

CHAPTER 1



Hello, World!

Your First Shell Program

A *shell script* is a file containing one or more commands that you would type on the command line. This chapter describes how to create such a file and make it executable. It also covers some other issues surrounding shell scripts, including what to name the files, where to put them, and how to run them.

I will begin with the first program traditionally demonstrated in every computer language: a program that prints “Hello, World!” in your terminal. It’s a simple program, but it is enough to demonstrate a number of important concepts. The code itself is the simplest part of this chapter. Naming the file and deciding where to put it are not complicated tasks, but they are important.

For most of this chapter, you will be working in a terminal. It could be a virtual terminal, a terminal window, or even a dumb terminal. In your terminal, the shell will immediately execute any commands you type (after you press Enter, of course).

You should be in your home directory, which you can find in the variable `$HOME`:

```
echo $HOME
```

You can find the current directory with either the `pwd` command or the `PWD` variable:

```
pwd
echo "$PWD"
```

If you are not in your home directory, you can get there by typing `cd` and pressing Enter at the shell prompt.

The Code

The code is nothing more than this:

```
echo Hello, World!
```

There are three words on this command line: the command itself and two arguments. The command, `echo`, prints its arguments separated by a single space and terminated with a newline.

The File

Before you turn that code into a script, you need to make two decisions: what you will call the file and where you will put it. The name should be unique (that is, it should not conflict with any other commands), and you should put it where the shell can find it.

The Naming of Scripts

Beginners often make the mistake of calling a trial script `test`. To see why that is bad, enter the following at the command prompt:

```
type test
```

The `type` command tells you what the shell will execute (and where it can be found if it is an external file) for any given command. In `bash`, `type -a test` will display all the commands that match the name `test`:

```
$ type test
test is a shell builtin
$ type -a test
test is a shell builtin
test is /usr/bin/test
```

As you can see, a command called `test` already exists; it is used to test file types and to compare values. If you call your script `test`, it will not be run when you type `test` at the shell prompt; the first command identified by `type` will be run instead. (I'll talk more about both `type` and `test` in later chapters.)

Typically, Unix command names are as short as possible. They are often the first two consonants of a descriptive word (for example, `mv` for **m**ove or `ls` for **l**ist) or the first letters of a descriptive phrase (for example, `ps` for **p**rocess **s**tatus or `sed` for **s**tream **e**ditor).

For this exercise, call the script `hw`. Many shell programmers add a suffix, such as `.sh`, to indicate that the program is a shell script. The script doesn't need it, and I use one only for programs that are being developed. My suffix is `-sh`, and when the program is finished, I remove it. A shell script becomes another command and doesn't need to be distinguished from any other type of command.

Selecting a Directory for the Script

When the shell is given the name of a command to execute, it looks for that name in the directories listed in the `PATH` variable. This variable contains a colon-separated list of directories that contain executable commands. This is a typical value for `$PATH`:

```
/bin:/usr/bin:/usr/local/bin:/usr/games
```

If your program is not in one of the `PATH` directories, you must give a pathname, either absolute or relative, for `bash` to find it. An *absolute* pathname gives the location from the root of the filesystem, such as `/home/chris/bin/hw`; a *relative* pathname is given in relation to the current working directory (which should currently be your home directory), as in `bin/hw`.

Commands are usually stored in directories named `bin`, and a user's personal programs are stored in a `bin` subdirectory in the `$HOME` directory. To create that directory, use this command:

```
mkdir bin
```

Now that it exists, it must be added to the `PATH` variable:

```
PATH=$PATH:$HOME/bin
```

For this change to be applied to every shell you open, add it to a file that the shell will *source* when it is invoked. This will be `.bash_profile`, `.bashrc`, or `.profile` depending on how `bash` is invoked. These files are sourced only for interactive shells, not for scripts.

Creating the File and Running the Script

Usually you would use a text editor to create your program, but for a simple script like this, it's not necessary to call up an editor. You can create the file from the command line using redirection:

```
echo echo Hello, World! > bin/hw
```

The greater-than sign (`>`) tells the shell to send the output of a command to the specified file, rather than to the terminal. You'll learn more about redirection in Chapter 2.

The program can now be run by calling it as an argument to the shell command:

```
bash bin/hw
```

That works, but it's not entirely satisfactory. You want to be able to type `hw`, without having to precede it with `bash`, and have the command executed. To do that, give the file execute permissions:

```
chmod +x bin/hw
```

Now the command can be run using just its name:

```
$ hw
Hello, World!
```

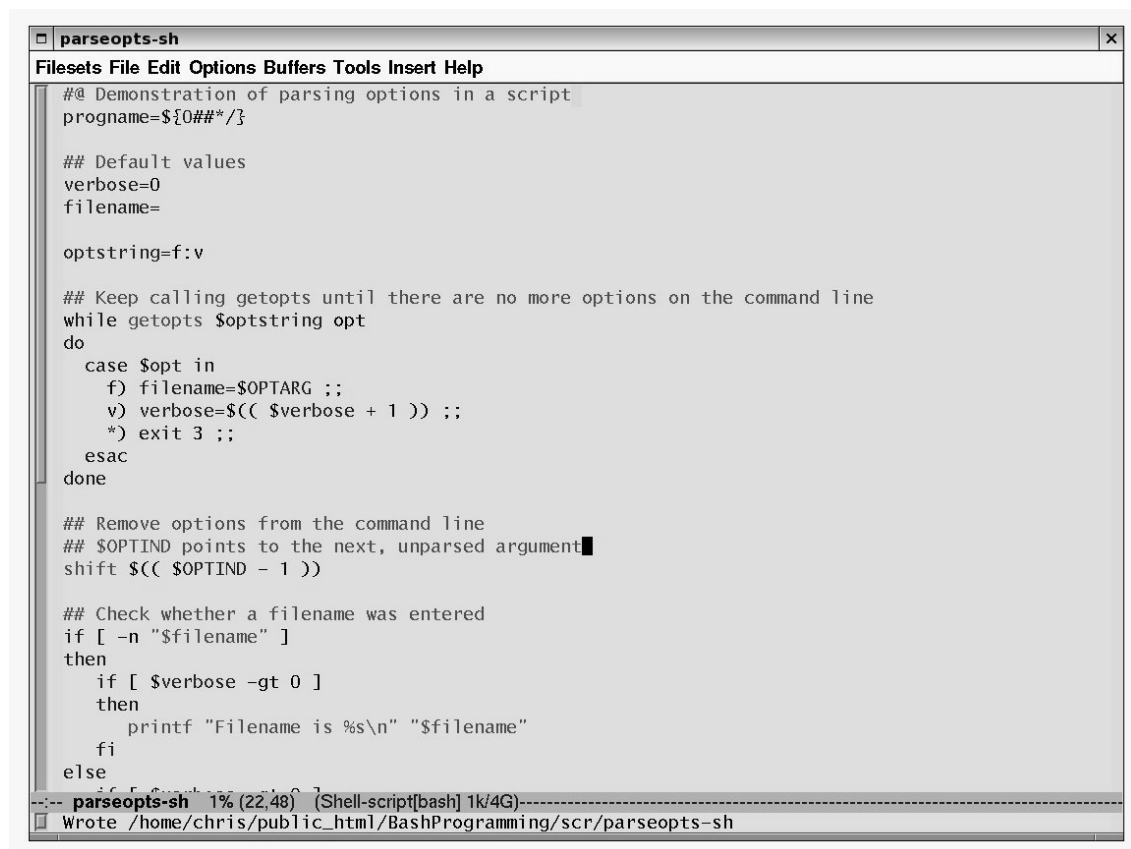
Choosing and Using a Text Editor

For many people, one of the most important pieces of computer software is a word processor. Although I am using one to write this book (OpenOffice.org Writer), it's not something I use often. The last time I used a word processor was four years ago when I wrote my previous book. A text editor, on the other hand, is an indispensable tool. I use one for writing e-mail, Usenet articles, shell scripts, PostScript programs, web pages, and more.

A text editor operates on plain-text files. It stores only the characters you type; it doesn't add any hidden formatting codes. If I type `A` and press Enter in a text editor and save it, the file will contain exactly two characters: `A` and a newline. A word-processor file containing the same text would be thousands of times larger. (With *abiword*, the file contains 2,526 bytes; the OpenOffice.org file contains 8,192 bytes.)

You can write scripts in any text editor, from the basic `e3` or `nano` to the full-featured `emacs` or `nedit`. The better text editors allow you to have more than one file open at a time. They make editing code easier with, for example, syntax highlighting, automatic indentation, autocompletion, spell checking, macros, search and replace, and undo. Ultimately, which editor you choose is a matter of personal preference. I use GNU `emacs` (see Figure 1-1).

■ **Note** In Windows text files, lines end with two characters: a *carriage return* (CR) and a *linefeed* (LF). On Unix systems, such as Linux, lines end with a single linefeed. If you write your programs in a Windows text editor, you must either save your files with Unix line endings or remove the carriage returns afterward.



```

parseopts-sh
Filesets File Edit Options Buffers Tools Insert Help
#@ Demonstration of parsing options in a script
progname=${0##*/}

## Default values
verbose=0
filename=

optstring=f:v

## Keep calling getopt until there are no more options on the command line
while getopt $optstring opt
do
  case $opt in
    f) filename=$OPTARG ;;
    v) verbose=$(( $verbose + 1 )) ;;
    *) exit 3 ;;
  esac
done

## Remove options from the command line
## $OPTIND points to the next, unparsed argument
shift $(( $OPTIND - 1 ))

## Check whether a filename was entered
if [ -n "$filename" ]
then
  if [ $verbose -gt 0 ]
  then
    printf "Filename is %s\n" "$filename"
  fi
else
  if [ $verbose -gt 0 ]
  then
    printf "No filename entered\n"
  fi
fi

```

----- parseopts-sh 1% (22,48) (Shell-script[bash] 1k/4G) -----
 Wrote /home/chris/public_html/BashProgramming/scr/parseopts-sh

Figure 1-1. Shell code in the GNU `emacs` text editor

Building a Better “Hello, World!”

Earlier in the chapter you created a script using redirection. That script was, to say the least, minimalist. All programs, even a one-liner, require documentation. Information should include at least the author, the date, and a description of the command. Open the file `bin/hw` in your text editor, and add the information in Listing 1-1 using *comments*.

Listing 1-1. hw

```
#!/bin/bash
#: Title      : hw
#: Date       : 2008-11-26
#: Author     : "Chris F.A. Johnson" <shell@cfajohnson.com>
#: Version    : 1.0
#: Description : print Hello, World!
#: Options    : None

printf "%s\n" "Hello, World!"
```

Comments begin with an octothorpe, or *hash* (`#`), at the beginning of a *word* and continue until the end of the line. The shell ignores them. I often add a character after the hash to indicate the type of comment. I can then search the file for the type I want, ignoring other comments.

The first line is a special type of comment called a *shebang* or *hash-bang*. It tells the system which *interpreter* to use to execute the file. The characters `#!` must appear at the very beginning of the first line; in other words, they must be the first two bytes of the file for it to be recognized.

Summary

The following are the commands, concepts, and variables you learned in this chapter.

Commands

- `pwd`: Prints the name of the current working directory
- `cd`: Changes the shell’s working directory
- `echo`: Prints arguments separated by a space and terminated by a newline
- `type`: Displays information about a command
- `mkdir`: Creates a new directory
- `chmod`: Modifies the permissions of a file
- `source`: a.k.a. `.` (`dot`): executes a script in the current shell environment
- `printf`: Prints the arguments as specified by a format string

Concepts

- *Script*: This is a file containing commands to be executed by the shell.
- *Word*: A word is a sequence of characters considered to be a single unit by the shell.
- *Output redirection*: You can send the output of a command to a file rather than the terminal using `> FILENAME`.
- *Variables*: These are entities that store values.
- *Comments*: These consist of an unquoted *word* beginning with `#`. All remaining characters on that line constitute a comment and will be ignored.
- *Shebang or hash-bang*: This is a hash and an exclamation mark (`#!`) followed by the path to the interpreter that should execute the file.
- *Interpreter*: This is a program that reads a file and executes the statements it contains. It may be a shell or another language interpreter such as `awk` or `python`.

Variables

- `PWD` contains the pathname of the shell's current working directory.
- `HOME` stores the pathname of the user's home directory.
- `PATH` is a colon-separated list of directories in which command files are stored. The shell searches these directories for commands it is asked to execute.

Exercises

1. Write a script that creates a directory called `bp1` inside `$HOME`. Populate this directory with two subdirectories, `bin` and `scripts`.
2. Write a script to create the "Hello, World!" script, `hw`, in `$HOME/bp1/bin/`; make it executable; and then execute it.