

# Building Blocks

**W**hen you're young, you learn numbers, and then later you learn how to combine and work with those numbers using symbols such as +, -, ×, and =. So far in this book, you've learned several commands, but each one has been run one at a time. Commands can actually be combined in more complex and more interesting ways, however, using various symbols such as |, >, >>, and <. This chapter takes a look at those building blocks that enable you to do some useful things with the commands you've learned and the commands you'll be examining in greater detail in subsequent chapters.

## Run Several Commands Sequentially

;

What if you have several commands you need to run consecutively, but some of them are going to take a long time, and you don't feel like babysitting your computer? For instance, what if you have a huge number of John Coltrane MP3s in a zipped archive file, and you want to unzip them, place them in a new subdirectory,

and then delete the archive file? Normally you'd have to run those commands one at a time, like this:

---

**NOTE:** In order to save space, I removed the owner and group from the long listing.

---

```
$ ls -l /home/scott/music
-rw-r--r-- 1437931 2005-11-07 17:19 JohnColtrane.zip
$ unzip /home/scott/music/JohnColtrane.zip
$ mkdir -p /home/scott/music/coltrane
$ mv /home/scott/music/JohnColtrane*.mp3
  ➔ /home/scott/music/coltrane/
$ rm /home/scott/music/JohnColtrane.zip
```

JohnColtrane.zip is a 1.4GB file, and even on a fast machine, unzipping that monster is going to take some time, and you probably have better things to do than sit there and wait. Command stacking to the rescue!

Command stacking puts all the commands you want to run on one line in your shell, with each specific command separated by a semicolon (;). Each command is then executed in sequential order and each must terminate—successfully or unsuccessfully—before the next one runs. It's easy to do, and it can really save you some time.

With command stacking, the previous series of commands now looks like this:

```
$ ls -l /home/scott/music
-rw-r--r-- 1437931 2005-11-07 17:19 JohnColtrane.zip
$ unzip /home/scott/music/JohnColtrane.zip ;
  ➔ mkdir -p /home/scott/music/coltrane ;
  ➔ mv /home/scott/music/JohnColtrane*.mp3
  ➔ /home/scott/music/coltrane/ ;
  ➔ rm /home/scott/music/JohnColtrane.zip
```

Of course, you can also use this method to introduce short delays as commands run. If you want to take a screenshot of everything you see in your monitor, just run the following command (this assumes you have the ImageMagick package installed, which virtually all Linux distributions do):

```
$ sleep 3 ; import -frame window.tif
```

The `sleep` command in this case waits three seconds, and then the screenshot is taken using `import`. The delay gives you time to minimize your terminal application and bring to the foreground any windows you want to appear in the screenshot. The `;` makes it easy to separate the commands logically so you get maximum use out of them.

---

**CAUTION:** Be very careful when command stacking, especially when deleting or moving files! Make sure what you typed is what you want because the commands will run, one right after another, and you might end up with unexpected surprises.

---

## Run Commands Only If the Previous Ones Succeed

&&

In the previous section, you saw that `;` separates commands, as in this example:

```
$ unzip /home/scott/music/JohnColtrane.zip ;  
➔mkdir -p /home/scott/music/coltrane ;  
➔mv /home/scott/music/JohnColtrane*.*mp3  
➔/home/scott/music/coltrane/ ;  
➔rm /home/scott/music/JohnColtrane.zip
```

What if you fat finger your command, and instead type this:

```
$ unzip /home/scott/JohnColtrane.zip ;
↳mkdir -p /home/scott/music/coltrane ;
↳mv /home/scott/music/JohnColtrane*.*mp3
  /home/scott/music/coltrane/ ;
↳rm /home/scott/music/JohnColtrane.zip
```

Instead of `unzip /home/scott/music/JohnColtrane.zip`, you accidentally enter **`unzip /home/scott/JohnColtrane.zip`**. You fail to notice this, so you go ahead and press Enter, and then get up and walk away. Your computer can't `unzip /home/scott/JohnColtrane.zip` because that file doesn't exist, so it blithely continues onward to the next command (`mkdir`), which it performs without a problem. However, the third command can't be performed (`mv`) because there aren't any MP3 files to move because `unzip` didn't work. Finally, the fourth command runs, deleting the zip file (notice that you provided the correct path this time) and leaving you with no way to recover and start over. Oops!

---

**NOTE:** Don't believe that this chain of events can happen? I did something very similar to it just a few days ago. Yes, I felt like an idiot.

---

That's the problem with using `;`—commands run in sequence, regardless of their successful completion. A better method is to separate the commands with `&&`, which also runs each command one after the other, but only if the previous one completes successfully (technically, each command must return an exit status of 0 for the next one to run). If a command fails, the entire chain of commands stops.

If you'd used `&&` instead of `;` in the sequence of previous commands, it would have looked like this:

```
$ unzip /home/scott/JohnColtrane.zip &&  
↳ mkdir -p /home/scott/music/coltrane &&  
↳ mv /home/scott/music/JohnColtrane.*mp3  
  /home/scott/music/coltrane/ &&  
↳ rm /home/scott/music/JohnColtrane.zip
```

Because the first `unzip` command couldn't complete successfully, the entire process stops. You walk back later to find that your series of commands failed, but `JohnColtrane.zip` still exists, so you can try once again. Much better!

Here are two more examples that show you just how useful `&&` can be. In Chapter 13, "Installing Software," you're going to learn about `apt`, a fantastic way to upgrade your Debian-based Linux box. When you use `apt`, you first update the list of available software, and then find out if there are any upgrades available. If the list of software can't be updated, you obviously don't want to bother looking for upgrades. To make sure the second process doesn't (uselessly) occur, separate the commands with `&&`:

```
# apt-get update && apt-get upgrade
```

Example two: You want to convert a PostScript file to a PDF using the `ps2pdf` command, print the PDF, and then delete the PostScript file. The best way to set up these commands is with `&&`:

```
$ ps2pdf foobar.ps && lpr foobar.pdf && rm foobar.ps
```

If you had instead used `;` and `ps2pdf` failed, the PostScript file would still end up in *nowheresville*, leaving you without a way to start over.

Now are you convinced that `&&` is often the better way to go? If there's no danger that you might delete a file, `;` might be just fine, but if one of your commands involves `rm` or something else from where there is no recovery, you'd better use `&&` and be safe.

## Run a Command Only If the Previous One Fails

||

The `&&` runs each command in sequence only if the previous one completes successfully. The `||` does the opposite: If the first command fails (technically, it returns an exit status that is not 0), only then does the second one run. Think of it like the words *either/or*—either run the first command or the second one.

The `||` is often used to send an alert to an administrator when a process stops. For instance, to ensure that a particular computer is up and running, an administrator might constantly query it with the `ping` command (you'll find out more about `ping` in Chapter 14, "Connectivity"); if `ping` fails, an email is sent to the administrator to let him know.

```
ping -c 1 -w 15 -n 72.14.203.104 ||
{
    echo "Server down" | mail -s 'Server down'
    ↪admin@google.com
}
```

---

**NOTE:** Wondering what the `|` is? Look ahead in this chapter to the "Use the Output of One Command As Input for Another" section to find out what it is and how to use it.

---

With just a bit of thought, you'll start to find many places where `||` can help you. It's a powerful tool that can really prove useful.

## Plug the Output of a Command into Another Command

**\$O**

Command substitution takes the output of a command and plugs it in to another command as though you had typed that output in directly. Surround the initial command that's run—the one that's going to produce the output that's plugged in—with `$()`. An example makes this much clearer.

Let's say you just arrived home from a family dinner, connected your digital camera to your Linux box, pulled the new photos off of it, and now you want to put them in a folder named today's date.

```
$ pwd
/home/scott/photos/family
$ ls -1F
2005-11-01/
2005-11-09/
2005-11-15/
$ date "+%Y-%m-%d"
2005-11-24
$ mkdir $(date "+%Y-%m-%d")
$ ls -1F
2005-11-01/
2005-11-09/
2005-11-15/
2005-11-24/
```

In this example, date "+%Y-%m-%d" is run first, and then the output of that command, 2005-11-24, is used by mkdir as the name of the new directory. This is powerful stuff, and as you look at the shell scripts written by others (which you can easily find all over the Web) you'll find that command substitution is used all over the place.

---

**NOTE:** In the past, you were supposed to surround the initial command with backticks, the ` character at the upper left of your keyboard. Now, however, you're better advised to use the characters used in this section: \$( ).

---

## Understand Input/Output Streams

To take advantage of the information in this chapter, you need to understand that there are three input/output streams for a Linux shell: standard input, standard output, and standard error. Each of these streams has a file descriptor (or numeric identifier), a common abbreviation, and a usual default.

For instance, when you're typing on your keyboard, you're sending input to standard input, abbreviated as stdin and identified as 0. When your computer presents output on the terminal, that's standard output, abbreviated as stdout and identified as 1. Finally, if your machine needs to let you know about an error and displays that error on the terminal, that's standard error, abbreviated as stderr and identified as 2.

Let's look at these three streams using a common command, **ls**. When you enter **ls** on your keyboard, that's using stdin. After typing **ls** and pressing Enter, the list of files and folders in a directory appears as stdout. If

you try to run `ls` against a folder that doesn't exist, the error message that appears on your terminal is courtesy of `stderr`.

Table 4.1 can help you keep these three streams straight.

Table 4.1 The Three Input/Output Streams

File Descriptor (Identifier)	Name	Common Abbreviation	Typical Default
0	Standard input	<code>stdin</code>	Keyboard
1	Standard output	<code>stdout</code>	Terminal
2	Standard error	<code>stderr</code>	Terminal

In this chapter, we're going to learn how to redirect input and output. Instead of having output appear on the terminal, for instance, you can redirect it to another program. Or instead of acquiring input from your typing, a program can get it from a file. After you understand the tricks you can play with `stdin` and `stdout`, there are many powerful things you can do.

## Use the Output of One Command As Input for Another

It's a maxim that Unix is made up of small pieces, loosely joined. Nothing embodies that principle more than the concept of pipes. A pipe is the `|` symbol on your keyboard, and when placed between two commands, it takes the output from the first and uses it as input for the second. In other words, `|` redirects `stdout` so it is sent to be `stdin` for the next command.

Here's a simple example that helps to make this concept clear. You already know about `ls`, and you're going to find out about the `less` command in Chapter 5, "Viewing Files." For now, know that `less` allows you to page through text files one screen at a time. If you run `ls` on a directory that has many files, such as `/usr/bin`, things just zoom by too fast to read. If you pipe the output of `ls` to `less`, however, you can page through the output one screen at a time.

```
$ pwd
/usr/bin
$ ls -l

zipinfo
zipnote
zipsplit
zsoelim
zxpdf
[Listing truncated due to length - 2318 lines!]
$ ls -l | less
411toppm
7z
7za
822-date
a2p
```

You see one screen of results at a time when you pipe the results of `ls -l` through to `less`, which makes it much easier to work with.

Here's a more advanced example that uses two commands discussed later: `ps` and `grep`. You'll learn in Chapter 9, "Finding Stuff: Easy," that `ps` lists running processes and in Chapter 12, "Monitoring System Resources," that `grep` helps find lines in files that match a pattern. Let's say that Firefox is acting strange,

and you suspect that multiple copies are still running in the background. The `ps` command lists every process running on your computer, but the output tends to be lengthy and flashes by in an instant. If you pipe the output of `ps` to `grep` and search for *firefox*, you'll be able to tell immediately if Firefox in fact is still running.

---

NOTE: In order to save space in this listing, I removed the owner, which in every instance was the same person.

---

```
$ ps ux
1504 0.8 4.4 75164 46124 ? S Nov20 1:19 kontakt
19003 0.0 0.1 3376 1812 pts/4 S+ 00:02 0:00 ssh
➔admin@david.hartley.com
21176 0.0 0.0 0 0 ? Z 00:14 0:00
➔[wine-preloader] <defunct>
24953 0.4 3.3 51856 34140 ? S 00:33 0:08 kdeinit:
➔kword /home/scott/documents/clientele/current
[Listing truncated for length]
$ ps ux | grep firefox
scott 8272 4.7 10.9 184072 112704 ? S1 Nov19 76:45
➔/opt/firefox/firefox-bin
```

From 58 lines of output to one—now that's much easier to read!

---

NOTE: Keep in mind that many programs can work with pipes, but not all. The text editor `vim` (or `pico`, `nano`, or `emacs`), for instance, takes over the entire shell so that all input from the keyboard is assumed to be directed at `vim`, while all output is displayed somewhere in the program. Because `vim` has total control of the shell, you can't pipe output using the program. You'll learn to recognize non-pipable programs as you use the shell over time.

---

## Redirect a Command's Output to a File

&gt;

Normally, output goes to your screen, otherwise known as `stdout`. If you don't want output to go to the screen and instead want it to be placed into a file, use the `>` (greater than) character.

```
$ pwd
/home/scott/music
$ ls -lF
Hank_Mobley/
Horace_Silver/
John_Coltrane/
$ ls -lF Hank_Mobley/* > hank_mobley.txt
$ cat hank_mobley.txt
1958_Peckin'_Time/
1960_Roll_Call/
1960_Soul_Station/
1961_Workout/
1963_No_Room_For_Squares/

$ ls -lF
Hank_Mobley/
hank_mobley.txt
Horace_Silver/
John_Coltrane/
```

Notice that before you used the `>`, the file `hank_mobley.txt` didn't exist. When you use `>` and redirect to a file that doesn't already exist, that file is created. Here's the big warning: If `hank_mobley.txt` had already existed, it would have been completely overwritten.

---

**CAUTION:** Once again: Be careful when using redirection, as you could potentially destroy the contents of a file that contains important stuff!

---

## Prevent Overwriting Files When Using Redirection

There's a way to prevent overwriting files when redirecting, however—the `noclobber` option. If you set `noclobber` to `on`, `bash` won't allow redirection to overwrite existing files without your explicit permission. To turn on `noclobber`, use this command:

```
$ set -o noclobber
```

At that point, if you want to use redirection and overwrite a file, use `>|` instead of just `>`, like this:

```
$ pwd
/home/scott/music
$ ls -lF
Hank_Mobley/
hank_mobley.txt
Horace_Silver/
John_Coltrane/
$ ls -lF Hank_Mobley/* > hank_mobley.txt
ERROR
$ ls -lF Hank_Mobley/* >| hank_mobley.txt
$ cat hank_mobley.txt
1958_Peckin'_Time/
1960_Roll_Call/
1960_Soul_Station/
1961_Workout/
1963_No_Room_For_Squares/
```

If you decide you don't like or need `noclobber`, you can turn it off again:

```
$ set +o noclobber
```

To permanently turn on `noclobber`, you need to add `set -o noclobber` to your `.bashrc` file.

## Append a Command's Output to a File

```
>>
```

As you learned previously, the `>` character redirects output from `stdout` to a file. For instance, you can redirect the output of the `date` command to a file easily enough:

```
$ date
Mon Nov 21 21:33:58 CST 2005
$ date > hank_mobley.txt
$ cat hank_mobley.txt
Mon Nov 21 21:33:58 CST 2005
```

Remember that `>` creates a new file if it doesn't already exist and overwrites a file that already exists. If you use `>>` instead of `>`, however, your output is appended to the bottom of the named file (and yes, if the file doesn't exist, it's created).

```
$ cat hank_mobley.txt
Mon Nov 21 21:33:58 CST 2005
$ ls -lF Hank_Mobley/* >> hank_mobley.txt
$ cat hank_mobley.txt
Mon Nov 21 21:33:58 CST 2005
1958_Peckin'_Time/
```

```
1960_Roll_Call/  
1960_Soul_Station/  
1961_Workout/  
1963_No_Room_For_Squares/
```

---

**CAUTION:** Be careful with `>>`. If you accidentally type `>` instead, you won't append, you'll overwrite!

---

## Use a File As Input for a Command

```
<
```

Normally your keyboard provides input to commands, so it is termed `stdin`. In the same way that you can redirect `stdout` to a file, you can redirect `stdin` so it comes from a file instead of a keyboard. Why is this handy? Some commands can't open files directly, and in those cases, the `<` (lesser than) is just what you need.

For instance, normally the `echo` command repeats what you type on `stdin`, as shown here:

```
$ echo "This will be repeated"  
This will be repeated.
```

However, you can use the `<` to redirect input, and the `echo` command uses the contents of a file instead of `stdin`. In this case, let's use the `hank_mobley.txt` file created in the previous section.

```
$ echo < hank_mobley.txt
Mon Nov 21 21:33:58 CST 2005
1958_Peckin'_Time/
1960_Roll_Call/
1960_Soul_Station/
1961_Workout/
1963_No_Room_For_Squares/
```

You're not going to use the `<` all the time, but it will be exactly what you need in a number of situations, so keep it in mind.

## Conclusion

This book started with some simple commands, but now you've learned the building blocks that allow you to combine those commands in new and interesting ways. Going forward, it's going to get more complicated, but it's going to get more powerful as well. The things covered in this chapter are a key factor in gaining that power. Now, onward!