file named `readme.txt` in the current directory, and uses an `if` statement to control, based on that test, which commands are run. It, too, could be typed directly at the prompt — any script can — but in this case that is not likely to be useful.

Both of the above are entirely "within Bash", in that they do not require Bash to run any external programs. (The commands `echo`, `if … then … else … fi`, and `[[ -e … ]]` are all built-in commands, implemented by Bash itself.) But, being a shell-scripting language, a large part of Bash's purpose is to run external programs. The following script demonstrates this ability:

```
if [[ -e config.txt ]] ; then
  echo 'The file "config.txt" already exists. Comparing with default . . .'
  diff -u config-default.txt config.txt > config-diff.txt
  echo 'A diff has been written to "config-diff.txt".'
else
  echo 'The file "config.txt" does not exist. Copying default . . .'
  cp config-default.txt config.txt
  echo '. . . done.'
fi
```

Here `diff` and `cp` are two common utility programs that, while not part of Bash, are found on most systems that have Bash. The above script assumes the presence of a default configuration file named `config-default.txt`, and checks for the presence of a configuration file named `config.txt`. If `config.txt` exists, then the script uses the external program `diff` to produce a "diff" (a report of the differences between, in this case, two files), so that the user can see what non-default configurations are in place. If `config.txt` does not exist, then the script uses the external program `cp` ("copy") to copy the default configuration file to `config.txt`.

As you can see, the external programs are run using the same sort of syntax as the built-in commands; they're both just "commands".

The above version of this script is very "verbose", in that it generates a great deal of output. A more typical script would likely not include the `echo` commands, since users are unlikely to need this level of information. In that case, we might use the # notation to include comments that are completely ignored by Bash, and do not appear to the user. Such comments are simply informative notes for someone reading the script itself:

```
if [[ -e config.txt ]] ; then
  # if config.txt exists:
  diff -u config-default.txt config.txt > config-diff.txt   # see what's changed
else
  # if config.txt does not exist:
  cp config-default.txt config.txt  # take the default
fi
```

But the above is simply for demonstration's sake. In reality, a script this simple does not require any comments at all.

Simple commands

A *simple command* consists of a sequence of *words* separated by spaces or tabs. The first word is taken to be the name of a command, and the remaining words are passed as *arguments* to the command. We have already seen a number of examples of simple commands; here are some more:

- `cd ..`
  - This command uses `cd` ("change directory"; a built-in command for navigating through the filesystem) to navigate "up" one directory.

- The notation `..` means "parent directory". For example, `/foo/bar/../baz.txt` is equivalent to `/foo/baz.txt`.

- `rm foo.txt bar.txt baz.txt`
  - Assuming the program `rm` ("remove") is installed, this command deletes the files `foo.txt`, `bar.txt`, and `baz.txt` in the current directory
  - Bash finds the program `rm` by searching through a configurable list of directories for a file named `rm` that is executable (as determined by its file-permissions).
- `/foo/bar/baz bip.txt`
  - This command runs the program located at `/foo/bar/baz`, passing `bip.txt` as the sole argument.
  - `/foo/bar/baz` must be executable (as determined by its file-permissions).
  - **Warning: Please ensure that there is NO SPACE between the forward slash and any files following it** e.g. assuming the "foo" folder exists in the "root" directory, then executing the following command: "rm -r / foo" will destroy your computer if performed with "sudo" access. You have been warned. If you don't understand the preceding, don't worry about it for the moment.
  - If `/foo/bar/baz` is a text-file rather than a binary program, and its first line begins with `#!`, then the remainder of that line determines the *interpreter* to use to run the file. For example, if the first line of `/foo/bar/baz` is `#!/bin/bash`, then the above command is equivalent to `/bin/bash /foo/bar/baz bip.txt`

That example with `/foo/bar/baz` bears special note, since it illustrates how you can create a Bash-script that can be run like an ordinary program: just include `#!/bin/bash` as the first line of the script (assuming that that is where Bash is located on your system; if not, adjust as needed), and make sure the script has the right file-permissions to be readable and executable. For the remainder of this book, all examples of complete shell scripts will begin with the line `#!/bin/bash`.

Quoting

We saw above that the command `rm foo.txt bar.txt baz.txt` removes three separate files: `foo.txt`, `bar.txt`, and `baz.txt`. This happens because Bash splits the command into four separate words based on whitespace, and three of those words become arguments to the program `rm`. But what if we need to delete a file whose name contains a space?

Caution:

In Unix and GNU/Linux distributions, file names can contain spaces, tabs, newlines, and even control characters.

Bash offers several *quoting mechanisms* that are useful for this case; the most commonly used are single-quotes `'` and double-quotes `"`. Either of these commands will delete a file named `this file.txt`:

```
rm 'this file.txt'
```

```
rm "this file.txt"
```

Within the quotation marks, the space character loses its special meaning as a word-separator. Normally we wrap an entire word in quotation marks, as above, but in fact, only the space itself actually needs to be enclosed; `this' 'file.txt` or `this" "file.txt` is equivalent to `'this file.txt'`.

Another commonly-used quoting mechanism is the backslash `\`, but it works slightly differently; it quotes (or "escapes") just one character. This command, therefore, is equivalent to the above:

```
rm this\ file.txt
```

In all of these cases, the quoting characters themselves are not passed in to the program. (This is called *quote removal*.) As a result, `rm` has no way of knowing whether it was invoked — for example — as `rm foo.txt` or as `rm 'foo.txt'`.

Filename expansion and tilde expansion

Bash supports a number of special notations, known as *expansions*, for passing commonly-used types of arguments to programs.

One of these is *filename expansion*, where a *pattern* such as `*.txt` is replaced with the names of all files matching that pattern. For example, if the current directory contains the files `foo.txt`, `bar.txt`, `this file.txt`, and `something.else`, then this command:

```
echo *.txt
```

is equivalent to this command:

```
echo 'bar.txt' 'foo.txt' 'this file.txt'
```

Here the asterisk `*` means "zero or more characters"; there are a few other special pattern characters (such as the question mark `?`, which means "exactly one character"), and some other pattern-matching rules, but this use of `*` is by far the most common use of patterns.

Filename expansion is not necessarily limited to files in the current directory. For example, if we want to list all files matching `t*.sh` inside the directory `/usr/bin`, we can write this:

```
echo /usr/bin/t*.sh
```

which may expand to something like this:

```
echo /usr/bin/test.sh /usr/bin/time.sh
```

If no files match a specified pattern, then no substitution happens; for example, this command:

```
echo asfasefasef*avzxv
```

will most likely just print `asfasefasef*avzxv`.

Caution:

If any filenames begin with a hyphen `-`, then filename-expansion can sometimes have surprising consequences. For example, if a directory contains two files, named `-n` and `tmp.txt`, then `cat *` expands to `cat -n tmp.txt`, and `cat` will interpret `-n` as an option rather than a filename; instead, it is better to write `cat ./*` or `cat -- *`, which expands to `cat ./-n ./tmp.txt` or `cat -- -n tmp.txt`, removing this problem.

What happens if we have an actual file named `*.txt` that we want to refer to? (Yes, filenames can contain asterisks!) Then we can use either of the quoting styles we saw above. Either of these:

```
cat '*.txt'
```

```
cat "*.txt"
```

will print the actual file `*.txt`, rather than printing all files whose names end in `.txt`.

Another, similar expansion is *tilde expansion*. Tilde expansion has a lot of features, but the main one is this: in a word that consists solely of a tilde `~`, or in a word that begins with `~/` (tilde-slash), the tilde is replaced with the full path to the current user's home directory. For example, this command:

```
echo ~/*.txt
```

will print out the names of all files named `*.txt` in the current user's home directory

Brace expansion

Similar to filename expansion is *brace expansion*, which is a compact way of representing multiple similar arguments. The following four commands are equivalent:

```
ls file1.txt file2.txt file3.txt file4.txt file5.txt
```

```
ls file{1,2,3,4,5}.txt
```

```
ls file{1..5..1}.txt
```

```
ls file{1..5}.txt
```

The first command lists each argument explicitly. The other three commands all use brace expansion to express the arguments more tersely: in the second command, all the possibilities `1` through `5` are given, separated by commas; in the third command, a numeric sequence is given ("from 1 to 5, incrementing by 1"); and the fourth command is the same as the third, but leaves the `1` implicit.

We can also list the files in the opposite order:

```
ls file5.txt file4.txt file3.txt file2.txt file1.txt
```

```
ls file{5,4,3,2,1}.txt
```

```
ls file{5..1..-1}.txt
```

```
ls file{5..1}.txt
```

with the default increment size being `-1` when the endpoint of the sequence is less than the starting-point.

Since in Bash, the first word of a command is the program that is run, we could also write the command this way:

```
{ls,file{1..5}.txt}
```

but obviously that is not conducive to readability. (The same sort of thing, incidentally, can be done with filename expansion.)

Brace expansion, like filename expansion, can be disabled by any of the quoting mechanisms; `'{'`, `"{"`, or `\{` produces an actual literal curly-brace.

Redirecting output

Bash allows a command's standard output (file-descriptor 1) to be sent to a file, rather than to the console. For example, the common utility program `cat` writes out a file to standard output; if we redirect its standard output to a file, then we have the effect of copying the contents of one file into another file.

If we want to overwrite the destination file with the command's output, we use this notation:

```
cat input.txt > output.txt
```

If we want to keep the existing contents of the destination file as they are, and merely append the command's output to the end, we use this notation:

```
cat input.txt >> output.txt
```

However, not everything that a program writes to the console goes through standard output. Many programs use standard error (file-descriptor 2) for error-messages and some types of "logging" or "side-channel" messages. If we wish for standard error to be combined with standard output, we can use either of these notations:

```
cat input.txt &>> output.txt
```

```
cat input.txt >> output.txt 2>&1
```

If we wish for standard error to be appended to a different file from standard output, we use this notation:

```
cat input.txt >> output.txt 2>> error.txt
```

In fact, we can redirect *only* standard error, while leaving standard output alone:

```
cat input.txt 2>> error.txt
```

In all of the above examples, we can replace `>>` with `>` if we want to overwrite the redirect-target rather than append to it.

Later we will see some more advanced things that we can do with output redirection.

Redirecting input

Just as Bash allows a program's output to be sent into a file, it also allows a program's input to be taken from a file. For example, the common Unix utility `cat` copies its input to its output, such that this command:

```
cat < input.txt
```

will write out the contents of `input.txt` to the console.

As we have already seen, this trick is not needed in this case, since `cat` can instead be instructed to copy a specified file to its output; the above command is simply equivalent to this one:

```
cat input.txt
```

This is the rule rather than the exception; most common Unix utilities that can take input from the console also have the built-in functionality to take their input from a file instead. In fact, many, including `cat`, can take input from multiple files, making them even more flexible than the above. The following command prints out `input1.txt` followed by `input2.txt`:

```
cat input1.txt input2.txt
```

Nonetheless, input redirection has its uses, some of which we will see later on.

A preview of pipelines

Although they are outside the scope of this chapter, we now have the background needed for a first look at pipelines. A *pipeline* is a series of commands separated by the pipe character `|`. Each command is run at the same time, and the output of each command is used as the input to the next command.

For example, consider this pipeline:

```
cat input.txt | grep foo | grep -v bar
```

We have already seen the `cat` utility; `cat input.txt` simply writes the file `input.txt` to its standard output. The program `grep` is a common Unix utility that filters ("greps", in Unix parlance) based on a pattern; for example, the command `grep foo` will print to its standard output any lines of input that contain the string `foo`. The command `grep -v bar` uses the `-v` option to invert the pattern; the command prints any lines of input that *don't* contain the string `bar`. Since the input to each command is the output of the previous command, the net result is that the pipeline prints any lines of `input.txt` that *do* contain `foo` and do *not* contain `bar`.

Variables

In a Bash script, there are a few different kinds of *parameters* that can hold *values*. One major kind of parameter is *variables*: named parameters. If you are familiar with almost any other imperative programming language (such as C, BASIC, Fortran, or Pascal), then you are already familiar with variables. The following simple script uses the variable `location` to hold the value `world`, and prints out a "Hello, world!" message:

```
location=world               # store "world" in the variable "location"
echo "Hello, ${location}!"   # print "Hello, world!"
```

As you can see, the string `${location}` in the second line was replaced by `world` before that command was run. This substitution is known as *variable expansion*, and it's more flexible than you might suspect. For example, it can even hold the name of the command to run:

```
cmd_to_run=echo                  # store "echo" in the variable "cmd_to_run"
"${cmd_to_run}" 'Hello, world!'  # print "Hello, world!"
```

In both of the above examples, we have used the notation `${variable_name}` to perform the variable expansion. The briefer notation `$variable_name` – without curly brackets – would have the same effect in these cases. Sometimes the curly brackets are necessary (for example, `${foo}bar` cannot be written as `$foobar`, because the latter would be interpreted as `${foobar}`), but they can usually be omitted, and real-world scripts usually omit them.

Of course, needless to say, the above are not very realistic examples; they demonstrate only *how* to use variables, not *why* or *when* to use them. If you are familiar with other imperative programming languages, then it is probably already obvious to you why and when you would use variables; if not, then this should become clear as you read through this book and see examples that use them more realistically.

You may have noticed that we have used double-quotes `"`, rather than single-quotes `'`, for character-strings that include variable expansions. This is because single-quotes prevent variable expansion; a command such as `echo '${location}'` will print the actual string `${location}`, rather than printing the value of a variable named `location`.

It is generally a good idea to wrap variable expansions in double-quotes, because otherwise the results of variable expansion will undergo filename expansion, as well as *word splitting* (whereby white-space is used to separate the words that make up a command). For example, this script:

```
foo='a  b*'     # store "a  b*" in the variable "foo"
echo $foo
```

is liable to print something like `a ba.txt bd.sh`, which is probably not what we want. Real-world scripts frequently do not include double-quotes except when they are clearly necessary, but this practice sometimes leads to confusing bugs.

Tip:

It is generally a good idea to wrap variable expansions in double-quotes; for example, use `"$var"` rather than `$var`.

A number of variables have special significance. For example, the variable `PATH` determines the list of directories that Bash should search in when trying to run an external program; if it is set to `/usr/bin:/bin`, then the command `cp src.txt dst.txt` will look to run either `/usr/bin/cp` or `/bin/cp`. The variable `HOME` is pre-initialized to the current user's home directory, and it determines the behavior of tilde-expansion. For example, if a script sets `HOME=/foo`, then `echo ~/bar` will print `/foo/bar`. (This will not actually change the user's home directory, however.)

Positional parameters

In most of the above commands — both those that run a built-in command, and those that use an external program — we have supplied one or more *arguments*, that indicate what the command should operate on. For example, when we invoke the common Unix utility `mkdir` ("make directory") to create a new directory, we invoke it with a command like this one:

```
mkdir tmp
```

where `tmp` is the name of the new directory to create.

And as we have seen, Bash scripts are themselves programs that can be run. So it goes without saying that they, too, can take arguments. These arguments are available to the program as its *positional parameters*. Earlier, we saw that variables are one kind of parameter. Positional parameters are very similar, but are identified by numbers rather than by names. For example, `$1` (or `${1}`) expands to the script's first argument. So, suppose we want to create a simple script called `mkfile.sh` that takes two arguments — a filename, and a line of text — and creates the specified file with the specified text. We can write it as follows:

```
#!/bin/bash
echo "$2" > "$1"
```

(Notice the line `#!/bin/bash` at the very beginning of the file; we covered that line at Basic commands. When you run this code, that line will guarantee that it will be interpreted by the Bash shell, even if you are running from another program or your computer has a non-standard configuration.)

and (after making it executable by running `chmod +x mkfile.sh`) we can run it as follows:

```
./mkfile.sh file-to-create.txt  'line to put in file'
```

We can also refer to all of the arguments at once by using `$@`, which expands to *all* of the positional parameters, in order. When wrapped in double-quotes, as `"$@"`, each argument becomes a separate word. (Note: the alternative `$*` is perhaps more common, but `"$*"` becomes a single word, with spaces in between the original parameters. `"$@"` is almost always preferable to either `$@` or `$*`, which allow an argument to become split into multiple words if it contains whitespace, and to `"$*"`, which combines multiple arguments into a single word.) This is often useful in concert with the built-in command `shift`, which removes the first positional parameter, such that `$2` becomes `$1`, `$3` becomes `$2`, and so on. For example, if we change `mkfile.sh` as follows:

```
#!/bin/bash
file="$1" # save the first argument as "$file"
shift # drop the first argument from "$@"
echo "$@" > "$file" # write the remaining arguments to "$file"
```

then we can run it as follows:

```
./mkfile.sh file-to-create.txt line to put in file
```

and all of the arguments, except the filename, will be written to the file.

The number of positional parameters is available as `$#`; for example, if `$#` is `3`, then the positional parameters are `$1`, `$2`, and `$3`.

Note that positional parameters beyond `$9` require the curly braces; should you need to refer to the tenth argument, for example, you must write `${10}` rather than `$10`. (The latter would be interpreted as `${1}0`.) That said, it is not usually a good idea to have so many arguments with specific meanings, since it is hard for users to keep track of them. If you find yourself specifically referring to your script's tenth argument, it may be worth re-evaluating your approach.

If you have some experience with Bash, or with Unix utilities, you've most likely noticed that many commands can take various "options", indicated with a leading hyphen `-`, in addition to their regular arguments. For example, `rm "$filename"` deletes an individual file, while `rm -r "$dirname"` deletes an entire directory with all of its contents. (The `-r` is short for "recursive": the command "recursively" deletes an entire tree of directories.) These options are actually just arguments. In `rm "$filename"`, there is just one argument (`"$filename"`), while in `rm -r "$dirname"` there are two (`-r` and `"$dirname"`). In a sense, there is nothing inherently special about these arguments, but this notation for options is so widespread as to be considered standard; many or most of the Bash built-in commands can accept various options, and later we will see various techniques for supporting options as arguments to our Bash scripts.

Exit status

When a process completes, it returns a small non-negative integer value, called its *exit status* or its *return status*, to the operating system. By convention, it returns zero if it completed successfully, and a positive number if it failed with an error. (This approach allows multiple different errors to be distinguished by using different positive numbers.) A Bash script can obey this convention by using the built-in command `exit`. The following command:

```
exit 4
```

terminates the shell script, returning an exit status of four, indicating some sort of error. When no exit status is specified (either because `exit` is run with no argument, or because the script ends without calling `exit`), the script returns the exit status of the last command it ran.

One way that exit statuses are used is with the Bash operators `&&` ("and") and `||` ("or"). If two commands are separated by `&&`, then the command on the left is run first, and the command on the right is only run if the first command succeeds. Conversely, if they are separated by `||`, then the command on the right is only run if the command on the left fails.

For example, suppose we want to delete the file `file.txt` and recreate it as a blank file. We can delete it using the common Unix utility `rm` ("remove"), and recreate it using the common Unix utility `touch`; so, we might write this:

```
rm file.txt
touch file.txt
```

But really, if `rm` fails, we don't want to run `touch`: we don't want to recreate the file if we failed to delete it to begin with. So, we can write this instead:

```
rm file.txt && touch file.txt
```

This is the same as before, except that it won't try to run `touch` unless `rm` has succeeded.

A third Boolean-like operator, `!` ("not"), inverts the exit status of a command. For example, this command:

```
! rm file.txt
```

is equivalent to `rm file.txt`, except that it will indicate success if `rm` indicates failure, and vice versa. (This is not usually useful when exit statuses are used as actual exit statuses, indicating success or failure, but we will soon see some extended uses of exit statuses where a "not" operation is more useful.)

The exit status of a command is (briefly) available as `$?`. This can be useful when it is necessary to distinguish between multiple different failure statuses; for example, the `grep` command (which searches for lines in a file that match a specified pattern) returns 0 if it finds a match, 1 if it finds no matches, and 2 if a genuine error occurs.

Conditional expressions and `if` statements

Very often, we want to run a certain command only if a certain condition is met. For example, we might want to run the command `cp source.txt destination.txt` ("copy the file `source.txt` to location `destination.txt`") if, and only if, `source.txt` exists. We can do that like this:

```
#!/bin/bash

if [[ -e source.txt ]] ; then
  cp source.txt destination.txt
fi
```

The above uses two built-in commands:

- The construction `[[ condition ]]` returns an exit status of zero (success) if *`condition`* is true, and a nonzero exit status (failure) if *`condition`* is false. In our case, *`condition`* is `-e source.txt`, which is true if and only if there exists a file named `source.txt`.
- The construction

  ```
  if command1 ; then
    command2
  fi
  ```

  first runs *`command1`*; if that completes successfully (that is, if its exit status is zero), then it goes on to run *`command2`*.

In other words, the above is equivalent to this:

```
#!/bin/bash

[[ -e source.txt ]] && cp source.txt destination.txt
```

except that it is more clear (and more flexible, in ways that we will see shortly).

In general, Bash treats a successful exit status (zero) as meaning "true" and a failed exit status (nonzero) as meaning "false", and vice versa. For example, the built-in command `true` always "succeeds" (returns zero), and the built-in command `false` always "fails" (returns one).

Caution:

In many commonly-used programming languages, zero is considered "false" and nonzero values are considered "true". Even in Bash, this is true within arithmetic expressions (which we'll see later on). But at the level of commands, the reverse is true: an exit status of zero means "successful" or "true" and a nonzero exit status means "failure" or "false".

Caution:

Be sure to include spaces before and after `[[` and `]]` so that Bash recognizes them as separate words. Something like `if[[` or `[[-e` will not work properly.

`if` statements

`if` statements are more flexible than what we saw above; we can actually specify *multiple* commands to run if the test-command succeeds, and in addition, we can use an `else` clause to specify one or more commands to run instead if the test-command *fails*:

```
#!/bin/bash

if [[ -e source.txt ]] ; then
  echo 'source.txt exists; copying to destination.txt.'
  cp source.txt destination.txt
else
  echo 'source.txt does not exist; exiting.'
  exit 1 # terminate the script with a nonzero exit status (failure)
fi
```

The commands can even include other `if` statements; that is, one `if` statement can be "nested" inside another. In this example, an `if` statement is nested inside another `if` statement's `else` clause:

```
#!/bin/bash

if [[ -e source1.txt ]] ; then
  echo 'source1.txt exists; copying to destination.txt.'
  cp source1.txt destination.txt
else
  if [[ -e source2.txt ]] ; then
    echo 'source1.txt does not exist, but source2.txt does.'
    echo 'Copying source2.txt to destination.txt.'
    cp source2.txt destination.txt
  else
    echo 'Neither source1.txt nor source2.txt exists; exiting.'
    exit 1 # terminate the script with a nonzero exit status (failure)
  fi
fi
```

This particular pattern — an `else` clause that contains exactly one `if` statement, representing a fallback-test — is so common that Bash provides a convenient shorthand notation for it, using `elif` ("else-if") clauses. The above example can be written this way:

```
#!/bin/bash

if [[ -e source1.txt ]] ; then
  echo 'source1.txt exists; copying to destination.txt.'
  cp source1.txt destination.txt
elif [[ -e source2.txt ]] ; then
  echo 'source1.txt does not exist, but source2.txt does.'
  echo 'Copying source2.txt to destination.txt.'
  cp source2.txt destination.txt
else
  echo 'Neither source1.txt nor source2.txt exists; exiting.'
  exit 1 # terminate the script with a nonzero exit status (failure)
fi
```

A single `if` statement can have any number of `elif` clauses, representing any number of fallback conditions.

Lastly, sometimes we want to run a command if a condition is false, without there being any corresponding command to run if the condition is true. For this we can use the built-in `!` operator, which precedes a command; when the command returns zero (success or "true"), the `!` operator changes returns a nonzero value (failure or "false"), and vice versa. For example, the following statement will copy `source.txt` to `destination.txt` unless `destination.txt` already exists:

```
#!/bin/bash

if ! [[ -e destination.txt ]] ; then
  cp source.txt destination.txt
fi
```

All those examples above are examples using the `test` expressions. Actually `if` just runs everything in `then` when the command in the statement returns 0:

```
# First build a function that simply returns the code given
returns() { return $*; }
# Then use read to prompt user to try it out, read `help read' if you have forgotten this.
read -p "Exit code:" exit
if (returns $exit)
  then echo "true, $?"
  else echo "false, $?"
fi
```

So the behavior of `if` is quite like the logical 'and' `&&` and 'or' `||` in some ways:

```
# Let's reuse the returns function.
returns() { return $*; }
read -p "Exit code:" exit

# if (        and                 ) else            fi
returns $exit && echo "true, $?" || echo "false, $?"

# The REAL equivalent, false is like `returns 1'
# Of course you can use the returns $exit instead of false.
# (returns $exit ||(echo "false, $?"; false)) && echo "true, $?"
```

Always notice that misuse of those logical operands may lead to errors. In the case above, everything was fine because plain `echo` is almost always successful.

Conditional expressions

In addition to the `-e` *`file`* condition used above, which is true if *`file`* exists, there are quite a few kinds of conditions supported by Bash's `[[ … ]]` notation. Five of the most commonly used are:

`-d` *`file`*

True if *`file`* exists and is a directory.

`-f file`
: True if *`file`* exists and is a regular file.

`string1 = string2`
: True if *`string1`* and *`string2`* are equal.

`string == pattern`
: True if *`string`* matches *`pattern`*. (*`pattern`* has the same form as a pattern in filename expansion; for example, unquoted `*` means "zero or more characters".)

`string != pattern`
: True if *`string`* does *not* match *`pattern`*.

In the last three types of tests, the value on the left is usually a variable expansion; for example, `[[ "$var" = 'value' ]]` returns a successful exit status if the variable named `var` contains the value `value`.

The above conditions just scratch the surface; there are many more conditions that examine files, a few more conditions that examine strings, several conditions for examining integer values, and a few other conditions that don't belong to any of these groups.

One common use for equality tests is to see if the first argument to a script (`$1`) is a special option. For example, consider our `if` statement above that tries to copy `source1.txt` or `source2.txt` to `destination.txt`. The above version is very "verbose": it generates a lot of output. Usually we don't want a script to generate quite so much output; but we may want users to be able to request the output, for example by passing in `--verbose` as the first argument. The following script is equivalent to the above `if` statements, but it only prints output if the first argument is `--verbose`:

```
#!/bin/bash

if [[ "$1" == --verbose ]] ; then
  verbose_mode=TRUE
  shift # remove the option from $@
else
  verbose_mode=FALSE
fi

if [[ -e source1.txt ]] ; then
  if [[ "$verbose_mode" == TRUE ]] ; then
    echo 'source1.txt exists; copying to destination.txt.'
  fi
  cp source1.txt destination.txt
elif [[ -e source2.txt ]] ; then
  if [[ "$verbose_mode" == TRUE ]] ; then
    echo 'source1.txt does not exist, but source2.txt does.'
    echo 'Copying source2.txt to destination.txt.'
  fi
  cp source2.txt destination.txt
else
  if [[ "$verbose_mode" == TRUE ]] ; then
    echo 'Neither source1.txt nor source2.txt exists; exiting.'
  fi
  exit 1 # terminate the script with a nonzero exit status (failure)
fi
```

Later, when we learn about shell functions, we will find a more compact way to express this. (In fact, even with what we already know, there is a more compact way to express this: rather than setting `$verbose_mode` to `TRUE` or `FALSE`, we can set `$echo_if_verbose_mode` to `echo` or `:`, where the colon `:` is a Bash built-in command that does nothing. We can then replace all uses of `echo` with `"$echo_if_verbose_mode"`. A command such as `"$echo_if_verbose_mode"` *`message`* would then become `echo` *`message`*, printing *`message`*, if verbose-mode is turned on, but would become `:` *`message`*, doing nothing, if verbose-mode is turned off. However, that approach might be more confusing than is really worthwhile for such a simple purpose.)

Combining conditions

To combine multiple conditions with "and" or "or", or to invert a condition with "not", we can use the general Bash notations we've already seen. Consider this example:

```
#!/bin/bash

if [[ -e source.txt ]] && ! [[ -e destination.txt ]] ; then
  # source.txt exists, destination.txt does not exist; perform the copy:
  cp source.txt destination.txt
fi
```

The test-command `[[ -e source.txt ]] && ! [[ -e destination.txt ]]` uses the `&&` and `!` operators that we saw above that work based on exit status. `[[ condition ]]` is "successful" if *`condition`* is true, which means that `[[ -e source.txt ]] && ! [[ -e destination.txt ]]` will only run `! [[ -e destination.txt ]]` if `source.txt` exists. Furthermore, `!` inverts the exit status of `[[ -e destination.txt ]]`, so that `! [[ -e destination.txt ]]` is "successful" if and only if `destination.txt` *doesn't* exist. The end result is that `[[ -e source.txt ]] && ! [[ -e destination.txt ]]` is "successful" — "true" — if and only if `source.txt` *does* exist and `destination.txt` does *not* exist.

The construction `[[ … ]]` actually has built-in internal support for these operators, such that we can also write the above this way:

```
#!/bin/bash

if [[ -e source.txt && ! -e destination.txt ]] ; then
  # source.txt exists, destination.txt does not exist; perform the copy:
  cp source.txt destination.txt
fi
```

but the general-purpose notations are often more clear; and of course, they can be used with any test-command, not just the `[[ … ]]` construction.

Notes on readability

The `if` statements in the above examples are formatted to make them easy for humans to read and understand. This is important, not only for examples in a book, but also for scripts in the real world. Specifically, the above examples follow these conventions:

- The commands within an `if` statement are indented by a consistent amount (by two spaces, as it happens). This indentation is irrelevant to Bash — it ignores whitespace at the beginning of a line — but is very important to human programmers. Without it, it is hard to see where an `if` statement begins and ends, or even to see that there is an `if` statement. Consistent indentation becomes even more important when there are `if` statements nested within `if` statements (or other control structures, of various kinds that we will see).
- The semicolon character `;` is used before `then`. This is a special operator for separating commands; it is mostly equivalent to a line-break, though there are some differences (for example, a comment always runs from `#` to the end of a line, never from `#` to `;`). We could write `then` at the beginning of a new line, and that is perfectly fine, but it's good for a single script to be consistent one way or the other; using a single, consistent appearance for ordinary constructs makes it easier to notice unusual constructs. In the real world, programmers usually put `then` at the end of the `if` or `elif` line, so we have followed that convention here.
- A newline is used after `then` and after `else`. These newlines are optional — they need not be (and cannot be) replaced with semicolons — but they promote readability by visually accentuating the structure of the `if` statement.
- Regular commands are separated by newlines, never semicolons. This is a general convention, not specific to `if` statements. Putting each command on its own line makes it easier for someone to "skim" the script and see roughly what it is doing.

These exact conventions are not particularly important, but it is good to follow consistent and readable conventions for formatting your code. When a fellow programmer looks at your code — or when you look at your code two months after writing it — inconsistent or illogical formatting can make it very difficult to understand what is going on.

Loops

Often we want to run the same sequence of commands, over and over again, with slight differences. For example, suppose that we want to take all files named `*.txt` and rename them to `*.txt.bak` ("backup"). We can use file-expansion to get the list of files named `*.txt`, but how do we use that list? There's no obvious command containing, say, `'foo.txt' 'bar.txt'`

`'baz.txt'`, that would perform the three moves we need. What we need is a `for` loop:

```
for file in *.txt  ; do
  mv "$file" "$file.bak"
done
```

The above takes the variable `file`, and assigns it in turn to each word in the expansion of `*.txt`. Each time, it runs the body of the loop. In other words, it is equivalent to the following:

```
file='foo.txt'
mv "$file" "$file.bak"
file='bar.txt'
mv "$file" "$file.bak"
file='baz.txt'
mv "$file" "$file.bak"
```

There is nothing special here about filename expansion; we can use the same approach to iterate over any other argument-list, such as the integers 1 through 20 (using brace expansion):

```
for i in {1..20} ; do
  echo "$i"
done
```

or the positional parameters `"$@"`:

```
for arg in "$@" ; do
  echo "$arg"
done
```

In fact, that specific case is so common that Bash provides the equivalent shorthand `for arg ; do`, with `in "$@"` being implied. (But it's probably better to use the explicit form anyway.)

Another kind of loop is the `while` loop. It is similar to an `if` statement, except that it loops repeatedly as long as its test-command continues to be successful. For example, suppose that we need to wait until the file `wait.txt` is deleted. One approach is to "sleep" for a few seconds, then "wake up" and see if it still exists. We can loop repeatedly in this fashion:

```
while [[ -e wait.txt ]] ; do
  sleep 3 # "sleep" for three seconds
done
```

Conversely, we can use an `until` loop to loop *until* a given command is successful; for example, the reverse of the above might be:

```
until [[ -e proceed.txt ]] ; do
  sleep 3 # "sleep" for three seconds
done
```

Of course, this is the same as combining `while` with `!`, but in some cases it may be more readable.

- Just like `if`, `while` judges `true` or `false` in the same way. Try it out yourself.

Shell functions

A *shell function* is a special type of variable that is essentially a script-within-a-script. This feature allows us to group a sequence of commands into a single named command, which is particularly useful if the sequence of commands needs to be run from many places within the script. A shell function can even consist of just a single command; this may be useful if the command is particularly

complicated, or if its significance would not immediately be obvious to a reader. (That is, shell functions can serve two purposes: they can save typing, and they can allow for more readable code by creating intuitively-named commands.) Consider the following script:

```
#!/bin/bash
# Usage:     get_password VARNAME
# Asks the user for a password; saves it as $VARNAME.
# Returns a non-zero exit status if standard input is not a terminal, or if the
# "read" command returns a non-zero exit status.
get_password() {
  if [[ -t 0 ]] ; then
    read -r -p 'Password:' -s "$1" && echo
  else
    return 1
  fi
}

get_password PASSWORD  && echo "$PASSWORD"
```

The above script creates a shell function called `get_password` that asks the user to type a password, and stores the result in a specified variable. It then runs `get_password PASSWORD` to store the password as `$PASSWORD`; and lastly, if the call to `get_password` succeeded (as determined by its exit status), the retrieved password is printed to standard output (which is obviously not a realistic use; the goal here is simply to demonstrate the behavior of `get_password`).

The function `get_password` doesn't do anything that couldn't be done without a shell function, but the result is much more readable. The function invokes the built-in command `read` (which reads a line of user input and saves it in one or more variables) with several options that most Bash programmers will not be familiar with. (The `-r` option disables a special meaning for the backslash character; the `-p` option causes a specified prompt, in this case `Password:`, to be displayed at the head of the line; and the `-s` option prevents the password from being displayed as the user types it in. Since the `-s` option also prevents the user's newline from being displayed, the `echo` command supplies a newline.) Additionally, the function uses the conditional expression `-t 0` to make sure that the script's input is coming from a terminal (a console), and not from a file or from another program that wouldn't know that a password is being requested. (This last feature is debatable; depending on the general functionality of the script, it may be better to accept a password from standard input regardless of its source, under the assumption that the source was designed with the script in mind.) The overall point is that giving sequence of commands a name — `get_password` — makes it much easier for a programmer to know what it does.

Within a shell function, the positional parameters (`$1`, `$2`, and so on, as well as `$@`, `$*`, and `$#`) refer to the arguments that the function was called with, *not* the arguments of the script that contains the function. If the latter are needed, then they need to be passed in explicitly to the function, using `"$@"`. (Even then, `shift` and `set` will only affect the positional parameters within the function, not those of the caller)

A function call returns an exit status, just like a script (or almost any command). To explicitly specify an exit status, use the `return` command, which terminates the function call and returns the specified exit status. (The `exit` command cannot be used for this, because it would terminate the entire script, just as if it were called from outside a function.) If no exit status is specified, either because no argument is given to the `return` command or because the end of the function is reached without having run a `return` command, then the function returns the exit status of the last command that was run.

Incidentally, either `function` or `( )` may be omitted from a function declaration, but at least one must be present. Instead of `function get_password ( )`, many programmers write `get_password()`. Similarly, the `{ … }` notation is not exactly required, and is not specific to functions; it is simply a notation for grouping a sequence of commands into a single compound command. The body of a function must be a compound command, such as a `{ … }` block or an `if` statement; `{ … }` is the conventional choice, even when all it contains is a single compound command and so could theoretically be dispensed with.

Subshells, environment variables, and scope

Caution:

Take your time with this section. These concepts are relatively straightforward once you understand them, but they are different in important ways from analogous concepts in other programming languages. Many programmers and systems administrators, including some who are experienced in Bash, find them counter-intuitive at first.

Subshells

In Bash, one or more commands can be wrapped in parentheses, causing those commands to be executed in a "subshell". (There are also a few ways that subshells can be created implicitly; we will see those later.) A subshell receives a copy of the surrounding context's "execution environment", which includes any variables, among other things; but any changes that the subshell makes to the execution environment are*not* copied back when the subshell completes. So, for example, this script:

```
#!/bin/bash

foo=bar
echo "$foo" # prints 'bar'

# subshell:
(
  echo "$foo" # prints 'bar' - the subshell inherits its parents' variables
  baz=bip
  echo "$baz" # prints 'bip' - the subshell can create its own variables
  foo=foo
  echo "$foo" # prints 'foo' - the subshell can modify inherited variables
)

echo "$baz" # prints nothing (just a newline) - the subshell's new variables are lost
echo "$foo" # prints 'bar' - the subshell's changes to old variables are lost
```

prints this:

```
bar
bar
bip
foo

bar
```

Tip:

If you need to call a function that modifies one or more variables, but you don't actually want those variables to be modified, you can wrap the function call in parentheses, so it takes place in a subshell. This will "isolate" the modifications and prevent them from affecting the surrounding execution environment. (That said: when possible, it's better to write functions in such a way that this problem doesn't arise to begin with. As we'll see soon, the `local` keyword can help with this.)

The same is true of function definitions; just like a regular variable, a function defined within a subshell is not visible outside the subshell.

A subshell also delimits changes to other aspects of the execution environment; in particular, the `cd` ("change directory") command only affects the subshell. So, for example, this script:

```
#!/bin/bash

cd /
pwd # prints '/'

# subshell:
(
  pwd # prints '/' - the subshell inherits the working directory
  cd home
  pwd # prints '/home' - the subshell can change the working directory
) # end of subshell

pwd # prints '/' - the subshell's changes to the working directory are lost
```

prints this:

```
/
/
/home
/
```

Tip:

If your script needs to change the working directory before running a given command, it's a good idea to use a subshell if possible. Otherwise it can become hard to keep track of the working directory when reading a script. (Alternatively, the `pushd` and `popd` built-in commands can be used to similar effect.)

An `exit` statement within a subshell terminates only that subshell. For example, this script:

```
#!/bin/bash
( exit 0 ) && echo 'subshell succeeded'
( exit 1 ) || echo 'subshell failed'
```

prints this:

```
subshell succeeded
subshell failed
```

Like in a script as a whole, `exit` defaults to returning the exit status of the last-run command, and a subshell that does not have an explicit `exit` statement will return the exit status of the last-run command.

Environment variables

We have already seen that, when a program is called, it receives a list of arguments that are explicitly listed on the command line. What we haven't mentioned is that it also receives a list of name-value pairs called "environment variables". Different programming languages offer different ways for a program to access an environment variable; C programs can use `getenv("variable_name")` (and/or accept them as a third argument to `main`), Perl programs can use `$ENV{'variable_name'}`, Java programs can use `System.getenv().get("variable_name")`, and so forth.

In Bash, environment variables are simply made into regular Bash variables. So, for example, the following script prints out the value of the `HOME` environment variable:

```
#!/bin/bash
echo "$HOME"
```

The reverse, however, is not true: regular Bash variables are not automatically made into environment variables. So, for example, this script:

```
#!/bin/bash
foo=bar
bash -c 'echo $foo'
```

will not print `bar`, because the variable `foo` is not passed into the `bash` command as an environment variable. (`bash -c` *`script arguments...`* runs the one-line Bash script *`script`*.)

To turn a regular Bash variable into an environment variable, we have to "export" it into the environment. The following script *does* print `bar`:

```
#!/bin/bash
export foo=bar
bash -c 'echo $foo'
```

Note that `export` doesn't just create an environment variable; it actually marks the Bash variable as an exported variable, and later assignments to the Bash variable will affect the environment variable as well. That effect is illustrated by this script:

```
#!/bin/bash
foo=bar
bash -c 'echo $foo'  # prints nothing
export foo
bash -c 'echo $foo'  # prints 'bar'
foo=baz
bash -c 'echo $foo'  # prints 'baz'
```

The `export` command can also be used to remove a variable from an environment, by including the `-n` option; for example, `export -n foo` undoes the effect of `export foo`. And multiple variables can be exported or unexported in a single command, such as `export foo bar` or `export -n foo bar`.

It's important to note that environment variables are only ever passed *into* a command; they are never received *back* from a command. In this respect, they are similar to regular Bash variables and subshells. So, for example, this command:

```
#!/bin/bash
export foo=bar
bash -c 'foo=baz'  # has no effect
echo "$foo" # print 'bar'
```

prints `bar`; the change to `$foo` inside the one-line script doesn't affect the process that invoked it. (However, it *would* affect any scripts that were called in turn *by* that script.)

If a given environment variable is desired for just one command, the syntax `var=value command` may be used, with the syntax of a variable assignment (or multiple variable assignments) preceding a command on the same line. (Note that, despite using the syntax of a variable assignment, this is very different from a normal Bash variable assignment, in that the variable is automatically exported into the environment, and in that it only exists for the one command. If you want avoid the confusion of similar syntax doing dissimilar things, you can use the common Unix utility `env` for the same effect. That utility also makes it possible to *remove* an environment variable for one command — or even to remove *all* environment variables for one command.) If `$var` already exists, and it's desired to include its actual value in the environment for just one command, that can be written as `var="$var" command`.

An aside: sometimes it's useful to put variable definitions — or function definitions — in one Bash script (say, `header.sh`) that can be called by another Bash script (say, `main.sh`). We can see that simply invoking that other Bash script, as `./header.sh` or as `bash ./header.sh`, will not work: the variable definitions in `header.sh` would not be seen by `main.sh`, not even if we

"exported" those definitions. (This is a common point of confusion: `export` exports variables into the environment so that other processes can see them, but they're still only seen by *child* processes, not by *parents*.) However, we can use the Bash built-in command `.` ("dot") or `source`, which runs an external file almost as though it were a shell function. If `header.sh` looks like this:

```
foo=bar
function baz ()
{
  echo "$@"
}
```

then this script:

```
#!/bin/bash
. header.sh
baz "$foo"
```

will print `'bar'`.

Scope

We have now seen some of the vagaries of variable scope in Bash. To summarize what we've seen so far:

- Regular Bash variables are scoped to the shell that contains them, including any subshells in that shell.
  - They are not visible to any child processes (that is, to external programs).
  - If they are created inside a subshell, they are not visible to the parent shell.
  - If they are modified inside a subshell, those modifications are not visible to the parent shell.
  - This is also true of functions, which in many ways are similar to regular Bash variables.
- Function-calls are not inherently run in subshells.
  - A variable modification within a function *is* generally visible to the code that calls the function.
- Bash variables that are exported into the environment are scoped to the shell that contains them, including any subshells *or child processes* in that shell.
  - The `export` built-in command can be used to export a variable into the environment. (There are other ways as well, but this is the most common way.)
  - They differ from non-exported variables only in that they are visible to child processes. In particular, they are still not visible to parent shells or parent processes.
- External Bash scripts, like other external programs, are run in child processes. The `.` or `source` built-in command can be used to run such a script internally, in which case it's not inherently run in a subshell.

To this we now add:

- Bash variables that are localized to a function-call are scoped to the function that contains them, including any functions called by that function.
  - The `local` built-in command can be used to localize one or more variables to a function-call, using the syntax `local var1 var2` or `local var1=val1 var2=val2`. (There are other ways as well — for example, the `declare` built-in command has the same effect — but this is probably the most common way.)
  - They differ from non-localized variables only in that they disappear when their function-call ends. In particular, they still *are* visible to subshells and child function-calls. Furthermore, like non-localized variables, they can be exported into the environment so as to be seen by child processes as well.

In effect, using `local` to localize a variable to a function-call is like putting the function-call in a subshell, except that it only affects the one variable; other variables can still be left non-"local".

Tip:

A variable that is set inside a function (either via assignment, or via a for-loop or other built-in command) should be marked as "local" using the built-in command `local`, so as to avoid accidentally affecting code outside the function, unless it is specifically desired that the caller see the new value.

It's important to note that, although local variables in Bash are very useful, they are not quite *as* local as local variables in most other programming languages, in that they're seen by child function-calls. For example, this script:

```
#!/bin/bash

foo=bar

function f1 ()
{
  echo "$foo"
}

function f2 ()
{
  local foo=baz
  f1 # prints 'baz'
}

f2
```

will actually print `baz` rather than `bar`. This is because the original value of `$foo` is hidden until `f2` returns. (In programming language theory, a variable like `$foo` is said to be "dynamically scoped" rather than "lexically scoped".)

One difference between `local` and a subshell is that whereas a subshell initially takes its variables from its parent shell, a statement like `local foo` immediately hides the previous value of `$foo`; that is, `$foo` becomes locally unset. If it is desired to initialize the local `$foo` to the value of the existing `$foo`, we must explicitly specify that, by using a statement like `local foo="$foo"`.

When a function exits, variables regain the values they had before their `local` declarations (or they simply become unset, if they had previously been unset). Interestingly, this means that a script such as this one:

```
#!/bin/bash

function f ()
{
  foo=baz
  local foo=bip
}

foo=bar
f
echo "$foo"
```

will actually print `baz`: the `foo=baz` statement in the function takes effect before the variable is localized, so the value `baz` is what is restored when the function returns.

And since `local` is simply an executable command, a function can decide at execution-time whether to localize a given variable, so this script:

```
#!/bin/bash

function f ()
{
  if [[ "$1" == 'yes' ]] ; then
    local foo
  fi
  foo=baz
}

foo=bar
```

```
f yes # modifies a localized $foo, so has no effect
echo "$foo" # prints 'bar'
f # modifies the non-localized $foo, setting it to 'baz'
echo "$foo" # prints 'baz'
```

will actually print

```
bar
baz
```

Pipelines and command substitution

As we have seen, a command's return value, taken strictly, is just a small non-negative integer intended to indicate success or failure. Its real output is what it writes to the standard output stream. By default, text written to the standard output stream is printed to the terminal, but there are a few ways that it can be "captured" and used as the command's true return value.

Pipelines

When a sequence of commands are linked together in a pipeline, the output of each command is passed as input to the next. This is a very powerful technique, since it lets us combine a number of small utility programs to create something complex.

Command substitution

Command substitution is a bit like variable expansion, but it runs a command and captures its output, rather than simply retrieving the value of a variable. For example, consider our `get_password` example above:

```
#!/bin/bash

function get_password ( )
# Usage:     get_password VARNAME
# Asks the user for a password; saves it as $VARNAME.
# Returns a non-zero exit status if standard input is not a terminal, or if the
# "read" command returns a non-zero exit status.
{
  if [[ -t 0 ]] ; then
    read -r -p 'Password:' -s "$1" && echo
  else
    return 1
  fi
}

get_password PASSWORD  && echo "$PASSWORD"
```

There is really no reason that the caller should have to save the password in a variable. If `get_password` simply printed the password to its standard output, then the caller could use command substitution, and use it directly:

```
#!/bin/bash

function get_password ( )
# Usage:     get_password
# Asks the user for a password; prints it for capture by calling code.
# Returns a non-zero exit status if standard input is not a terminal, or if
# standard output *is* a terminal, or if the "read" command returns a non-zero
# exit status.
{
  if [[ -t 0 ]] && ! [[ -t 1 ]] ; then
    local PASSWORD
    read -r -p 'Password:' -s PASSWORD && echo >&2
    echo "$PASSWORD"
  else
    return 1
  fi
}

echo "$(get_password)"
```

To evaluate `"$(get_password)"`, Bash runs the command `get_password` in a subshell, capturing its standard output, and replaces `$(get_password)` with the captured output.

In addition to the notation `$(…)`, an older notation `` `…` `` (using backquotes) is also supported, and still quite commonly found. The two notations have the same effect, but the syntax of `` `…` `` is more restrictive, and in complex cases it can be trickier to get right.

Command substitution allows nesting; something like `a "$(b "$(c)")"` is allowed. (It runs the command `c`, using its output as an argument to `b`, and using the output of *that* as an argument to `a`.)

A command substitution can actually contain a sequence of commands, rather than just one command. The output of all of these commands is captured. As we've seen previously, semicolons can be used instead of newlines to separate commands; that is particularly commonly done in command substitution. A command substitution can even contain variable assignments and function definitions (though, since the substituted commands run within a subshell, variable assignments and function definitions inside the command will not be seen outside it; they are only useful if they are used inside the substituted commands).

Shell arithmetic

Arithmetic expressions in Bash are closely modeled on those in C, so they are very similar to those in other C-derived languages, such as C++, Java, Perl, JavaScript, C#, and PHP. One major difference is that Bash only supports integer arithmetic (whole numbers), not floating-point arithmetic (decimals and fractions); something like `3 + 4` means what you'd expect (7), but something like `3.4 + 4.5` is a syntax error. Something like `13 / 5` is fine, but performs integer division, so evaluates to 2 rather than to 2.6.

Arithmetic expansion

Perhaps the most common way to use arithmetic expressions is in *arithmetic expansion*, where the result of an arithmetic expression is used as an argument to a command. Arithmetic expansion is denoted `$(( … ))`. For example, this command:

```
echo $(( 3 + 4 * (5 - 1) ))
```

prints `19`.

expr (deprecated)

Another way to use arithmetic expressions is using the Unix program "expr", which was popular before Bash supported math.[2] Similar to Arithmetic expansion, this command:

```
echo `expr 3 + 4 \* \( 5 - 1 \)`
```

prints `19`. Note that using "expr" requires an escape character "\" before the multiplication operator "*" and parentheses. Further note the spaces between each operator symbol, including the parentheses.

Numeric operators

In addition to the familiar notations `+` (addition) and `-` (subtraction), arithmetic expressions also support `*` (multiplication), `/` (integer division, described above), `%` (modulo division, the "remainder" operation; for example, 11 divided by 5 is 2 remainder 1, so `11 % 5` is `1`), and `**` ("exponentiation", i.e. involution; for example, $2^4 = 16$, so `2 ** 4` is `16`).

The operators `+` and `-`, in addition to their "binary" (two-operand) senses of "addition" and "subtraction", have "unary" (one-operand) senses of "positive" and "negative". Unary `+` has basically no effect; unary `-` inverts the sign of its operand. For example, `-(3*4)` evaluates to `-12`, and `-(-(3*4))` evaluates to `12`.

Referring to variables

Inside an arithmetic expression, shell variables can be referred to directly, without using variable expansion (that is, without the dollar sign `$`). For example, this:

```
i=2+3
echo $(( 7 * i ))
```

prints `35`. (Note that `i` is evaluated first, producing 5, and then it's multiplied by 7. If we had written `$i` rather than `i`, mere string substitution would have been performed; `7 * 2+3` equals 14 + 3, that is, 17 — probably not what we want.)

The previous example shown using "expr":

```
i=`expr 2 + 3`
echo `expr 7 \* $i`
```

prints `35`.

Assigning to variables

Shell variables can also be assigned to within an arithmetic expression. The notation for this is similar to that of regular variable assignment, but is *much* more flexible. For example, the previous example could be rewritten like this:

```
echo $(( 7 * (i = 2 + 3) ))
```

except that this sets `$i` to `5` rather than to `2+3`. Note that, although arithmetic expansion looks a bit like command substitution, it is *not* executed in a subshell; this command actually sets `$i` to `5`, and later commands can use the new value. (The parentheses inside the arithmetic expression are just the normal mathematical use of parentheses to control the order of operations.)

In addition to the simple assignment operator `=`, Bash also supports compound operators such as `+=`, `-=`, `*=`, `/=`, and `%=`, which perform an operation followed by an assignment. For example, `(( i *= 2 + 3 ))` is equivalent to `(( i = i * (2 + 3) ))`. In each case, the expression as a whole evaluates to the new value of the variable; for example, if `$i` is `4`, then `(( j = i *= 3 ))` sets both `$i` and `$j` to `12`.

Lastly, Bash supports increment and decrement operators. The increment operator `++` increases a variable's value by 1; if it *precedes* the variable-name (as the "pre-increment" operator), then the expression evaluates to the variable's *new* value, and if it *follows* the variable-name (as the "post-increment" operator), then the expression evaluates to the variable's *old* value. For example, if `$i` is `4`, then `(( j = ++i ))` sets both `$i` and `$j` to `5`, while `(( j = i++ ))` sets `$i` to `5` and `$j` to `4`. The decrement operator `--` is exactly the same, except that it decreases the variable's value by 1. Pre-decrement and post-decrement are completely analogous to pre-increment and post-increment.

Arithmetic expressions as their own commands

A command can consist entirely of an arithmetic expression, using either of the following syntaxes:

```
(( i = 2 + 3 ))
```

```
let 'i = 2 + 3'
```

Either of these commands will set `$i` to `5`. Both styles of command return an exit status of zero ("successful" or "true") if the expression evaluates to a non-zero value, and an exit status of one ("failure" or "false") if the expression evaluates to zero. For example, this:

```
(( 0 )) || echo zero
(( 1 )) && echo non-zero
```

will print this:

```
zero
non-zero
```

The reason for this counterintuitive behavior is that in C, zero means "false" and non-zero values (especially one) mean "true". Bash maintains that legacy inside arithmetic expressions, then translates it into the usual Bash convention at the end.

The comma operator

Arithmetic expressions can contain multiple sub-expressions separated by commas `,`. The result of the last sub-expression becomes the overall value of the full expression. For example, this:

```
echo $(( i = 2 , j = 2 + i , i * j ))
```

sets `$i` to `2`, sets `$j` to `4`, and prints `8`.

The `let` built-in actually supports multiple expressions directly without needing a comma; therefore, the following three commands are equivalent:

```
(( i = 2 , j = 2 + i , i * j ))
```

```
let 'i = 2 , j = 2 + i , i * j'
```

```
let 'i = 2' 'j = 2 + i' 'i * j'
```

Comparison, Boolean, and conditional operators

Arithmetic expressions support the integer comparison operators `<`, `>`, `<=` (meaning ≤), `>=` (meaning ≥), `==` (meaning =), and `!=` (meaning ≠). Each evaluates to `1` for "true" or `0` for "false".

They also support the Boolean operators `&&` ("and"), which evaluates to `0` if either of its operands is zero, and to `1` otherwise; `||` ("or"), which evaluates to `1` if either of its operands is nonzero, and to `0` otherwise; and `!` ("not"), which evaluates to `1` if its operand is zero, and to `0` otherwise. Aside from their use of zero to mean "false" and nonzero values to mean "true", these are just like the operators `&&`, `||`, and `!` that we've seen outside arithmetic expressions. Like those operators, these are "short-cutting" operators that do not evaluate their second argument if their first argument is enough to determine a result. For example, `(( ( i = 0 ) && ( j = 2 ) ))` will not evaluate the `( j = 2 )` part, and therefore will not set `$j` to `2`, because the left operand of `&&` is zero ("false").

And they support the conditional operator *`b ? e1 : e2`*. This operator evaluates *`e1`*, and returns its result, if *`b`* is nonzero; otherwise, it evaluates *`e2`* and returns its result.

These operators can be combined in complex ways:

```
(( i = ( ( a > b && c < d + e || f == g + h ) ? j : k ) ))
```

Arithmetic for-loops

Above, we saw one style of for-loop, that looked like this:

```
# print all integers 1 through 20:
for i in {1..20} ; do
  echo $i
done
```

Bash also supports another style, modeled on the for-loops of C and related languages, using shell arithmetic:

```
# print all integers 1 through 20:
for (( i = 1 ; i <= 20 ; ++i )) ; do
  echo $i
done
```

This for-loop uses three separate arithmetic expressions, separated by semicolons `;` (and not commas `,` — these are completely separate expressions, not just sub-expressions). The first is an initialization expression, run before the loop begins. The second is a test expression; it is evaluated before every potential loop iteration (including the first), and if it evaluates to zero ("false"), then the loop exits. The third is a counting expression; it is evaluated at the end of each loop iteration. In other words, this for-loop is exactly equivalent to this while-loop:

```
# print all integers 1 through 20:
(( i = 1 ))
while (( i <= 20 )) ; do
  echo $i
  (( ++i ))
done
```

but, once you get used to the syntax, it makes it more clear what is going on.

Bitwise operators

In addition to regular arithmetic and Boolean operators, Bash also offers "bitwise" operators, meaning operators that operate on integers *qua* bit-strings rather than *qua* integers. If you are not already familiar with this concept, you can safely ignore these.

Just as in C, the bitwise operators are `&` (bitwise "and"), `|` (bitwise "or"), `^` (bitwise "exclusive or"), `~` (bitwise "not"), `<<` (bitwise left-shift), and `>>` (bitwise right-shift), as well as `&=` and `|=` and `^=` (which include assignment, just like `+=`).

Integer literals

An integer constant is expressed as an *integer literal*. We have already seen many of these; `34`, for example, is an integer literal denoting the number 34. All of our examples have been *decimal* (base ten) integer literals, which is the default; but in fact, literals may be expressed in any base in the range 2–64, using the notation `base#value` (with the base itself being expressed in base-ten). For example, this:

```
echo $(( 12 ))        # use the default of base ten (decimal)
echo $(( 10#12 ))     # explicitly specify base ten (decimal)
echo $(( 2#1100 ))    # base two (binary)
echo $(( 8#14 ))      # base eight (octal)
echo $(( 16#C ))      # base sixteen (hexadecimal)
echo $(( 8 + 2#100 )) # eight in base ten (decimal), plus four in base two (binary)
```

will print `12` six times. (Note that this notation only affects how an integer literal is interpreted. The result of the arithmetic expansion is still expressed in base ten, regardless.)

For bases 11 through 36, the English letters A through Z are used for digit-values 10 through 35. This is not case-sensitive. For bases 37 through 64, however, it is specifically the lowercase English letters that are used for digit-values 10 through 35, with the uppercase letters being used for digit-values 36 through 61, the at-sign `@` being used for digit-value 62, and the underscore `_` being used for digit-value 63. For example, `64#@A3` denotes 256259 ($62 \times 64^2 + 36 \times 64 + 3$).

There are also two special notations: prefixing a literal with `0` indicates base-eight (octal), and prefixing it with `0x` or `0X` indicates base-sixteen (hexadecimal). For example, `030` is equivalent to `8#30`, and `0x6F` is equivalent to `16#6F`.

Integer variables

A variable may be declared as an integer variable — that is, its "integer attribute" may be "set" — by using this syntax:

```
declare -i n
```

After running the above command, any subsequent assignments to `n` will automatically cause the right-hand side to be interpreted as an arithmetic expression. For example, this:

```
declare -i n
n='2 + 3 > 4'
```

is more or less equivalent to this:

```
n=$((2 + 3 > 4))
```

except that the first version's `declare -i n` will continue to affect later assignments as well.

In the first version, note the use of quotes around the right-hand side of the assignment. Had we written `n=2 + 3 > 4`, it would have meant "run the command `+` with the argument `3`, passing in the environment variable `n` set to `2`, and redirecting standard output into the file `4`"; which is to say, setting a variable's integer attribute doesn't affect the overall parsing of assignment statements, but merely controls the interpretation of the value that is finally assigned to the variable.

We can "unset" a variable's integer attribute, turning off this behavior, by using the opposite command:

```
declare +i n
```

The `declare` built-in command has a number of other uses as well: there are a few other attributes a variable can have, and `declare` has a few other features besides turning attributes on and off. In addition, a few of its properties bear note:

- As with `local` and `export`, the argument can be a variable assignment; for example `declare -i n=2+3` sets `$n`'s integer attribute and sets it to `5`.
- As with `local` and `export`, multiple variables (and/or assignments) can be specified at once; for example, `declare -i m n` sets both `$m`'s integer attribute and `$n`'s.
- When used inside a function, `declare` implicitly localizes the variable (unless the variable is already local), which also has the effect of locally unsetting it (unless the assignment syntax is used).

Non-integer arithmetic

As mentioned above, Bash shell arithmetic only supports integer arithmetic. However, external programs can often be used to obtain similar functionality for non-integer values. In particular, the common Unix utility `bc` is often used for this. The following command:

```
echo "$(echo '3.4 + 2.2' | bc)"
```

prints `5.6`. Needless to say, since `bc` is not so tightly integrated with Bash as shell arithmetic is, it is not as convenient; for example, something like this:

```
# print the powers of two, from 1 to 512:
for (( i = 1 ; i < 1000 ; i *= 2 )) ; do
  echo $i
done
```

would, to support non-integers, become something like this:

```
# print the powers of one-half, from 1 to 1/512:
i=1
while [ $( echo "$i > 0.001" | bc ) = 1 ] ; do
  echo $i
  i=$( echo "scale = scale( $i) + 1 ; $i / 2" | bc )
done
```

Part of this is because we can no longer use an arithmetic for-loop; part of it is because referring to variables and assigning to variables is trickier now (since `bc` is not aware of the shell's variables, only its own, unrelated ones); and part of it is because `bc` communicates with the shell only via input and output.

External Programs

Bash, as a shell, is actually a 'glue' language. It helps programs to cooperate with each other, and benefits from it. Always Search The Internet for what you want -- there are lots of command line utilities available.

Using whiptail

Whiptail is a program that allows shell scripts to display dialog boxes to the user for informational purposes, or to get input from the user in a friendly way. Whiptail is included by default on Debian and various other GNU/Linux distributions.

> From the GNU/Linux Dictionary: whiptail is a "dialog" replacement using newt instead of ncurses.
> From its README: whiptail is designed to be drop-in compatible with dialog(1), but has fewer features: some dialog boxes are not implemented, such as tailbox, timebox, calendarbox, etc.

See Bash Shell Scripting/Whiptail

Using AWK

See AWK and `man awk` (`man gawk`).

Using sed

See sed and `man sed`.

Using grep

See grep and `man grep`.

Using man, info and help

These three programs are where you can find help or reference from. `man` displays roff manual pages, `info` displays texinfo documentations, while `help` displays builtin helps.

Appending `--long-help`, `--help` or `--usage` to a command-line program may also gives you the usage information. Possible synonyms include `-H` and `-h`.

Just try these out:

```
man --help
man man

info --help
man info
info info

help help
```

Pressing `h` in `man` and `info`'s interfaces can also give you some direction.

Input/Output

The read built-in

From `help read`:

```
read: read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u
fd] [name ...]

    Read a line from the standard input and split it into fields.
```

read is great for both user inputs and reading standard inputs/piping.

An example of user input:

```
# 'readline'  prompt          default    variable name
read -e -p "Do this:"  -i "destroy the ship"  command

echo "$command"
```

Or even simpler:

```
pause() { read -n 1 -p "Press any key to continue..." ; }
```

Hello-world level example of stdout operation:

```
echo 'Hello, world!'  | { read hello
echo $hello }
```

Just be creative. For example, in many ways read can replace whiptail. Here is an example, extracted from Arthur200000's shell script:

```
# USAGE
#   yes_or_no "title" "text" height width ["yes text"] ["no text"]
# INPUTS
#   $LINE = (y/n)          - If we should use line-based input style(read)
#   $_DEFAULT = (optional) - The default value for read
yes_or_no() {
  if [ "$LINE" == "y" ]; then
    echo -e "\e[1m$1\e[0m"
    echo '$2' | fold -w $4 -s
    while read -e -n 1 -i "$_DEFAULT" -p "Y for ${5:-Yes}, N for ${6:-No}[Y/N]" _yesno; do
      case $_yesno in
        [yY]|1)
          return 0
          ;;
```

```
        [nN]|0)
          return 1
          ;;
        *)
          echo -e "\e[1;31mINVALID INPUT\x21\e[0m"
      esac
  else whiptail --title  "${1:-Huh?}" --yesno "${2:-Are you sure?}" ${3:-10} ${4:-80}\
         --yes-button  "${5:-Yes}" --no-button  "$6{:-No}"; return $?
  fi
}

# USAGE
#   user_input var_name ["title"] ["prompt"] [height] [width]
# INPUTS
#   $LINE = (y/n)          - If we should use line-based input style(read)
#   $_DEFAULT = (optional) - The default value for read; defaults to nothing.
user_input(){
  if [ "$LINE" == "y" ]; then
    echo -e "\e[1m${2:-Please Enter:}\e[0m" | fold -w ${4:-80} -s
    read -e -i "${_DEFAULT}" -p "${3:->}" $1
  else
    eval "$1"=$(whiptail --title  "$2" --inputbox  "$3" 3>&1 1>&2 2>&3)
  fi
}
```

Shell redirection

In shells, redirection is used for file I/O. The most common usage of is to redirect standard streams (stdin, stdout and stderr) to accept input from another program via piping, to save program output as a file, and to suppress program output by redirecting a stream to `/dev/null`.

Piping

Index of symbols

Symbol	Explanation
`!`	• Logically negates the exit status of a pipeline. For example, if `grep YES votes.txt` returns `0`, then `! grep YES votes.txt` returns `1`, but is otherwise equivalent. • Also supported by the `[ ... ]` builtin, and inside conditional expressions. For example, if `[[ -e file.txt ]]` is true, then `[[ ! -e file.txt ]]` is false. • Also supported in arithmetic expressions. For example, if `$i` is nonzero, then `$(( ! i ))` is `0`. • See also `#!` below.
`"…"`	• Quotes an argument (or part of an argument) so that it is not split by whitespace into multiple arguments, but without preventing parameter expansion and command substitution internally • See also `$"…"` below.
`#`	• Introduces a comment (which continues to the end of the line). For example, the command `foo bar baz # bip` is equivalent to the command `foo bar baz`, because the comment `# bip` is removed. • Inside an arithmetic expression, an integer literal of the form *b*`#`*n* is interpreted in base *b*. For example, `2#110110` is binary 110110, i.e. fifty-four • See also `#!` below. • See also `$#` below.
`#!`	• (Typically "shebang" when read aloud.) Used at the very beginning of an executable script to specify the interpreter that should be used to run it. For example, if the first line of `script.pl` is `#!/usr/bin/perl`, and `script.pl` has executable permissions, then `./script.pl` is roughly equivalent to `/usr/bin/perl ./script.pl` • The first line of a Bash script is generally either `#!/bin/bash` or `#!/bin/sh`. (The former is generally considered preferable.)
`$`	• Introduces various types of expansions, notably parameter expansion (as in `$`*`var`* or `${`*`var`*`}`), command substitution (as in `$(`*`command`*`)`), and arithmetic expansion (as in `$((`*`expression`*`))`).
`$"…"`	• A variant of `"…"` (see above) that supports locale-specific translation. (Unless you're writing scripts for use in multiple languages, e.g. both English and French, you don't need to worry about this.)
`$#`	• The number of positional parameters (arguments to a script or function). For example, if a script is invoked as `script.sh a b c`, then `$#` will be `3`. Builtins that modify positional parameters, such as `shift` and `set`, affect `$#` as well.
%	The modulus operator. Returns the remainder resulting from integer division. E.g. 5%2 = 1
&	Ampersand. Commonly used to start a command in the background. E.g. firefox &
'	Single quote. Used to quote text literally
(	Open parenthesis. Used to denote the beginning of a subshell, among other things.
)	Closing parenthesis. Used to denote the "EOF" of a subshell.
*	Asterisk. Denotes multiplication. E.g. 5*2 = 10
+	Plus. Denotes addition. E.g. 5+2 = 7
,	Comma. Used for separation. E.g. ls file{1,2,3}
-	Hyphen. Denotes subtraction. E.g. 5-2 = 3
.	Full Stop.
/	Forward slash. Denotes integer division (e.g. 5/2=2) or part of a path (e.g. /home/user)
:	Colon.
;	Semicolon. Separates lines if no newline/EOL exists. E.g. echo hello; echo world

<	Open angle bracket. Used for input redirection
=	Equality sign. Used to assign variables and check equality
>	Closing angle bracket. Used for output redirection.
?	Question Mark.
@	At sign. Typically used as a variable containing all arguments passed to the environment as $@
[	Open square bracket. Used as a more visually appealing alternative to test. E.g. if [condition] ; then etc
\	Backslash. Most commonly used for escaping. E.g. rm file\ with\ a\ bunch\ of\ spaces.txt
]	Closing square bracket. Closes test enclosures
^	Caret.
_	Underscore.
`…`	• Triggers command substitution; equivalent to $(…), but is somewhat more error-prone.
{	Open curly brace. Used for specific variable expansion. E.g. (where var = "hello ") echo "${var}world" will print "hello world", echo "$varworld" will generate an error, expecting a variable called varworld.
|	Pipe. Used for redirecting input to output. Specifically, it takes the output of the command on the left hand side, runs the program on the right side, and then passes the contents of the first command's output to the second, as if it were being typed from a keyboard. 'ls -l | grep Desk' is equivalent to running "grep Desk", and then manually typing what ls -l would have output. Every press of the return key would then trigger grep until ^D is pressed to pass the EOF.
}	Closing curly brace.
~	Tilde. Typically used to refer to the home directory. Logged in as "mrwhite", cd ~ (or just cd) would go to /home/mrwhite. Logged in as another user, the same effect could be achieved with 'cd ~mrwhite'.

See also

- Images with BASH src code

Related Wikibooks

- A Quick Introduction to Unix often involves the Bash shell.
- Guide to Unix often involves the Bash shell.
- The Android/Terminal IDE uses Bash as its default command shell.
- Mac OS X Tiger/A Quick Look Under the Hood uses Bash as its default command shell.
- Shell Programming
- Bourne Shell Scripting - Bash is almost Bourne-Shell compatible.

External links

- *Bash Reference Manual* – gnu.org
- *GNU Bash Reference Manual* for Bash 3.2 – network-theory.co.uk
- "Section - What does {some strange unix command name} stand for?"
- Wooledge BashGuide
- *Advanced Bash-Scripting Guide*, by Mendel Cooper

- "Bash - GNU Project - Free Software Foundation". https://www.gnu.org/software/bash/. Retrieved 28 November 2017.
- http://www.sal.ksu.edu/faculty/tim/unix_sg/bash/math.html