Introduction

You are invited to make additions or modifications at http://mywiki.wooledge.org/BashGuide so long as you can keep them accurate. Please test any code samples you write.

All the information here is presented without any warranty or guarantee of accuracy. Use it at your own risk. When in doubt, please consult the man pages or the GNU info pages as the authoritative references.

Contents

About This Guide

This guide aims to aid people interested in learning to work with BASH. It aspires to teach good practice techniques for using BASH, and writing simple scripts.

This guide is targeted at beginning users. It assumes no advanced knowledge -- just the ability to login to a Unix-like system and open a command-line (terminal) interface. It will help if you know how to use a text editor; we will not be covering editors, nor do we endorse any particular editor choice. Familiarity with the fundamental Unix tool set, or with other programming languages or programming concepts, is not required, but those who have such knowledge may understand some of the examples more quickly.

If something is unclear to you, you are invited to report this (use http://mywiki.wooledge.org/BashGuideFeedback, or the `#bash` channel on `irc.freenode.org`) so that it may be clarified in this document for future readers.

You are invited to contribute to the development of this document by extending it or correcting invalid or incomplete information.

The primary maintainer(s) of this document:

- -- Lhunath (primary author)
- -- GreyCat

A Definition

BASH is an acronym for ***B**ourne **A**gain **Sh**ell*. It is based on the *Bourne* shell and is mostly compatible with its features.

Shells are command interpreters. They are applications that provide users with the ability to give commands to their operating system interactively, or to execute batches of commands quickly. In no way are they required for the execution of programs; they are merely a layer between system function calls and the user.

Think of a shell as a way for you to speak to your system. Your system doesn't need it for most of its work, but it is an excellent interface between you and what your system can offer. It allows you to perform basic math, run basic tests and execute applications. More importantly, it allows you to combine these operations and connect applications to each other to perform complex and automated tasks.

BASH is **not** your operating system. It is not your window manager. It is not your terminal (but it oftens runs *inside* your terminal). It does not control your mouse or keyboard. It does not configure your system, activate your screensaver, or open your files when you double-click them. It is generally not involved in launching applications from your window manager or desktop environment. It's important to understand that BASH is only an interface for you to execute statements (using BASH syntax), either at the interactive BASH prompt or via BASH scripts.

In The Manual: Introduction

Shell: A (possibly interactive) command interpreter, acting as a layer between the user and the system.
Bash: The Bourne Again Shell, a *Bourne* compatible shell.

Using Bash

Most users that think of BASH think of it as a prompt and a command line. That is BASH in *interactive mode*. BASH can also run in *non-interactive mode*, as when executing scripts. We can use scripts to automate certain logic. Scripts are basically lists of commands (just like the ones you can type on the command line), but stored in a file. When a script is executed, all these commands are (generally) executed sequentially, one after another.

We'll start with the basics in an *interactive shell*. Once you're familiar with those, you can put them together in scripts.

Important!
You should make yourself familiar with the `man` and `apropos` commands on the shell. They will be vital to your self-tutoring.

```
$ man man
$ man apropos
```

In this guide, the `$` at the beginning of a line represents your BASH prompt. Traditionally, a shell prompt either ends with `$`, `%` or `#`. If it ends with `$`, this indicates a shell that's compatible with the Bourne shell (such as a POSIX shell, or a Korn shell, or Bash). If it ends with `%`, this indicates a *C shell* (csh or tcsh); this guide does not cover C shell. If it ends with `#`, this indicates that the shell is running as the system's superuser account (`root`), and that you should be extra careful.

Your actual BASH prompt will probably be much longer than `$`. Prompts are often highly individualized.

The `man` command stands for "manual"; it opens documentation (so-called "man pages") on various topics.

You use it by running the command `man [topic]` at the BASH prompt, where `[topic]` is the name of the "page" you wish to read. Note that many of these "pages" are considerably longer than one printed page; nevertheless, the name persists. Each command (application) on your system is likely to have a man page. There are pages for other things too, such as system calls or specific configuration files. In this guide, we will only be covering commands.

Note that if you're looking for information on BASH built-ins (commands provided by BASH, not by external applications) you should look in `man bash` instead. BASH's manual is extensive and detailed. It is an excellent reference, albeit more technical than this guide.

Bash also offers a `help` command which contains brief summaries of its built-in commands (which we'll discuss in depth later on).

```
$ help
$ help read
```

In the FAQ:
Is there a list of which features were added to specific releases (versions) of Bash?

Interactive mode: A mode of operation where a prompt asks you for one command at a time.

Script: A file that contains a sequence of commands to execute one after the other.

Contents

The guide has been divided into sections, which are intended to be read roughly in the order presented. If you skip ahead to a specific section, you might find yourself missing some background information from previous sections. (Links to relevant sections are not always provided when a topic is mentioned.)

Commands and Arguments

BASH reads commands from its input (which is usually either a terminal or a file). Each line of input that it reads is treated as a *command* -- an instruction to be carried out. (There are a few advanced cases, such as commands that span multiple lines, but we won't worry about that just yet.)

Bash divides each line into *words* at each whitespace character (spaces and tabs). The first word it finds is the name of the command to be executed. All the remaining words become *arguments* to that command (options, filenames, etc.).

Assume you're in an empty directory. (If you want to try this code out, you can create and go into an empty directory called `test` by running: `mkdir test; cd test`.)

```
$ ls                # List files in the current directory (no output: no files).
$ touch a b c       # Create files 'a', 'b' and 'c'.
$ ls                # List all files again; this time the output shows 'a', 'b' and
'c'.
a  b  c
```

The `ls` command prints out the names of the files in the current directory. The first time we run `ls`, we get no output, because there are no files yet.

The # character, when it is at the beginning of a word, indicates a *comment*. Comments are ignored by the shell; they are meant only for humans to read. Everything from the # to the end of the current line is considered a comment, and ignored. If you're running these examples in your own shell, you don't have to type the comments; but even if you do, things will still work.

`touch` is an application that changes the *Last Modified* time of a file to the current time. If the filename that it's given does not exist yet, it simply creates that file, as a new and empty file. In this example, we passed three arguments. `touch` creates a file for each argument. `ls` shows us that three files have been created.

```
$ rm *              # Remove all files in the current directory.
$ ls                # List files in the current directory (no output: no files).
$ touch a   b c     # Create files 'a', 'b' and 'c'.
$ ls                # List all files again; this time the output shows 'a', 'b' and
'c'.
a  b  c
```

`rm` is an application that removes all the files that it was given. `*` is a glob. It basically means *all files in the current directory*. You will read more about globs later.

Now, did you notice that there are several spaces between `a` and `b`, and only one between `b` and `c`? Also, notice that the files that were created by `touch` are no different than the first time. You now know that the amount of whitespace between arguments does not matter. This is important to know. For example:

```
$ echo This is a test.
This is a test.
$ echo This    is    a    test.
This is a test.
```

`echo` is a command that writes its arguments to *standard output* (which in our case is the terminal). In this example, we provide the `echo` command with four arguments: 'This', 'is', 'a' and 'test.'. `echo` takes these arguments, and prints them out one by one with a space in between. In the second case, the exact same thing happens. The extra spaces make no difference. If we actually want the extra whitespace, we need to pass the sentence as one single argument. We can do this by using quotes:

```
$ echo "This    is    a    test."
This    is    a    test.
```

Quotes group everything inside them into a single argument. This argument is

'`This    is    a    test.`', properly spaced. Note that the quotes are not part of the argument; BASH removes them before handing the argument to `echo`. `echo` prints this single argument out just like it always does.

Be very careful to avoid the following:

```
$ ls                                      # There are two files in the current
directory.
The secret voice in your head.mp3  secret
$ rm The secret voice in your head.mp3        # Executes rm with 6 arguments; not 1!
rm: cannot remove `The': No such file or directory
rm: cannot remove `voice': No such file or directory
rm: cannot remove `in': No such file or directory
rm: cannot remove `your': No such file or directory
rm: cannot remove `head.mp3': No such file or directory
$ ls                                      # List files in the current directory: It
is still there.
The secret voice in your head.mp3             # But your file 'secret' is now gone!
```

You need to make sure you quote filenames properly. If you don't you'll end up deleting the wrong things! `rm` takes filenames as arguments. If your filenames have spaces and you do not quote them, BASH thinks each word is a separate argument. BASH hands each argument to `rm` separately, like individually wrapped slices of processed cheese. `rm` treats each argument as a separate file.

The above example tried to delete a file for each word in the filename of the song, instead of keeping the filename intact. That caused our file `secret` to be deleted, and our song to remain behind!

This is what we should have done:

```
$ rm "The secret voice in your head.mp3"
```

Arguments are separated from the command name and from each other by white space. This is important to remember. For example, the following is **wrong**:

```
$ [-f file]
```

You want the `[` command name to be separated from the arguments `-f`, `file` and `]`. If you do not separate `[` and `-f` from each other with whitespace, BASH will think you are trying to execute the command name `[-f` and look in `PATH` for a program named `[-f`. Additionally, the arguments `file` and `]` need to be separated by spaces. The `[` command expects its last argument to be `]`. The correct command separates all arguments with spaces:

```
$ [ -f file ]
```

(We'll cover the `[` command in more detail later. We see a lot of people who are confused by it, and think that they can omit the white space between it and its arguments, so we need to present this particular example very early.)

And of course, if your filename contains whitespace or other special characters, it should be quoted:

```
$ [ -f "my file" ]
```

NOTE: Please have a good look at Arguments, Quotes, WordSplitting and http://wiki.bashhackers.org/syntax/words if all this isn't very clear to you yet. It is important that you have a good grasp of how the shell interprets the statements you give it before you continue this guide.

> **Good Practice:**
> **You should *always* quote sentences or strings that belong together, even if it's not absolutely necessary. This will keep you alert and reduce the risk of human error in your scripts.**
> **For example, you should always quote arguments to the `echo` command.**

> **In the FAQ:**
> **I'm trying to put a command in a variable, but the complex cases always fail!**
> **How can I handle command-line arguments (options) to my script easily?**

> *Arguments*: These are the optional additional words you can specify when running commands. They are given after the command's name ('`ls -l foo`' executes `ls` with two arguments).
> *Quotes*: The two forms of quotes (`'` and `"`) are used to protect certain special characters inside them from being interpreted as special by Bash. The difference between `'` and `"` will be discussed later.

Strings

The term *string* refers to a sequence of characters which is treated as a single unit. The term is used loosely throughout this guide, as well as in almost every other programming language.

In BASH programming, almost *everything* is a string. When you type a command, the command's name is a string, and each argument is a string. Variable names are strings, and the contents of variables are strings as well. A filename is a string, and most files *contain* strings. They're everywhere!

An entire command can also be considered a string. This is not normally a useful point of view, but it illustrates the fact that *parts* of strings can sometimes be considered strings in their own right. A string which is part of a larger string is called a *substring*.

Strings do not have any intrinsic meaning. Their meaning is defined by how and where they are used.

Let's try another example. With your favorite editor, write a shopping list and save it with the file name "list", and use `cat` to show it:

```
$ cat list
shampoo
tissues
milk (skim, not whole)
```

We typed a command: `cat list`. The shell reads this command as a string, and then divides it into the substrings `cat` and `list`. As far as the shell is concerned, `list` has no meaning. It's just a string with four characters in it. `cat` receives the argument `list`, which is a string that it interprets as a filename. The string `list` has become meaningful because of *how it was used*.

The file happens to contain some text, which we see on our terminal. The entire file content, taken as a whole, is a string, but that string is not meaningful. However, if we divide the file into *lines* (and therefore treat each line as a separate string), then we see each individual *line* has meaning.

We can divide the final line into words, but these words are not meaningful by themselves. We can't buy `(skim` at the store, and we might get the wrong kind of `milk`. Dividing the lines into words is not a useful thing to do in this example. But the shell doesn't know any of this -- only *you* do!

So, when you are dealing with commands, and data, and variables -- all of which are just strings to the shell -- you have all the responsibility. You need to be sure everything that needs to be separated can be separated properly, and everything that needs to stay together stays together properly.

We'll touch on these concepts repeatedly as we continue.

Types of Commands

Bash understands several different types of commands: aliases, functions, builtins, keywords, and executables.

- **Aliases**: Aliases are a way of shortening commands. They are only used in **interactive** shells, not in **scripts**. (This is one of the *very* few differences between a script and an interactive shell.) An alias is a *name* that is mapped to a certain *string*. Whenever that *name* is used as a command name, it is replaced by the *string* before executing the command. So, instead of executing:

  ```
  $ nmap -Pn -A --osscan-limit 192.168.0.1
  ```

  You could use an alias like this:

  ```
  $ alias nmapp='nmap -Pn -A --osscan-limit'
  $ nmapp 192.168.0.1
  ```

  Aliases are limited in power; the replacement only happens in the first word. If you want more flexibility, use a function. Aliases are only useful as simple textual shortcuts.

- **Functions**: Functions in Bash are somewhat like aliases, but more powerful. Unlike aliases, they can be used in **scripts**. A function contains shell commands, very much like a small script; they can even take arguments and create local variables. When a function is called, the commands in it are executed. Functions will be covered in depth later in the guide.

- **Builtins**: Bash has some basic commands built into it, such as `cd` (change directory), `echo` (write output), and so on. You can think of them as functions that are provided already.

- **Keywords**: Keywords are quite like builtins, but the main difference is that special parsing rules apply to them. For example, `[` is a bash builtin, while `[[` is a bash keyword. They are both used for testing stuff, but since `[[` is a keyword rather than a builtin, it benefits from a few special parsing rules which make it a lot easier to use:

  ```
  $ [ a < b ]
  -bash: b: No such file or directory
  $ [[ a < b ]]
  ```

  The first example returns an error because bash tries to redirect the file `b` to the command `[ a ]` (See File Redirection). The second example actually does what you expect it to. The character `<` no longer has its special meaning of *File Redirection* operator when it's used in a `[[` command.

- **Executables**: The last kind of command that can be executed by bash is an *executable*, also called an *external command* or *application*. Executables are invoked by using a pathname. If the executable is in the current directory, use `./myprogram`. If it's in the `/usr/local/bin` directory, use `/usr/local/bin/myprogram`. To make life a little easier for you, though, Bash uses a variable that tells it where to find applications in case you just know the name of the application but not its full pathname. This variable is called `PATH`, and it is a set of directory names separated by colons -- for example, `/bin:/usr/bin`. When a command is specified in Bash without a pathname (e.g. `myprogram`, or `ls`), and it isn't an alias, function, builtin or keyword, Bash searches through the directories in `PATH`, in order from left to right, to see whether they contain an executable of the name you typed.

Tip:
You can use the `type` command to figure out the type of a command.
For example:

```
$ type rm
rm is hashed (/bin/rm)
$ type cd
cd is a shell builtin
```

In The Manual: Simple Commands

In the FAQ:
What is the difference between test, [and [[?
How can I make an alias that takes an argument?

Alias: A name that is mapped to a string. Whenever that name is used as a command, it is replaced by the string it has mapped.

Function: A name that is mapped to a set of commands. Whenever that name is used as a command, the function is called with the arguments provided after the function's name on the command line. Functions are the basic method of making new commands.

Builtin: Certain commands have been built into Bash. These are handled internally whenever they are executed on the command line (and do not create a new process).

Application: A program that can be executed by referring to its pathname (`/bin/ls`), or simply by its name if its location is in your `PATH` variable. Running an application creates a new process.

Scripts

A script is basically a sequence of commands in a file. Bash reads the file and processes the commands in order. It only moves on to the next command when the current one has ended, unless the current one has been executed asynchronously (in the background). Don't worry too much about the latter case yet -- you'll learn about how that works later on.

Virtually any example that you see in this guide can be used in a script just as well as on the command line.

Making a script is easy. Just make a new file, and put this in it at the top:

```
#!/usr/bin/env bash
```

This header (also called the *hashbang* or *shebang*) makes sure that whenever your script is executed, Bash will be used as its interpreter. The way it works, is that when the kernel executes a non-binary, it looks at the first line of the file. If the line begins with `#!`, the kernel uses the line to determine the interpreter that the code should be passed to. (There are other valid ways to do this, as well -- see below.) The `#!` must be at the very start of the file, with no spaces or blank lines before them. Your script's commands should all appear on separate lines below this.

Please do not be fooled by examples on the Internet that use `/bin/sh` as interpreter. **`sh` is not `bash`**. Even though `sh`'s syntax and `bash`'s look very much alike and even though most `bash` scripts will run in `sh`, a lot of the examples in this guide only apply to `bash` and will just break or cause unexpected behaviour in `sh`.

Also, please refrain from giving your scripts that stupid `.sh` extension. It serves no purpose, and it's completely misleading (since it's going to be a `bash` script, not an `sh` script).

And by the way, it's perfectly fine if you use *Windows* to write your scripts, but if at all possible, **avoid using *Notepad* for writing scripts**. *Microsoft Notepad* can only make files with DOS-style line-endings. That means that each line you make in notepad will be ended by two characters: a *Carriage Return* and a *Newline* character. Bash reads lines as terminated by *Newline* characters only. As a result, the *Carriage Return* character will cause you incredible headache if you don't know it's there (very weird error messages). If at all possible, *use a decent editor* like Vim, Emacs, kate, GEdit, GVIM or xemacs. If you don't, then you will need to remove the carriage returns from your scripts before running them.

Once your script file has been made, you can call it like this:

```
$ bash myscript
```

In this example, we execute `bash` and tell it to read our script. When we do this, the `#!` line is just a comment. Bash does not do anything at all with it.

Alternatively, you can give your script executable permissions. When you do this, you can actually execute the script as an application instead of calling Bash manually:

```
$ chmod +x myscript
$ ./myscript
```

When executed in this way, the `#!` line tells the operating system (OS) what interpreter to use. The OS runs `/usr/bin/env`, which in turn runs `bash`, which reads our script. BASH itself ignores the # header line, just like last time.

Some people like to keep their scripts in a personal directory. Others like to keep their scripts somewhere in the `PATH` variable. Most like to do both at once. Here's what I suggest you do:

```
$ mkdir -p "$HOME/bin"
$ echo 'PATH="$HOME/bin:$PATH"' >> "$HOME/.bashrc"
$ exec bash
```

The first command will make a directory called `bin` in your home directory. It is traditional for directories that contain commands to be named `bin`, even when those commands are scripts and not compiled ("*binary*") programs. The second command will add a line to your `.bashrc` file which adds the directory we just made to the beginning of the `PATH` variable. Every new instance of Bash will now check for executable scripts in your `bin` directory. Finally, the third line replaces our current instance of Bash with a new one, which reads the `.bashrc` file.

Changes to DotFiles (such as `.bashrc`) never have an immediate effect. You have to take some step to reread the files. In the example above, we used `exec bash` to replace the running shell. If you wanted, you could close your existing terminal and open a new one. Bash would then initialize itself again by reading `.bashrc` (and possibly other files). Or, you could just execute that line of code on the command line (`PATH="$HOME/bin:$PATH"`) or manually process your `.bashrc` file in the running shell by running `source "$HOME/.bashrc"`.

In any case, we can now put our script in our `bin` directory and execute it as a normal command (we no longer need to prepend our script's name with its path, which was the `./` part in the previous examples):

```
$ mv myscript "$HOME/bin"
$ myscript
```

Tip:
While you're defining the interpreter in your header, you might want to take the time to explain your script's function and expected arguments a little too:

```
#! /usr/bin/env bash
#
#   scriptname argument [argument] ...
#
# A short explanation of your script's purpose.
#
# Copyright [date], [name]
```

Tip:
You can use this header to specify up to one word of optional arguments that you want to pass to the interpreter. For example, the following arguments will turn on some verbose debugging:

```
#! /bin/bash -xv
```

But that requires knowing where Bash is installed, and it's not always in `/bin`. Unfortunately, you can't use this:

```
#! /usr/bin/env bash -xv
```

because that would require two words of arguments in the header line. Unix doesn't allow that. You can do this instead:

```
#! /usr/bin/env bash
set -xv
```

For more hints, see Debugging.

Header: The header of a script determines the application that will function as its interpreter (e.g. `bash`, `sh`, `perl`, ...). Colloquially, this is also called a *shebang* -- a slang term derived by combining *hash* (`#`) and *bang* (`!`). You will probably see the word *shebang* used more often than *header*, particularly since *header* has several other meanings in different contexts, while *shebang* only means one thing.

Special Characters

There are several special characters in BASH that have a non-literal meaning. When we use these characters, Bash evaluates these characters and their meaning, but usually does not pass them on to the underlying commands. These are also called *metacharacters*.

Here are a few of those special characters, and what they do:

- ***[whitespace]***: Whitespace (spaces, tabs and newlines). Bash uses whitespace to determine where words begin and end. The first word of each command is used as the command name; any additional words become arguments to that command.
- **`$`**: Expansion character. This character is used in most substitutions, including parameter expansion (variable substitution). More about this later.
- **`'text'`**: Single quotes protect the text inside from any kind of expansion by the shell and keep it from being split into multiple words or arguments. They also prevent the special meaning of all special characters inside.
- **`"text"`**: Double quotes protect the text inside from being split into multiple words or arguments, but they permit substitutions to occur. They prevent the special meaning of *most* special characters inside them -- basically, all except for the `$` and some others.
- **`#`**: Comment character. Any word beginning with # begins a *comment* that extends to the next newline. Comments are not processed by the shell.
- **`;`**: Command separator. The semicolon is used to separate multiple commands from each other if the user chooses to keep them on the same line. It's *basically* the same thing as a newline.
- **`\`**: Escape character. The backslash prevents the next character from being used in any special sort of way. This works inside double quotes, or outside of all quotes, but *not* inside single quotes.
- **`~`**: The tilde is a shortcut for your home directory. By itself, or when followed by a `/`, it is the same as `$HOME`. When followed by a username, it means *that user's* home directory. Examples:
  `cd ~john/bin; cp coolscript ~/bin`
- **`>`** or **`<`**: Redirection characters. These characters are used to modify (redirect) the input and/or output of a command. Redirections will be covered later.
- **`|`**: Pipelines allow you to send the output of one command as the input to another command.
- **`[[ expression ]]`**: Test expression. This evaluates the conditional expression as a logical statement, to determine whether it's "true" or "false".
- **`{ commands; }`** : Command grouping. The commands inside the braces are treated as though they were only one command. It is convenient for places where Bash syntax requires only one command to be present, and you don't feel a function is warranted.
- **`` `command` ``**, **`$(command)`**: Command substitution (The latter form is **highly** preferred.) Command substitution executes the inner command first, and then replaces the whole `` `...` `` or `$(...)` with that command's standard output.
- **`(command)`**: Subshell Execution. This executes the command in a new bash shell, instead of in the current one, like a safe sandbox. If the command causes side effects (like changing variables), those changes will have no effect on the current shell.
- **`((expression))`**: Arithmetic Command. Inside the parentheses, operators such as +, -, * and / are seen as mathematical operators. This can be used for assignments like `((a=b+7))` as well as tests like `if ((a < b))`. More on this later.
- **`$((expression))`**: Arithmetic Substitution. Comparable to the above, but this expression is replaced with the result of its arithmetic evaluation. Example:
  `echo "The average is $(( (a+b)/2 ))"`.

Some examples:

```
$ echo "I am $LOGNAME"
I am lhunath
$ echo 'I am $LOGNAME'
I am $LOGNAME
$ # boo
$ echo An open\ \ \ space
An open   space
$ echo "My computer is $(hostname)"
My computer is Lyndir
$ echo boo > file
$ echo $(( 5 + 5 ))
10
$ (( 5 > 0 )) && echo "Five is bigger than zero."
Five is bigger than zero.
```

In The Manual: Shell Syntax

Special Characters: Characters that have a special meaning to Bash. Usually their meaning is interpreted and then they are removed from the command before executing it.

Parameters

Parameters are a sort of named space in memory you can use to retrieve or store information. Generally speaking, they will store string data, but can also be used to store integers, indexed and associative arrays.

Parameters come in two flavors: *variables* and *special parameters*. Special parameters are read-only, pre-set by BASH, and used to communicate some type of internal status. Variables are parameters that you can create and update yourself. Variable names are bound by the following rule:

Name: A word consisting only of letters, digits and underscores, and beginning with a letter or an underscore. Also referred to as an *identifier*.

To store data in a variable, we use the following *assignment* syntax:

```
$ varname=vardata
```

This command assigns the data `vardata` to the variable by name of `varname`.

Please note that you *cannot* use spaces around the = sign in an assignment. If you write this:

```
# This is wrong!
$ varname = vardata
```

BASH will not know that you are attempting to assign something. The parser will see `varname` with no = and treat it as a command name, and then pass = and `vardata` to it as arguments.

To access the data stored in a variable, we use parameter expansion. Parameter expansion is the *substitution* of a parameter by its value, which is to say, the syntax tells bash that you want to use the contents of the variable. After that, Bash *may still perform additional manipulations on the result*. This is a very important concept to grasp correctly, because it is very much unlike the way variables are handled in other programming languages!

To illustrate what parameter expansion is, let's use this example:

```
$ foo=bar
$ echo "Foo is $foo"
```

When Bash is about to execute your code, it first changes the command by taking your parameter expansion (the `$foo`), and replacing it by the contents of `foo`, which is `bar`. The command becomes:

```
$ echo "Foo is bar"
Foo is bar
```

Now, Bash is ready to execute the command. Executing it shows us the simple sentence on screen.

It is important to understand that parameter expansion causes the `$parameter` to be **replaced** by its contents, because of the following case which relies on an understanding of the previous chapter on argument splitting:

```
$ song="My song.mp3"
$ rm $song
rm: My: No such file or directory
rm: song.mp3: No such file or directory
```

Why did this not work? Because Bash replaced your `$song` by its contents, being `My song.mp3`; then it performed word splitting; and only THEN executed the command. It was as if you had typed this:

```
$ rm My song.mp3
```

And according to the rules of word splitting, Bash thought you meant for `My` and `song.mp3` to mean two different files, because there is white space between them and it wasn't quoted. How do we fix this? We remember to **put double quotes around every parameter expansion!**

```
$ rm "$song"
```

> *Parameters*: Parameters store data that can be retrieved through a symbol or a name.

Special Parameters and Variables

Let's get our vocabulary straight before we get into the real deal. There are *Parameters* and *Variables*. Variables are actually just one kind of parameter: parameters that are denoted by a name. Parameters that aren't variables are called *Special Parameters*. I'm sure you'll understand things better with a few examples:

```
$ # Some parameters that aren't variables:
$ echo "My shell is $0, and has these options set: $-"
My shell is -bash, and has these options set: himB
$ # Some parameters that ARE variables:
$ echo "I am $LOGNAME, and I live at $HOME."
I am lhunath, and I live at /home/lhunath.
```

Please note: Unlike PHP/Perl/... parameters do NOT start with a $-sign. The $-sign you see in the examples merely causes the parameter that follows it to be *expanded*. Expansion basically means that the shell replaces the parameter by its content. As such, `LOGNAME` is the parameter (variable) that contains your username. `$LOGNAME` is an expression that will be replaced with the content of that variable, which in my case is `lhunath`.

I think you've got the drift now. Here's a summary of most of the *Special Parameters*:

Parameter Name	**Usage**	**Description**
0	`"$0"`	Contains the name, or the path, of the script. This is not always reliable.
1 2 etc.	`"$1"` etc.	*Positional Parameters* contain the arguments that were passed to the current script or function.
*	`"$*"`	Expands to all the words of all the positional parameters. Double quoted, it expands to a single string containing them all, separated by the first character of the **IFS** variable (discussed later).
@	`"$@"`	Expands to all the words of all the positional parameters. Double quoted, it expands to a list of them all as individual words.
#	`$#`	Expands to the number of positional parameters that are currently set.
?	`$?`	Expands to the exit code of the most recently completed foreground command.
$	`$$`	Expands to the PID (process ID number) of the current shell.
!	`$!`	Expands to the PID of the command most recently executed in the background.
_	`"$_"`	Expands to the last argument of the last command that was executed.

And here are a few examples of *Variables* that the shell provides for you:

- **BASH_VERSION**: Contains a string describing the version of Bash.
- **HOSTNAME**: Contains the hostname of your computer, I swear. Either short or long form, depending on how your computer is set up.
- **PPID**: Contains the PID of the parent process of this shell.
- **PWD**: Contains the current working directory.
- **RANDOM**: Each time you expand this variable, a (pseudo)random number between 0 and 32767 is generated.
- **UID**: The ID number of the current user. Not reliable for security/authentication purposes, alas.
- **COLUMNS**: The number of characters that fit on one line in your terminal. (The width of your terminal in characters.)
- **LINES**: The number of lines that fit in your terminal. (The height of your terminal in characters.)
- **HOME**: The current user's home directory.
- **PATH**: A colon-separated list of paths that will be searched to find a command, if it is not an alias, function, builtin command, or shell keyword, and no pathname is specified.
- **PS1**: Contains a string that describes the format of your shell prompt.
- **TMPDIR**: Contains the directory that is used to store temporary files (by the shell).

(There are many more -- see the manual for a comprehensive list.) Of course, you aren't restricted to only these variables. Feel free to define your own:

```
$ country=Canada
$ echo "I am $LOGNAME and I currently live in $country."
I am lhunath and I currently live in Canada.
```

Notice what we did to assign the value `Canada` to the variable `country`. Remember that you are **NOT allowed to have any spaces before or after that equals sign**!

```
$ language = PHP
-bash: language: command not found
$ language=PHP
$ echo "I'm far too used to $language."
I'm far too used to PHP.
```

Remember that Bash is not Perl or PHP. You need to be very well aware of how *expansion* works to avoid **big** trouble. If you don't, you'll end up creating very dangerous situations in your scripts, especially when making this mistake with `rm`:

```
$ ls
no secret  secret
$ file='no secret'
$ rm $file
rm: cannot remove `no': No such file or directory
```

Imagine we have two files, `no secret` and `secret`. The first contains nothing useful, but the second contains the secret that will save the world from impending doom. Unthoughtful as you are, you forgot to **quote** your parameter expansion of `file`. Bash expands the parameter and the result is `rm no secret`. Bash splits the arguments up by their whitespace as it normally does, and `rm` is passed two arguments: 'no' and 'secret'. As a result, it fails to find the file `no` and it deletes the file `secret`. *The secret is lost!*

Good Practice:
You should always keep parameter expansions properly quoted. This prevents the whitespace or the possible globs inside of them from giving you gray hair or unexpectedly wiping stuff off your computer. The only good PE, is a quoted PE.

In The Manual: Shell Parameters, Shell Variables

In the FAQ:
How can I concatenate two variables? How do I append a string to a variable?
How can I access positional parameters after $9?

Variable: A variable is a kind of parameter that you can create and modify directly. It is denoted by a name, which must begin with a letter or underscore (_), and must consist only of letters, digits, and the underscore. Variable names are case-sensitive.

Expansion: Expansion happens when a parameter is prefixed by a dollar sign. Bash takes the parameter's value and replaces the parameter's expansion by its value before executing the command. This is also called *substitution*.

Variable Types

Although Bash is not a typed language, it does have a few different types of variables. These types define the kind of content they are allowed to have. Type information is stored internally by Bash.

- **Array**: `declare -a` *variable*: The variable is an array of strings.
- **Associative array**: `declare -A` *variable*: The variable is an associative array of strings (bash 4.0 or higher).
- **Integer**: `declare -i` *variable*: The variable holds an integer. Assigning values to this variable automatically triggers *Arithmetic Evaluation*.
- **Read Only**: `declare -r` *variable*: The variable can no longer be modified or unset.
- **Export**: `declare -x` *variable*: The variable is marked for export which means it will be inherited by any child process.

Arrays are basically indexed lists of strings. They are very convenient for their ability to store multiple strings together without relying on a *delimiter* to split them apart (which is tedious when done correctly and error-prone when not).

Defining variables as integers has the advantage that you can leave out some syntax when trying to assign or modify them:

```
$ a=5; a+=2; echo $a; unset a
52
$ a=5; let a+=2; echo $a; unset a
7
$ declare -i a=5; a+=2; echo $a; unset a
7
$ a=5+2; echo $a; unset a
5+2
$ declare -i a=5+2; echo $a; unset a
7
```

However, in practice the use of `declare -i` is exceedingly rare. In large part, this is because it creates behavior that can be surprising to anyone trying to maintain the script, who misses the `declare` statement. Most experienced shell scripters prefer to use explicit arithmetic commands (`let` or `((...))`) when they want to perform arithmetic.

It is also rare to see an explicit declaration of an array using `declare -a`. It is sufficient to write `array=(...)` and Bash will know that the variable is now an array. The exception to this is the associative array, which *must* be declared explicitly: `declare -A myarray`.

String: A string is a sequence of characters.
Array: An array is a list of strings indexed by numbers.
Integer: An integer is a whole number (positive, negative or zero).
Read Only: Parameters that are read-only cannot be modified or unset.
Export: Variables that are marked for export will be inherited by any child process. Variables inherited in this way are called *Environment Variables*.

In the FAQ:
How can I use array variables?

Parameter Expansion

Parameter Expansion is the term that refers to any operation that causes a parameter to be expanded (replaced by content). In its most basic appearance, the expansion of a parameter is achieved by prefixing that parameter with a `$` sign. In certain situations, additional curly braces around the parameter's name are required:

```
$ echo "'$USER', '$USERs', '${USER}s'"
'lhunath', '', 'lhunaths'
```

This example illustrates what basic parameter expansions (PE) look like. The second PE results in an empty string. That's because the parameter `USERs` is empty. We did not intend to have the `s` be part of the parameter name. Since there's no way Bash could know you want a literal `s` appended to the parameter's value, you need to use curly braces to mark the beginning and end of the parameter name. That's what we do in the third PE in our example above.

Parameter Expansion also gives us tricks to modify the string that will be expanded. These operations can be terribly convenient:

```
$ for file in *.JPG *.jpeg
> do mv "$file" "${file%.*}.jpg"
> done
```

The code above can be used to rename all JPEG files with a `.JPG` or a `.jpeg` extension to have a normal `.jpg` extension. The expression `${file%.*}` cuts off everything from the end starting with the last period (`.`). Then, in the same quotes, a new extension is appended to the expansion result.

Here's a summary of most of the PE tricks that are available:

Syntax	**Description**
`${parameter:-word}`	**Use Default Value**. If '`parameter`' is unset or null, '`word`' (which may be an expansion) is substituted. Otherwise, the value of '`parameter`' is substituted.
`${parameter:=word}`	**Assign Default Value**. If '`parameter`' is unset or null, '`word`' (which may be an expansion) is assigned to '`parameter`'. The value of '`parameter`' is then substituted.
`${parameter:+word}`	**Use Alternate Value**. If '`parameter `' is null or unset, nothing is substituted, otherwise '`word`' (which may be an expansion) is substituted.
`${parameter:offset:length}`	**Substring Expansion**. Expands to up to '`length`' characters of '`parameter`' starting at the character specified by '`offset`' (0-indexed). If '`:length`' is omitted, go all the way to the end. If '`offset`' is negative (use parentheses!), count backward from the end of '`parameter`' instead of forward from the beginning.
`${#parameter}`	The length in characters of the value of '`parameter`' is substituted.
`${parameter#pattern}`	The '`pattern`' is matched against the **beginning** of '`parameter`'. The result is the expanded value of

	'`parameter`' with the shortest match deleted.
`${parameter##pattern}`	As above, but the *longest* match is deleted.
`${parameter%pattern}`	The '`pattern`' is matched against the **end** of '`parameter`'. The result is the expanded value of '`parameter`' with the shortest match deleted.
`${parameter%%pattern}`	As above, but the *longest* match is deleted.
`${parameter/pat/string}`	Results in the expanded value of '`parameter`' with the first (unanchored) match of '`pat`' replaced by '`string`'.
`${parameter//pat/string}`	As above, but every match of '`pat`' is replaced.

You will learn them through experience. They come in handy far more often than you think they might. Here are a few examples to kickstart you:

```
$ file="$HOME/.secrets/007"; \
> echo "File location: $file"; \
> echo "Filename: ${file##*/}"; \
> echo "Directory of file: ${file%/*}"; \
> echo "Non-secret file: ${file/secrets/not_secret}"; \
> echo; \
> echo "Other file location: ${other:-There is no other file}"; \
> echo "Using file if there is no other file: ${other:=$file}"; \
> echo "Other filename: ${other##*/}"; \
> echo "Other file location length: ${#other}"
File location: /home/lhunath/.secrets/007
Filename: 007
Directory of file: /home/lhunath/.secrets
Non-secret file: /home/lhunath/.not_secret/007

Other file location: There is no other file
Using file if there is no other file: /home/lhunath/.secrets/007
Other filename: 007
Other file location length: 26
```

Remember the difference between `${v#p}` and `${v##p}`. The doubling of the # character means patterns will become greedy. The same goes for `%`:

```
$ version=1.5.9; echo "MAJOR: ${version%%.*}, MINOR: ${version#*.}."
MAJOR: 1, MINOR: 5.9.
$ echo "Dash: ${version/./-}, Dashes: ${version//./-}."
Dash: 1-5.9, Dashes: 1-5-9.
```

Note: You cannot use multiple PEs together. If you need to execute multiple PEs on a parameter, you will need to use multiple statements:

```
$ file=$HOME/image.jpg; file=${file##*/}; echo "${file%.*}"
image
```

Good Practice:
You may be tempted to use external applications such as `sed`, `awk`, `cut`, `perl` or others to modify your strings. Be aware that all of these require an extra process to be started, which in some cases can cause slowdowns. Parameter Expansions are the perfect alternative.

In The Manual: Shell Parameter Expansion

In the FAQ:
How do I do string manipulations in bash?
How can I rename all my *.foo files to *.bar, or convert spaces to underscores, or convert upper-case file names to lower case?
How can I use parameter expansion? How can I get substrings? How can I get a file without its extension, or get just a file's extension?
How do I get the effects of those nifty Bash Parameter Expansions in older shells?
How do I determine whether a variable is already defined? Or a function?

Parameter Expansion: Any expansion (see earlier definition) of a parameter. Certain operations are possible during this expansion that are performed on the value that will be expanded.

Patterns

BASH offers three different kinds of *pattern matching*. Pattern matching serves two roles in the shell: selecting filenames within a directory, or determining whether a string conforms to a desired format.

On the command line you will mostly use *globs*. These are a fairly straight-forward form of patterns that can easily be used to match a range of files, or to check variables against simple rules.

The second type of pattern matching involves *extended globs*, which allow more complicated expressions than regular globs.

Since version `3.0`, Bash also supports *regular expression* patterns. These will be useful mainly in scripts to test user input or parse data. (You can't use a regular expression to select filenames; only globs and extended globs can do that.)

Pattern: A pattern is a string with a special format designed to match filenames, or to check, classify or validate data strings.

Glob Patterns

Globs are a very important concept in Bash, if only for their incredible convenience. Properly understanding globs will benefit you in many ways. Globs are basically patterns that can be used to match filenames or other strings.

Globs are composed of normal characters and metacharacters. Metacharacters are characters that have a special meaning. These are the metacharacters that can be used in globs:

- *****: Matches any string, including the null string.
- **?**: Matches any single character.
- **[...]**: Matches any one of the enclosed characters.

Globs are implicitly *anchored* at both ends. What this means is that a glob must match a *whole* string (filename or data string). A glob of `a*` will not match the string `cat`, because it only matches the `at`, not the whole string. A glob of `ca*`, however, would match `cat`.

Here's an example of how we can use glob patterns to expand to filenames:

```
$ ls
a  abc  b  c
$ echo *
a abc b c
$ echo a*
a abc
```

Bash sees the glob, for example `a*`. It *expands* this glob, by looking in the current directory and matching it against all files there. Any filenames that match the glob are gathered up and sorted, and then the list of filenames is used in place of the glob. As a result, the statement `echo a*` is replaced by the statement `echo a abc`, which is then executed.

When a glob is used to match *filenames*, the `*` and `?` characters cannot match a slash (`/`) character. So, for instance, the glob `*/bin` might match `foo/bin` but it cannot match `/usr/local/bin`. When globs match *patterns*, the `/` restriction is removed.

Bash performs filename expansions *after* word splitting has already been done. Therefore, filenames generated by a glob will not be split; they will always be handled correctly. For example:

```
$ touch "a b.txt"
$ ls
a b.txt
$ rm *
$ ls
```

Here, `*` is expanded into the single filename "`a b.txt`". This filename will be passed as a single argument to `rm`. Using globs to enumerate files is **always** a better idea than using `` `ls` `` for that purpose. Here's an example with some more complex syntax which we will cover later on, but it will illustrate the reason very well:

```
$ ls
a b.txt
$ for file in `ls`; do rm "$file"; done
rm: cannot remove `a': No such file or directory
rm: cannot remove `b.txt': No such file or directory
$ for file in *; do rm "$file"; done
$ ls
```

Here we use the `for` command to go through the output of the `ls` command. The `ls` command prints the string `a b.txt`. The `for` command splits that string into words over which it iterates. As a result, `for` iterates over first `a`, and then `b.txt`. Naturally, this is **not** what we want. The glob, however, expands in the proper form. It results in the string "`a b.txt`", which `for` takes as a single argument.

In addition to filename expansion, globs may also be used to check whether data matches a specific format. For example, we might be given a filename, and need to take different actions depending on its extension:

```
$ filename="somefile.jpg"
$ if [[ $filename = *.jpg ]]; then
> echo "$filename is a jpeg"
> fi
```

```
somefile.jpg is a jpeg
```

The `[[` keyword and the `case` keyword (which we will discuss in more detail later) both offer the opportunity to check a string against a glob -- either regular globs, or extended globs, if the latter have been enabled.

Good Practice:
You should always use globs instead of `ls` (or similar) to enumerate files. Globs will always expand safely and minimize the risk for bugs.
You can sometimes end up with some very weird filenames. Most scripts aren't tested against all the odd cases that they may end up being used with. Don't let your script be one of those!

In The Manual: Pattern Matching

In the FAQ:
How can I use a logical AND/OR/NOT in a shell pattern (glob)?

Glob: A glob is a string that can match certain strings or filenames.

Extended Globs

Bash also supports a feature called *Extended Globs*. These globs are more powerful in nature; technically, they are equivalent to regular expressions, although the syntax looks different than most people are used to. This feature is turned off by default, but can be turned on with the `shopt` command, which is used to toggle **sh**ell **opt**ions:

```
$ shopt -s extglob
```

- **?(list)**: Matches zero or one occurrence of the given patterns.
- ***(list)**: Matches zero or more occurrences of the given patterns.
- **+(list)**: Matches one or more occurrences of the given patterns.
- **@(list)**: Matches one of the given patterns.
- **!(list)**: Matches anything except one of the given patterns.

The list inside the parentheses is a list of globs or extended globs separated by the `|` character. Here's an example:

```
$ ls
names.txt  tokyo.jpg  california.bmp
$ echo !(*jpg|*bmp)
names.txt
```

Our extended glob expands to anything that does not match the `*jpg` or the `*bmp` pattern. Only the text file passes for that, so it is expanded.

Regular Expressions

Regular expressions (regex) are similar to *Glob Patterns*, but they can only be used for pattern matching, not for filename matching. Since 3.0, Bash supports the `=~` operator to the `[[` keyword. This operator matches the string that comes before it against the regex pattern that follows it. When the string matches the pattern, `[[` returns with an exit code of `0` ("true"). If the string does not match the pattern, an exit code of `1` ("false") is returned. In case the pattern's syntax is invalid, `[[` will abort the operation and return an exit code of `2`.

Bash uses the *Extended Regular Expression* (`ERE`) dialect. We will not cover regexes in depth in this guide, but if you are interested in this concept, please read up on RegularExpression, or Extended Regular Expressions.

Regular Expression patterns that use capturing groups (parentheses) will have their captured strings assigned to the `BASH_REMATCH` variable for later retrieval.

Let's illustrate how regex can be used in Bash:

```
$ langRegex='(..)_(..)'
$ if [[ $LANG =~ $langRegex ]]
> then
>     echo "Your country code (ISO 3166-1-alpha-2) is ${BASH_REMATCH[2]}."
>     echo "Your language code (ISO 639-1) is ${BASH_REMATCH[1]}."
> else
>     echo "Your locale was not recognised"
> fi
```

Be aware that regex parsing in Bash has changed between releases `3.1` and `3.2`. Before `3.2` it was safe to wrap your regex pattern in quotes but this has changed in `3.2`. Since then, regex should always be unquoted. You should protect any special characters by escaping it using a backslash. The best way to always be compatible is to put your regex in a variable and expand that variable in `[[` without quotes, as we showed above.

Good Practice:
Since the way regex is used in `3.2` is also valid in `3.1` we *highly* recommend you just never quote your regex. Remember to keep special characters properly escaped!

For cross-compatibility (to avoid having to escape parentheses, pipes and so on) use a variable to store your regex, e.g. `re='^\*( >| *Applying |.*\.diff|.*\.patch)'; [[ $var =~ $re ]]` This is much easier to maintain since you only write ERE syntax and avoid the need for shell-escaping, as well as being compatible with all 3.x BASH versions.

See also Chet Ramey's Bash FAQ, section E14.

In The Manual: Regex(3)

In the FAQ:
I want to check if [[$var == foo or $var == bar or $var == more ... without repeating $var n times.

Regular Expression: A regular expression is a more complex pattern that can be used to match specific strings (but unlike globs cannot expand to filenames).

Brace Expansion

Then, there is *Brace Expansion*. Brace Expansion technically does not fit in the category of patterns, but it is similar. Globs only expand to actual filenames, but brace expansions will expand to any possible permutation of their contents. Here's how they work:

```
$ echo th{e,a}n
then than
$ echo {/home/*,/root}/.*profile
/home/axxo/.bash_profile /home/lhunath/.profile /root/.bash_profile /root/.profile
$ echo {1..9}
1 2 3 4 5 6 7 8 9
$ echo {0,1}{0..9}
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
```

The brace expansion is replaced by a list of words, just like a glob is. However, these words aren't necessarily filenames, and they are not sorted (`than` would have come before `then` if they were).

Brace expansion happens *before* filename expansion. In the second `echo` command above, we used a combination of brace expansion and globs. The brace expansion goes first, and we get:

```
$ echo /home/*/.*profile /root/.*profile
```

After the brace expansion, the globs are expanded, and we get the filenames as the final result.

Brace expansions can only be used to generate lists of words. They cannot be used for pattern matching.

TestsAndConditionals

Sequential execution of commands is one thing, but to achieve any advanced logic in your scripts or your command line one-liners, you'll need tests and conditionals. Tests determine whether something is true or false. Conditionals are used to make decisions which determine the execution flow of a script.

Exit Status

Every command results in an exit code whenever it terminates. This exit code is used by whatever application started it to evaluate whether everything went OK. This exit code is like a return value from functions. It's an integer between 0 and 255 (inclusive). Convention dictates that we use 0 to denote success, and any other number to denote failure of some sort. The specific number is entirely application-specific, and is used to hint as to what exactly went wrong.

For example, the `ping` command sends ICMP packets over the network to a certain host. That host normally responds to this packet by sending the exact same one right back. This way, we can check whether we can communicate with a remote host. `ping` has a range of exit codes which can tell us what went wrong, if anything did:

From the Linux `ping` manual:

> **If ping does not receive any reply packets at all it will exit with code 1. If a packet count and deadline are both specified, and fewer than count packets are received by the time the deadline**

has arrived, it will also exit with code 1. On other error it exits with code 2. Otherwise it exits with code 0. This makes it possible to use the exit code to see if a host is alive or not.

The special parameter `?` shows us the exit code of the last foreground process that terminated. Let's play around a little with `ping` to see its exit codes:

```
$ ping God
ping: unknown host God
$ echo $?
2
$ ping -c 1 -W 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
--- 1.1.1.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ echo $?
1
```

Good Practice:
You should make sure that your scripts always return a non-zero exit code if something unexpected happened in their execution. You can do this with the `exit` builtin:

```
rm file || { echo 'Could not delete file!' >&2; exit 1; }
```

In The Manual: Exit Status

Exit Code / *Exit Status*: Whenever a command ends it notifies its parent (which in our case will always be the shell that started it) of its exit status. This is represented by a number ranging from 0 to 255. This code is a hint as to the success of the command's execution.

Control Operators (&& and ||)

Now that we know what exit codes are, and that an exit code of '0' means the command's execution was successful, we'll learn to use this information. The easiest way of performing a certain action depending on the success of a previous command is through the use of *control operators*. These operators are `&&` and `||`, which respectively represent a logical AND and a logical OR. These operators are used between two commands, and they are used to control whether the second command should be executed depending on the success of the first. This concept is called *conditional execution*.

Let's put that theory in practice:

```
$ mkdir d && cd d
```

This simple example has two commands, `mkdir d` and `cd d`. You could use a semicolon there to separate the commands and execute them sequentially; but we want something more. In the above example, BASH will execute `mkdir d`, then `&&` will check the result of the `mkdir` application after it finishes. If the `mkdir` application was successful (exit code 0), then Bash will execute the next command, `cd d`. If `mkdir d` failed, and returned a non-0 exit code, Bash will skip the next command, and we will stay in the current directory.

Another example:

```
$ rm /etc/some_file.conf || echo "I couldn't remove the file"
rm: cannot remove `/etc/some_file.conf': No such file or directory
I couldn't remove the file
```

`||` is much like `&&`, but it does the exact opposite. It only executes the next command if the first **failed**. As such, the message is only echoed if the `rm` command was unsuccessful.

In general, it's *not* a good idea to string together multiple different control operators in one command (we will explore this in the next section). `&&` and `||` are quite useful in simple cases, but not in complex ones. In the next few sections we'll show some other tools you can use for decision-making.

Good Practice:
It's best not to get overzealous when dealing with conditional operators. They can make your script hard to understand, especially for a person that's assigned to maintain it and didn't write it himself.

In The Manual: Lists of Commands

Control Operators: These operators are used to link commands together. They check the exit code of the previous command to determine whether or not to execute the next command in the sequence.

Grouping Statements

Using conditional operators is easy and terse if we want to do simple error checking. Things get a bit more dangerous, though, when we want to run multiple statements if a condition holds true, or if we need to evaluate multiple conditions.

Suppose you want to delete a file if it contains a certain "good" word but also doesn't contain another "bad" word. Using `grep` (a command that checks its input for patterns), we translate these conditions to:

```
grep -q goodword "$file"           # exit status 0 (success) if "$file" contains
'goodword'
! grep -q "badword" "$file"        # exit status 0 (success) if "$file" does not
contain 'badword'
```

We use `-q` (quiet) on grep because we don't want it to output the lines that match; we just want the exit code to be set.

The `!` in front of a command causes Bash to *negate* the command's exit status. If the command returns 0 (success), the `!` turns it into a failure. Likewise, if the command returns non-zero (failure), the `!` turns it into a success.

Now, to put these conditions together and delete the file as a result of both holding true, we could use **Conditional Operators**:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && rm "$file"
```

This works great. (In fact, we can string together as many `&&` as we want, without any problems.) Now, imagine we want to show an error message in case the deletion of the file failed:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && rm "$file" || echo "Couldn't
delete: $file" >&2
```

This looks OK, at first sight. If `rm`'s exit code is not `0` (success), then the `||` operator will trigger the next command and `echo` the error message (`>&2`: to standard error).

But there's a problem. When we have a sequence of commands separated by *Conditional Operators*, Bash looks at every one of them, in order from left to right. The exit status is carried through from whichever command was most recently executed, and skipping a command doesn't change it.

So, imagine the first `grep` fails (sets the exit status to 1). Bash sees a `&&` next, so it skips the second `grep` altogether. Then it sees another `&&`, so it also skips the `rm` which follows that one. Finally, it sees a `||` operator. Aha! The exit status is "failure", and we have a `||`, so Bash executes the `echo` command, and tells us that it couldn't delete a file -- even though it never actually *tried* to! That's not what we want.

This doesn't sound too bad when it's just a wrong error message you receive, but if you're not careful, this **will** eventually happen on more dangerous code. You wouldn't want to accidentally delete files or overwrite files as a result of a failure in your logic!

The failure in our logic is in the fact that we **want** the `rm` and the `echo` statements to belong together. The `echo` is related to the `rm`, not to the `grep`s. So what we need is to *group* them. Grouping is done using curly braces:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && { rm "$file" || echo
"Couldn't delete: $file" >&2; }
```

(Note: don't forget that you need a semicolon or newline before the closing curly brace!)

Now we've grouped the `rm` and `echo` command together. That effectively means the group is considered **one statement** instead of several. Going back to our situation of the first `grep` failing, instead of Bash trying the `&& rm "$file"` statement, it will now try the `&& { ... }` statement. Since it is preceded by a `&&` and the last command it ran failed (the failed `grep`), it will skip this group and move on.

Command grouping can be used for more things than just *Conditional Operators*. We may also want to group them so that we can redirect input to a group of statements instead of just one:

```
{
    read firstLine
    read secondLine
    while read otherLine; do
        something
    done
} < file
```

Here we're redirecting `file` to a group of commands that read input. The file will be opened when the command group starts, stay open for the duration of it, and be closed when the command group finishes. This way, we can keep sequentially reading lines from it with multiple commands.

Another common use of grouping is in simple error handling:

```
# Check if we can go into appdir.  If not, output an error and exit the script.
```

```
cd "$appdir" || { echo "Please create the appdir and try again" >&2; exit 1; }
```

Conditional Blocks (if, test and [[)

`if` is a shell keyword that executes a command (or a set of commands), and checks that command's exit code to see whether it was successful. Depending on that exit code, `if` executes a specific, different, block of commands.

```
$ if true
> then echo "It was true."
> else echo "It was false."
> fi
It was true.
```

Here you see the basic outline of an *if-statement*. We start by calling `if` with the command `true`. `true` is a builtin command that always ends successfully. `if` runs that command, and once the command is done, `if` checks the exit code. Since `true` always exits successfully, `if` continues to the `then`-block, and executes that code. Should the `true` command have failed somehow, and returned an unsuccessful exit code, the `if` statement would have skipped the `then` code, and executed the `else` code block instead.

Different people have different preferred styles for writing `if` statements. Here are some of the common styles:

```
if COMMANDS
then OTHER COMMANDS
fi

if COMMANDS
then
    OTHER COMMANDS
fi

if COMMANDS; then
    OTHER COMMANDS
fi
```

There are some commands designed specifically to *test* things and return an exit status based on what they find. The first such command is `test` (also known as `[`). A more advanced version is called `[[`. `[` or `test` is a normal command that reads its arguments and does some checks with them. `[[` is much like `[`, but it's special (a shell keyword), and it offers far more versatility. Let's get practical:

```
$ if [ a = b ]
> then echo "a is the same as b."
> else echo "a is not the same as b."
> fi
a is not the same as b.
```

`if` executes the command `[` (remember, you don't **need** an `if` to run the `[` command!) with the arguments `a`, `=`, `b` and `]`. `[` uses these arguments to determine what must be checked. In this case, it checks whether the string `a` (the first argument) is equal (the second argument) to the string `b` (the third argument), and if this is the case, it will exit successfully. However, since we know this is not the case, `[` will not exit successfully (its exit code will be 1). `if` sees that `[` terminated unsuccessfully and executes the code in the `else` block.

Now, to see why `[[` is so much more interesting and trustworthy than `[`, let us highlight some possible problems with `[`:

```
$ myname='Greg Wooledge' yourname='Someone Else'
$ [ $myname = $yourname ]
-bash: [: too many arguments
```

Can you guess what caused the problem?

`[` was executed with the arguments `Greg`, `Wooledge`, `=`, `Someone`, `Else` and `]`. That is 6 arguments, not 4! `[` doesn't understand what test it's supposed to execute, because it expects either the first or second argument to be an operator. In our case, the operator is the third argument. Yet another reason why quotes are so terribly important. Whenever we type whitespace in Bash that belongs together with the words before or after it, **we need to quote it**, and the same thing goes for parameter expansions:

```
$ [ "$myname" = "$yourname" ]
```

This time, `[` sees an operator (=) in the second argument and it can continue with its work.

To help us out a little, the Korn shell introduced (and Bash adopted) a new style of conditional test. Original as the Korn shell authors are, they called it `[[`. `[[` is loaded with several very interesting features which are missing from `[`.

One of the features of `[[` is pattern matching:

```
$ [[ $filename = *.png ]] && echo "$filename looks like a PNG file"
```

Another feature of `[[` helps us in dealing with parameter expansions:

```
$ [[ $me = $you ]]           # Fine.
$ [[ I am $me = I am $you ]] # Not fine!
-bash: conditional binary operator expected
-bash: syntax error near `am'
```

This time, `$me` and `$you` did not need quotes. Since `[[` isn't a normal command (like `[` is), but a *shell keyword*, it has special magical powers. It parses its arguments before they are expanded by Bash and does the expansion itself, taking the result as a single argument, even if that result contains whitespace. (In other words, `[[` does not allow word-splitting of its arguments.) *However*, be aware that simple strings still have to be quoted properly. `[[` can't know whether your literal whitespace in the statement is intentional or not; so it splits it up just like Bash normally would. Let's fix our last example:

```
$ [[ "I am $me" = "I am $you" ]]
```

Also; there is a subtle difference between quoting and not quoting the **right-hand side** of the comparison in `[[`. The = operator does pattern matching by default, whenever the *right-hand side* is **not** quoted:

```
$ foo=[a-z]* name=lhunath
$ [[ $name = $foo   ]] && echo "Name $name matches pattern $foo"
Name lhunath matches pattern [a-z]*
$ [[ $name = "$foo" ]] || echo "Name $name is not equal to the string $foo"
Name lhunath is not equal to the string [a-z]*
```

The first test checks whether `$name` matches the *pattern* in `$foo`. The second test checks whether `$name` is equal to the *string* in `$foo`. The quotes really do make that much difference -- a subtlety worth noting.

Remember: Always quote stuff if you are unsure. If `foo` **really** contains a pattern instead of a string (a **rare** thing to want -- you would normally write the pattern out literally: `[[ $name = [a-z]* ]]`), you will get a safe error here and you can come and fix it. If you neglect to quote, bugs can become very hard to find, since the broken code may not fail immediately.

You could also combine several `if` statements into one using `elif` instead of `else`, where each test indicates another possibility:

```
$ name=lhunath
$ if [[ $name = "George" ]]
> then echo "Bonjour, $name"
> elif [[ $name = "Hans" ]]
> then echo "Goeie dag, $name"
> elif [[ $name = "Jack" ]]
> then echo "Good day, $name"
> else
> echo "You're not George, Hans or Jack.  Who the hell are you, $name?"
> fi
```

Now that you've got a decent understanding of quoting issues that may arise, let's have a look at some of the other features that `[` and `[[` were blessed with:

- Tests supported by `[` (also known as `test`):
  - **-e FILE**: True if file exists.
  - **-f FILE**: True if file is a regular file.
  - **-d FILE**: True if file is a directory.
  - **-h FILE**: True if file is a symbolic link.
  - **-p PIPE**: True if pipe exists.
  - **-r FILE**: True if file is readable by you.
  - **-s FILE**: True if file exists and is not empty.
  - **-t FD** : True if FD is opened on a terminal.
  - **-w FILE**: True if the file is writable by you.
  - **-x FILE**: True if the file is executable by you.
  - **-O FILE**: True if the file is effectively owned by you.
  - **-G FILE**: True if the file is effectively owned by your group.
  - **FILE -nt FILE**: True if the first file is newer than the second.
  - **FILE -ot FILE**: True if the first file is older than the second.
  - **-z STRING**: True if the string is empty (it's length is zero).
  - **-n STRING**: True if the string is not empty (it's length is not zero).
  - **STRING = STRING**: True if the first string is identical to the second.
  - **STRING != STRING**: True if the first string is not identical to the second.
  - **STRING < STRING**: True if the first string sorts before the second.
  - **STRING > STRING**: True if the first string sorts after the second.
  - **EXPR -a EXPR**: True if both expressions are true (logical AND).
  - **EXPR -o EXPR**: True if either expression is true (logical OR).
  - **! EXPR**: Inverts the result of the expression (logical NOT).
  - **INT -eq INT**: True if both integers are identical.

- **INT -ne INT**: True if the integers are not identical.
- **INT -lt INT**: True if the first integer is less than the second.
- **INT -gt INT**: True if the first integer is greater than the second.
- **INT -le INT**: True if the first integer is less than or equal to the second.
- **INT -ge INT**: True if the first integer is greater than or equal to the second.

- Additional tests supported only by `[[`:
    - **STRING = (or ==) PATTERN**: Not string comparison like with `[` (or `test`), but *pattern matching* is performed. True if the string matches the glob pattern.
    - **STRING =~ REGEX**: True if the string matches the regex pattern.
    - **(EXPR)**: Parantheses can be used to change the evaluation precedence.
    - **EXPR && EXPR**: Much like the '-a' operator of `test`, but does not evaluate the second expression if the first already turns out to be false.
    - **EXPR || EXPR**: Much like the '-o' operator of `test`, but does not evaluate the second expression if the first already turns out to be true.

Some examples? Sure:

```
$ test -e /etc/X11/xorg.conf && echo 'Your Xorg is configured!'
Your Xorg is configured!
$ test -n "$HOME" && echo 'Your homedir is set!'
Your homedir is set!
$ [[ boar != bear ]] && echo "Boars aren't bears."
Boars aren't bears!
$ [[ boar != b?ar ]] && echo "Boars don't look like bears."
$ [[ $DISPLAY ]] && echo "Your DISPLAY variable is not empty, you probably have Xorg
running."
Your DISPLAY variable is not empty, you probably have Xorg running.
$ [[ ! $DISPLAY ]] && echo "Your DISPLAY variable is not not empty, you probably don't
have Xorg running."
```

Good Practice:
Whenever you're making a Bash script, you should always use `[[` rather than `[`.
Whenever you're making a Shell script, which may end up being used in an environment where Bash is not available, you should use `[`, because it is far more portable. (While being built in to Bash and some other shells, `[` should be available as an external application as well; meaning it will work as argument to, for example, find's -exec and xargs.)
Don't ever use the `-a` or `-o` tests of the `[` command. Use multiple `[` commands instead (or use `[[` if you can). POSIX doesn't define the behavior of `[` with complex sets of tests, so you never know what you'll get.

```
if [ "$food" = apple ] && [ "$drink" = tea ]; then
  echo "The meal is acceptable."
fi
```

In The Manual: Conditional Constructs

In the FAQ:
How can I group expressions, e.g. (A AND B) OR C?
What is the difference between the old and new test commands ([and [[)?

How do I determine whether a variable contains a substring?
How can I tell whether a variable contains a valid number?

if (keyword): Execute a list of commands and then, depending on their exit code, execute the code in the following `then` (or optionally `else`) block.

Conditional Loops (while, until and for)

Now you've learned how to make some basic decisions in your scripts. However, that's not enough for every kind of task we might want to script. Sometimes we need to repeat things. For that, we need to use a *loop*. There are two basic kinds of loops (plus a couple of variants), and using the correct kind of loop will help you keep your scripts readable and maintainable.

The two basic kinds of loops are called `while` and `for`. The `while` loop has a variant called `until` which simply reverses its check; and the `for` loop can appear in two different forms. Here's a summary:

- **`while`** ***command***: Repeat so long as command is executed successfully (exit code is 0).
- **`until`** ***command***: Repeat so long as command is executed unsuccessfully (exit code is not 0).
- **`for`** ***variable*** **`in`** ***words***: Repeat the loop for each word, setting *variable* to each word in turn.
- **`for ((`** ***expression*****`;`** ***expression*****`;`** ***expression*** **`))`**: Starts by evaluating the first arithmetic expression; repeats the loop so long as the second arithmetic expression is successful; and at the end of each loop evaluates the third arithmetic expression.

Each loop form is followed by the key word `do`, then one or more commands in the *body*, then the key word `done`. The `do` and `done` are similar to the `then` and `fi` (and possible `elif` and/or `else`) from the `if` statement we saw earlier. Their job is to tell us where the body of the loop begins and ends.

In practice, the loops are used for different kinds of tasks. The `for` loop (first form) is appropriate when we have a list of things, and we want to run through that list sequentially. The `while` loop is appropriate when we don't know exactly how many times we need to repeat something; we simply want it to keep going until we find what we're looking for.

Here are some examples to illustrate the differences and also the similarities between the loops. (Remember: on most operating systems, you press **Ctrl-C** to kill a program that's running on your terminal.)

```
$ while true
> do echo "Infinite loop"
> done
```

```
$ while ! ping -c 1 -W 1 1.1.1.1; do
> echo "still waiting for 1.1.1.1"
> sleep 1
> done
```

```
$ (( i=10 )); while (( i > 0 ))
> do echo "$i empty cans of beer."
> (( i-- ))
> done
$ for (( i=10; i > 0; i-- ))
> do echo "$i empty cans of beer."
> done
```

```
$ for i in {10..1}
> do echo "$i empty cans of beer."
> done
```

The last three loops achieve exactly the same result, using different syntax. You'll encounter this many times in your shell scripting experience. There will nearly always be multiple approaches to solving a problem. The test of your skill soon won't be about solving a problem as much as about how *best* to solve it. You must learn to pick the best angle of approach for the job. Usually, the main factors to take into account will be the simplicity and flexibility of the resulting code. My personal favorite is the last of the examples. In that example I used *Brace Expansion* to generate the words; but there are other ways, too.

Let's take a closer look at that last example, because although it looks the easier of the two `fors`, it can often be the trickier, if you don't know exactly how it works.

As I mentioned before: `for` runs through a list of words and puts each one in the loop index variable, one at a time, and then loops through the body with it. The tricky part is how Bash decides what the words are. Let me explain myself by expanding the braces from that previous example:

```
$ for i in 10 9 8 7 6 5 4 3 2 1
> do echo "$i empty cans of beer."
> done
```

Bash takes the characters between `in` and the end of the line, and splits them up into words. This splitting is done on spaces and tabs, just like argument splitting. However, if there are any unquoted substitutions in there, they will be word-split as well (using IFS). All these split-up words become the iteration elements.

As a result, be VERY careful not to make the following mistake:

```
$ ls
The best song in the world.mp3
$ for file in $(ls *.mp3)
> do rm "$file"
> done
rm: cannot remove `The': No such file or directory
rm: cannot remove `best': No such file or directory
rm: cannot remove `song': No such file or directory
rm: cannot remove `in': No such file or directory
rm: cannot remove `the': No such file or directory
rm: cannot remove `world.mp3': No such file or directory
```

You should already know to quote the `$file` in the `rm` statement; but what's going wrong here? Bash expands the command substitution (`$(ls *.mp3)`), replaces it by its output, and *then* performs word splitting on it (because it was unquoted). Essentially, Bash executes `for file in The best song in the world.mp3`. *Boom, you are dead.*

You want to quote it, you say? Let's add another song:

```
$ ls
The best song in the world.mp3  The worst song in the world.mp3
$ for file in "$(ls *.mp3)"
> do rm "$file"
> done
rm: cannot remove `The best song in the world.mp3  The worst song in the world.mp3': No
such file or directory
```

Quotes will indeed protect the whitespace in your filenames; but they will do more than that. The quotes will protect **all the whitespace** from the output of `ls`. There is no way Bash can know which parts of the output of `ls` represent filenames; it's not psychic. The output of `ls` is a simple string, and Bash treats it as such. The `for` puts the whole quoted output in `i` and runs the `rm` command with it. *Damn, dead again.*

So what do we do? As suggested earlier, globs are your best friend:

```
$ for file in *.mp3
> do rm "$file"
> done
```

This time, Bash **does** know that it's dealing with filenames, and it **does** know what the filenames are, and as such it can split them up nicely. The result of expanding the glob is this: `for file in "The best song in the world.mp3" "The worst song in the world.mp3"`. Problem solved!

Now let's look at the `while` loop. The `while` loop is very interesting for its capacity to execute commands until something interesting happens. Here are a few examples of how `while` loops are very often used:

```
$ # The sweet machine; hand out sweets for a cute price.
$ while read -p $'The sweet machine.\nInsert 20c and enter your name: ' name
> do echo "The machine spits out three lollipops at $name."
> done
```

```
$ # Check your email every five minutes.
$ while sleep 300
> do kmail --check
> done
```

```
$ # Wait for a host to come back online.
$ while ! ping -c 1 -W 1 "$host"
> do echo "$host is still unavailable."
> done; echo -e "$host is available again.\a"
```

The `until` loop is barely ever used, if only because it is pretty much exactly the same as `while !`. We could rewrite our last example using an `until` loop:

```
$ # Wait for a host to come back online.
$ until ping -c 1 -W 1 "$host"
> do echo "$host is still unavailable."
> done; echo -e "$host is available again.\a"
```

In practice, most people simply use `while !` instead.

Lastly, you can use the `continue` builtin to skip ahead to the next iteration of a loop without executing the rest of the body, and the `break` builtin to jump out of the loop and continue with the script after it. This works in both `for` and `while` loops.

In The Manual: Looping Constructs

In the FAQ:
How can I run a command on all files with the extention .gz?

How can I use numbers with leading zeros in a loop, e.g. 01, 02?
How can I find and deal with file names containing newlines, spaces or both?
How can I rename all my *.foo files to *.bar, or convert spaces to underscores, or convert upper-case file names to lower case?
Can I do a spinner in Bash?
I want to check to see whether a word is in a list (or an element is a member of a set).

Loop: A loop is a structure that is designed to repeat the code within until a certain condition has been fulfilled. At that point, the loop stops and the code beyond it is executed.
for (keyword): A `for`-loop is a type of loop that sets a variable to each of a list of values in turn, and repeats until the list is exhausted.
while (keyword): A `while`-loop is a type of loop that continues to run its code so long as a certain command (run before each iteration) executes successfully.
until (keyword): An `until`-loop is a type of loop that continues to run its code so long as a certain command (run before each iteration) executes unsuccessfully.

Choices (case and select)

Sometimes you want to build application logic depending on the content of a variable. This could be implemented by taking a different branch of an `if` statement depending on the results of testing against a glob:

```
shopt -s extglob

if [[ $LANG = en* ]]; then
    echo 'Hello!'
elif [[ $LANG = fr* ]]; then
    echo 'Salut!'
elif [[ $LANG = de* ]]; then
    echo 'Guten Tag!'
elif [[ $LANG = nl* ]]; then
    echo 'Hallo!'
elif [[ $LANG = it* ]]; then
    echo 'Ciao!'
elif [[ $LANG = es* ]]; then
    echo 'Hola!'
elif [[ $LANG = @(C|POSIX) ]]; then
    echo 'hello world'
else
    echo 'I do not speak your language.'
fi
```

But all these comparisons are a bit redundant. Bash provides a keyword called `case` exactly for this kind of situation. A `case` statement basically enumerates several possible *Glob Patterns* and checks the content of your parameter against these:

```
case $LANG in
    en*) echo 'Hello!' ;;
    fr*) echo 'Salut!' ;;
    de*) echo 'Guten Tag!' ;;
    nl*) echo 'Hallo!' ;;
    it*) echo 'Ciao!' ;;
    es*) echo 'Hola!' ;;
    C|POSIX) echo 'hello world' ;;
    *)   echo 'I do not speak your language.' ;;
esac
```

Each choice in a `case` statement consists of a pattern (or a list of patterns with `|` between them), a right parenthesis, a block of code that is to be executed if the string matches one of those patterns, and two semi-colons to denote the end of the block of code (since you might need to write it on several lines). `case` stops matching patterns as soon as one is successful. Therefore, we can use the `*` pattern in the end to match any case that has not been caught by the other choices.

Another construct of choice is the `select` construct. This statement smells like a loop and is a convenience statement for generating a menu of choices that the user can choose from.

The user is presented by choices and asked to enter a number reflecting his choice. The code in the `select` block is then executed with a variable set to the choice the user made. If the user's choice was invalid, the variable is made empty:

```
$ echo "Which of these does not belong in the group?"; \
> select choice in Apples Pears Crisps Lemons Kiwis; do
> if [[ $choice = Crisps ]]
> then echo "Correct!  Crisps are not fruit."; break; fi
> echo "Errr... no.  Try again."
> done
```

The menu reappears so long as the `break` statement is not executed. In the example the `break` statement is only executed when the user makes the correct choice.

We can also use the `PS3` variable to define the prompt the user replies on. Instead of showing the question before executing the `select` statement, we could choose to set the question as our prompt:

```
$ PS3="Which of these does not belong in the group (#)? "; \
> select choice in Apples Pears Crisps Lemons Kiwis; do
> if [[ $choice = Crisps ]]
> then echo "Correct!  Crisps are not fruit."; break; fi
> echo "Errr... no.  Try again."
> done
```

All of these conditional constructs (`if`, `for`, `while`, and `case`) can be *nested*. This means you could have a `for` loop with a `while` loop inside it, or any other combination, as deeply as you need to solve your problem.

```
# A simple menu:
while true; do
    echo "Welcome to the Menu"
    echo "  1. Say hello"
    echo "  2. Say good-bye"

    read -p "-> " response
    case $response in
        1) echo 'Hello there!' ;;
        2) echo 'See you later!'; break ;;
        *) echo 'What was that?' ;;
    esac
done

# Alternative: use a variable to terminate the loop instead of an
# explicit break command.

quit=
while test -z "$quit"; do
    echo "...."
    read -p "-> " response
    case $response in
        ...
        2) echo 'See you later!'; quit=y ;;
        ...
    esac
done
```

Good Practice:
A select statement makes a simple menu simple, but it doesn't offer much flexibility. If you want something more elaborate, you might prefer to write your own menu using a while loop, some echo or printf commands, and a read command.

In The Manual: Conditional Constructs

In the FAQ:
I want to check if [[$var == foo or $var == bar or $var = more ... without repeating $var n times.
How can I handle command-line arguments (options) to my script easily?

case (keyword): The `case` statement evaluates a parameter's value against several given patterns (choices).
select (keyword): The `select` statement offers the user the choice of several options and executes a block of code with the user's choice in a parameter. The menu repeats until a `break` command is executed.

Arrays

As mentioned earlier, BASH provides three types of parameters: Strings, Integers and Arrays.

Strings are without a doubt the most used parameter type. But they are also the most misused parameter type. It is important to remember that a string holds just **one** element. Capturing the output of a command, for instance, and putting it in a string parameter means that parameter holds just **one** string of characters, regardless of whether that string represents twenty filenames, twenty numbers or twenty names of people.

And as is always the case when you put multiple items in a single string, these multiple items must be somehow delimited from each other. We, as humans, can usually decipher what the different filenames are when looking at a string. We assume that, perhaps, each line in the string represents a filename, or each word represents a filename. While this assumption is understandable, it is also inherently flawed. Each single filename can contain every character you might want to use to separate the filenames from each other in a string. That means there's technically no telling where the first filename in the string ends, because there's no character that can say: "I denote the end of this filename" because that character itself could be part of the filename.

Often, people make this mistake:

```
# This does NOT work in the general case
$ files=$(ls ~/*.jpg); cp $files /backups/
```

When this would probably be a better idea (using array notation, which is explained later, in the next section):

```
# This DOES work in the general case
$ files=(~/*.jpg); cp "${files[@]}" /backups/
```

The first attempt at backing up our files in the current directory is flawed. We put the output of `ls` in a string called `files` and then use the **unquoted** `$files` parameter expansion to cut that string into arguments (relying on *Word Splitting*). As mentioned before, argument and word splitting cuts a string into pieces wherever there is whitespace. Relying on it means we assume that none of our filenames will contain any whitespace. If they do, the filename will be cut in half or more. Conclusion: **bad**.

The only safe way to represent **multiple** string elements in Bash is through the use of arrays. An array is a type of variable that **maps integers to strings**. That basically means that it holds a numbered list of strings. Since each of these strings is a separate entity (element), it can safely contain any character, even whitespace.

For the best results and the least headaches, remember that if you have a list of things, you should always put it in an array.

Unlike some other programming languages, Bash does not offer lists, tuples, etc. Just arrays, and associative arrays (which are new in Bash 4).

> *Array*: An array is a numbered list of strings: It maps integers to strings.

Creating Arrays

There are several ways you can create or fill your array with data. There is no one single true way: the method you'll need depends on where your data comes from and what it is.

The easiest way to create a simple array with data is by using the `=()` syntax:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
```

This syntax is great for creating arrays with static data or a known set of string parameters, but it gives us very little flexibility for adding lots of array elements. If you need more flexibility, you can also specify explicit indexes:

```
$ names=([0]="Bob" [1]="Peter" [20]="$USER" [21]="Big Bad John")
# or...
$ names[0]="Bob"
```

Notice that there is a gap between indices 1 and 20 in this example. An array with holes in it is called a *sparse array*. Bash allows this, and it can often be quite useful.

If you want to fill an array with filenames, then you'll probably want to use *Globs* in there:

```
$ photos=(~/"My Photos"/*.jpg)
```

Notice here that we quoted the `My Photos` part because it contains a space. If we hadn't quoted it, Bash would have split it up into `photos=('~/My' 'Photos/'*.jpg )` which is obviously **not** what we want. Also notice that we quoted **only** the part that contained the space. That's because we cannot quote the ~ or the `*`; if we do, they'll become literal and Bash won't treat them as special characters anymore.

Unfortunately, its really easy to **equivocally** create arrays with a bunch of filenames in the following way:

```
$ files=$(ls)    # BAD, BAD, BAD!
$ files=($(ls))  # STILL BAD!
```

Remember to **always avoid** using `ls.` The first would create a **string** with the output of `ls`. That string cannot possibly be used safely for reasons mentioned in the `Arrays` introduction. The second is closer, but it still splits up filenames with whitespace.

This is the right way to do it:

```
$ files=(*)      # Good!
```

This statement gives us an array where each filename is a separate element. The `*` is a glob pattern for `any string` which pathname-expands into all the filenames in the current directory (just like it would in eg. `rm *`). After the pathname expansion, the command will look like `files=([each file in the current directory that matches *])` which assigns all of the files to the array `files`. Perfect!

This section that we're about to introduce contains some advanced concepts. If you get lost, you may want to return here after you've read the whole guide. You can skip ahead to Using Arrays if you want to keep things simple.

Now, sometimes we want to build an array from a string or the output of a command. Commands (generally) just output strings: for instance, running a `find` command will enumerate filenames, and separate these filenames with newlines (putting each filename on a separate line). So to parse that one big string into an array we need to tell Bash where each element ends. (Note, this is a bad example, because filenames can **contain** a newline, so it is not safe to delimit them with newlines! But see below.)

Breaking up a string is what `IFS` is used for:

```
$ IFS=. read -a ip_elements <<< "127.0.0.1"
```

Here we use `IFS` with the value `.` to cut the given IP address into array elements wherever there's a `.`, resulting in an array with the elements `127`, `0`, `0` and `1`.

(The builtin command `read` and the `<<<` operator will be covered in more depth in the Input and Output chapter.)

We could do the same thing with a `find` command, by setting `IFS` to a newline. But then our script would fail when someone creates a filename with a newline in it (either accidentally or maliciously).

So, is there any way to get a list of elements from an external program (like `find`) into a Bash array? In general, the answer is yes, provided there is a reliable way to delimit the elements.

In the specific case of filenames, the answer to this problem is NUL bytes. A NUL byte is a byte which is just all zeros: `00000000`. Bash strings can't contain NUL bytes, because of an artifact of the "C" programming language: NUL bytes are used in C to mark the end of a string. Since Bash is written in C and uses C's native strings, it inherits that behavior.

A data stream (like the output of a command, or a file) can contain NUL bytes. Streams are like strings with three big differences: they are read sequentially (you usually can't jump around); they're *unidirectional* (you can read from them, or write to them, but typically not both); and they can contain NUL bytes.

File *names* cannot contain NUL bytes (since they're implemented as C strings by Unix), and neither can the vast majority of human-readable things we would want to store in a script (people's names, IP addresses, etc.). That makes NUL a great candidate for separating elements in a stream. Quite often, the command whose output you want to read will have an option that makes it output its data separated by NUL bytes rather than newlines or something else. `find` (on GNU and BSD, anyway) has the option `-print0`, which we'll use in this example:

```
files=()
while read -r -d ''; do
    files+=("$REPLY")
done < <(find /foo -print0)
```

This is a safe way of parsing a command's output into strings. Understandably, it looks a little confusing and convoluted at first. So let's take it apart:

The first line `files=()` creates an empty array named `files`.

We're using a while loop that runs a `read` command each time. The `read` command uses the `-d ''` option to specify the delimiter and it interprets the empty string as a NUL byte (`\0`) (as Bash arguments can not contain NULs). This means that instead of reading a line at a time (up to a newline), we're reading up to a NUL byte. It also uses `-r` to prevent it from treating backslashes specially.

Once `read` has read some data and encountered a NUL byte, the `while` loop's body is executed. We put what we read (which is in the parameter `REPLY`) into our array.

To do this, we use the `+=()` syntax. This syntax adds one or more element(s) to the end of our array.

Finally, the `< <(..)` syntax is a combination of *File Redirection* (<) and *Process Substitution* (`<(..)`). Omitting the technical details for now, we'll simply say that this is how we send the output of the `find` command into our `while` loop.

The `find` command itself uses the `-print0` option as mentioned before to tell it to separate the filenames it finds with a NUL byte.

Good Practice:
Arrays are a safe list of strings. They are perfect for storing multiple filenames.
If you have to parse a stream of data into component elements, there must be a way to tell where each element starts and ends. The NUL byte is very often the best choice for this job.
If you have a list of things, keep it in list form as long as possible. Don't smash it into a string or a file until you absolutely have to. If you do have to write it out to a file and read it back in later, keep in mind the delimiter problem we mentioned above.

In The Manual: Arrays

In the FAQ:
How can I use array variables?
How can I use variable variables (indirect variables, pointers, references) or associative arrays?
How can I find and deal with file names containing newlines, spaces or both?
I set variables in a loop. Why do they suddenly disappear after the loop terminates? Or, why can't I pipe data to read?

Using Arrays

Walking over array elements is really easy. Because an array is such a safe medium of storage, we can simply use a for loop to iterate over its elements:

```
$ for file in "${myfiles[@]}"; do
>     cp "$file" /backups/
> done
```

Notice the syntax used to **expand** the array here. We use the **quoted** form: `"${myfiles[@]}"`. Bash replaces this syntax with each element in the array properly quoted – similar to how positional parameters (arguments that were passed to the current script or function) are expanded.

The following two examples have the same effect:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ for name in "${names[@]}"; do echo "$name"; done
```

```
$ for name in "Bob" "Peter" "$USER" "Big Bad John"; do echo "$name"; done
```

The first example creates an array named `names` which is filled up with a few elements. Then the array is expanded into these elements, which are then used by the `for` loop. In the second example, we skipped the array and just passed the list of elements directly to `for`.

Remember to **quote** the `${arrayname[@]}` expansion properly. If you don't, you'll lose all benefit of having used an array at all: leaving arguments unquoted means you're telling Bash it's OK to wordsplit them into pieces and break everything again.

The above example expanded the array in a `for`-loop statement. But you can expand the array anywhere you want to put its elements as arguments; for instance in a `cp` command:

```
myfiles=(db.sql home.tbz2 etc.tbz2)
cp "${myfiles[@]}" /backups/
```

This runs the `cp` command, replacing the `"${myfiles[@]}"` part with every filename in the `myfiles` array, properly quoted. After expansion, Bash will effectively run this:

```
cp "db.sql" "home.tbz2" "etc.tbz2" /backups/
```

`cp` will then copy the files to your `/backups/` directory.

You can also expand single array elements by referencing their element number (called **index**). Remember that by default, arrays are *zero-based*, which means that their **first element** has the index **zero**:

```
$ echo "The first name is: ${names[0]}"
$ echo "The second name is: ${names[1]}"
```

(You could create an array with no element 0. Remember what we said about *sparse arrays* earlier -- you can have "holes" in the sequence of indices, and this applies to the beginning of the array as well as the middle. It's your responsibility as the programmer to know which of your arrays are potentially sparse, and which ones are not.)

There is also a second form of expanding all array elements, which is `"${arrayname[*]}"`. This form is **ONLY** useful for converting arrays into a single string with all the elements joined together. The main purpose for this is outputting the array to humans:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ echo "Today's contestants are: ${names[*]}"
Today's contestants are: Bob Peter lhunath Big Bad John
```

Notice that in the resulting string, there's no way to tell where the names begin and end! This is why we keep everything separate as long as possible.

Remember to still keep everything nicely **quoted**! If you don't keep `${arrayname[*]}` quoted, once again Bash's *Wordsplitting* will cut it into bits.

You can combine `IFS` with `"${arrayname[*]}"` to indicate the character to use to delimit your array elements as you merge them into a single string. This is handy, for example, when you want to comma delimit names:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ ( IFS=,; echo "Today's contestants are: ${names[*]}" )
Today's contestants are: Bob,Peter,lhunath,Big Bad John
```

Notice how in this example we put the `IFS=,; echo ...` statement in a Subshell by wrapping `(` and `)` around it. We do this because we don't want to change the default value of `IFS` in the main shell. When the subshell exits, `IFS` still has its default value and no longer just a comma. This is important because `IFS` is used for a lot of things, and changing its value to something non-default will result in very odd behavior if you don't expect it!

Alas, the `"${array[*]}"` expansion only uses the *first* character of `IFS` to join the elements together. If we wanted to separate the names in the previous example with a comma and a space, we would have to use some other technique (for example, a `for` loop).

The `printf` command deserves special mention here, because it's a supremely elegant way to dump an array:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ printf "%s\n" "${names[@]}"
Bob
Peter
lhunath
Big Bad John
```

Of course, a `for` loop offers the ultimate flexibility, but `printf` and its implicit looping over arguments can cover many of the simpler cases. It can even produce NUL-delimited streams, perfect for later retrieval:

```
$ printf "%s\0" "${myarray[@]}" > myfile
```

One final tip: you can get the number of elements of an array by using `${#array[@]}`

```
$ array=(a b c)
$ echo ${#array[@]}
3
```

> **Good Practice:**
> **Always quote your array expansions properly, just like you would your normal parameter expansions.**
> **Use `"${myarray[@]}"` to expand all your array elements and ONLY use `"${myarray[*]}"` when you want to merge all your array elements into a single string.**

Associative Arrays

Until recently, Bash could only use numbers (more specifically, non-negative integers) as keys of arrays. This means you could not "map" or "translate" one string to another. This is something a lot of people missed. People began to (ab)use variable indirection as a means to address the issue.

Since Bash 4 was released, there is no longer any excuse to use indirection (or **worse**, `eval`) for this purpose. You can now use full-featured associative arrays.

To create an associative array, you need to declare it as such (using `declare -A`). This is to guarantee backward compatibility with the standard indexed arrays. Here's how you do that:

```
$ declare -A fullNames
$ fullNames=( ["lhunath"]="Maarten Billemont" ["greycat"]="Greg Wooledge" )
$ echo "Current user is: $USER.  Full name: ${fullNames[$USER]}."
Current user is: lhunath.  Full name: Maarten Billemont.
```

With the same syntax as for indexed arrays, you can iterate over the keys of associative arrays:

```
$ for user in "${!fullNames[@]}"
> do echo "User: $user, full name: ${fullNames[$user]}."; done
User: lhunath, full name: Maarten Billemont.
User: greycat, full name: Greg Wooledge.
```

Two things to remember, here: First, the order of the keys you get back from an associative array using the `${!array[@]}` syntax is unpredictable; it won't necessarily be the order in which you assigned elements, or any kind of sorted order.

Second, you cannot omit the `$` if you're using a parameter as the key of an associative array. With standard indexed arrays, the `[...]` part is actually an arithmetic context (really, you can do math there without an explicit `$((...))` markup). In an arithmetic context, a *Name* can't possibly be a valid number, and so BASH assumes it's a parameter and that you want to use its content. This doesn't work with associative arrays, since a *Name* could just as well be a valid associative array key.

Let's demonstrate with examples:

```
$ indexedArray=( "one" "two" )
$ declare -A associativeArray=( ["foo"]="bar" ["alpha"]="omega" )
$ index=0 key="foo"
$ echo "${indexedArray[$index]}"
one
$ echo "${indexedArray[index]}"
one
$ echo "${indexedArray[index + 1]}"
two
$ echo "${associativeArray[$key]}"
bar
$ echo "${associativeArray[key]}"

$ echo "${associativeArray[key + 1]}"
```

As you can see, both `$index` and `index` work fine with indexed arrays. They both evaluate to `0`. You can even do math on it to increase it to `1` and get the second value. No go with associative arrays, though. Here, we need to use `$key`; the others fail.

Input and Output

Input and output in Bash scripts is a complex topic, because there is a great deal of flexibility in how it's done. This chapter will only scratch the surface of what is possible.

Input refers to any information that your program receives (or reads). Input to a Bash script can come from several different places:

- Command-line arguments (which are placed in the positional parameters)
- Environment variables, inherited from whatever process started the script
- Files
- Anything else a *File Descriptor* can point to (pipes, terminals, sockets, etc.). This will be discussed below.

Output refers to any information that your program produces (or writes). Output from a Bash script can also go to lots of different places:

- Files
- Anything else a File Descriptor can point to
- Command-line arguments to some other program
- Environment variables passed to some other program

Input and output are the bread and butter of shell script programming. Figuring out where your input comes from, what it looks like, and what you must do to it in order to produce your desired output are core requirements for almost all scripts.

Command-line Arguments

For many scripts, the first (or the only) input we will care about are the arguments received by the script on the command line. As we saw in the Parameters chapter, there are some *Special Parameters* available to every script which contain these arguments. These are called the *Positional Parameters*. They are a very simple numerically indexed array of strings (in fact, in the POSIX shell, they are the *only* array the shell has). The first positional parameter is referred to with `$1`; the second, with `$2`; and so on. After the 9th one, you must use curly braces to refer to them: `${10}`, `${11}`, etc. But in practice, it's exceedingly rare that you would ever need to do that, because there are better ways to deal with them as a group.

In addition to referring to them one at a time, you may also refer to the entire set of positional parameters with the `"$@"` substitution. The double quotes here are **extremely important**. If you don't use the double quotes, each one of the positional parameters will undergo word splitting and globbing. You don't want that. By using the quotes, you tell Bash that you want to preserve each parameter as a separate word.

Another way to deal with the positional parameters is to eliminate each one as it is used. There is a special builtin command named `shift` which is used for this purpose. When you issue the `shift` command, the first positional parameter (`$1`) goes away. The second one becomes `$1`, the third one becomes `$2`, and so on down the line. So, if you wish, you can write a loop that keeps using `$1` over and over.

In real scripts, a combination of these techniques is used. A loop to process `$1` as long as it begins with a `-` takes care of the options. Then, when all the options have been processed and shifted away, everything that's left (in `"$@"`) is presumably a filename that we want to process.

For brevity, we will not include examples of argument processing here. Instead, we will refer to the FAQ where those examples have already been written.

Good Practice:
Identify where your input comes from before you start writing. If you get to design the data flow into your script, then choose a way that makes sense for the kind of data you're dealing with. If you need to pass filenames, passing them as arguments is an excellent approach, because each one is encapsulated as a word, ready to go.

In the FAQ:
How can I handle command-line arguments (options) to my script easily?

The Environment

Every program inherits certain information, resources, privileges and restrictions from its parent process. (For a more advanced discussion of this topic, see process management.) One of those resources is a set of variables called *Environment Variables*.

In Bash, environment variables work very much like the regular shell variables we're used to. The only real difference is that they're already set when the script starts up; we don't have to set them ourselves.

Traditionally, environment variables have names that are all capital letters, such as `PATH` or `HOME`. This helps you avoid creating any variables that would conflict with them; as long as your variables all contain at least one lower-case letter, you should never have to worry about accidentally colliding with the environment. (Bash's special variables are also capitalized, such as `PIPESTATUS`. This is done for the exact same reason -- so you can avoid having your variables trampled by Bash.)

Passing information to a program through the environment is useful in many situations. One of those is user preference. Not every user on a Unix-like system has the same likes and dislikes in applications, and in some cases, they may not all speak the same language. So, it's useful for users to be able to tell *every* application they run what their favorite editor is (the `EDITOR` environment variable), or what language they speak (the various environment variables that compose the user's locale). Environment variables can be set in each user's DotFiles, and then they will be passed automatically to every program the user runs from the login session.

Environment variables can also be tweaked on the fly extremely easily (more easily than if the same information were stored in a file). When you run a command in Bash, you have the option of specifying a temporary environment change which only takes effect for the duration of that command. This is done by putting `VAR=value` in front of the command. Here is an example:

```
$ ls /tpm
ls: no se puede acceder a /tpm: No existe el fichero o el directorio
$ LANG=C ls /tpm
ls: cannot access /tpm: No such file or directory
```

The `LANG=C` temporary environment will not cause the user's locale to change for anything other than the one command where it was typed.

In a script, if you know that some information is in an environment variable, you can just use it like any other variable:

```
if [[ $DISPLAY ]]; then
    xterm -e top
else
    top
fi
```

This runs `xterm -e top` if the environment variable `DISPLAY` is set (and not empty); otherwise, it runs `top`.

If you want to put information into the environment for your child processes to inherit, you use the `export` command:

```
export MYVAR=something
```

The tricky part here is that your environment changes are only inherited by your **descendants**. You can't change the environment of a program that is already running, or of a program that you don't run.

Changing the environment and then running some other program is extremely common. A script that does this as its primary task is called a WrapperScript.

Good Practice:
Don't use all-capital variable names in your scripts, unless they are environment variables. Use lower-case or mixed-case variable names, to avoid accidents.

In the FAQ:
I'm trying to write a script that will change directory (or set a variable), but after the script finishes, I'm back where I started (or my variable isn't set)!

File Descriptors

File Descriptors (in short: FDs) are the way programs refer to files, or to other resources that work like files (such as pipes, devices, sockets, or terminals). FDs are kind of like pointers to sources of data, or places data can be written. When something reads from or writes to that FD, the data is read from or written to that FD's resource.

By default, every new process starts with three open FDs:

- *Standard Input* (`stdin`): File Descriptor 0
- *Standard Output* (`stdout`): File Descriptor 1
- *Standard Error* (`stderr`): File Descriptor 2

In an interactive shell, or in a script running on a terminal, the *Standard Input* is how bash sees the characters you type on your keyboard. The *Standard Output* is where the program sends most of its normal information so that the user can see it, and the *Standard Error* is where the program sends its error messages.

GUI applications also have these FDs, but they don't normally work with them. Usually, they do all their user interaction via the GUI, making it hard for BASH to control them. As a result, we'll stick to simple terminal applications. With those, we can easily feed data to them on their Standard Input, and read data from them on their Standard Output and Standard Error.

Let's make these definitions a little more concrete. Here's a demonstration of how Standard Input and Standard Output work:

```
$ read -p "What is your name? " name; echo "Good day, $name.  Would you like some tea?"
What is your name? lhunath
Good day, lhunath.  Would you like some tea?
```

`read` is a command that reads information from `stdin` and stores it in a variable. We specified `name` to be that variable. Once `read` has read a line of information from `stdin`, it finishes and lets `echo` display a message. `echo` sends its output to `stdout`. `stdin` and `stdout` are connected to your terminal. When a program reads from a terminal, it receives keystrokes from your keyboard; when it writes to a terminal, characters are displayed on your monitor. As a result, you can type in your name and are then greeted with a friendly message on your monitor, offering you a cup of tea.

So what is `stderr`? Let's demonstrate:

```
$ rm secrets
rm: cannot remove `secrets': No such file or directory
```

Unless you have a file called `secrets` in your current directory, that `rm` command will fail and show an error message explaining what went wrong. Error messages like these are by convention displayed on `stderr`.

`stderr` is also connected to your terminal's output device, just like `stdout`. As a result, error messages display on your monitor just like the messages on `stdout`. However, the distinction between `stdout` and `stderr` makes it easy to keep errors separated from the application's normal messages. For example, a script might wish to log `stderr` messages in a special place for long-term storage. Some people also like to use wrappers to make all the output on `stderr` red, so that they can see the error messages more clearly. (This doesn't work as well as one might wish, but some people find it good enough for some tasks.)

In shell scripts, FDs are always referenced by number. In the next section, we will see some of the ways we can work with FDs using their numbers.

In the Manual: The read builtin

Good Practice:
Remember that when you create scripts, you should send your custom error messages to the `stderr` FD. This is a convention and it is very convenient when applications follow the convention. As such, so should you! You're about to learn redirection soon, but let me show you quickly how it's done:

```
echo "Uh oh.  Something went really bad.." >&2
```

File Descriptor: A numeric index referring to one of a process's open files. Each command has at least three basic descriptors: FD 0 is `stdin`, FD 1 is `stdout` and FD 2 is `stderr`.

Redirection

The most basic form of input/output manipulation in BASH is *Redirection*. *Redirection* is used to change the data source or destination of a program's FDs. That way, you can send output to a file instead of the terminal, or have an application read from a file instead of from the keyboard.

Redirections are performed by BASH (or any other shell), *before* the shell runs the command to which the redirections are applied.

In The Manual: Redirections

Redirection: the practice of changing a FD to read its input from, or send its output to, a different location.

File Redirection

File Redirection involves changing a single FD to point to a file. Let's start with an output redirection:

```
$ echo "It was a dark and stormy night.  Too dark to write." > story
$ cat story
It was a dark and stormy night.  Too dark to write.
```

The > operator begins an *output redirection*. The redirection applies only to one command (in this case, an `echo` command). It tells BASH that when BASH runs the command, `stdout` should point to a file, rather than wherever it was pointing before.

As a result, the `echo` command will not send its output to the terminal; rather, the `> story` redirection **changes the destination of the `stdout` FD** so that it now points to a file called `story`. Be aware that this redirection occurs before the `echo` command is executed. By default, Bash doesn't check to see whether that file `story` exists first; it just opens the file, and if there was already a file by that name, its former contents are lost. If the file doesn't exist, it is created as an empty file, so that the FD can be pointed to it.

This behaviour can be toggled with *Shell Options* (see later).

It should be noted that this redirection is in effect only for the single `echo` command it was applied to. Other commands executed after that will continue sending their output to the script's `stdout` location.

We then use the application `cat` to print out the contents of that file. `cat` is an application that reads the contents of all the files you pass it as arguments. It then writes each file one after another on `stdout`. In essence, it con**cat**enates the contents of all the files you pass it as arguments.

Warning: Far too many code examples and shell tutorials on the Internet tell you to use `cat` whenever you need to read the contents of a file. **This is not necessary!** `cat` only serves well to concatenate multiple files together, or as a quick tool on the shell prompt to see what's inside a file. You should **NOT** use `cat` to pipe files to commands in your scripts. Instead, you should use a redirection. Please keep this warning in mind. Useless use of `cat` will result in an extra process to create, and using a pipe instead of a redirection takes away an application's ability to skip back and forth inside the input file.

When we use `cat` without passing any kind of arguments, it obviously doesn't know what files to read. In this case, `cat` will just read from `stdin` instead of from a file (much like `read`). Since `stdin` is normally not a regular file, starting `cat` without any arguments will seem to do nothing:

```
$ cat
```

It doesn't even give you back your shell prompt! What's going on? `cat` is still reading from `stdin`, which is your terminal. Anything you type on your keyboard now will be sent to `cat` as soon as you hit the *Enter* key. With each line you type, `cat` will do what it normally does: display it reads on `stdout`, the same way as when it displayed our story on `stdout`.

```
$ cat
test?
test?
```

Why does it say `test?` twice now? First of all, terminals are actually more complicated than they appear; they have different modes of operation. The mode we are using in this example is called *canonical mode*, and in this mode, the terminal shows you each character as you type it, and lets you perform extremely simple editing (such as using the Backspace key) on your input. The stuff you type is not actually sent to the application until you press Enter.

As you type `test?`, you will see it echoed on the screen by the terminal itself. Once you press Enter, the whole line becomes available to the application (`cat`) that's reading from the terminal. `cat` reads the line from `stdin`, and then shows it on `stdout`, which is also your terminal; hence, the second line: `test?`.

You can press *Ctrl+D* to send your terminal the *End of File* character. That'll cause `cat` to think `stdin` has closed. It will stop reading, and terminate. BASH will see that `cat` has terminated, and return you to your prompt.

Now let's use an *input redirection* to attach a file to `stdin`, so that `stdin` is no longer reading from our keyboard, but instead, now reads from the file:

```
$ cat < story
The story of William Tell.

It was a cold december night.  Too cold to write.
```

The result of this is exactly the same as the result from our previous `cat story`; except this time, the way it works is a little different. In our first example, `cat` opened an FD to the file `story` and read its contents through that FD. In the second example, `cat` simply reads from `stdin`, just like it did when it was reading from our keyboard. However, this time, the `< story` operation has **modified** `cat`'s `stdin` so that its data source is the file `story` rather than our keyboard.

Redirection operators can be preceded by a number. That number denotes the FD that will be changed.

Let's summarize with some examples:

- **`command > file`**: Send the `stdout` of command to `file`.
- **`command 1> file`**: Send the `stdout` of command to `file`. Since `stdout` is FD 1, that's the number we put in front of the redirection operator. This is identical to the previous example, because FD 1 is the default for the **>** operator.
- **`command < file`**: Use the contents of `file` when `command` reads from `stdin`.
- **`command 0< file`**: Use the contents of `file` when `command` reads from `stdin`, exactly as in the previous example, since FD 0 (`stdin`) is the default for the **<** operator.

The number for the `stderr` FD is 2. So, let's try sending `stderr` to a file:

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2> errors
```

In this example, we're looping over each directory (or file) in `/home`. We then try to delete the file `secret` in each of them. Some `homedir`s may not have a secret, or we may not have permission to remove it. As a result, the `rm` operation will fail and send an error message on `stderr`.

You may have noticed that our redirection operator isn't on `rm`, but it's on that `done` thing. Why is that? Well, this way, the redirection applies to all output to `stderr` made inside the whole loop. Technically, what happens is BASH opens the file named `errors` and points `stderr` to it before the loop begins, then closes it when the loop ends. Any commands run inside the loop (such as `rm`) *inherit* the open FD from BASH.

Let's see what the result of our loop was:

```
$ cat errors
rm: cannot remove `/home/axxo/secret': No such file or directory
rm: cannot remove `/home/lhunath/secret': No such file or directory
```

Two error messages in our error log file. Two people that didn't have a `secret` file in their home directory.

If you're writing a script, and you expect that running a certain command may fail on occasion, but don't want the script's user to be bothered by the possible error messages that command may produce, you can silence an FD. Silencing it is as easy as normal *File Redirection*. We're just going to send all output to that FD into the system's black hole:

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2> /dev/null
```

The file `/dev/null` is **always** empty, no matter what you write to it or read from it. As such, when we write our error messages to it, they just disappear. The `/dev/null` file remains as empty as ever before.

That's because it's not a normal file; it's a *virtual* device. Some people call `/dev/null` the *bit bucket*.

There is one last thing you should learn about *File Redirection*. It's interesting that you can make error log files like this to keep your error messages; but as I mentioned before, Bash destroys the existing contents of a file when it redirects to it. As a result, each time we run our loop to delete secret files, our log file will be truncated empty before we fill it up again with new error messages. What if we'd like to keep a record of any error messages generated by our loop? What if we don't want that file to be truncated each time we start our loop? The solution is achieved by doubling the redirection operator. > becomes >>. >> will not empty a file; it will just append new data to the end of it!

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2>> errors
```

Hooray!

By the way, the space between the redirection operator and the filename is optional. Some people write `> file` and some write `>file`. Both ways are correct.

Good Practice:
It's a good idea to use redirection whenever an application needs file data and is built to read data from `stdin`. A lot of bad examples on the Internet tell you to pipe (see later) the output of `cat` into processes; but this is nothing more than a very *bad* idea.
When designing an application that could be fed data from a variety of different sources, it is often best simply to have your application read from `stdin`; that way, the user can use redirection to feed it whatever data she wishes. An application that reads standard input in a generalized way is called a *filter*.

File Descriptor Manipulation

Now that you know how to manipulate process input and output by sending it to and reading it from files, let's make it a little more interesting still.

It's possible to change the source and desination of FDs to point to or from files, as you know. It's also possible to copy one FD to another. Let's prepare a simple testbed:

```
$ echo "I am a proud sentence." > file
```

We've made a file called `file`, and written a proud sentence into it.

There's an application called `grep` that we've seen briefly in a previous chapter. `grep` is like duct tape: you can use it in almost any project (whether it's a good idea or not). It basically takes a *search pattern* as its first argument and maybe some filenames as extra arguments. Just like `cat`, `grep` also uses `stdin` if you don't specify any files. `grep` reads the files (or `stdin` if none were provided) and searches for the *search pattern* you gave it. Most versions of `grep` even support a `-r` switch, which makes it take directories as well as files as extra arguments, and then searches all the files and directories in those directories that you gave it. Here's an example of how `grep` can work:

```
$ ls house/
drawer  closet  dustbin  sofa
$ grep -r socks house/
```

```
house/sofa:socks
```

In this silly example we have a directory called `house` with several pieces of furniture in it as files. If we're looking for our `socks` in each of those files, we send `grep` to search the directory `house/`. `grep` will search everything in there, open each file and look through its contents. In our example, `grep` finds `socks` in the file `house/sofa`; presumably tucked away under a pillow. You want a more realistic example? Sure:

```
$ grep "$HOSTNAME" /etc/*
/etc/hosts:127.0.0.1       localhost Lyndir
```

Here we instruct `grep` to search for whatever `$HOSTNAME` expands to in whatever files `/etc/*` expands to. It finds my hostname, which is `Lyndir`, in the file `/etc/hosts`, and shows me the line in that file that contains the *search pattern*.

OK, now that you understand `grep`, let's continue with our *File Descriptor Manipulation*. Remember that we created a file called `file`, and wrote a proud sentence to it? Let's use `grep` to find where that proud sentence is now:

```
$ grep proud *
file:I am a proud sentence.
```

Good! `grep` found our sentence in `file`. It writes the result of its operation to `stdout` which is shown on our terminal. Now let's see if we can make `grep` send an error message, too:

```
$ grep proud file 'not a file'
file:I am a proud sentence.
grep: not a file: No such file or directory
```

This time, we instruct `grep` to search for the string `proud` in the files '`file`' and '`not a file`'. `file` exists, and the sentence is in there, so `grep` happily writes the result to `stdout`. It moves on to the next file to scan, which is '`not a file`'. `grep` can't open this file to read its content, because it doesn't exist. As a result, `grep` emits an error message on `stderr` which is still connected to our terminal.

Now, how would you go about silencing this `grep` statement completely? We'd like to send all the output that appears on the terminal to a file instead; let's call it `proud.log`:

```
# Not quite right....
$ grep proud file 'not a file' > proud.log 2> proud.log
```

Does that look about right? We first use > to send `stdout` to `proud.log`, and then use 2> to send `stderr` to `proud.log` as well. Almost, but not quite. If you run this command (at least on some computers), and then look in `proud.log`, you'll see there's only an error message, not the output from `stdout`. We've created a very bad condition here. We've created two FDs that both point to the same file, independently of each other. The results of this are not well-defined. Depending on how the operating system handles FDs, some information written via one FD may clobber information written through the other FD.

```
$ echo "I am a very proud sentence with a lot of words in it, all for you." > file2
$ grep proud file2 'not a file' > proud.log 2> proud.log
$ cat proud.log
grep: not a file: No such file or directory
of words in it, all for you.
```

What happened here? `grep` opened `file2` first, found what we told it to look for, and then wrote our very proud sentence to `stdout` (FD 1). FD 1 pointed to `proud.log`, so the information was written to that file. However, we also had another FD (FD 2) pointed to this same file, and specifically, pointed to the *beginning* of this file. When `grep` tried to open '`not a file`' to read it, it couldn't. Then, it wrote an error message to `stderr` (FD 2), which was pointing to the beginning of `proud.log`. As a result, the second write operation overwrote information from the first one!

We need to prevent having two independent FDs working on the same destination or source. We can do this by *duplicating* FDs:

```
$ grep proud file 'not a file' > proud.log 2>&1
```

In order to understand these, you need to remember: always read file redirections from left to right. This is the order in which Bash processes them. First, `stdout` is changed so that it points to our `proud.log`. Then, we use the `>&` syntax to duplicate FD 1 and put this duplicate in FD 2.

A duplicate FD works differently from having two independent FDs pointing to the same place. Write operations that go through either one of them are exactly the same. There won't be a mix-up with one FD pointing to the start of the file while the other has already moved on.

Be careful not to confuse the order:

```
$ grep proud file 'not a file' 2>&1 > proud.log
```

This will duplicate `stderr` to where `stdout` points (which is the terminal), and then `stdout` will be redirected to `proud.log`. As a result, `stdout`'s messages will be logged, but the error messages will still go to the terminal. *Oops*.

Note:
For convenience, Bash also makes yet another form of redirection available to you. The `&>` redirection operator is actually just a shorter version of what we did here; redirecting both `stdout` and `stderr` to a file :

```
$ grep proud file 'not a file' &> proud.log
```

This is the same as `> proud.log 2>&1`, but not portable to BourneShell. It is not recommended practice, but you should recognize it if you see it used in someone else's scripts.

TODO: Moving FDs and Opening FDs RW.

In the FAQ:
How can I redirect the output of multiple commands at once?
How can I redirect the output of 'time' to a variable or file?
How do I use dialog to get input from the user?
How can I redirect stderr to a pipe?
Tell me all about 2>&1 -- what's the difference between 2>&1 >foo and >foo 2>&1, and when do I use which?

Heredocs And Herestrings

Sometimes storing data in a file is overkill. We might only have a tiny bit of it -- enough to fit conveniently in the script itself. Or we might want to redirect the contents of a variable into a command, without having to write it to a file first.

```
$ grep proud <<END
> I am a proud sentence.
> END
I am a proud sentence.
```

This is a *Heredoc* (or *Here Document*). *Heredoc*s are useful if you're trying to embed short blocks of multi-line data inside your script. (Embedding larger blocks is bad practice. You should keep your logic (your code) and your input (your data) separated, preferably in different files, unless it's a small data set.)

In a *Heredoc*, we choose a word to act as a sentinel. It can be any word; we used `END` in this example. Choose one that won't appear in your data set. All the lines that follow the *first* instance of the sentinel, up to the *second* instance, become the `stdin` for the command. The second instance of the sentinel word has to be a line all by itself.

There are a few different options with *Heredocs*. Normally, you can't indent them -- any spaces you use for indenting your script will appear in the `stdin`. The terminator string (in our case `END`) must be at the beginning of the line.

```
echo "Let's test abc:"
if [[ abc = a* ]]; then
    cat <<END
        abc seems to start with an a!
END
fi
```

Will result in:

```
Let's test abc:
        abc seems to start with an a!
```

You can avoid this by temporarily removing the indentation for the lines of your *Heredoc*s. However, that distorts your pretty and consistent indentation. There is an alternative. If you use `<<-END` instead of `<<END` as your *Heredoc* operator, Bash removes any `tab` characters in the beginning of each line of your *Heredoc* content before sending it to the command. That way you can still use tabs (but not spaces) to indent your *Heredoc* content with the rest of your code. Those tabs will not be sent to the command that receives your *Heredoc*. You can also use tabs to indent your sentinel string.

Bash substitutions are performed on the contents of the *Heredoc* by default. However, if you quote the word that you're using to delimit your *Heredoc*, Bash won't perform any substitutions on the contents. Try this example with and without the quote characters, to see the difference:

```
$ cat <<'XYZ'
> My home directory is $HOME
> XYZ
My home directory is $HOME
```

The most common use of *Heredocs* is dumping documentation to the user:

```
usage() {
    cat <<EOF
usage: foobar [-x] [-v] [-z] [file ...]
A short explanation of the operation goes here.
It might be a few lines long, but shouldn't be excessive.
EOF
}
```

Now let's check out the very similar but more compact *Herestring*:

```
$ grep proud <<<"I am a proud sentence"
I am a proud sentence.
```

This time, `stdin` reads its information straight from the string you put after the <<< operator. This is very convenient to send data that's in variables into processes:

```
$ grep proud <<<"$USER sits proudly on his throne in $HOSTNAME."
lhunath sits proudly on his throne in Lyndir.
```

*Herestring*s are shorter, less intrusive and overall more convenient than their bulky *Heredoc* counterpart. However, they are not portable to the Bourne shell.

Later on, you will learn about pipes and how they can be used to send the output of a command into another command's `stdin`. Many people use pipes to send the output of a variable as `stdin` into a command. However, for this purpose, *Herestring*s should be preferred. They do not create a subshell and are lighter both to the shell and to the style of your shell script:

```
$ echo 'Wrap this silly sentence.' | fmt -t -w 20
Wrap this silly
   sentence.
$ fmt -t -w 20 <<< 'Wrap this silly sentence.'
Wrap this silly
   sentence.
```

Technically, *Heredocs* and *Herestrings* are themselves redirects just like any other. As such, additional redirections can occur on the same line, all evaluated in the usual order.

```
$ cat <<EOF > file
> My home dir is $HOME
> EOF
$ cat file
My home dir is /home/greg
```

Good Practice:
Long heredocs are usually a bad idea because scripts should contain logic, not data. If you have a large document that your script needs, you should ship it in a separate file along with your script. Herestrings, however, come in handy quite often, especially for sending variable content (rather than files) to filters like `grep` or `sed`.

Pipes

Now that you can effortlessly manipulate *File Descriptors* to direct certain types of output to certain files, it's time you learn some more ingenious tricks available through I/O redirection.

You can use *File Redirection* to write output to files or read input from files. But what if you want to connect the output of one application directly to the input of another? That way, you could build a sort of chain to process output. If you already know about `FIFO`s, you could use something like this to that end:

```
$ ls
$ mkfifo myfifo; ls
myfifo
$ grep bea myfifo &
[1] 32635
$ echo "rat
> cow
> deer
> bear
> snake" > myfifo
bear
```

We use the `mkfifo` command to create a new file in the current directory named '`myfifo`'. This is no ordinary file, however, but a `FIFO` (which stands for *First In, First Out*). `FIFO`s are special files that serve data on a `First In, First Out`-basis. When you read from a `FIFO`, you will only receive data as soon as another process writes to it. As such, a `FIFO` never really contains any data. So long as no process writes to it, any read operation on the `FIFO` will **block** as it waits for data to become available. The same works for writes to the `FIFO` -- they will block until another process reads from the `FIFO`.

In our example, the `FIFO` called `myfifo` is read from by `grep`. `grep` waits for data to become available on the `FIFO`. That's why we append the grep command with the `&` operator, which puts it in the background. That way, we can continue typing and executing commands while `grep` runs and waits for data. Our `echo` statement feeds data to the `FIFO`. As soon as this data becomes available, the running `grep` command reads it in and processes it. The result is displayed. We have successfully sent data from the `echo` command to the `grep` command.

But these temporary files are a real annoyance. You may not have write permissions. You need to remember to clean up any temporary files you create. You need to make sure that data is going in and out, or the `FIFO` might just end up blocking for no reason.

For these reasons, another feature is made available: *Pipes*. A pipe basically just connects the `stdout` of one process to the `stdin` of another, effectively *piping* the data from one process into another. The entire set of commands that are piped together is called a *pipeline*. Let's try our above example again, but using pipes:

```
$ echo "rat
> cow
> deer
> bear
> snake" | grep bea
bear
```

The pipe is created using the `|` operator between two commands that are connected with the pipe. The former command's `stdout` is connected to the latter command's `stdin`. As a result, `grep` can read `echo`'s output and display the result of its operation, which is `bear`.

Pipes are widely used as a means of post-processing application output. `FIFO`s are, in fact, also referred to as `named pipes`. They accomplish the same results as the pipe operator, but through a filename.

Note:
The pipe operator creates a subshell environment for each command. This is important to know

because any variables that you modify or initialize inside the second command will appear unmodified outside of it. Let's illustrate:

```
$ message=Test
$ echo 'Salut, le monde!' | read message
$ echo "The message is: $message"
The message is: Test
$ echo 'Salut, le monde!' | { read message; echo "The message is: $message"; }
The message is: Salut, le monde!
$ echo "The message is: $message"
The message is: Test
```

Once the pipeline ends, so do the subshells that were created for it. Along with those subshells, any modifications made in them are lost. So be careful!

Good Practice:
Pipes are a very attractive means of post-processing application output. You should, however, be careful not to over-use pipes. If you end up making a pipeline that consists of three or more applications, it is time to ask yourself whether you're doing things a smart way. You might be able to use more application features of one of the post-processing applications you've used earlier in the pipe. Each new command in a pipeline causes a new subshell and a new application to be loaded. It also makes it very hard to follow the logic in your script!

In The Manual: Pipelines

In the FAQ:
I set variables in a loop. Why do they suddenly disappear after the loop terminates? Or, why can't I pipe data to read?
How can two processes communicate using named pipes (fifos)?
How can I redirect stderr to a pipe?
How can I read a file line-by-line?
Tell me all about 2>&1 -- what's the difference between 2>&1 >foo and >foo 2>&1, and when do I use which?

Miscellaneous Operators

Aside from the standard I/O operators, bash also provides a few more advanced operators that make life on the shell that much nicer.

Process Substitution

A cousin of the pipe is the process substitution operator, which comes in two forms: `<()` and `>()`. It's a convenient way to use named pipes without having to create temporary files. Whenever you think you need a temporary file to do something, process substitution might be a better way to handle things.

What it does, is basically run the command inside the parentheses. With the `<()` operator, the command's output is put in a named pipe (or something similar) that's created by bash. The operator itself in your command is replaced by the filename of that file. After your whole command finishes, the file is cleaned up.

Here's how we can put that into action: Imagine a situation where you want to see the difference between the output of two commands. Ordinarily, you'd have to put the two outputs in two files and `diff` those:

```
$ head -n 1 .dictionary > file1
$ tail -n 1 .dictionary > file2
$ diff -y file1 file2
Aachen                                                        | zymurgy
$ rm file1 file2
```

Using the *Process Substitution* operator, we can do all that with a one-liner and no need for manual cleanup:

```
$ diff -y <(head -n 1 .dictionary) <(tail -n 1 .dictionary)
Aachen                                                        | zymurgy
```

The `<(..)` part is replaced by the temporary FIFO created by bash, so `diff` actually sees something like this:

```
$ diff -y /dev/fd/63 /dev/fd/62
```

Here we see how bash runs `diff` when we use process substitution. It runs our `head` and `tail` commands, redirecting their respective outputs to the "files" `/dev/fd/63` and `/dev/fd/62`. Then it runs the `diff` command, passing those filenames where originally we had put the process substitution operators.

The actual implementation of the temporary files differs from system to system. In fact, you can see what the above would actually look like to `diff` on your box by putting an `echo` in front of our command:

```
$ echo diff -y <(head -n 1 .dictionary) <(tail -n 1 .dictionary)
diff -y /dev/fd/63 /dev/fd/62
```

The `>(..)` operator is much like the `<(..)` operator, but instead of redirecting the command's output to a file, we redirect the file to the command's input. It's used for cases where you're running a command that writes to a file, but you want it to write to another command instead:

```
$ tar -cf >(ssh host tar xf -) .
```

Good Practice:
Process Substitution gives you a concise way to create temporary FIFOs automatically. They're less flexible than creating your own named pipes by hand, but they're perfect for common short commands like `diff` that need filenames for their input sources.

Compound Commands

BASH offers numerous ways to combine simple commands to achieve our goals. We've already seen some of them in practice, but now let's look at a few more.

Bash has constructs called *compound commands*, which is a catch-all phrase covering several different concepts. We've already seen some of the compound commands Bash has to offer -- `if` statements, `for` loops, `while` loops, the `[[` keyword, `case` and `select`. We won't repeat that information again here.

Instead, we'll explore the other compound commands we haven't seen yet: subshells, command grouping, and arithmetic evaluation.

In addition, we'll look at *functions* and *aliases*, which aren't compound commands, but which work in a similar way.

In the manual: Compound Commands

Subshells

A SubShell is similar to a child process, except that more information is inherited. Subshells are created implicitly for each command in a pipeline. They are also created explicitly by using parentheses around a command:

```
$ (cd /tmp || exit 1; date > timestamp)
$ pwd
/home/lhunath
```

When the subshell terminates, the `cd` command's effect is gone -- we're back where we started. Likewise, any variables that are set during the subshell are not remembered. You can think of subshells as temporary shells. See SubShell for more details.

Note that if the `cd` failed in that example, the `exit 1` would have terminated the subshell, but *not* our interactive shell. As you can guess, this is quite useful in real scripts.

In the manual: Command Grouping

Command grouping

We've already touched on this subject in Grouping Statements, though it pays to repeat it in the context of this chapter.

Commands may be grouped together using curly braces. Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow. All compound commands such as if statements and while loops do this as well, but command groups do *only* this. In that sense, command groups can be thought of as "null compound commands" in that they have no effect other than to group commands. They look a bit like subshells, with the difference being that command groups are executed in the same shell as everything else, rather than a new one. This is both faster and allows things like variable assignments to be visible outside of the command group.

All commands within a command group are within the scope of any redirections applied to a command group (or any compound command):

```
$ { echo "Starting at $(date)"; rsync -av . /backup; echo "Finishing at $(date)"; }
>backup.log 2>&1
```

The above example truncates and opens the file backup.log on stdout, then points stderr at where stdout is currently pointing (backup.log), then runs each command with those redirections applied. The file descriptors remain open until all commands within the command group complete before they are automatically closed. This means backup.log is only opened a single time, not opened and closed for each command. The next example demonstrates this better:

```
$ echo "cat
> mouse
> dog" > inputfile
$ for var in {a..c}; do read -r "$var"; done < inputfile
$ echo "$b"
mouse
```

Notice how we didn't actually use a command group here. As previously explained, "for" being a compound command behaves just like a command group. It would have been extremely difficult to read the second line from a file without allowing for multiple `read` commands to read from a single open FD without rewinding to the start each time. Contrast with this:

```
$ read -r a < inputfile
$ read -r b < inputfile
$ read -r c < inputfile
$ echo "$b"
cat
```

That's not what we wanted at all!

Command groups are also useful to shorten certain common tasks:

```
$ [[ -f $CONFIGFILE ]] || { echo "Config file $CONFIGFILE not found" >&2; exit 1; }
```

The logical "or" now executes the command group if $CONFIGFILE doesn't exist rather than just the first simple command. A subshell would not have worked here, because the `exit 1` in a command group terminates the entire shell -- which is what we want here.

Compare that with a differently formatted version:

```
$ if [[ ! -f $CONFIGFILE ]]; then
> echo "Config file $CONFIGFILE not found" >&2
> exit 1
> fi
```

If the command group is on a single line, as we've shown here, then there *must* be a semicolon before the closing **}**, (i.e { ...; last command**;** }) otherwise, Bash would think **}** is an argument to the final command in the group. If the command group is spread across multiple lines, then the semicolon may be replaced by a newline:

```
$ {
>  echo "Starting at $(date)"
>  rsync -av . /backup
>  echo "Finishing at $(date)"
> } > backup.log 2>&1
```

If redirections are used on a simple command, they only apply to the command itself, not parameter or other expansions. Command groups make all contents including expansions apply even in the case of a single simple command:

```
$ { echo "$(cat)"; } <<<'hi'
hi
$ { "$(</dev/stdin)" <<<"$_"; } <<<'cat'
hi
```

The second command (which you don't need to fully understand) would require killing the shell if the command group weren't present, since the shell would be reading from the tty to determine the command to execute. It also illustrates that the command still gets the redirect applied to it, while the expansion gets that of the command group.

In the manual: Command Grouping

Arithmetic Evaluation

Bash has several different ways to say we want to do arithmetic instead of string operations. Let's look at them one by one.

The first way is the `let` command:

```
$ unset a; a=4+5
$ echo $a
4+5
$ let a=4+5
$ echo $a
9
```

You may use spaces, parentheses and so forth, if you quote the expression:

```
$ let a='(5+2)*3'
```

For a full list of operators availabile, see `help let` or the manual.

Next, the actual *arithmetic evaluation* compound command syntax:

```
$ ((a=(5+2)*3))
```

This is equivalent to `let`, but we can also use it as a *command*, for example in an `if` statement:

```
$ if (($a == 21)); then echo 'Blackjack!'; fi
```

Operators such as `==`, `<`, `>` and so on cause a comparison to be performed, inside an arithmetic evaluation. If the comparison is "true" (for example, `10 > 2` is true in arithmetic -- but not in strings!) then the compound command exits with status 0. If the comparison is false, it exits with status 1. This makes it suitable for testing things in a script.

Although not a compound command, an *arithmetic substitution* (or *arithmetic expression*) syntax is also available:

```
$ echo "There are $(($rows * $columns)) cells"
```

Inside `$((...))` is an *arithmetic context*, just like with `((...))`, meaning we do arithmetic (multiplying things) instead of string manipulations (concatenating `$rows`, space, asterisk, space, `$columns`). `$((...))` is also portable to the POSIX shell, while `((...))` is not.

Readers who are familiar with the C programming language might wish to know that `((...))` has many C-like features. Among them are the ternary operator:

```
$ ((abs = (a >= 0) ? a : -a))
```

and the use of an integer value as a truth value:

```
$ if ((flag)); then echo "uh oh, our flag is up"; fi
```

Note that we used variables inside `((...))` without prefixing them with `$`-signs. This is a special syntactic shortcut that Bash allows inside arithmetic evaluations and arithmetic expressions.

There is one final thing we must mention about `((flag))`. Because the inside of `((...))` is C-like, a variable (or expression) that evaluates to *zero* will be considered *false* for the purposes of the arithmetic evaluation. Then, because the evaluation is false, it will *exit* with a status of 1. Likewise, if the expression inside `((...))` is *non-zero*, it will be considered *true*; and since the evaluation is true, it will *exit* with status 0. This is potentially *very* confusing, even to experts, so you should take some time to think about this. Nevertheless, when things are used the way they're intended, it makes sense in the end:

```
$ flag=0      # no error
$ while read line; do
>   if [[ $line = *err* ]]; then flag=1; fi
> done < inputfile
$ if ((flag)); then echo "oh no"; fi
```

In the manual: Arithmetic Expansion, Shell Arithmetic

Functions

Functions are very nifty inside Bash scripts. They are blocks of commands, much like normal scripts you might write, except they don't reside in separate files, and they don't cause a separate process to be executed. However, they take arguments just like scripts -- and unlike scripts, they can affect variables inside your script, if you want them to. Take this for example:

```
$ sum() {
>   echo "$1 + $2 = $(($1 + $2))"
> }
```

This will do absolutely nothing when run. This is because it has only been stored in memory, much like a variable, but it has not yet been called. To run the function, you would do this:

```
$ sum 1 4
1 + 4 = 5
```

Amazing! We now have a basic calculator, and potentially a more economic replacement for a five year-old.

A note on scope: if you choose to embed functions within script files, as many will find more convenient, then you need to understand that the parameters you pass to the script are not necessarily the parameters that are passed to the function. To wrap this function inside a script, we would write a file containing this:

Toggle line numbers

```
#!/bin/bash
sum() {
    echo "$1 + $2 = $(($1 + $2))"
}
sum "$1" "$2"
```

As you can see, we passed the script's two parameters to the function within, but we could have passed anything we wanted (though, doing so in this situation would only confuse users trying to use the script).

Functions serve a few purposes in a script. The first is to isolate a block of code that performs a specific task, so that it doesn't clutter up other code. This helps you make things more readable, when done in moderation. (Having to jump all over a script to track down 7 functions to figure out what a single command does has the opposite effect, so make sure you do things that make sense.) The second is to allow a block of code to be reused with slightly different arguments.

Here's a slightly less silly example:

Toggle line numbers

```
#!/bin/bash
open() {
    case "$1" in
        *.mp3|*.ogg|*.wav|*.flac|*.wma) xmms "$1";;
        *.jpg|*.gif|*.png|*.bmp)        display "$1";;
        *.avi|*.mpg|*.mp4|*.wmv)        mplayer "$1";;
    esac
}
for file; do
    open "$file"
done
```

Here, we define a *function* named `open`. This function is a block of code that takes a single argument, and based on the *pattern* of that argument, it will either run `xmms`, `display` or `mplayer` with that argument. Then, a `for` loop iterates over all of the *script's* positional parameters. (Remember, `for file` is equivalent to `for file in "$@"` and both of them iterate over the full set of positional parameters.) The `for` loop calls the `open` function for each parameter.

As you may have observed, the function's parameters are different from the script's parameters.

Functions may also have *local variables*, declared with the `local` or `declare` keywords. This lets you do work without potentially overwriting important variables from the caller's namespace. For example,

Toggle line numbers

```
#!/bin/bash
count() {
    local i
    for ((i=1; i<=$1; i++)); do echo $i; done
    echo 'Ah, ah, ah!'
}
for ((i=1; i<=3; i++)); do count $i; done
```

The `local` variable `i` inside the function is stored differently from the variable `i` in the outer script. This allows the two loops to operate without interfering with each other's counters.

Functions may also call themselves *recursively*, but we won't show that today. *Maybe later!*

In the manual: Shell Functions

Aliases

Aliases are superficially similar to functions at first glance, but upon closer examination, they have entirely different behavior.

- Aliases do not work in scripts, at all. They only work in interactive shells.
- Aliases cannot take arguments.
- Aliases will not invoke themselves recursively.
- Aliases cannot have local variables.

Aliases are essentially keyboard shortcuts intended to be used in `.bashrc` files to make your life easier. They usually look like this:

```
$ alias ls='ls --color=auto'
```

Bash checks the first word of every simple command to see whether it's an *alias*, and if so, it does a simple text replacement. Thus, if you type

```
$ ls /tmp
```

Bash acts as though you had typed

```
$ ls --color=auto /tmp
```

If you wanted to duplicate this functionality with a function, it would look like this:

```
$ unalias ls
$ ls() { command ls --color=auto "$@"; }
```

As with a *command group*, we need a `;` before the closing `}` of a function if we write it all in one line. The special built-in command `command` tells our function **not** to call itself recursively; instead, we want it to call the `ls` command that it would have called if there hadn't been a function by that name.

Aliases are useful as long as you don't try to make them work like functions. If you need complex behavior, use a function instead.

Destroying Constructs

To remove a function or variable from your current shell environment use the *unset* command. It is usually a good idea to be explicit about whether a function or variable is to be unset using the *-f* or *-v* flags respectively. If unspecified, variables take precedence over functions.

```
$ unset -f myfunction
```

```
$ unset -v 'myArray[2]' # unset element 2 of myArray. The quoting is important to
prevent globbing.
```

To remove an alias, use the *unalias* command.

```
$ unalias rm
```

In the manual: Bourne Shell Builtins

In the FAQ: ...

Sourcing

When you call one script from another, the new script inherits the environment of the original script, just like running any other program in UNIX. Explaining what this means is out of the scope of this guide, but for the most part you can consider the environment to be the current working directory, open file descriptors, and environment variables, which you can view using the `export` command.

When the script that you ran (or any other program, for that matter) finishes executing, its environment is discarded. The environment of the first script will be same as it was before the second script was called, although of course some of bash's special parameters may have changed (such as the value of `$?`, the return value of the most recent command). This means, for example, you can't simply run a script to change your current working directory for you.

In the FAQ:
I'm trying to write a script that will change directory (or set a variable), but after the script finishes, I'm back where I started (or my variable isn't set)!

What you can do is to *source* the script, instead of running it as a child. You can do this using the `.` (dot) command:

```
. ./myscript   #runs the commands from the file myscript in this environment
```

This is often called *dotting in* a script. The `.` tells BASH to read the commands in `./myscript` and run them in the current shell environment. Since the commands are run in the current shell, they can change the current shell's variables, working directory, open file descriptors, functions, etc.

Note that Bash has a second name for this command, `source`, but since this works identically to the `.` command, it's probably easier to just forget about it and use the `.` command, as that will work everywhere.

Job Control

Though not typically used in scripts, job control is very important in interactive shells. Job control allows you to interact with background jobs, suspend foreground jobs, and so on.

Theory

On Posix systems, jobs are implemented as "process groups", with one process being the leader of the group. Each tty (terminal) has a single "foreground process group" that is allowed to interact with the terminal. All other process groups with the same controlling tty are considered background jobs, and can be either running or suspended.

A job is suspended when its process group leader receives one of the signals `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU`. `SIGTTIN` and `SIGTTOU` are automatically sent whenever a background job tries to read from or write to the terminal---this is why `cat &` is immediately suspended rather than running in the background.

Certain keypresses at the terminal cause signals to be sent to all processes in the foreground process group. These can be configured with `stty`, but are usually set to the defaults:

- Ctrl-Z sends `SIGTSTP` to the foreground job (usually suspending it)
- Ctrl-C sends `SIGINT` to the foreground job (usually terminating it)
- Ctrl-\ sends `SIGQUIT` to the foreground job (usually causing it to dump core and abort)

Practice

Job control is on by default in interactive shells. It can be turned on for scripts with `set -m` or `set -o monitor`.

A foreground job can be suspended by pressing Ctrl-Z. There is no way to refer to the foreground job in `bash`: if there is a foreground job other than bash, then bash is waiting for that job to terminate, and hence cannot execute any code (even traps are delayed until the foreground job terminates). The following commands, therefore, work on background (and suspended) jobs only.

Job control enables the following commands:

- **`fg`** [*jobspec*]: bring a background job to the foreground.
- **`bg`** [*jobspec* ...]: run a suspended job in the background.
- **`suspend`**: suspend the shell (mostly useful when the parent process is a shell with job control).

Other commands for interacting with jobs include:

- **`jobs`** [*options*] [*jobspec* ...]: list suspended and background jobs. Options include `-p` (list process IDs only), `-s` (list only suspended jobs), and `-r` (list only running background jobs). If one or more jobspecs are specified, all other jobs are ignored.
- **`kill`** can take a jobspec instead of a process ID.
- **`disown`** tells `bash` to forget about an existing job. This keeps bash from automatically sending `SIGHUP` to the processes in that job, but also means it can no longer be referred to by jobspec.

So, what does all that mean? Job control allows you to have multiple things going on within a single terminal session. (This was terribly important in the old days when you only had one terminal on your desk, and no way to create virtual terminals to add more.) On modern systems and hardware, you have more choices available -- you could for example run `screen` or `tmux` within a terminal to give you virtual terminals. Or within an X session, you could open more `xterm` or similar terminal emulators (and you can mix the two together as well).

But sometimes, a simple job control "suspend and background" comes in handy. Maybe you started a backup and it's taking longer than you expected. You can suspend it with Ctrl-Z and then put it in the background with `bg`, and get your shell prompt back, so that you can work on something else in the same terminal session while that backup is running.

Job Specifications

A job specification or "jobspec" is a way of referring to the processes that make up a job. A jobspec may be:

- **`%`*n*** to refer to job number *n*.
- **`%`*str*** to refer to a job which was started by a command beginning with *str*. It is an error if there is more than one such job.
- **`%?`*str*** to refer to a job which was started by a command containing *str*. It is an error if there is more than one such job.
- **`%%`** or **`%+`** to refer to the current job: the one most recently started in the background, or suspended from the foreground. `fg` and `bg` will operate on this job if no **jobspec** is given.
- **`%-`** for the previous job (the job that was `%%` before the current one).

It is possible to run an arbitrary command with a jobspec, using `jobs -x ''cmd args...''`. This replaces arguments that look like jobspecs with the PIDs of the corresponding process group leaders, then runs the command. For example, `jobs -x strace -p %%` will attach `strace` to the current job (most useful if it is running in the background rather than suspended).

Finally, a bare jobspec may be used as a command: `%1` is equivalent to `fg %1`, while `%1 &` is equivalent to `bg %1`.

See Also

ProcessManagement discusses practices for working with multiple processes. There's also an example of the use of job control within a script.

Practices

Choose Your Shell

The first thing you should do before starting a shell script, or any kind of script or program for that matter, is enumerate the requirements and the goal of that script. Then evaluate what the best tool is to accomplish those goals.

BASH may be easy to learn and write in, but it isn't always fit for the job.

There are a lot of tools in the basic toolset that can help you. If you just need AWK, then don't make a shell script that invokes AWK. Just make an AWK script. If you need to retrieve data from an HTML or XML file in a reliable manner, Bash is also **the wrong tool for the job**. You should consider XPath/XSLT instead, or a language that has a library available for parsing XML or HTML.

If you decide that a shell script is what you want, then first ask yourself these questions:

- In the foreseeable future, might your script be needed in an environment where Bash is not available *by default*?
  - If so, **then consider sh instead.** `sh` is a POSIX shell and its features are available in any shell that complies with the POSIX standard. You can rely on the fact that any POSIX system will be able to run your script. You will have to balance the needs of portability against the Bash features you'd be unable to use.
  - *Keep in mind that this guide does not apply to sh!* The Bashism page gives some alternatives, but it is not complete.
- Are you certain that in all environments you will run and might want to run this script in the future Bash 3.x (or 4.x) will be available to you?
  - **If not, you should limit yourself to features of Bash 2.x ONLY.**

If the above questions do not limit your choices, use all the Bash features you require, and then make note of which version of Bash is required to run your script.

Using Bash 3 or higher means you can avoid ancient scripting techniques. They have been replaced with far better alternatives for *very good reasons*.

- Stay away from scripting examples you see on the Web unless you understand them completely. Mostly any scripts you will find on the Web are broken in some way. Do not copy/paste from them.
- Always use the correct shebang. If you're writing a Bash script, you **need** to put `#!/usr/bin/env bash` at the top of your script. Omitting this header or using the `#!/bin/sh` header is **wrong**. In the latter case you will no longer be able to use any Bash features. You will be limited to the POSIX shell scripting standard (even if your `/bin/sh` links to Bash).
- When writing Bash scripts, do **not** use the `[` command. Bash has a far better alternative: `[[`. Bash's `[[` is more reliable in many ways and there are no advantages whatsoever to sticking with the old relic. You'll also miss out on a lot of features provided by `[[` that `[` does not have (like pattern matching).
- It's also time you forgot about `` `...` ``. It isn't consistent with the syntax of *Expansion* and is terribly hard to nest without a dose of painkillers. Use `$(...)` instead.
- And for heaven's sake, "**Use more quotes!**" Protect your strings and parameter expansions from WordSplitting. Word splitting will eat your babies if you don't quote things properly.
- Learn to use parameter expansions instead of `sed` or `cut` to manipulate simple strings in Bash. If you want to remove the extension from a filename, use `${filename%.*}` instead of `` `echo "$filename" | sed 's/\.[^.]*$//'` `` or some other dinosaur.
- Use built-in math instead of `expr` to do simple calculations, *especially* when just incrementing a variable. If the script you're reading uses ``x=`expr $x + 1` `` then it's not something you want to mimic.

Quoting

Word Splitting is the demon inside Bash that is out to get unsuspecting newcomers or even veterans who let down their guard.

If you do not understand how *Word Splitting* works or when it is applied you should be very careful with your strings and your *Parameter Expansion*s. I suggest you read up on WordSplitting if you doubt your knowledge.

The best way to protect yourself from this beast is to quote all your strings. Quotes keep your strings in one piece and prevent *Word Splitting* from tearing them open. Allow me to illustrate:

```
$ echo Push that word             away from me.
Push that word away from me.
$ echo "Push that word             away from me."
Push that word             away from me.
```

Now, don't think *Word Splitting* is about collapsing spaces. What really happened in this example is that the first command passed each word in our sentence as a separate argument to `echo`. Bash split our sentence up in words using the whitespace between them to determine where each argument begins and ends. In the second example Bash is forced to keep the whole quoted string together. This means it's not split up into arguments and the whole string is passed to `echo` **as one argument**. `echo` prints out each argument it gets with a space in between them. You should understand the basics of *Word Splitting* now.

This is where it gets dangerous: *Word Splitting* does not just happen on literal strings. It also happens **after *Parameter Expansion***! As a result, on a dull and tired day, you might just be stupid enough to make this mistake:

```
$ sentence="Push that word             away from me."
$ echo $sentence
Push that word away from me.
$ echo "$sentence"
Push that word             away from me.
```

As you can see, in the first `echo` command, we were negligent and left out the quotes. That was a mistake. Bash expanded our sentence and then used *Word Splitting* to split the resulting expansion up into arguments to use for `echo`. In our second example, the quotes around the *Parameter Expansion* of `sentence` make sure Bash **does not** split it up into multiple arguments around the whitespace.

It's not just spaces you need to protect. *Word Splitting* occurs on spaces, tabs and newlines, or whatever characters are in the `IFS` variable. Here's another example to show you just how badly you can break things if you neglect to use quotes:

```
$ echo "$(ls -al)"
total 8
drwxr-xr-x   4 lhunath users 1 2007-06-28 13:13 "."/
drwxr-xr-x 102 lhunath users 9 2007-06-28 13:13 ".."/
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "a"
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "b"
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "c"
drwxr-xr-x   2 lhunath users 1 2007-06-28 13:13 "d"/
drwxr-xr-x   2 lhunath users 1 2007-06-28 13:13 "e"/
$ echo $(ls -al)
total 8 drwxr-xr-x 4 lhunath users 1 2007-06-28 13:13 "."/ drwxr-xr-x 102 lhunath users
9 2007-06-28 13:13 ".."/ -rw-r--r-- 1 lhunath users 0 2007-06-28 13:13 "a" -rw-r--r-- 1
lhunath users 0 2007-06-28 13:13 "b" -rw-r--r-- 1 lhunath users 0 2007-06-28 13:13 "c"
drwxr-xr-x 2 lhunath users 1 2007-06-28 13:13 "d"/ drwxr-xr-x 2 lhunath users 1 2007-
06-28 13:13 "e"/
```

In some very rare occasions it may be desired to leave out the quotes. That's if you **need** *Word Splitting* to take place:

```
$ friends="Marcus JJ Thomas Michelangelo"
$ for friend in $friends
> do echo "$friend is my friend!"; done
Marcus is my friend!
```

```
JJ is my friend!
Thomas is my friend!
Michelangelo is my friend!
```

But, honestly? You should use arrays for nearly all of these cases. Arrays have the benefit that they separate strings without the need for an explicit delimiter. That means your strings in the array can contain **any** valid (non-NUL) character, without the worry that it might be your string delimiter (like the space is in our example above). Using arrays in our example above gives us the ability to add middle or last names of our friends:

```
$ friends=( "Marcus The Rich" "JJ The Short" "Timid Thomas" "Michelangelo The Mobster" )
$ for friend in "${friends[@]}"
> do echo "$friend is my friend"; done
```

Note that in our previous `for` we used an *unquoted* `$friends`. This allowed Bash to split our `friends` string up into words. In this last example, we quoted the `${friends[@]}` *Parameter Expansion*. Quoting an expansion of an array through the `@` index makes Bash expand that array into a sequence of its elements, **where each element is wrapped in quotes**.

Readability

Almost as important as the result of your code is the **readability of your code**.

Chances are that you aren't just going to write a script once and then forget about it. If so, you might as well delete it after using it. If you plan to continue using it, you should also plan to continue maintaining it. Unlike your house your code won't get dirty over time, but you will learn new techniques and new approaches constantly. You will also gain insight about how your script is used. All that new information you gather since the completion of your initial code should be used to maintain your code in such a way that it constantly improves. Your code should keep growing more user-friendly and more stable.

> Trust me when I say, no piece of code is ever 100% finished, with the exception of some very short and useless chunks of nothingness.

To make it easier for yourself to keep your code healthy and improve it regularly you should keep an eye on the readability of what you write. When you return to a long loop after a year has passed since your last visit to it and you wish to improve it, add a feature, or debug something about it, believe me when I say you'd rather see this:

Toggle line numbers

```
friends=( "Marcus The Rich" "JJ The Short" "Timid Thomas" "Michelangelo The Mobster" )

# Say something significant about my friends.
for name in "${friends[@]}"; do

    # My first friend (in the list).
    if [[ $name = "${friends[0]}" ]]; then
        echo "$name was my first friend."

    # My friends those names start with M.
    elif [[ $name = M* ]]; then
        echo "$name starts with an M"

    # My short friends.
    elif [[ " $name " = *" Short "* ]]; then
        echo "$name is a shorty."

    # Friends I kind of didn't bother to remember.
    else
        echo "I kind of forgot what $name is like."

    fi
done
```

Than be confronted with something like this:

Toggle line numbers

```
x=(       Marcus\ The\ Rich JJ\ The\ Short
  Timid\ Thomas Michelangelo\ The\ Mobster)
for name in "${x[@]}"
  do if [ "$name" = "$x" ]; then echo $name was my first friend.
 elif
   echo $name    |   \
  grep -qw Short
    then echo $name is a shorty.
 elif [ "x${name:0:1}" = "xM" ]
     then echo $name starts   with an M; else
echo I kind of forgot what $name \
 is like.; fi; done
```

And yes, I know this is an exaggerated example, but I've seen some authentic code that actually has a lot in common with that last example.

For your own health, keep these few points in mind:

- *Healthy whitespace gives you breathing space*. Indent your code properly and consistently. Use blank lines to separate paragraphs or logic blocks.
- *Avoid backslash-escaping*. It's counter-intuitive and boggles the mind when overused. Even in small examples it takes your mind more effort to understand `a\ b\ c` than it takes to understand `'a b c'`.
- *Comment your way of thinking before you forget*. You might find that even code that looks totally common sense right now could become the subject of "*What the hell was I thinking when I wrote this?*" or "*What is **this** supposed to do?*".
- *Consistency prevents mind boggles*. Be consistent in your naming style. Be consistent in your use of capitals. Be consistent in your use of shell features. In coding, unlike in the bedroom, it's *good* to be simple and predictable.

Bash Tests

The `test` command, also known as `[`, is an application that usually resides somewhere in `/usr/bin` or `/bin` and is used a lot by shell programmers to perform certain tests on files and variables. In a number of shells, including Bash, `test` is also implemented as a shell builtin.

It can produce surprising results, especially for people starting shell scripting that think [ ] is part of the shell syntax.

If you use `sh`, you have little choice but to use `test` as it is the only way to do most of your testing.

If however you are using Bash to do your scripting (and I presume you are since you're reading this guide), then you can also use the `[[` keyword. While it still behaves in many ways like a command, it presents several advantages over the traditional `test` command.

Let me illustrate how `[[` can be used to replace `test`, and how it can help you to avoid some of the common mistakes made by using `test`:

```
$ var=''
$ [ $var = '' ] && echo True
-bash: [: =: unary operator expected
$ [ "$var" = '' ] && echo True
True
$ [[ $var = '' ]] && echo True
True
```

`[ $var = '' ]` expands into `[ = '' ]`. The first thing `test` does is count its arguments. Since we're using the `[` form, we'll just strip off the mandatory `]` argument at the end. In the first example, `test` sees two arguments: = and `''`. It knows that if it has two arguments, the first one has to be a *unary operator* (an operator that takes one operand). But = is not a unary operator (it's binary -- it requires *two* operands), so `test` blows up.

Yes, `test` did not see our empty `$var` because Bash expanded it into nothingness before `test` could even see it. Moral of the story? **Use more quotes!** Using quotes, `[ "$var" = '' ]` expands into `[ "" = '' ]` and `test` has no problem.

Now, `[[` can see the whole command before it's being expanded. It sees `$var`, and not the expansion of `$var`. As a result, there is no need for the quotes at all! **`[[` is safer**.

```
$ var=
$ [ "$var" < a ] && echo True
-bash: a: No such file or directory
$ [ "$var" \< a ] && echo True
True
$ [[ $var < a ]] && echo True
True
```

In this example we attempted a string comparison between an empty variable and '`a`'. We're surprised to see the first attempt does not yield `True` even though we think it should. Instead, we get some weird error that implies Bash is trying to open a file called '`a`'.

We've been bitten by *File Redirection*. Since `test` is just an application, the < character in our command is interpreted (as it should) as a *File Redirection* operator instead of the string comparison operator of `test`. Bash is instructed to open a file '`a`' and connect it to `stdin` for reading. To prevent this, we need to escape < so that `test` receives the operator rather than Bash. This makes our second attempt work.

Using `[[` we can avoid the mess altogether. `[[` sees the < operator before Bash gets to use it for *Redirection* -- problem fixed. Once again, **`[[` is safer**.

Even more dangerous is using the > operator instead of our previous example with the < operator. Since > triggers output *Redirection* it will create a file called '`a`'. As a result, **there will be no error message warning us that we've committed a sin**! Instead, our script will just break. Even worse, we might overwrite some important file! It's up to us to *guess* where the problem is:

```
$ var=a
$ [ "$var" > b ] && echo True || echo False
```

```
True
$ [[ "$var" > b ]] && echo True || echo False
False
```

Two different results, great. Trust me when I say you can always trust `[[` more than `[`. `[ "$var" > b ]` is expanded into `[ "a" ]` and the output of that is being redirected into a new file called '`b`'. Since `[ "a" ]` is the same as `[ -n "a" ]` and that basically tests whether the `"a"` string is non-empty, the result is a success and the `echo True` is executed.

Using `[[` we get our expected scenario where `"a"` is tested against `"b"` and since we all know `"a"` sorts before `"b"` this triggers the `echo False` statement. And this is how you can break your script without realizing it. You will however have a suspiciously empty file called '`b`' in your current directory.

So believe me when I say, **`[[` is safer than `[`**. Because everybody inevitably makes programming errors. People usually don't intend to introduce bugs in their code. It just happens. So don't pretend you can use `[` and "*You'll be careful not to make these mistakes*", because I can assure you that you **will**.

Besides `[[` provides the following features over `[`:

- **`[[` can do glob pattern matching:**

  ```
  [[ abc = a* ]]
  ```

- **`[[` can do regex pattern matching (Bash 3.1+):**

  ```
  [[ abb =~ ab+ ]]
  ```

The only advantage of `test` is its portability.

Don't Ever Do These

The Bash shell allows you to do quite a lot of things, offering you considerable flexibility. Unfortunately, it does very little to discourage misuse and other ill-advised behavior. It hopes people will find out for themselves that certain things should be avoided at all costs.

Unfortunately many people don't care enough to want to find out for themselves. They write without thinking things through and many awful and dangerous scripts end up in production environments or in Linux distributions. The result of these, and even your very own scripts written in a time of neglect, can often be **DISASTROUS**.

That said, for the good of your scripts and for the rest of mankind, **Never Ever Do Anything Along These Lines**:

- **`ls -l | awk '{ print $8 }'`**

  DON'T EVER parse the output of `ls`! The `ls` command's output cannot be trusted for a number of reasons.

  1. For one, `ls` will mangle the filenames of files if they contain characters unsupported by your locale. As a result, parsing filenames out of `ls` is NEVER guaranteed to actually give you a file that you will be able to find. `ls` might have replaced certain characters in the filename by question marks.
  2. Secondly, `ls` separates lines of data by newlines. This way, every bit of information on a file is on a new line. **UNFORTUNATELY filenames themselves can ALSO contain**

newlines. This means that if you have a filename that contains a newline in the current directory, it will completely break your parsing and as a result, break your script!

3. Last but not least, the output format of `ls -l` is NOT guaranteed consistent across platforms. Some systems omit the group ID by default, for instance, and reverse the effect of the `-g` switch. Some systems use two fields for the modification time, and some use three. On the systems that use three, the third field can be either a year, or an HH:MM concatenation, depending on how old the file is.

There are alternatives to using ls in many cases. If you need to work with file modification times, you can typically use Bash Tests. If none of those things is possible, *I recommend you switch to a different language, like python or perl*.

- **`if echo "$file" | fgrep .txt; then`**
  `ls *.txt | grep story`

  DON'T EVER test or filter filenames with `grep`! Unless your `grep` pattern is really smart it will probably not be trustworthy.

  In the first example above, the test would match both `story.txt` AND `story.txt.exe`. If you make `grep` patterns that are smart enough, they'll probably be so ugly, massive and unreadable that you shouldn't use them anyway.

  The alternative is called globbing. Bash has a feature called *Pathname Expansion*. This will help you enumerate all files that match a certain pattern. Also, you can use globs to test whether a filename matches a certain glob pattern (in a `case` or `[[` command).

- **`cat file | grep pattern`**

  Don't use `cat` to feed a single file's content to a filter. `cat` is a utility used to concatenate the contents of several files together.

  To feed the contents of a file to a process you will probably be able to pass the filename as an argument to the program (`grep 'pattern' /my/file`, `sed 'expression' /my/file`, ...).

  If the manual of the program does not specify any way to do this, you should use redirection (`read column1 column2 < /my/file`, `tr ' ' '\n' < /my/file`, ...).

- **`for line in $(<file); do`**

  Don't use a for loop to read the lines of a file. Use a while read loop instead.

- **`for number in $(seq 1 10); do`**

  For the love of god and all that is holy, do not use `seq` to count.

  Bash is able enough to do the counting for you. You do not need to spawn an external application (especially a single-platform one) to do some counting and then pass that application's output to Bash for word splitting. Learn the syntax of `for` already!

  You should use this in Bash 3.x: `for number in {1..10}`, or this in Bash 2.x: `for ((i=1; i<=10; i++))`.

  If you actually wanted a stream of numbers separated by newlines as test input, consider `printf '%d\n' {1..10}`

- **``i=`expr $i + 1` ``**

  `expr` is a relic of ancient Rome. Do not wield it.

  It was used in scripts written for shells with very limited capabilities. You're basically spawning a new process, calling another C program to do some math for you and return the result as a string to bash. Bash can do all this itself and so much faster, more reliably (no numbers->string->number conversions) and in all, better.

  You should use this in Bash: `let i++` or `((i++))`

  Even POSIX sh can do arithmetic: `i=$(($i+1))`. It only lacks the `++` operator and the

`((...))` command (it has only the `$((...))` substitution).

Debugging

Very often you will find yourself clueless as to why your script isn't acting the way you want it to. Resolving this problem is always just a matter of common sense and debugging techniques.

Diagnose the Problem

Unless you know what exactly the **problem** is, you most likely won't come up with a *solution* anytime soon. So make sure you understand what exactly goes wrong. Evaluate the symptoms and/or error messages.

Try to formulate the problem as a sentence. This will also be vital if you're going to ask other people for help with your problem. You don't want them to have to go through your whole script or run it so that they understand what's going on. No; **you** need to make the problem perfectly clear to **yourself** and to anybody trying to help you. *This requirement stands until the day the human race invents means of telepathy*.

Minimize the Codebase

If staring at your code doesn't give you a divine inspiration, the next thing you should do is try to minimize your codebase to isolate the problem.

Don't worry about preserving the functionality of your script. The only thing you want to preserve is the logic of the code block that seems buggy.

Often, the best way to do this is to copy your script to a new file and start deleting everything that seems irrelevant from it. Alternatively, you can make a new script that does something similar in the same code fashion, and keep adding structure until you duplicate the problem.

As soon as you delete something that makes the problem go away (or add something that makes it appear), you'll have found where the problem lies. Even if you haven't precisely pinpointed the issue, at least you're not staring at a massive script anymore, but hopefully at a stub of no more than 3-7 lines.

For example, if you have a script that lets you browse images in your image folder by date, and for some reason you can't manage to iterate over your images in the folder properly, it suffices to reduce the script to this part:

```
for image in $(ls -R "$imgFolder"); do
    echo "$image"
done
```

Your actual script will be far more complex, and the inside of the `for` loop will also be far longer. But the essence of the problem is this code. Once you've reduced your problem to this it may be easier to see the problem you're facing. Your `echo` spits out parts of image names; it looks like all whitespace is replaced by newlines. That must be because `echo` is run once for each chunk terminated by whitespace, not for every image name (as a result, it seems the output has split open image names with whitespace in them). With this reduced code, it's easier to see that the cause is actually your `for` statement that splits up the output of `ls` into words. That's because `ls` is **UNPARSABLE** in a bugless manner (do not ever use `ls` in scripts, unless if you want to show its output to a user).

We can't use a recursive glob (unless we're in bash 4), so we have to use `find` to retrieve the filenames. One fix would be:

```
find "$imgFolder" -print0 | while IFS= read -r -d '' image; do
    echo "$image"
done
```

Now that you've fixed the problem in this tiny example, it's easy to merge it back into the original script.

Activate Bash's Debug Mode

If you still don't see the error of your ways, Bash's debugging mode might help you see the problem through the code.

When Bash runs with the `x` option turned on, it prints out every command it executes before executing it (to standard error). That is, **after any expansions have been applied**. As a result, you can see exactly what's happening as each line in your code is executed. Pay very close attention to the quoting used. Bash uses quotes to show you exactly which strings are passed as a single argument.

There are three ways of turning on this mode.

- Run the script with `bash -x`:

  ```
  $ bash -x ./mybrokenscript
  ```

- Modify your script's header:

  ```
  #!/bin/bash -x
  [.. script ..]
  ```

  Or:

  ```
  #!/usr/bin/env bash
  set -x
  ```

- Or add `set -x` somewhere in your code to turn on this mode for only a specific block of your code:

  ```
  #!/usr/bin/env bash
  [..irrelevant code..]
  set -x
  [..relevant code..]
  set +x
  [..irrelevant code..]
  ```

Because the debugging output goes to stderr, you will generally see it on the screen, if you are running the script in a terminal. If you would like to log it to a file, you can tell Bash to send all stderr to a file:

```
exec 2>> /path/to/my.logfile
set -x
```

A nice feature of bash version >= 4.1 is the variable BASH_XTRACEFD. This allows you to specify the file descriptor to write the `set -x` debugging output to. In older versions of bash, this output always goes to stderr, and it is difficult if not impossible to keep it separate from normal output (especially if you are logging stderr to a file, but you need to see it on the screen to operate the program). Here's a nice way to use it:

```
# dump set -x data to a file
```

```
# turns on with a filename as $1
# turns off with no params
# note that FD 4 should not be used elsewhere in the script
setx_output()
{
    if [[ $1 ]]; then
        exec 4>>"$1"
        BASH_XTRACEFD=4
        set -x
    else
        set +x
        exec 4>&-
    fi
}
```

If you have a complicated mess of scripts, you might find it helpful to change PS4 before setting -x:

```
export PS4='+$BASH_SOURCE:$LINENO:$FUNCNAME: '
```

Step Your Code

If the script goes too fast for you, you can enable code-stepping. The following code uses the `DEBUG` trap to inform the user about what command is about to be executed and wait for his confirmation do to so. Put this code in your script, at the location you wish to begin stepping:

```
trap '(read -p "[$BASH_SOURCE:$LINENO] $BASH_COMMAND?")' DEBUG
```

The Bash Debugger

The Bash Debugger Project is a gdb-style debugger for bash, available from http://bashdb.sourceforge.net/

The Bash Debugger will allow you to walk through your code and help you track down bugs.

Reread the Manual

If your script still doesn't seem to agree with you, maybe your perception of the way things work is wrong. Try going back to the manual (or this guide) to re-evaluate whether commands do exactly what you think they do, or the syntax is what you think it is. Very often people misunderstand what `for` does, how *Word Splitting* works, or how often they should use quotes.

Keep the tips and good practice guidelines in this guide in mind as well. They often help you avoid bugs and problems with scripts.

I mentioned this in the *Scripts* section of this guide too, but it's worth repeating it here. First of all, make sure your script's **header** is actually **`#! /bin/bash`**. If it is **missing** or if it's something like **`#! /bin/sh`** then you deserve the problems you're having. That means you're probably not even using Bash to run your code. Obviously that'll cause issues. Also, make sure you have no *Carriage Return* characters at the ends of your lines. This is the cause of scripts written in *Microsoft Windows(tm)*. You can get rid of these fairly easily like this:

```
$ tr -d '\r' < myscript > tmp && mv tmp myscript
```

Read the FAQ / Pitfalls

The BashFAQ and BashPitfalls pages explain common misconceptions and issues encountered by other Bash scripters. It's very likely that your problem will be described there in some shape or form.

To be able to find your problem in there, you'll obviously need to have *Diagnosed* it properly. You'll need to know what you're looking for.

Ask Us on IRC

There are people in the `#bash` channel almost 24/7. This channel resides on the `freenode` IRC network. To reach us, you need an IRC client. Connect it to `irc.freenode.net`, and `/join #bash`.

Make **sure** that you know what your real problem is and have stepped through it on paper, so you can explain it well. We don't like having to guess at things. Start by explaining what you're trying to do with your script.

Either way, please have a look at this page before entering `#bash`: XyProblem.

FullBashGuide (last edited 2012-12-02 17:35:01 by geirha)