

INFO 101 – Introduction to Computing and Security / Tutorial – week 4

Prof. Dr. Rui Abreu and Prof. Dr. Franz Wotawa
IST, University of Lisbon, Portugal and Technical University of Graz, Austria
rui@computer.org, wotawa@ist.tugraz.at

November 12, 2020

1 Introduction

This time the tutorial has only one objective, i.e.:

- Using the command line shell and shell scripts including regular expressions for improved automation.

In addition, you find at the end (appendix A and B) two files covering the issue of the binary subtractor component. Please have a look at these documents.

2 Using shell scripts

In this week, we discuss the use of shell scripts. In the past weeks we had a look at the bash shell and its commands. We used this commands to search for files with a certain filename (`find`) and had a look the file content (`grep` or `egrep`). We further discussed the use of regular expressions in the context of the bash shell to search for particular patterns. We now will use the information about the bash shell gained for the next step, i.e., writing programs (so called shell scripts) that can be used to put different commands together and use variables to further increase its applicability.

Let us start first, we a simple example. Let us assume that we want to know all files in our home directory that end with ".txt". In particular, we are not so much interested in all files but the number of files we have in our home directory. We can easily provide this information using the following 2 shell commands:

```
find $HOME -name "*.txt" > tmp.txt
wc -l tmp.txt
```

The `find` will search for files with the given pattern and store all results in a file `tmp.txt`. Note that `>` is used to move the output of a command to a given file. Otherwise, the output would be displayed at the standard output, which usually is the shell itself. In the second line, we only count the number of lines. On my computer the output would be:

```
ruimaranhao@greyhound:test ruimaranhao$ wc -l tmp.txt
3450 tmp.txt
```

Let us now put all the commands in a file. We do this using `nano` or `pico`. When calling `nano first_shell_script`, the editor will open an empty file. Add the following content to the first line:

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

Let us now put both line in the file below the first line. Write the content to `first_shell_script` and close the editor. Note that the first line of the script must include the link to the bash shell, which is usually in the director `/bin`. This maybe different in your shell. You may also have a different shell like `ksh` or `sh`. Look for the shell and the directory in your system!

When you use `ls -lsa` in the directory you stored the file, you will see an entry similar to the following:

```
8 -rw-r--r--  1 ruimaranhao  staff      65 18 Nov 13:00 first_shell_script
```

There we see that we can read and write the file, and others can read it, but we cannot execute it. When using `chmod +x first_shell_script`, we are now able to execute the file. Note that `chmod` changes the access mode of a given file or directory. Have a look at the man page as well! It is also possible to set modes only for the user.

```
8 -rwxr-xr-x  1 ruimaranhao  staff      65 18 Nov 13:00 first_shell_script
```

In particular, all users can read and execute the file, which is fine. Let us try to start the script by typing `first_shell_script` in your bash shell. What we see is:

```
ruimaranhao@greyhound:test ruimaranhao$ first_shell_script
-bash: first_shell_script: command not found
```

The reason is that the shell interpreter is using the variable `$PATH` that stores all directories where executable files are searched for. If we want to add the current directory `'.'` to this path, we first have to find out all directories in `$PATH`, which can be done using `echo $PATH`. `echo` is printing the content of a variable or a given string. Afterwards, we can use `$PATH=the result of echo:.` The last `:.` adds the directory `'.'` to the path and `':'` is a separator between paths. Note that this change is only valid for the current session. If you close the shell, you have to do it again. This can be changed in all systems. However, for this purpose, we have to find the right setting file and to modify it. Another, shorter way of adding a new path to `$PATH` is to write `PATH=${PATH}:.` in your shell. In this case, the value of `$PATH` is extended with `:.` and stored in `$PATH`. What is the difference between `$PATH`, `PATH`, and `${PATH}`? `$PATH` and `${PATH}` are used when accessing the value of the variable `PATH`. The `{ }` are used in all situations where there is no space after the variable name.

When now calling `first_shell_script` we obtain an answer from the script. It is also worth noting that when not changing `$PATH`, we can access the shell script by adding the directory, e.g., `./first_shell_script`.

Here is again our first shell script:

```
#!/bin/bash

find $HOME -name "*.txt"  > tmp.txt
wc -l tmp.txt
```

Let us now improve the script. We may not only want to search in our home directory but in other directories as well. We also may not want to search for `".txt"` only. For this purpose shell scripts allow to specify parameters. In a shell script we can access the parameters using the variables `$1` and `$2`. Without any problems, we can add up to 9 parameters. The modified shell script, we call `second_shell_script` now looks like:

```
#!/bin/bash

find $1 -name $2 > tmp.txt
wc -l tmp.txt
```

Note, for changing the script, we can copy the first shell script using the `cp` command and use `nano` for changing its content. Let us try our 2nd version using the command

```
second_shell_script $HOME "*.text".
```

This time we are not searching for files with the extension `".txt"` but with the extension `".text"`. Let us now have a look at the following shell script `all_words.sh`

```
#!/bin/bash
#
# This shell script searches in a file for words that match
# a given pattern and return the number of these words
#
# first parameter : a regular expression
# second parameter: a file

tmp=.tmp.all_words

if [ -e $tmp ]; then
    echo "Temporary file $tmp exist!"
    echo "Please remove this file before calling the script!"
    exit -1
fi

egrep -n $1 -o --color=always $2 > $tmp
wc -l $tmp | cut -c 1-9
rm $tmp
```

This script returns the occurrences a words in a file matching a given pattern. We again generate a temporary file (`.tmp.all_words`) that we remove in the last line. At the beginning of the file, we use a variable `tmp` to store the filename of the temporary file. Note that between the variable, `'='`, and the filename there should be no space! In addition, we use an `if` statement to check whether the temporary file exists. This is done using `-e`. If yes, we use `echo` to output a text and `exit` with `-1` as parameter to leave the shell script. Note that in case the shell successfully terminates `exit` would have a value of `0`. Negative numbers always indicate that an error happened. Sometimes, commands or shell scripts also use other positive numbers to distinguish different behaviors.

In the above shell script, we also added a new command `cut`. This command only gives us back the first 9 characters of the output of `wc`. Hence, we will only return the number of words. However, this solution is not that good, because there are some spaces before the number in the output. And even worst, if the output of `wc` changes and not 9 characters are needed, we would not get the right answer. The following shell script provides us with this feature:

```
#!/bin/bash
#
# This shell script searches in a file for words that match
# a given pattern and return the number of these words
#
# first parameter : a regular expression
# second parameter: a file

tmp=.tmp.all_words

if [ -e $tmp ]; then
    echo "Temporary file $tmp exist!"
    echo "Please remove this file before calling the script!"
    exit -1
fi

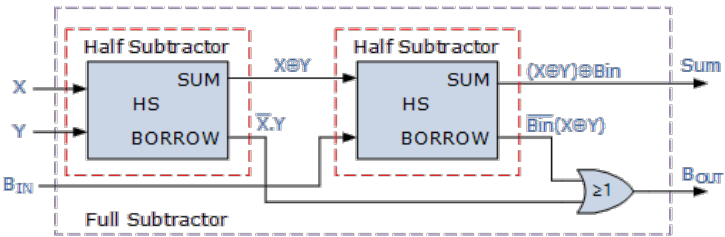
egrep -n $1 -o --color=always $2 > $tmp
no_lines='wc -l $tmp | awk '{ print $1 }''
rm $tmp
echo $no_lines
```

What we see is the new command `awk`. `awk` takes a text and stores all words separated by white spaces in variables `$1`, `$2`, etc. Hence, since we are interested in the first word, we use `$1` and store the result in a variable `no_lines` which we output using `echo`.

In Appendix C, you find more commands and more information including examples for shell scripts. Note that for being able to write the shell script for the 4th assignment, you do not need to read and understand all the different features of a shell script outlined in Appendix C.

Appendix A

From <https://www.electronics-tutorials.ws/combination/binary-subtractor.html>



Binary Subtractor

The Binary Subtractor is another type of combinational arithmetic circuit that produces an output which is the subtraction of two binary numbers

As their name implies, a **Binary Subtractor** is a decision making circuit that subtracts two binary numbers from each other, for example, $X - Y$ to find the resulting difference between the two numbers.

Unlike the Binary Adder which produces a SUM and a CARRY bit when two binary numbers are added together, the *binary subtractor* produces a DIFFERENCE, D by using a BORROW bit, B from the previous column. Then obviously, the operation of subtraction is the opposite to that of addition.

We learnt from our maths lessons at school that the minus sign, “-” is used for a subtraction calculation, and when one number is subtracted from another, a borrow is required if the subtrahend is greater than the minuend. Consider the simple subtraction of the two denary (base 10) numbers below.

123	X	(Minuend)
<u>- 78</u>	Y	(Subtrahend)
45	DIFFERENCE	

We can not directly subtract 8 from 3 in the first column as 8 is greater than 3, so we have to borrow a 10, the base number, from the next column and add it to the minuend to produce 13 minus 8. This “borrowed” 10 is then return back to the subtrahend of the next column once the difference is found. Simple school math’s, borrow a 10 if needed, find the difference and return the borrow.

The subtraction of one binary number from another is exactly the same idea as that for subtracting two decimal numbers but as the *binary number system* is a Base-2 numbering system which uses “0” and “1” as its two independent digits, large binary numbers which are to be subtracted from each other are therefore represented in terms of “0’s” and “1’s”.

Binary Subtraction

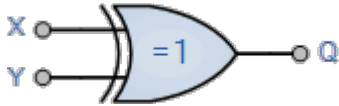
Binary Subtraction can take many forms but the rules for subtraction are the same whichever process you use. As binary notation only has two digits, subtracting a “0” from a “0” or a “1” leaves the result unchanged as $0-0 = 0$ and $1-0 = 1$. Subtracting a “1” from a “1” results in a “0”, but subtracting a “1” from a “0” requires a borrow. In other words $0 - 1$ requires a borrow.

Binary Subtraction of Two Bits

0	1	1 (borrow)	$1 \rightarrow 0$
<u>-0</u>	<u>-0</u>	<u>-1</u>	<u>-1</u>
0	1	0	1

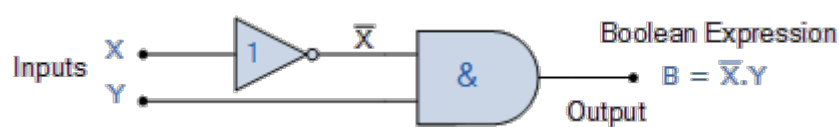
For the simple 1-bit subtraction problem above, if the borrow bit is ignored the result of their binary subtraction resembles that of an Exclusive-OR Gate. To prevent any confusion in this tutorial between a binary subtractor input labelled, B and the resulting borrow bit output from the binary subtractor also being labelled, B, we will label the two input bits as X for the minuend and Y for the subtrahend. Then the resulting truth table is the difference between the two input bits of a single binary subtractor is given as:

2-input Exclusive-OR Gate

Symbol	Truth Table		
 <p>2-input Ex-OR Gate</p>	Y	X	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0

As with the Binary Adder, the difference between the two digits is only a “1” when these two inputs are not equal as given by the Ex-OR expression. However, we need an additional output to produce the borrow bit when input $X = 0$ and $Y = 1$. Unfortunately there are no standard logic gates that will produce an output for this particular combination of X and Y inputs.

But we know that an AND Gate produces an output “1” when both of its inputs X and Y are “1” (HIGH) so if we use an inverter or NOT Gate to complement the input X before it is fed to the AND gate, we can produce the required borrow output when $X = 0$ and $Y = 1$ as shown below.

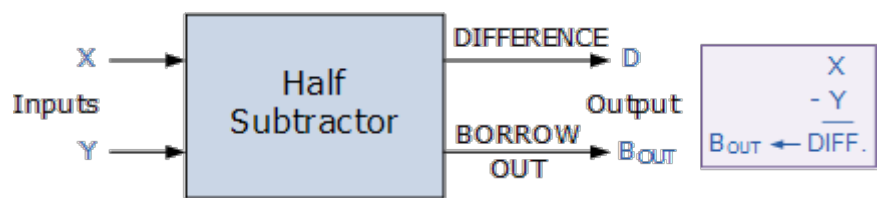


Then by combining the Exclusive-OR gate with the NOT-AND combination results in a simple digital binary subtractor circuit known commonly as the **Half Subtractor** as shown.

A Half Subtractor Circuit

A half subtractor is a logical circuit that performs a subtraction operation on two binary digits. The half subtractor produces a sum and a borrow bit for the next stage.

Half Subtractor with Borrow-out



Symbol	Truth Table			
	Y	X	DIFFERENCE	BORROW
	0	0	0	0
	0	1	1	0
	1	0	1	1
	1	1	0	0

From the truth table of the half subtractor we can see that the DIFFERENCE (D) output is

the result of the Exclusive-OR gate and the Borrow-out (Bout) is the result of the NOT-AND combination. Then the Boolean expression for a half subtractor is as follows.

For the **DIFFERENCE** bit:

$$D = X \text{ XOR } Y = X \oplus Y$$

For the **BORROW** bit

$$B = \text{not-}X \text{ AND } Y = \bar{X}.Y$$

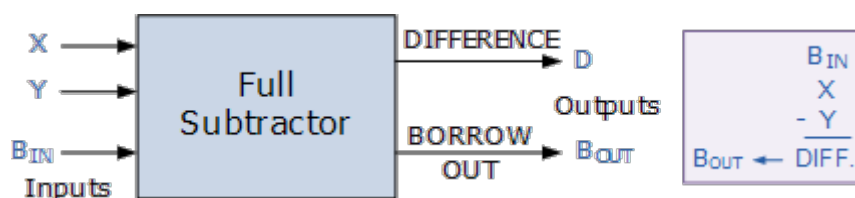
If we compare the Boolean expressions of the half subtractor with a half adder, we can see that the two expressions for the SUM (adder) and DIFFERENCE (subtractor) are exactly the same and so they should be because of the Exclusive-OR gate function. The two Boolean expressions for the binary subtractor BORROW is also very similar to that for the adders CARRY. Then all that is needed to convert a half adder to a half subtractor is the inversion of the minuend input X.

One major disadvantage of the *Half Subtractor* circuit when used as a binary subtractor, is that there is no provision for a “Borrow-in” from the previous circuit when subtracting multiple data bits from each other. Then we need to produce what is called a “full binary subtractor” circuit to take into account this borrow-in input from a previous circuit.

A Full Binary Subtractor Circuit

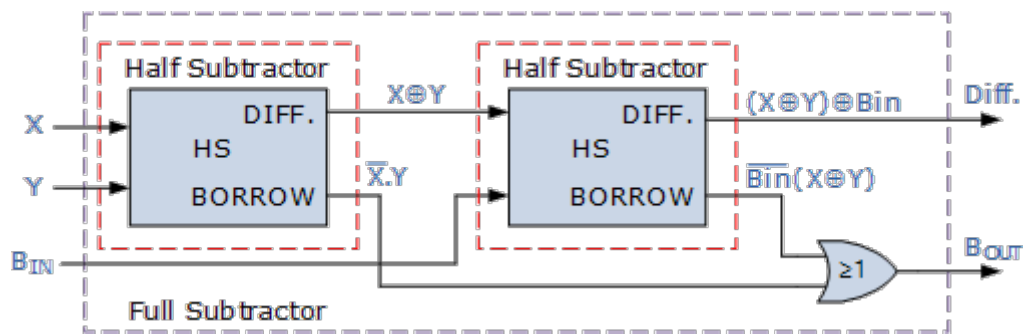
The main difference between the **Full Subtractor** and the previous **Half Subtractor** circuit is that a full subtractor has three inputs. The two single bit data inputs X (minuend) and Y (subtrahend) the same as before plus an additional *Borrow-in* (B-in) input to receive the borrow generated by the subtraction process from a previous stage as shown below.

Full Subtractor Block Diagram



Then the combinational circuit of a “full subtractor” performs the operation of subtraction on three binary bits producing outputs for the difference D and borrow B-out. Just like the binary adder circuit, the full subtractor can also be thought of as two half subtractors connected together, with the first half subtractor passing its borrow to the second half subtractor as follows.

Full Subtractor Logic Diagram



As the full subtractor circuit above represents two half subtractors cascaded together, the truth table for the full subtractor will have eight different input combinations as there are three input variables, the data bits and the *Borrow-in*, B_{IN} input. Also includes the difference output, D and the Borrow-out, B_{OUT} bit.

Full Subtractor Truth Table

Symbol	Truth Table				
	B-in	Y	X	Diff.	B-out
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	1
	0	1	1	0	0
	1	0	0	1	1
	1	0	1	0	0
	1	1	0	0	1
	1	1	1	1	1

Then the Boolean expression for a full subtractor is as follows.

For the **DIFFERENCE** (D) bit:

$$D = (\bar{X}.\bar{Y}.B_{IN}) + (\bar{X}.Y.\bar{B}_{IN}) + (X.\bar{Y}.\bar{B}_{IN}) + (X.Y.B_{IN})$$

which can be simplified too:

$$D = (X \text{ XOR } Y) \text{ XOR } B_{IN} = (X \oplus Y) \oplus B_{IN}$$

For the **BORROW OUT** (B_{OUT}) bit:

$$B_{OUT} = (\bar{X}.\bar{Y}.B_{IN}) + (\bar{X}.Y.\bar{B}_{IN}) + (\bar{X}.Y.B_{IN}) + (X.Y.B_{IN})$$

which will also simplify too:

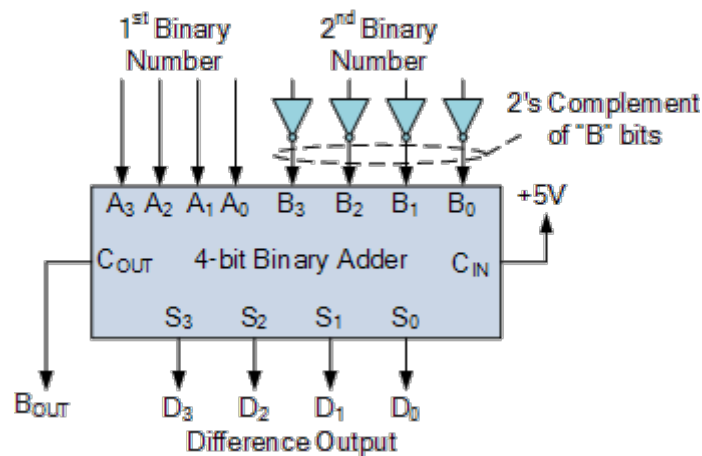
$$B_{OUT} = \bar{X} \text{ AND } Y \text{ OR } (\overline{X \text{ XOR } Y})B_{IN} = \bar{X}.Y + (\overline{X \oplus Y})B_{IN}$$

An n-bit Binary Subtractor

As with the binary adder, we can also have n number of 1-bit full binary subtractor connected or “cascaded” together to subtract two parallel n-bit numbers from each other. For example two 4-bit binary numbers. We said before that the only difference between a full adder and a full subtractor was the inversion of one of the inputs.

So by using an n-bit adder and n number of inverters (NOT Gates), the process of subtraction becomes an addition as we can use two’s complement notation on all the bits in the subtrahend and setting the carry input of the least significant bit to a logic “1” (HIGH).

Binary Subtractor using 2’s Complement



Then we can use a 4-bit full-adder ICs such as the 74LS283 and CD4008 to perform subtraction simply by using two’s complement on the subtrahend, B inputs as $X - Y$ is the same as saying, $X + (-Y)$ which equals X plus the two’s complement of Y .

If we wanted to use the 4-bit adder for addition once again, all we would need to do is set the carry-in (C_{IN}) input LOW at logic “0”. Because we can use the 4-bit adder IC such as the 74LS83 or 74LS283 as a full-adder or a full-subtractor they are available as a single adder/subtractor circuit with a single control input for selecting between the two operations.

Appendix B

Two's Complement

Thomas Finley, April 2000

Contents and Introduction

- [Contents and Introduction](#)
- [Conversion from Two's Complement](#)
- [Conversion to Two's Complement](#)
- [Arithmetic with Two's Complement](#)
- [Why Inversion and Adding One Works](#)

Two's complement is not a complicated scheme and is not well served by anything lengthy. Therefore, after this introduction, which explains what two's complement is and how to use it, there are mostly examples.

Two's complement is the way every computer I know of chooses to represent integers. To get the two's complement negative notation of an integer, you write out the number in binary. You then invert the digits, and add one to the result.

Suppose we're working with 8 bit quantities (for simplicity's sake) and suppose we want to find how -28 would be expressed in two's complement notation. First we write out 28 in binary form.

0 0 0 1 1 1 0 0

Then we invert the digits. 0 becomes 1, 1 becomes 0.

1 1 1 0 0 0 1 1

Then we add 1.

1 1 1 0 0 1 0 0

That is how one would write -28 in 8 bit binary.

Conversion from Two's Complement

Use the number 0xFFFFFFFF as an example. In binary, that is:

1 1

What can we say about this number? It's first (leftmost) bit is 1, which means that this represents a number that is negative. That's just the way that things are in two's complement: a leading 1 means the number is negative, a leading 0 means the number is 0 or positive.

To see what this number is a negative of, we reverse the sign of this number. But how to do that? The class notes say (on 3.17) that to reverse the sign you simply invert the bits (0 goes to 1, and 1 to 0) and add one to the resulting number.

The inversion of that binary number is, obviously:

0 0

Then we add one.

So the negative of 0xFFFFFFFF is 0x00000001, more commonly known as 1. So 0xFFFFFFFF is -1.

Note that this works both ways. If you have -30, and want to represent it in 2's complement, you take the binary representation of 30:

Invert the digits.

And add one.

Converted back into hex, this is 0xFFFFFE2. And indeed, suppose you have this code:

That should yield an output of -30. Try it out if you like.

One of the nice properties of two's complement is that addition and subtraction is made very simple. With a system like two's complement, the circuitry for addition and subtraction can be unified, whereas otherwise they would have to be treated as separate operations.

In the examples in this section, I do addition and subtraction in two's complement, but you'll notice that every time I do actual operations with binary numbers I am always adding.

Suppose we want to add two numbers 69 and 12 together. If we're to use decimal, we see the sum is 81. But let's use binary instead, since that's what the computer uses.

Example 2

Now suppose we want to subtract 12 from 69. Now, $69 - 12 = 69 + (-12)$. To get the negative of 12 we take its binary representation, invert, and add one.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
```

Invert the digits.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1
```

And add one.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0
```

The last is the binary representation for -12. As before, we'll add the two numbers together.

	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Carry Row			
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1		
																			(69)			
+	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0
																			(-12)			
<hr/>																						
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1		
																			(57)			

We result in 57, which is $69 - 12$.

Example 3

Lastly, we'll subtract 69 from 12. Similar to our operation in example 2, $12 - 69 = 12 + (-69)$. The two's complement representation of 69 is the following. I assume you've had enough illustrations of inverting and adding one.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1
```

So we add this number to 12.

									1	1	1	Carry Row				
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
																(12)
+	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
																(-69)
<hr/>																
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
																(-57)

This results in $12 - 69 = -57$, which is correct.

Why Inversion and Adding One Works

Invert and add one. Invert and add one. It works, and you may want to know why. If you don't care, skip this, as it is hardly essential. This is only intended for those curious as to why that rather strange technique actually makes mathematical sense.

Inverting and adding one might sound like a stupid thing to do, but it's actually just a mathematical shortcut of a rather straightforward computation.

Borrowing and Subtraction

Remember the old trick we learned in first grade of "borrowing one's" from future ten's places to perform a subtraction? You may not, so I'll go over it. As an example, I'll do 93702 minus 58358.

$$\begin{array}{r} 93702 \\ - 58358 \\ \hline \end{array}$$

Now, then, what's the answer to this computation? We'll start at the least significant digit, and subtract term by term. We can't subtract 8 from 2, so we'll borrow a digit from the next most significant place (the tens place) to make it 12 minus 8. 12 minus 8 is 4, and we note a 1 digit above the ten's column to signify that we must remember to subtract by one on the next iteration.

$$\begin{array}{r} 1 \\ 93702 \\ - 58358 \\ \hline 4 \end{array}$$

This next iteration is 0 minus 5, and minus 1, or 0 minus 6. Again, we can't do 0 minus 6, so we borrow from the next most significant figure once more to make that 10 minus 6, which is 4.

$$\begin{array}{r} 11 \\ 93702 \\ - 58358 \\ \hline 44 \end{array}$$

This next iteration is 7 minus 3, and minus 1, or 7 minus 4. This is 3. We don't have to borrow this time.

$$\begin{array}{r} 11 \\ 93702 \\ - 58358 \\ \hline 344 \end{array}$$

This next iteration is 3 minus 8. Again, we must borrow to make this 13 minus 8, or 5.

$$\begin{array}{r} 1 \ 11 \\ 93702 \\ - 58358 \\ \hline 5344 \end{array}$$

This next iteration is 9 minus 5, and minus 1, or 9 minus 6. This is 3. We don't have to borrow this time.

$$\begin{array}{r} 1 \ 11 \\ 93702 \\ - 58358 \\ \hline 35344 \end{array}$$

So 93702 minus 58358 is 35344.

Borrowing and it's Relevance to the Negative of a Number

When you want to find the negative of a number, you take the number, and subtract it from zero. Now, suppose we're really stupid, like a computer, and instead of simply writing a negative sign in front of a number A when we subtract A from 0, we actually go through the steps of subtracting A from 0.

Take the following idiotic computation of 0 minus 3:

$$\begin{array}{r}
 000000 \\
 - \quad 3 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 1 \\
 000000 \\
 - \quad 3 \\
 \hline
 7
 \end{array}
 \qquad
 \begin{array}{r}
 11 \\
 000000 \\
 - \quad 3 \\
 \hline
 97
 \end{array}
 \qquad
 \begin{array}{r}
 111 \\
 000000 \\
 - \quad 3 \\
 \hline
 997
 \end{array}
 \qquad
 \begin{array}{r}
 1111 \\
 000000 \\
 - \quad 3 \\
 \hline
 9997
 \end{array}$$

Et cetera, et cetera. We'd wind up with a number composed of a 7 in the one's digit, a 9 in every digit more significant than the 10^0 's place.

The Same in Binary

We can do more or less the same thing with binary. In this example I use 8 bit binary numbers, but the principle is the same for both 8 bit binary numbers (chars) and 32 bit binary numbers (ints). I take the number 75 (in 8 bit binary that is 01001011_2) and subtract that from zero.

Sometimes I am in the position where I am subtracting 1 from zero, and also subtracting another borrowed 1 against it.

$$\begin{array}{r}
 00000000 \\
 - 01001011 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 1 \\
 00000000 \\
 - 01001011 \\
 \hline
 1
 \end{array}
 \qquad
 \begin{array}{r}
 11 \\
 00000000 \\
 - 01001011 \\
 \hline
 01
 \end{array}
 \qquad
 \begin{array}{r}
 111 \\
 00000000 \\
 - 01001011 \\
 \hline
 101
 \end{array}
 \qquad
 \begin{array}{r}
 1111 \\
 00000000 \\
 - 01001011 \\
 \hline
 0101
 \end{array}$$

$$\begin{array}{r}
 11111 \\
 00000000 \\
 - 01001011 \\
 \hline
 10101
 \end{array}
 \qquad
 \begin{array}{r}
 111111 \\
 00000000 \\
 - 01001011 \\
 \hline
 110101
 \end{array}
 \qquad
 \begin{array}{r}
 1111111 \\
 00000000 \\
 - 01001011 \\
 \hline
 0110101
 \end{array}
 \qquad
 \begin{array}{r}
 11111111 \\
 00000000 \\
 - 01001011 \\
 \hline
 10110101
 \end{array}$$

If we wanted we could go further, but there would be no point. Inside of a computer the result of this computation would be assigned to an eight bit variable, so any bits beyond the eighth would be discarded.

With the fact that we'll simply disregard any extra digits in mind, what difference would it make to the end result to have subtracted 01001011 from 100000000 (a one bit followed by 8 zero bits) rather than 0? There is none. If we do that, we wind up with the same result:

$$\begin{array}{r}
 11111111 \\
 100000000 \\
 - 01001011 \\
 \hline
 010110101
 \end{array}$$

So to find the negative of an n-bit number in a computer, subtract the number from 0 or subtract it from 2^n . In binary, this power of two will be a one bit followed by n zero bits.

In the case of 8-bit numbers, it will answer just as well if we subtract our number from $(1 + 11111111)$ rather than 100000000 .

$$\begin{array}{r}
 1 \\
 + 11111111 \\
 - 01001011 \\
 \hline
 \end{array}$$

In binary, when we subtract a number A from a number of all 1 bits, what we're doing is inverting the bits of A. So the subtract operation is the equivalent of inverting the bits of the number. Then,

we add one.

So, to the computer, taking the negative of a number, that is, subtracting a number from 0, is the same as inverting the bits and adding one, which is where the trick comes from.

Thomas Finley 2000

Appendix C

Unix shell scripting with ksh/bash

Course Handout: (last update 22 March 2012)

These notes may be found at <http://www.dartmouth.edu/~rc/classes/ksh>. The online version has many links to additional information and may be more up to date than the printed notes

UNIX shell scripting with ksh/bash

The goals of this class are to enable you to:

- Learn what kinds of problems are suited to shell scripts
- Review the most commonly used Unix commands that are useful in shell scripts.
- Write simple shell scripts using the Bourne, Korn or Bash shells

These notes are intended for use in a 2-part class, total duration 3 hours.

Assumptions:

It is assumed that you already know how to:

- log in and get a command line window (any shell)
- run basic commands, navigate directories
- use simple I/O redirection and pipes
- use a text editor (any one)
- look up details of command usage in man pages

Example commands are shown **like this**. Many commands are shown with links to their full man pages ([sh](#))

Output from commands is shown like `this`; optional items are *[in brackets]*.

Some descriptions in these notes have more detail available, and are denoted like this:

More details of this item would appear here. The printed notes include all of the additional information

Permission is granted to download and use these notes and example scripts, as long as all copyright notices are kept intact. Some of the examples are taken from texts or online resources which have granted permission to redistribute.

These notes are updated from time to time. The "development" set of notes are <http://northstar-www.dartmouth.edu/~richard/classes/ksh> (Dartmouth only)

Richard Brittain, Dartmouth College Computing Services.
© 2003,2004,2010 Dartmouth College.
Comments and questions, contact Richard.Brittain @ dartmouth.edu



Table of Contents

1. [What is a shell script](#)
2. [Why use shell scripts](#)
3. [History](#)
4. [Feature comparison](#)
5. [Other scripting languages](#)
6. [ksh/bash vs sh](#)
7. [Basics](#)
8. [Filename Wildcards](#)
9. [Variables](#)
10. [Preset Variables](#)
11. [Arguments](#)
12. [Shell options](#)
13. [Command substitution](#)
14. [I/O redirection and pipelines](#)
15. [Input and output](#)
16. [Conditional Tests](#)
17. [Conditional Tests \(contd.\)](#)
18. [Flow control](#)
19. [Flow control \(contd.\)](#)
20. [Conditional test examples](#)
21. [Miscellaneous](#)
22. [Manipulating Variables](#)
23. [Functions](#)
24. [Advanced I/O](#)
25. [Wizard I/O](#)
26. [Coprocesses](#)
27. [Arrays](#)
28. [Signals](#)
29. [Security](#)
30. [Style](#)
31. [Examples](#)
32. [Common external commands](#)
33. [References](#)

What is a Shell Script

- A text file containing commands which could have been typed directly into the shell.

There is no difference in syntax between interactive command line use and placing the commands in a file. Some commands are only useful when used interactively (e.g. command line history recall) and other commands are too complex to use interactively.

- The shell itself has limited capabilities -- the power comes from using it as a "glue" language to combine the standard Unix utilities, and custom software, to produce a tool more useful than the component parts alone.
- Any shell can be used for writing a shell script. To allow for this, the first line of every script is:

`#!/path/to/shell` (e.g. **`#!/bin/ksh`**).

The **`#!`** characters tell the system to locate the following pathname, start it up and feed it the rest of the file as input. Any program which can read commands from a file can be started up this way, as long as it recognizes the **`#`** comment convention. The program is started, and then the script file is given to it as an argument. Because of this, the script must be readable as well as executable. Examples are perl, awk, tcl and python.

- Any file can be used as input to a shell by using the syntax:

`ksh myscript`

- If the file is made executable using **`chmod`**, it becomes a new command and available for use (subject to the usual \$PATH search).

`chmod +x myscript`

A shell script can be as simple as a sequence of commands that you type regularly. By putting them into a script, you reduce them to a single command.

Example: ex0 [display](#), [text](#)

```
1: #!/bin/sh
2: date
3: pwd
4: du -k
```

Why use Shell Scripts

- Combine lengthy and repetitive sequences of commands into a single, simple command.
- Generalize a sequence of operations on one set of data, into a procedure that can be applied to any similar set of data.
(e.g. apply the same analysis to every data file on a CD, without needing to repeat the commands)
- Create new commands using combinations of utilities in ways the original authors never thought of.
- Simple shell scripts might be written as shell aliases, but the script can be made available to all users and all processes. Shell aliases apply only to the current shell.
- Wrap programs over which you have no control inside an environment that you can control.
e.g. set environment variables, switch to a special directory, create or select a configuration file, redirect output, log usage, and then run the program.
- Create customized datasets on the fly, and call applications (e.g. matlab, sas, idl, gnuplot) to work on them, or create customized application commands/procedures.
- Rapid prototyping (but avoid letting prototypes become production)

Typical uses

- System boot scripts (/etc/init.d)
- System administrators, for automating many aspects of computer maintenance, user account creation etc.
- Application package installation tools
Other tools may create fancier installers (e.g. tcl/tk), but can not be assumed to be installed already. Shell scripts are used because they are very portable. Some software comes with a complete installation of the tool it wants to use (tcl/tk/python) in order to be self contained, but this leads to software bloat.
- Application startup scripts, especially unattended applications (e.g. started from cron or at)
- Any user needing to automate the process of setting up and running commercial applications, or their own code.

AUTOMATE, AUTOMATE, AUTOMATE

History of Shells

sh

aka "Bourne" shell, written by Steve Bourne at AT&T Bell Labs for Unix V7 (1979). Small, simple, and (originally) *very few internal commands*, so it called external programs for even the simplest of tasks. It is always available on everything that looks vaguely like Unix.

csh

The "C" shell. (Bill Joy, at Berkeley). Many things in common with the Bourne shell, but many enhancements to improve interactive use. The internal commands used only in scripts are **very** different from "sh", and similar (by design) to the "C" language syntax.

tcsh

The "TC" shell. Freely available and based on "csh". It has many additional features to make interactive use more convenient.

We use it as the default interactive shell for new accounts on all of our public systems.

Not many people write scripts in [t]csh. See [Csh Programming Considered Harmful](#) by Tom Christiansen for a discussion of problems with programming csh scripts.

ksh

The "Korn" shell, written by David Korn of AT&T Bell Labs (now AT&T Research). Written as a major upgrade to "sh" and backwards compatible with it, but has many internal commands for the most frequently used functions. It also incorporates many of the features from tcsh which enhance interactive use (command line history recall etc.).

It was slow to gain acceptance because earlier versions were encumbered by AT&T licensing. This shell is now freely available on all systems, but sometimes not installed by default on "free" Unix. There are two major versions. ksh88 was the version incorporated into AT&T SVR4 Unix, and may still be installed by some of the commercial Unix vendors. ksh93 added more features, primarily for programming, and better POSIX compliance.

POSIX 1003.2 Shell Standard.

Standards committees worked over the Bourne shell and added many features of the Korn shell (ksh88) and C shell to define a standard set of features which all compliant shells must have.

On most systems, /bin/sh is now a POSIX compliant shell. Korn shell and Bash are POSIX compliant, but have many features which go beyond the standard. On Solaris, the POSIX/XPG4 commands which differ slightly in behaviour from traditional SunOS commands are located in /usr/xpg4/bin

bash

The "Bourne again" shell. Written as part of the GNU/Linux Open Source effort, and the default shell for Linux and Mac OS-X. It is a functional clone of sh, with additional features to enhance interactive use, add POSIX compliance, and partial ksh compatability.

zsh

A freeware functional clone of sh, with parts of ksh, bash and full POSIX compliance, and many new interactive command-line editing features. It was installed as the default shell on early MacOSX systems.

Comparison of shell features

All the shells just listed share some common features, and the major differences in syntax generally only affect script writers. It is not unusual to use one shell (e.g. tcsh) for interactive use, but another (sh or ksh) for writing scripts.

Core Similarities (and recap of basic command line usage)

Each of these items is discussed in more detail later.

- Parse lines by whitespace, search for external commands using `$PATH`.
- Can run a shell script using `shellname scriptfile`, or run a single command using `shellname -c "command"`
- Pass expanded command line arguments to programs; get exit status back.
- Pass environment variables to programs.
- Expand filename wildcards using `[]*?`. Each shell has some additional wildcard metacharacters, but these are common to all shells.
- Standard I/O redirection and piping with `<, >, >>, |`
- A few internal functions (`cd`)
- Backgrounding commands with `&`
- Quoting rules: "double quotes" protect most things, but allow `$var` interpretation; 'single quotes' protect all metacharacters from interpretation.
- Home directory expansion using `~user` (except for `sh`)
- `#` comments
- Command substitution using ``command`` (backtics)
- Expand variables using `$varname` syntax
- Conditional execution using `&&` and `||`
- Line continuation with `"\"`

Principal Differences

between sh (+derivatives), and csh (+derivatives).

- Syntax of all the flow control constructs and conditional tests.
- Syntax for string manipulation inside of scripts
- Syntax for arithmetic manipulation inside of scripts
- Syntax for setting local variables (used only in the script) and environment variables (which are passed to child processes).
`setenv` vs `export`
- Syntax for redirecting I/O streams other than stdin/stdout
- Login startup files (`.cshrc` and `.login`, vs `.profile`) and default options
- Reading other shell scripts into the current shell (`source filename`, vs `. filename`)
- Handling of signals (interrupts)

Other Scripting Languages

There are many other programs which read a file of commands and carry out a sequence of actions. The "**#!/path/to/program**" convention allows any of them to be used as a scripting language to create new commands. Some are highly specialized, and some are much more efficient than the equivalent shell scripts at certain tasks. There is never only one way to perform a function, and often the choice comes down to factors like:

- what is installed already - many other scripting languages are not available by default
- what similar code already exists
- what you are most familiar with and can use most efficiently. Your time is always more expensive than computer cycles.

Some major players (all of these are freely available) in the general purpose scripting languages are:

- **awk**

A pattern matching and data (text and numeric) manipulation tool. Predates perl. Installed on all Unix systems. Often used in combination with shell scripts.

- **perl**

The most used scripting language for Web CGI applications and system administration tasks. Perl is harder to learn, and is usually installed by default now. It is more efficient and has an enormous library of functions available. You could use Perl for almost all scripting tasks, but the syntax is very different to the shell command line

- **python**

An object-oriented scripting language. Commonly installed by default on modern systems.

- **tcl/tk**

Tool Command Language. Another general purpose scripting language. The "tk" component is a scripted interface to standard X-windows graphical components, so the combination is often used to create graphical user interfaces.

Ksh93 can be extended by linking to shared libraries providing additional internal commands. One example of an extended shell is **tksh** which incorporates Tcl/Tk with ksh and allows generation of scripts using both languages. It can be used for prototyping GUI applications.

ksh/bash vs sh

Ksh and bash are both supersets of sh. For maximum portability, even to very old computers, you should stick to the commands found in sh. Where possible, ksh or bash-specific features will be noted in the following pages. In general, the newer shells run a little faster and scripts are often more readable because logic can be expressed more cleanly using the newer syntax. Many commands and conditional tests are now internal.

The philosophy of separate Unix tools each performing a single operation was followed closely by the designers of the original shell, so it had very few internal commands and used external tools for very trivial operations (like `echo` and `()`). Ksh and bash internally perform many of the basic string and numeric manipulations and conditional tests. Occasional problems arise because the internal versions of some commands like `echo` are not fully compatible with the external utility they replaced.

The action taken every time a shell needs to run an external program is to locate the program (via `$PATH`), `fork()`, which creates a second copy of the shell, adjust the standard input/output for the external program, and `exec()`, which replaces the second shell with the external program. This process is computationally expensive (relatively), so when the script does something trivial many times over in a loop, it saves a lot of time if the function is handled internally.

If you follow textbooks on Bourne shell programming, all of the advice should apply no matter which of the Bourne-derived shells you use. Unfortunately, many vendors have added features over the years and achieving complete portability can be a challenge. Explicitly writing for ksh (or bash) and insisting on that shell being installed, can often be simpler.

The sh and ksh *man* pages use the term *special command* for the internal commands - handled by the shell itself.

Basic sh script syntax

The most basic shell script is a list of commands exactly as could be typed interactively, prefaced by the `#!` magic header. All the parsing rules, filename wildcards, `$PATH` searches etc., which were summarized above, apply.

In addition:

`#` as the first non-whitespace character on a line
flags the line as a comment, and the rest of the line is completely ignored. Use comments liberally in your scripts, as in all other forms of programming.

`\` as the last character on a line
causes the following line to be logically joined before interpretation. This allows single very long commands to be entered in the script in a more readable fashion. You can continue the line as many times as needed.

This is actually just a particular instance of `\` being to *escape*, or remove the special meaning from, the following character.

`;` as a separator between words on a line
is interpreted as a newline. It allows you to put multiple commands on a single line. There are few occasions when you **must** do this, but often it is used to improve the layout of compound commands.

Example: ex1 [display](#), [text](#)

```

1: #!/bin/ksh
2: # For the purposes of display, parts of the script have
3: # been rendered in glorious technicolor.
4: ## Some comments are bold to flag special sections
5:
6: # Line numbers on the left are not part of the script.
7: # They are just added to the HTML for reference.
8:
9: # Built-in commands and keywords (e.g. print) are in blue
10: # Command substitutions are purple. Variables are black
11: print "Disk usage summary for $USER on `date`"
12:
13: # Everything else is red - mostly that is external
14: # commands, and the arguments to all of the commands.
15: print These are my files      # end of line comment for print
16: # List the files in columns
17: ls -C
18: # Summarize the disk usage
19: print
20: print Disk space usage
21: du -k
22: exit 0

```

Exit status

Every command (program) has a *value* or *exit status* which it returns to the calling program. This is separate from any output generated. The exit status of a shell script can be explicitly set using `exit N`, or it defaults to the value of the last command run.

The exit status is an integer 0-255. Conventionally 0=success and any other value indicates a problem. Think of it as only one way for everything to work, but many possible ways to fail. If the command was terminated by a signal, the value is 128 plus the signal value.

Filename Wildcards

The following characters are interpreted by the shell as filename wildcards, and any word containing them is replaced by a sorted list of all the matching files.

Wildcards may be used in the directory parts of a pathname as well as the filename part. If no files match the wildcard, it is left unchanged. Wildcards are not full regular expressions. Sed, grep, awk etc. work with more flexible (and more complex) string matching operators.

- ***
Match zero or more characters.
 - ?**
Match any single character
 - [...]**
Match any single character from the bracketed set. A range of characters can be specified with **[-]**
 - [!...]**
Match any single character NOT in the bracketed set.
- An initial "." in a filename does not match a wildcard unless explicitly given in the pattern. In this sense filenames starting with "." are hidden. A "." elsewhere in the filename is not special.
 - Pattern operators can be combined

Example:

chapter[1-5].* could match **chapter1.tex**, **chapter4.tex**, **chapter5.tex.old**. It would not match **chapter10.tex** or **chapter1**

Shell Variables

Scripts are not very useful if all the commands and options and filenames are explicitly coded. By using variables, you can make a script generic and apply it to different situations. Variable names consist of letters, numbers and underscores ([a-zA-Z0-9_], cannot start with a number, and are case sensitive. Several special variables (always uppercase names) are used by the system -- resetting these may cause unexpected behaviour. Some special variables may be read-only. Using lowercase names for your own variables is safest.

Setting and exporting variables

srcfile=dataset1

Creates (if it didn't exist) a variable named "srcfile" and sets it to the value "dataset1". If the variable already existed, it is overwritten. Variables are treated as text strings, unless the context implies a numeric interpretation. You can make a variable always be treated as a number. Note there must be **no spaces** around the "=".

set

Display all the variables currently set in the shell

unset srcfile

Remove the variable "srcfile"

srcfile=

Give the variable a null value, (not the same as removing it).

export srcfile

Added srcfile to the list of variables which will be made available to external program through the environment. If you don't do this, the variable is local to this shell instance.

export

List all the variables currently being exported - this is the environment which will be passed to external programs.

Using variables

\$srcfile

Prefacing the variable name with \$ causes the *value* of the variable to be substituted in place of the name.

\${srcfile}

If the variable is not surrounded by whitespace (or other characters that can't be in a name), the name must be surrounded by "{}" braces so that the shell knows what characters you intend to be part of the name.

Example:

```
datafile=census2000
# Tries to find $datafile_part1, which doesn't exist
echo $datafile_part1.sas
# This is what we intended
echo ${datafile}_part1.sas
```

Conditional modifiers

There are various ways to conditionally use a variable in a command.

\${datafile-default}

Substitute the value of **\$datafile**, if it has been defined, otherwise use the string "default". This is an easy way to allow for optional variables, and have sensible defaults if they haven't been set. If **datafile** was undefined, it remains so.

\${datafile=default}

Similar to the above, except if **datafile** has not been defined, set it to the string "default".

\${datafile+default}

If variable **datafile** has been defined, use the string "default", otherwise use null. In this case the actual value **\$datafile** is not used.

\${datafile?"error message"}

Substitute the value of **\$datafile**, if it has been defined, otherwise display **datafile: error message**. This is used for diagnostics when a variable should have been set and there is no sensible default value to use.

Placing a colon (:) before the operator character in these constructs has the effect of counting a *null* value the same as an undefined variable. Variables may be given a null value by setting them to an empty string, e.g. **datafile=** .

Example: echo \${datafile:-mydata.dat}

Echo the value of variable **datafile** if it has been set and is non-null, otherwise echo "mydata.dat".

Variable assignment command prefix

It is possible to export a variable just for the duration of a single command using the syntax:

var=value command args

Preset Shell Variables

Several special variables are used by the system -- you can use these, but may not be able to change them. The special variables use uppercase names, or punctuation characters. Some variables are set by the login process and inherited by the shell (e.g. **\$USER**), while others are used only by the shell.

Try running **set** or **env**

These are some of the more commonly used ones:

Login environment

\$USER, \$LOGNAME

Preset to the currently logged-in username.

\$PATH

The list of directories that will be searched for external commands. You can change this in a script to make sure you get the programs you intend, and don't accidentally get other versions which might have been installed.

\$TERM

The terminal type in which the shell session is currently executing. Usually "xterm" or "vt100". Many programs need to know this to figure out what special character sequences to send to achieve special effects.

\$PAGER

If set, this contains the name of the program which the user prefers to use for text file viewing. Usually set to "more" or "less" or something similar. Many programs which need to present multipage information to the user will respect this setting (e.g. **man**). This isn't actually used by the shell itself, but shell scripts should honour it if they need to page output to the user.

\$EDITOR

If set, this contains the name of the program which the user prefers to use for text file editing. A program which needs to have the user manually edit a file might choose to start up this program instead of some built-in default (e.g. "crontab -e". This also determines the default command-line-editing behaviour in interactive shells.

Shell internal settings

\$PWD

Always set the current working directory (readonly)

\$OLDPWD

The previous directory (before the most recent **cd** command). However, changing directories in a script is often dangerous.

\$? (readonly)

Set to the exit status of the last command run, so you can test success or failure. Every command resets this so it must be saved immediately if you want to use it later.

\$-

Set to the currently set options flags.

\$IFS

Internal Field Separators: the set of characters (normally space and tab) which are used to parse a command line into separate arguments. This may be set by the user for special purposes, but things get very confusing if it isn't changed back.

Process ID variables

\$\$ (readonly)

Set to the process ID of the current shell - useful in making unique temporary files, e.g. /tmp/\$0.\$\$

\$PPID (readonly)

Set to the process ID of the parent process of this shell - useful for discovering how the script was called.

\$! (readonly)

Set to the process ID of the last command started in background - useful for checking on background processes.

ksh/bash additional features

\$SECONDS (readonly)

Integer number of seconds since this shell was started. Can be used for timing commands.

\$RANDOM

Every time it is evaluated, **\$RANDOM** returns a random integer in the range 0-32k. **RANDOM** may be set to "seed" the random number generator.

\$LINENO (readonly)

Always evaluates to the current line number of the script being executed - useful for debugging.

Command Line (positional) arguments

To customize the behaviour of a script at run time, you can give it any number of arguments on the command line.

These are often filenames, but can be interpreted by the script in any way. Options are often specified using the "-flag" convention used by most Unix programs, and a ksh command [getopts](#) is available to help parse them.

The shell expands wildcards and makes variable and command substitutions as normal, then parses the resulting words by whitespace (actually special variable `$IFS`), and places the resulting text strings into the *positional variables* as follows:

`$0, $1, $2, ... $9`

The first 9 arguments are made available directly as `$1-$9`. To access more than 9, use `shift`, or `*, $*`. The variable `$0` contains the name of the script itself.

`${10}, ${11}, ...`

Positional arguments greater than 9 are set by ksh and bash. Remember to use braces to refer to them.

`shift`

discard `$1` and renumber all the other variables. "`shift N`" will shift `N` arguments at once.

`$#`

contains the number of arguments that were set (not including `$0`).

`$*`

contains all of the arguments in a single string, with one space separating them.

`$@`

similar to `$*`, but if used in quotes, it effectively quotes each argument and keeps them separate. If any argument contains whitespace, the distinction is important.

e.g. if the argument list is: `a1 a2 "a3 which contains spaces" a4`

then: `$1=a1, $2=a2, $3=a3 which contains spaces, $4=a4`

and: `$*=a1 a2 a3 which contains spaces a4`

and: `"$@"="a1" "a2" "a3 which contains spaces" "a4"`

Only using the form `"$@"` preserves quoted arguments. If the arguments are being passed from the script directly to some other program, it may make a big difference to the meaning.

Example: ex7 [display, text](#)

```
1: #!/bin/sh
2: #
3: # Check positional argument handling
4: echo "Number of arguments: $#"
```

```
5: echo "\$0 = $0"
6:
7: echo "Loop over \$*"
8: for a in $*; do
9:     echo "\"$a\""
```

```
10: done
11:
12: echo "Loop over \"\$@\""
13: for a in "$@"; do
14:     echo "\"$a\""
```

```
15: done
```

Setting new positional arguments

The `set` command, followed by a set of arguments, creates a new set of positional arguments. This is often used, assuming the original arguments are no longer needed, to parse a set of words (possibly using different field separators). Arguments may be reset any number of times.

Example: ex2 [display, text](#)

```
1: #!/bin/sh
2: # Find an entry in the password file
3: pwent=`grep '^root:' /etc/passwd`
4: # Turn off globbing - passwd lines often contain '*'
5: set -o noglob
6: # The "full name" and other comments are in
7: # field 5, colon delimited. Get this field using shell word splitting
8: OIFS=$IFS; IFS=: ; set $pwent; IFS=$OIFS
9: echo $5
```

Example: pickrandom [display, text](#)

Selects a random file from a directory. Uses the ksh RANDOM feature.

```
1: #!/bin/ksh
2:
```



```
3: # Select a random image from the background logo collection
4: # This could be used to configure a screen saver, for example.
5: #
6: # This works even if the filenames contain spaces.
7:
8: # switch to the logos directory to avoid long paths
9: logos=/afs/northstar/common/usr/lib/X11/logos/backgrounds
10: cd $logos
11:
12: # '*' is a filename wildcard to match all files in the current directory
13: set *
14:
15: # Use the syntax for arithmetic expressions. "%" is the modulo operator
16: # Shift arguments by a random number between 0 and the number of files
17: shift $((RANDOM % $#))
18:
19: # Output the resulting first argument
20: echo "$logos/$1"
```

Shell options

Startup options. **ksh -options scriptname**

- x
echo line to stderr before executing it
- n
read commands and check for syntax errors, but do not execute.
- a
all variables are automatically exported
- f
disable wildcard filename expansion (globbing)
- set -x**
Set an option within a shell script
- \$-**
contains the currently set option letters

There are many other options, not often needed. Options in ksh and bash can also be set using long names (e.g. **-o noglob** instead of **-f**). Many options are unique to ksh or bash.

Command Substitution

sh syntax

``command``

A command (plus optional arguments) enclosed in backticks is executed and the standard output of that command is substituted. If the command produces multiline output, the newlines are retained. If the resultant string is displayed, unquoted, using **echo**, newlines and multiple spaces will be removed.

ksh/bash syntax

`$(command)`

This syntax is functionally the same as backticks, but commands can be more easily nested.

`$(<file)`

This is equivalent to ``cat file``, but implemented internally for efficiency.

Example: ex3 [display](#), [text](#)

```
1: #!/bin/ksh
2:
3: echo Today is `date`
4:
5: file=/etc/hosts
6: echo The file $file has $(wc -l < $file) lines
7:
8: hostname -s > myhostname
9: echo This system has host name $(<myhostname)
```

I/O redirection and pipelines

Any simple command (or shell function, or compound command) may have its input and output redirected using the following operators. This is performed by the shell *before* the command is run.

Output redirection

> filename

Standard output (file descriptor 1) is redirected to the named file. The file is overwritten unless the **noclobber** option is set. The file is created if it does not exist.

The special device file **/dev/null** can be used to explicitly discard unwanted output. Reading from **/dev/null** results in an End of File status.

>> filename

Standard output is appended to the named file. The file is created if it does not exist.

>| filename

Output redirect, and override the *noclobber* option, if set.

Input redirection

< filename

Standard input (file descriptor 0) is redirected to the named file. The file must already exist.

Command pipelines

command | command [| command ...]

Pipe multiple commands together. The standard output of the first command becomes the standard input of the second command. All commands run simultaneously, and data transfer happens via memory buffers. This is one of the most powerful constructs in Unix. *Compound* commands may also be used with pipes. Pipes play very nicely with multiprocessor systems.

No more than one command in a pipeline should be interactive (attempt to read from the terminal). This construct is much more efficient than using temporary files, and most standard Unix utilities are designed such that they work well in pipelines.

The exit status of a pipeline is the exit status of the last command. In compound commands, a pipeline can be used anywhere a simple command could be used.

Input and Output

Shell scripts can generate output directly or read input into variables using the following commands:

Script output

echo

Print arguments, separated by spaces, and terminated by a newline, to stdout. Use quotes to preserve spacing. Echo also understands C-like escape conventions.

Beware that the shell may process backslashes before **echo** sees them (may need to double backslash). Internal in most shells, but was originally external.

\b backspace **\c** print line without new-line (some versions)

\f form-feed **\n** new-line

\r carriage return **\t** tab

\v vertical tab **** backslash

\On where n is the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number representing that character.

-n

suppress newline

print (ksh internal)

Print arguments, separated by spaces, and terminated by a newline, to stdout. Print observes the same escape conventions as echo.

-n

suppress newline

-r

raw mode - ignore \-escape conventions

-R

raw mode - ignore \-escape conventions and -options except -n.

Script input

read var1 var2 rest

read a line from stdin, parsing by \$IFS, and placing the words into the named variables. Any left over words all go into the last variable. A \ as the last character on a line removes significance of the newline, and input continues with the following line.

-r

raw mode - ignore \-escape conventions

Example: ex4a [display, text](#)

```
1: #!/bin/sh
2: echo "Testing interactive user input: enter some keystrokes and press return"
3: read x more
4: echo "First word was \"${x}\""
5: echo "Rest of the line (if any) was \"${more}\""
```

Conditional tests for [...] and [[...]] commands

Most of the useful flow-control operators involve making some conditional test and branching on the result (true/false). The test can be either the `test` command, or its alias, `[`, or the ksh/bash built-in `[[...]]` command, which has slightly different options, or it can be *any command which returns a suitable exit status*. Zero is taken to be "True", while any non-zero value is "False". Note that this is backwards from the C language convention.

File tests

-e file
True if *file* exists (can be of any type).

-f file
True if *file* exists and is an ordinary file.

-d file
True if *file* exists and is a directory.

-r file
True if *file* exists and is readable
Similarly, **-w** = writable, **-x** = executable, **-L** = is a symlink.

-s file
True if *file* exists and has size greater than zero

-t filedescriptor
True if the open *filedescriptor* is associated with a terminal device. E.g. this is used to determine if standard output has been redirected to a file.

Character string tests

-n "string"
true if *string* has non-zero length

-z "string"
true if *string* has zero length

With `[`, the argument must be quoted, because if it is a variable that has a null value, the resulting expansion (`[-z]`) is a syntax error. An expansion resulting in `"` counts as a null string.
For `[` only, a quoted string alone is equivalent to the `-n` test, e.g. `["$var"]`. In older shells for which `[` is an external program, the only way to test for a null string is:
if ["x\$var" = "x"]
This is rarely needed now, but is still often found.

\$variable = text
True if *\$variable* matches *text*.

\$variable < text
True if *\$variable* comes before (lexically) *text*
Similarly, **>** = comes after

More conditional tests for [...] and [[...]] commands

Arithmetic tests

`$variable -eq number`

True if *\$variable*, interpreted as a number, is equal to *number*.

`$variable -ne number`

True if *\$variable*, interpreted as a number, is **not** equal to *number*.

Similarly, **`-lt`** = less than, **`-le`** = less than or equal, **`-gt`** = greater than, **`-ge`** = greater than or equal

Additional tests for [[...]] (ksh and bash)

`$variable = pattern`

True if *\$variable* matches *pattern*. If *pattern* contains no wildcards, then this is just an exact text match. The same wildcards as used for filename matching are used.

The *pattern* must not be quoted. Since `[[...]]` is internal to the shell, the pattern in this case is treated differently and not filename-expanded as an external command would require.

`file1 -nt file2`

True if *file1* is newer than *file2*.

Similarly **`-ot`** = older than

`file1 -ef file2`

true if *file1* is effectively the same as *file2*, after following symlinks and hard links.

Negating and Combining tests

Tests may be negated by prepending the **`!`** operator, and combined with boolean **AND** and **OR** operators using the syntax:

`conditional -a conditional, conditional -o conditional`

AND and **OR** syntax for **`test`** and **`[`**

`conditional && conditional, conditional || conditional`

AND and **OR** syntax for **`[[...]]`**

Parentheses may be inserted to resolve ambiguities or override the default operator precedence rules.

Examples:

```
if [[ -x /usr/local/bin/lserve && \
    -w /var/logs/lserve.log ]]; then
    /usr/local/bin/lserve >> /var/logs/lserve.log &
fi

pwent=`grep '^richard:' /etc/passwd`
if [ -z "$pwent" ]; then
    echo richard not found
fi
```

Flow Control and Compound Commands

A *list* in these descriptions is a simple command, or a pipeline. The value of the *list* is the value of the last simple command run in it.

A *list* can also be a set of simple commands or pipelines separated by ";,&&,&&||,|&". For the compound commands which branch on the success or failure of some *list*, it is usually **|** or **|&**, but can be anything.

Conditional execution: if/else

list && list

Execute the first *list*. If true (success), execute the second one.

list || list

Execute the first *list*. If false (failure), execute the second one.

Example:

```
mkdir tempdir && cp workfile tempdir
```

```
sshd || echo "sshd failed to start"
```

You can use both forms together (with care) - they are processed left to right, and && must come first.

Example:

```
mkdir tempdir && cp workfile tempdir || \
echo "Failed to create tempdir"
```

if list; then list ; elif list; then list; else list; fi

Execute the first *list*, and if true (success), execute the "then" list, otherwise execute the "else" list. The "elif" and "else" lists are optional.

Example:

```
if [ -r $myfile ]
then
    cat $myfile
else
    echo $myfile not readable
fi
```

Looping: 'while' and 'for' loops

while list; do list; done

until list; do list; done

Execute the first *list* and if true (success), execute the second *list*. Repeat as long as the first *list* is true. The **until** form just negates the test.

Example: ex4 [display](#), [text](#)

```
1: #!/bin/ksh
2: count=0
3: max=10
4: while [[ $count -lt $max ]]
5: do
6:     echo $count
7:     count=$((count + 1))
8: done
9: echo "Value of count after loop is: $count"
```

for identifier [in words]; do; list; done

Set *identifier* in turn to each word in *words* and execute the *list*. Omitting the "in words" clause implies using \$@, i.e. the *identifier* is set in turn to each positional argument.

Example:

```
for file in *.dat
do
    echo Processing $file
done
```

As with most programming languages, there are often several ways to express the same action. Running a command and then explicitly examining **\$?** can be used instead of some of the above.

Compound commands can be thought of as running in an implicit subshell. They can have I/O redirection independent of the rest of the script. Setting of variables in a real subshell does not leave them set in the parent script. Setting variables in implicit subshells varies in behaviour among shells. Older **sh** could not set variables in an implicit subshell and then use them later, but current **ksh** can

do this (mostly).

Example: ex11 [display, text](#)

Reading a file line by line. The book by Randal Michael contains 12 example ways to read a file line by line, which vary tremendously in efficiency. This example shows the simplest and fastest way.

```
1: #!/bin/sh
2:
3: # Demonstrate reading a file line-by-line, using I/O
4: # redirection in a compound command
5: # Also test variable setting inside an implicit subshell.
6: # Test this under sh and ksh and compare the output.
7:
8: line="TEST"
9: save=
10:
11: if [ -z "$1" ]; then
12:     echo "Usage: $0 filename"
13: else
14:     if [ -r $1 ]; then
15:         while read line; do
16:             echo "$line"
17:             save=$line
18:         done < $1
19:     fi
20: fi
21: echo "End value of \$line is $line"
22: echo "End value of \$save is $save"
```

Flow Control and Compound Commands (contd.)

Case statement: pattern matching

case word in pattern) list;; esac

Compare *word* with each *pattern*) in turn, and executes the first *list* for which the *word* matches. The *patterns* follow the same rules as for filename wildcards.

(ksh and bash only) A pattern-list is a list of one or more patterns separated from each other with a |. Composite patterns can be formed with one or more of the following:

? (pattern-list)

Optionally matches any one of the given patterns.

*** (pattern-list)**

Matches zero or more occurrences of the given patterns.

+ (pattern-list)

Matches one or more occurrences of the given patterns.

@ (pattern-list)

Matches exactly one of the given patterns.

! (pattern-list)

Matches anything, except one of the given patterns.

Example:

```
case $filename in
*.dat)
    echo Processing a .dat file
    ;;
*.sas)
    echo Processing a .sas file
    ;;
*)
    # catch anything else that doesn't match patterns
    echo "Don't know how to deal with $filename"
    ;;
esac
```

Miscellaneous flow control and subshells

break [n]

Break out of the current (or n'th) enclosing loop. Control jumps to the next statement after the loop

continue [n];

Resume iteration of the current (or n'th) enclosing loop. Control jumps to the top of the loop, which generally causes re-evaluation of a **while** or processing the next element of a **for**.

. filename

Read the contents of the named file into the current shell and execute as if in line. Uses \$PATH to locate the file, and can be passed positional parameters. This is often used to read in shell functions that are common to multiple scripts. There are security implications if the pathname is not fully specified.

(...) Command grouping

Commands grouped in "()" are executed in a subshell, with a separate environment (can not affect the variables in the rest of the script).

Conditional Test Examples

As with most aspects of shell scripting, there are usually several possible ways to accomplish a task. Certain idioms show up commonly. These are five ways to examine and branch on the initial character of a string.

Use **case** with a pattern:

```
case $var in
/*) echo "starts with /" ;;

```

Works in all shells, and uses no extra processes

Use **cut**:

```
if [ "`echo $var | cut -c1`" = "/" ] ; then .

```

Works in all shells, but inefficiently uses a pipe and external process for a trivial task.

Use POSIX variable truncation:

```
if [ "${var%${var#?}}" = "/" ] ; then

```

Works with ksh, bash and other POSIX-compliant shells. Not obvious if you have not seen this one before. Fails on old Bourne shells. Dave Taylor in "Wicked Cool Shell Scripts" likes this one.

Use POSIX pattern match inside of `[[...]]`:

```
if [[ $var = /* ]]; then

```

Works with ksh, bash and other POSIX-compliant shells. Note that you must use `[[...]]` and no quotes around the pattern.

The `[[...]]` syntax is handled internally by the shell and can therefore interpret "wildcard" patterns differently than an external command. An unquoted wildcard is interpreted as a pattern to be matched, while a quoted wildcard is taken literally. The `[...]` syntax, even if handled internally, is treated as though it were external for backward compatability. This requires that wildcard patterns be expanded to matching filenames.

Use ksh (93 and later) and bash variable substrings:

```
if [ "${var:0:1}" = "/" ] ; then

```

ksh93 and later versions, and bash, have a syntax for directly extracting substrings by character position.

```
${varname:start:length}

```

Example: ex17 [display](#), [text](#)

Miscellaneous internal commands

The shells (ksh in particular) have many more internal commands. Some are used more in interactive shells. The commands listed here are used in scripts, but don't conveniently fit elsewhere in the class.

eval args

The args are read as input to the shell and the resulting command executed. Allows "double" expansion of some constructs. For example, constructing a variable name out of pieces, and then obtaining the value of that variable.

```
netdev=NETDEV_
NETDEV_1=hme0          # As part of an initialization step defining multiple devices

devnum=1               # As part of a loop over those devices
ifname=$netdev$devnum # construct a variable name NETDEV_1
eval device=\${ifname} # evaluate it - device is set to hme0
```

exec command args

The command is executed *in place* of the current shell. There is no return from an exec. I/O redirection may be used. This is also used to change the I/O for the current shell.

:

The line is variable-expanded, but otherwise treated as a comment. Sometimes used as a synonym for "true" in a loop.

```
while :; do
    # this loop will go forever until broken by
    # a conditional test inside, or a signal
done
```

unset var ...

Remove the named variables. This is not the same as setting their values to null.

typeset [+/- options] [name[=value]] ... (ksh only, bash uses **declare** for similar functions)

Set attributes and values for shell variables and functions. When used inside a function, a local variable is created. Some of the options are:

-L[n]

Left justify and remove leading blanks. The variable always has length n if specified.

-R[n]

Right justify and fill with leading blanks. The variable always has length n if specified.

-l

The named variable is always treated as an integer. This makes arithmetic faster. The reserved word **integer** is an alias for **typeset -i**.

-Z[n]

As for -R, but fill with zeroes if the value is a number

-i

Lower-case convert the named variables

-u

Upper-case convert the named variables

-r

Mark the variables as readonly

-x

Export the named variables to the environment

-ft

The variables are taken as function names. Turn on execution tracing.

Manipulating Variables (ksh/bash only)

Text variables

The *pattern* in the following uses the same wildcards as for filename matching.

```
${#var}
    returns the length of $var in characters
${var%pattern}
    removes the shortest suffix of $var patching pattern
${var%%pattern}
    removes the longest suffix of $var patching pattern
${var#pattern}
    removes the shortest prefix of $var patching pattern
${var##pattern}
    removes the longest prefix of $var patching pattern
```

Numeric variables

```
$(( integer expression ))
    The $(( ... )) construction interprets the contents as an arithmetic expression (integer only). Variables are referenced by name without the "$". Most of the arithmetic syntax of the 'C' language is supported, including bit manipulations (*,+, -,!,&,<,>). Use parentheses for changing precedence).
```

Examples

```
datapath=/data/public/project/trials/set1/datafile.dat

filename=${datapath##*/}
    filename is set to "datafile.dat" since the longest prefix pattern matching "*/" is the leading directory path (compare basename)
path=${datapath%/*}
    path is set to "/data/public/project/trials/set1" since the shortest suffix pattern matching "/*" is the filename in the last directory (compare dirname)

i=$((i+1))
    often used in while loops
```

Shell Functions

All but the earliest versions of **sh** allow you define *shell functions*, which are visible only to the shell script and can be used like any other command. Shell functions take precedence over external commands if the same name is used. Functions execute in the same process as the caller, and must be defined before use (appear earlier in the file). They allow a script to be broken into maintainable chunks, and encourage code reuse between scripts.

Defining functions

identifier() { list; }

POSIX syntax for shell functions. Such functions do not restrict scope of variables or signal traps. The identifier follows the rules for variable names, but uses a separate namespace.

function identifier { list; }

Ksh and bash optional syntax for defining a function. These functions may define local variables and local signal traps and so can more easily avoid side effects and be reused by multiple scripts.

A function may read or modify any shell variable that exists in the calling script. Such variables are *global*.

(ksh and bash only) Functions may also declare *local* variables in the function using **typeset** or **declare**. Local variables are visible to the current function and any functions called by it.

return [n], exit [n]

Return from a function with the given value, or exit the whole script with the given value.

Without a **return**, the function returns when it reaches the end, and the value is the exit status of the last command it ran.

Example:

```
die()
{
    # Print an error message and exit with given status
    # call as: die status "message" ["message" ...]
    exitstat=$1; shift
    for i in "$@"; do
        print -R "$i"
    done
    exit $exitstat
}
```

Calling functions.

Functions are called like any other command. The output may be redirected independantly of the script, and arguments passed to the function. Shell option flags like **-x** are unset in a function - you must explicitly set them in each function to trace the execution. Shell functions may even be backgrounded and run asynchronously, or run as coprocesses (ksh).

Example:

```
[ -w $filename ] || \
die 1 "$file not writeable" "check permissions"
```

Example: Backgrounded function call. ex12 [display, text](#)

```
1: #!/bin/sh
2:
3: background()
4: {
5:     sleep 10
6:     echo "Background"
7:     sleep 10
8:     # Function will return here - if backgrounded, the subprocess will exit.
9: }
10:
11: echo "ps before background function"
12: ps
13: background &
14: echo "My PID=$$"
15: echo "Background function PID=$!"
16: echo "ps after background function"
17: ps
18: exit 0
```

Example:

```
vprint()
{
    # Print or not depending on global "$verbosity"
```

```
# Change the verbosity with a single variable.
# Arg. 1 is the level for this message.
level=$1; shift
if [[ $level -le $verbosity ]]; then
    print -R $*
fi
}

verbosity=2
vprint 1 This message will appear
vprint 3 This only appears if verbosity is 3 or higher
```

Reusable functions

By using only command line arguments, not global variables, and taking care to minimise the side effects of functions, they can be made reusable by multiple scripts. Typically they would be placed in a separate file and read with the "." operator.

Functions may generate output to stdout, stderr, or any other file or filehandle. Messages to stdout may be captured by command substitution (``myfunction``), which provides another way for a function to return information to the calling script. Beware of side-effects (and reducing reusability) in functions which perform I/O.

Advanced I/O

Unix I/O is performed by assigning file descriptors to files or devices, and then using those descriptors for reading and writing. Descriptors 0, 1, and 2 are always used for stdin, stdout and stderr respectively. Stdin defaults to the keyboard, while stdout and stderr both default to the current terminal window.

Redirecting for the whole script

Redirecting stdout, stderr and other file descriptors for the whole script can be done with the **exec** command.

exec > outfile < infile

with no command, the **exec** just reassigns the I/O of the current shell.

exec n>outfile

The form **n<**, **n>** opens file descriptor **n** instead of the default stdin/stdout. This can then be used with **read -u** or **print -u**.

Explicitly opening or duplicating file descriptors

One reason to do this is to save the current state of stdin/stdout, temporarily reassign them, then restore them.

>&n

standard output is moved to whatever file descriptor **n** is currently pointing to

<&n

standard input is moved to whatever file descriptor **n** is currently pointing to

n>file

file descriptor **n** is opened for writing on the named *file*.

n>&1

file descriptor **n** is set to whatever file descriptor 1 is currently pointing to.

Example Sending messages to stderr (2) instead of stdout (1)

```
echo "Error: program failed" >&2
```

Echo always writes to stdout, but stdout can be temporarily reassigned to duplicate stderr (or other file descriptors). Conventionally Unix programs send error messages to stderr to keep them separated from stdout.

Input and output to open file descriptors (ksh)

Printing to file descriptors (usually more efficient than open/append/close):

print -u n args

print to file descriptor *n*.

-p

write to the pipe to a coprocess (opened by **l&**)

Reading from file descriptors other than stdin:

read -u n var1 var2 rest

read a line from file descriptor **n**, parsing by \$IFS, and placing the words into the named variables. Any left over words all go into the last variable.

-p

read from the pipe to a coprocess (opened by **l&**)

Closing file handles

<&-

standard input is explicitly closed

>&-

standard output is explicitly closed

For example, to indicate to another program downstream in a pipeline that no more data will be coming. All file descriptors are closed when a script exits.

I/O redirection operators are evaluated left-to-right. This makes a difference in a statement like: **>filename 2>&1**. (Many books with example scripts get this wrong)

"Here" documents

<< [-]string

redirect input to the temporary file formed by everything up the matching *string* at the start of a line. Allows for placing file content inline in a script.

Example: ex5 [display](#), [text](#)

```
1: #!/bin/sh
2: echo "Example of unquoted here document, with variable and command substitution"
3:
4: cat <<EOF
5: This text will be fed to the "cat" program as
6: standard input. It will also have variable
7: and command substitutions performed.
8: I am logged in as $USER and today is `date`
9: EOF
10: echo
11: echo "Example of quoted here document, with no variable or command substitution"
12: # The terminating string must be at the start of a line.
13: cat <<"EndOfInput"
14: This text will be fed to the "cat" program as standard
15: input. Since the text string marking the end was quoted, it does not get
16: variable and command substitutions.
17: I am logged in as $USER and today is `date`
18: EndOfInput
```

Example: duplex [display](#), [text](#)

```
1: #!/bin/sh
2: # Add in the magic postscript preface to perform
3: # duplex printer control for Xerox docuprint.
4:
5: # To have this script send the files directly to the printer, use
6: # a subshell to collect the output of the two 'cat' commands.
7:
8: ## (
9: cat << EOP
10: %!PS
11: %%BeginFeature: *Duplex DuplexTumble
12: <</Duplex true /Tumble false>> setpagedevice
13: %%EndFeature
14: EOP
15: cat "$@"
16: ## ) | lpr
```

Wizard Level I/O

More complicated manipulations of file descriptors can be arranged. Two such examples are shown here:

This short test script can be used to generate suitable output.

ex13: [display, text](#)

```
echo "This goes to stdout"
echo "This goes to stdout and has foo in the line"
echo "This goes to stderr" >&2
exit 99
```

Pass stderr of a command into a pipeline for further processing

Example: ex14 [display, text](#)

```
exec 3>&1
./ex13.sh 2>&1 1>&3 3>&- | sed 's/stderr/STDERR/' 1>&2
```

We duplicate stdout to another file descriptor (3), then run the first command with stderr redirected to stdout and stdout redirected to the saved descriptor (3). The result is piped into other commands as needed. The output of the pipeline is redirected back to stderr, so that stdout and stderr of the script as a whole are what we expect.

```
1: #!/bin/sh
2: # Example 14
3: # Take stderr from a command and pass it into a pipe
4: # for further processing.
5:
6: # Uses ex13.sh to generate some output to stderr
7: # stdout of ex13 is processed normally
8:
9: # Save a copy of original stdout
10: exec 3>&1
11:
12: # stdout from ex13.sh is directed to the original stdout (3)
13: # stderr is passed into the pipe for further processing.
14: # stdout from the pipe is redirected back to stderr
15: ./ex13.sh 2>&1 1>&3 3>&- | sed 's/stderr/STDERR/' 1>&2
16:
17: # 3 is closed before running the command, just in case it cares
18: # about inheriting open file descriptors.
```

Capture the exit status of a command in the middle of a pipeline

Example: ex15 [display, text](#)

```
exec 3>&1
ex13stat=$((./ex13.sh; echo $? >&4) | grep 'foo' 1>&3) 4>&1`
```

This script uses nested subshells captured in backtics. Again we first duplicate stdout to another file descriptor (3). The inner subshell runs the first command, then writes the exit status to fd 4. The outer subshell redirects 4 to stdout so that it is captured by the backtics. Standard output from the first command (inner subshell) is passed into the pipeline as normal, but the final output of the pipeline is redirected to 3 so that it appears on the original stdout and is not captured by the backtics.

If any of the commands really care about inheriting open file descriptors that they don't need then a more correct command line closes the descriptors before running the commands.

```
1: #!/bin/sh
2: # Example 15
3:
4: # Uses ex13.sh to generate some output and give us an
5: # exit status to capture.
6:
7: # Get the exit status of ex13 into $ex13stat.
8: # stdout of ex13 is processed normally
9:
10: # Save a copy of stdout
11: exec 3>&1
12: # Run a subshell, with 4 duplicated to 1 so we get it in stdout.
13: # Capture the output in ``
14: # ex13stat=$(( ... ) 4>&1`
15: # Inside the subshell, run another subshell to execute ex13,
16: # and echo the status code to 4
17: # (./ex13.sh; echo $? >&4)
18: # stdout from the inner subshell is processed normally, but the
19: # subsequent output must be directed to 3 so it goes to the
20: # original stdout and not be captured by the ``
21: ex13stat=$((./ex13.sh; echo $? >&4) | grep 'foo' 1>&3) 4>&1`
22:
23: echo Last command status=$?
24: echo ex13stat=$ex13stat
25:
26: # If any of the commands really care about inheriting open file
```

```

27: # descriptors that they don't need then a more correct command line
28: # closes the descriptors before running the commands
29: exec 3>&1
30: ex13stat=`(./ex13.sh 3>&- 4>&- ; echo $? >&4) | \
31:   grep 'foo' 1>&3 3>&- 4>&- ) 4>&1`
32: echo Last command status=$?
33: echo ex13stat=$ex13stat

```

Combine the above two techniques:

Example: ex16 [display, text](#)

```

exec 3>&1
ex13stat=`(./ex13.sh 2>&1 1>&3 3>&- 4>&- ; echo $? >&4) | \
sed s/err/ERR/ 1>&2 3>&- 4>&- ) 4>&1`

1: #!/bin/sh
2: # Example 16
3:
4: # Uses ex13.sh to generate some output and give us an
5: # exit status to capture.
6:
7: # Get the exit status of ex13 into ex13stat.
8: # stderr of ex13 is processed by the pipe, stdout
9: # is left alone.
10:
11: # Save a copy of stdout
12: exec 3>&1
13:
14: # Run a subshell, with 4 copied to 1 so we get it in stdout.
15: # Capture the output in backticks
16: # ex13stat=`( ) 4>&1`
17:
18: # In the subshell, run another subshell to execute ex13, and
19: # echo the status code to 4
20: # (./ex13.sh; echo $? >&4)
21:
22: # stdout from the inner subshell is directed to the original stdout (3)
23: # stderr is passed into the pipe for further processing.
24: # stdout from the pipe is redirected back to stderr
25:
26: # Close the extra descriptors before running the commands
27: exec 3>&1
28: ex13stat=`(./ex13.sh 2>&1 1>&3 3>&- 4>&- ; echo $? >&4) | \
29:   sed s/err/ERR/ 1>&2 3>&- 4>&- ) 4>&1`
30:
31: echo Last command status=$?
32: echo ex13stat=$ex13stat
33:

```

A practical application of this would be running a utility such as [dd](#) where the exit status is important to capture, but the error output is overly chatty and may need to be filtered before delivering to other parts of a script.

Coprocesses and Background jobs

Scripts can start any number of background jobs (any external command), which run in parallel with the parent script, and asynchronously. Processes which require no further interaction or synchronization (fire and forget) are easy. Interaction with background jobs is tricky. You can use signals, pipes, named pipes, or disk files for communication.

command &

Start *command* as a background process. Control returns immediately to the shell.

bgpid=\$!

The special variable **\$!** contains the process ID of the last background job that was started. You can save that and examine the process later (**ps -p \$bgpid**) or send it a signal (**kill -HUP \$bgpid**).

ksh coprocesses

Coprocesses are a way of starting a separate process which runs asynchronously, but has stdin/stdout connected to the parent script via pipes.

command |&

Start a coprocess with a 2-way pipe to it

read -p var

Read from the pipe to the coprocess, instead of standard input

print -p args

Write to the pipe connected to the coprocess, instead of standard output

Multiple coprocesses can be handled by moving the special file descriptors connected to the pipes onto standard input and output, and or to explicitly specified file descriptors.

exec <&p

The input from the coprocess is moved to standard input

exec >&p

The output from the coprocess is moved to standard output

Example: ex9 [display, text](#)

A script wants to save a copy of all output in a file, but also wants a copy to the screen. This is equivalent to always running the script as

script | tee outfile

```

1: #!/bin/ksh
2:
3: # If we have not redirected standard output, save a copy of
4: # the output of this script into a file, but still send a
5: # copy to the screen.
6:
7: if [[ -t 1 ]] ; then
8:   # Only do this if fd 1 (stdout) is still connected
9:   # to a terminal
10:
11:   # We want the standard output of the "tee" process
12:   # to go explicitly to the screen (/dev/tty)
13:   # and the second copy goes into a logfile named $0.out
14:
15:   tee $0.out >/dev/tty |&
16:
17:   # Our stdout all goes into this coprocess
18:   exec 1>&p
19: fi
20:
21: # Now generate some output
22: print "User activity snapshot on $(hostname) at $(date)"
23: print
24: who

```

Example: ex10 [display, text](#)

Start a coprocess to look up usernames in some database. It is faster to run a single process than to run a separate lookup for each user.

```

1: #!/bin/ksh
2: # This example uses a locally written tool for Dartmouth Name Directory lookups
3:
4: # Start the dndlookup program as a coprocess
5: # Tell it to output only the canonical full name, and to not print multiple matches
6: dndlookup -fname -u |&
7:
8: # move the input/output streams so we
9: # can use other coprocesses too
10: exec 4>&p
11: exec 5<&p
12:
13: echo "Name file contents:"

```

```
14: cat namefile
15: echo
16:
17: # read the names from a file "namefile"
18: while read uname; do
19:     print -u4 $uname
20:     read -u5 dndname
21:     case $dndname in
22:         *many\ matches*)
23:             # handle case where the name wasn't unique
24:             print "Multiple matches to \"$uname\" in DND"
25:             ;;
26:         *no\ match*)
27:             # handle case where the name wasn't found
28:             print "No matches to \"$uname\" in DND"
29:             ;;
30:         *)
31:             # we seem to have a hit - process the
32:             # canonical named retrieved from dndlookup
33:             print "Unique DND match: full name for \"$uname\" is \"$dndname\""
34:             ;;
35:     esac
36:     sleep 2
37: done < namefile
38:
39: # We've read all the names, but the coprocess
40: # is still running. Close the pipe to tell it
41: # we have finished.
42: exec 4>&-
```

Both `ksh` and `bash` implement arrays of variables, but in somewhat different ways.

Array elements are set with the syntax: `myarray[index]=value` and referenced with the syntax `${myarray[index]}`

Example: getauthlogs [display](#), [text](#)

18/11/2018, 14:22

```
81:             hostnames[$index]="${hostlookup $(hextodec $iphex)}"
82:             fi
83:             echo "$frontpart from ${hostnames[$index]}"
84:         else
85:             echo "$frontpart from $(hextodec $iphex)"
86:         fi
87:     ;;
88: *)
89:     echo "$line"
90: ;;
91: esac
92: else
93:     # No ip translation, just echo the whole line
94:     echo "$line"
95: fi
96: done
97:
```

Delivering and Trapping Signals

Unix signals (software interrupts) can be sent as asynchronous events to shell scripts, just as they can to any other program. The default behaviour is to ignore some signals and immediately exit on others. Scripts may detect signals and divert control to a handler function or external program. This is often used to perform clean-up actions before exiting, or restart certain procedures. Execution resumes where it left off, if the signal handler returns. Signal traps must be set separately inside of shell functions. Signals can be sent to a process with **kill**.

trap handler sig ...

handler is a command to be read (evaluated first) and executed on receipt of the specified *sigs*. Signals can be specified by name or number (see kill(1)) e.g. **HUP, INT, QUIT, TERM**. A Ctrl-C at the terminal generates a **INT**.

- A handler of **-** resets the signals to their default values
- A handler of **' '** (null) ignores the signals
- Special signal values are as follows:

EXIT

the handler is called when the function exits, or when the whole script exits. The exit signal has value 0.

ERR (ksh)

the handler is called when any command has a non-zero exit status

DEBUG (ksh)

the handler is called after *each* command.

Example: ex8 [display](#), [text](#)

```
1: #!/bin/bash
2: # Try this under bash, ksh and sh
3:
4: trap huphandler HUP
5: trap '' QUIT
6: trap exithandler TERM INT
7:
8: huphandler()
9: {
10:     echo 'Received SIGHUP'
11:     echo "continuing"
12: }
13:
14: exithandler()
15: {
16:     echo 'Received SIGTERM or SIGINT'
17:     exit 1
18: }
19: ## Execution starts here - infinite loop until interrupted
20: # Use ":" or "true" for infinite loop
21: # SECONDS is built-in to bash and ksh. It is number of seconds since script started
22: : is like a comment, but it is evaluated for side effects and evaluates to true
23: seconds=0
24: while : ; do
25:     # while true; do
26:     sleep 5
27:     seconds=$((seconds + 5))
28:     echo -n "$SECONDS $seconds - "
29: done
```

Exit handlers can be defined to clean up temporary files or reset the state of devices. This can be useful if the script has multiple possible exit points.

Security issues in shell scripts

Shell scripts are often used by system administrators and are run as a privileged user.

- Don't use set-UID scripts.

Most systems don't even allow a script to be made set-UID. It is impossible (due to inherent race conditions) to ensure that a set-uid script cannot be compromised. Use wrapper programs like **sudo** instead.

- Always explicitly set **\$PATH** at the start of a script, so that you know exactly which external programs will be used.
- If possible, don't use temporary files. If they cannot be avoided, use **\$TMPDIR**, and create files safely (e.g. **mktemp**).

Often scripts will write to a fixed, or trivially generated temporary filename in /tmp. If the file already exists and you don't have permission to overwrite it, the script will fail. If you do have permission to overwrite it, you will delete the previous contents. Since /tmp is public write, another user may create files in it, or possibly fill it completely.

Example:

1. A link is created by an unprivileged user in /tmp: **/tmp/scratch -> /vmunix**

2. A root user runs a script that blindly writes a scratch file to /tmp/scratch, and overwrites the operating system.

Environment variable **\$TMPDIR** is often used to indicate a preferred location for temporary files (e.g., a per-user directory). Some systems may use **\$TMP** or **\$TEMP**. Safe scratch files can be made by creating a new directory, owned and writeable only by you, then creating files in there.

Example:

```
(umask 077 && mkdir /tmp/tmpdir.$$) || exit 1
```

or (deluxe version)

```
tmp=${TMPDIR:-/tmp}
tmp=$tmp/tmpdir.$RANDOM.$RANDOM.$$
(umask 077 && mkdir $tmp) || {
    echo "Could not create temporary directory" 1>&2
    exit 1
}
```

Alternatively, many systems have **mktemp** to safely create a temporary file and return the filename, which can be used by the script and then deleted.

- Check exit status of everything you do.
- Don't trust user input
 - contents of files
 - data piped from other programs
 - file *names*. Output of filename generation with wildcards, or directly from **ls** or **find**

Example:

Consider the effects of a file named "**myfile;cd /;rm ***" if processed, unquoted, by your script.

One possible way to protect against weirdo characters in file names:

```
# A function to massage a list of filenames
# to protect weirdo characters
# e.g. find ... | protect_filenames | xargs command
#
# We are backslash-protecting the characters \"'\" ?*;
protect_filenames()
{
    sed -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\'/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\"/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\;/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\?/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\*/g \
        -es/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\;/g
}
```

If using GNU **find** and **xargs**, there is a much cleaner option to null-terminate generated pathnames.

Style

Shell scripts are very frequently written quickly for a single purpose, used once and discarded. They are also as frequently kept and used many times, and migrate into other uses, but often do not receive the same level of testing and debugging that other software would be given in the same situation. It *is* possible to apply general principles of good software engineering to shell scripts.

- Preface scripts with a statement of purpose, author, date and revision notes
- Use a revision control system for complex scripts with a long lifetime
- Assume your script *will* have a long lifetime unless you are certain it won't
- Document any non-standard external utilities which your script needs
- Document your scripts with inline comments - you'll need them in a few months when you edit it.
- Treat standard input and output in the normal way, so that your script can be used in combination with other programs (the Unix toolkit philosophy)
- Be consistent in the format of your output, so that other programs can rely on it
- Use options to control behaviour such as verbosity of output. Overly chatty programs are very hard to combine with other utilities
- Use interactive features (prompting the user for information) **very sparingly**. Doing so renders the script unuseable in pipeline combinations with other programs, or in unattended operations.
- Test (a lot)

When not to use shell scripts

- If an existing tool already does what you need - use it.
- If the script is more than a few hundred lines, you are probably using the wrong tool.
- If performance is horrible, and the script is used a lot, you might want to consider another language.

Some longer examples

The class accounts have directories with all of the examples from the books by Blinn, Michael, Rosenblatt, and Taylor. These can also be downloaded (see the References page). Some of these are linked below (but not included in the printed notes), with additional comments.

Download a [compressed tar file](#) of all example scripts used in these notes.

- **postprint** [display](#), [text](#)
A wrapper script for printing a mix of text and postscript files
- **checkpath** [display](#), [text](#)
Check all the directories in the `$PATH` for possibly conflicting programs.
- **run-with-timeout** [display](#), [text](#)
Run a command with a timeout. Kill the command if it hasn't finished when the timeout expires.
- **MailPkg** [display](#), [text](#)
Tar, compress, split and uuencode a set of files for mailing. (Blinn)
- **Ptree** (original) [display](#), [text](#)
- **Ptree** (ksh version) [display](#), [text](#)
Runs "ps" to get a process listing and then reformats to show the process family hierarchies. The original example is pure Bourne shell and inefficient. The ksh version is a fairly simple translation to use ksh internal commands where possible, and avoid writing scratch files, and runs very much faster. (Blinn).

This entire tutorial was created from individual HTML pages using a content management system written as ksh scripts (heavily using sed to edit the pages), coordinated by [make](#).

You can even write an entire web server as a shell script. This one is part of the [LEAF](#) (Linux Embedded Appliance Firewall) project. This wouldn't be suitable for much load, but handles occasional queries on static HTML and CGI scripts. (www.nisi.ab.ca/lrp/Packages/weblet.htm)

- **sh-httpd** [display](#), [text](#)

A Toolkit of commonly used external commands

The following commands are very frequently used in shell scripts. Many of them are used in the examples in these notes. This is just a brief recap -- see the man pages for details on usage. The most useful are flagged with *****.

Most of these commands will operate on a one or more named files, or will operate on a stream of data from standard input if no files are named.

Listing, copying and moving files and directories

ls *

list contents of a directory, or list details of files and directories.

mkdir; rmdir *

Make and Remove directories.

rm; cp; mv *

Remove (delete), Copy and Move (rename) files and directories

touch *

Update the last modified timestamp on a file, to make it appear to have just been written.

If the file does not exist, a new zero-byte file is created, which is often useful to signify that an event has occurred.

tee

Make a duplicate copy of a data stream - used in pipelines to send one copy to a log file and a second copy on to another program. (Think plumbing).

Displaying text, files or parts of files

echo *

Echo the arguments to standard output -- used for messages from scripts. Some versions of "sh", and all csh/ksh/bash shells internalized "echo".

Conflicts sometimes arise over the syntax for echoing a line with no trailing CR/LF. Some use "\c" and some use option "-n". To avoid these problems, ksh also provides the "print" command for output.

cat *

Copy and concatenate files; display contents of a file

head, tail *

Display the beginning of a file, or the end of it.

cut

Extract selected fields from each line of a file. Often awk is easier to use, even though it is a more complex program.

wc

Count lines, words and characters in the input.

Compression and archiving

compress; gzip; zip; tar *

Various utilities to compress/uncompress individual files, combine multiple files into a single archive, or do both.

Sorting and searching for patterns

sort *

Sort data alphabetically or numerically.

grep *

Search a file for lines containing character patterns. The patterns can be simple fixed text, or very complex regular expressions.

The name comes from "Global Regular Expression and Print" -- a function from the Unix editors which was used frequently enough to warrant getting its own program.

uniq *

Remove duplicate lines, and generate a count of repeated lines.

wc *

Count lines, words and characters in a file.

System information (users, processes, time)

date *

Display the current date and time (flexible format). Useful for conditional execution based on time, and for timestamping

output.

ps *

List the to a running processes.

kill *

Send a signal (interrupt) to a running process.

id

Print the user name and UID and group of the current user (e.g. to distinguish priviledged users before attempting to run programs which may fail with permission errors)

who

Display who is logged on the system, and from where they logged in.

uname *

Display information about the system, OS version, hardware architecture etc.

mail *

Send mail, from a file or standard input, to named recipients. Since scripts are often used to automate long-running background jobs, sending notification of completion by mail is a common trick.

logger

Place a message in the central system logging facility. Scripts can submit messages with all the facilities available to compiled programs.

hostname

Display the hostname of the current host - useful to keep track of where your programs are running

Conditional tests

test; [*

The conditional test, used extensively in scripts, is also an external program which evaluates the expression given as an argument and returns true (0) or false (1) exit status. The name "[" is a link to the "test" program, so a line like:

if [-w logfile]

actually runs a program "[", with arguments "-w logfile]", and returns a true/false value to the "if" command.

In ksh and most newer versions of sh, "[" is replaced with a compatible internal command, but the argument parsing is performed as if it were an external command. Ksh also provides the internal "[[" operator, with simplified syntax.

Stream Editing

awk *

A pattern matching and data manipulation utility, which has its own scripting language. It also duplicates much functionality from 'sed', 'grep', 'cut', 'wc', etc.

Complex scripts can be written entirely using awk, but it is frequently used just to extract fields from lines of a file (similar to 'cut').

sed *

Stream Editor. A flexible editor which operates by applying editing rules to every line in a data stream in turn.

Since it makes a single pass through the file, keeping only a few lines in memory at once, it can be used with infinitely large data sets. It is mostly used for global search and replace operations. It is a superset of 'tr', 'grep', and 'cut', but is more complicated to use.

tr

Transliterate - perform very simple single-character edits on a file.

Finding and comparing files

find *

Search the filesystem and find files matching certain criteria (name pattern, age, owner, size, last modified etc.)

xargs *

Apply multiple filename arguments to a named command and run it.

Xargs is often used in combination with "find" to apply some command to all the files matching certain criteria. Since "find" may result in a very large list of pathnames, using the results directly may overflow command line buffers. Xargs avoids this problem, and is much more efficient than running a command on every pathname individually.

diff *

Compare two files and list the differences between them.

basename *pathname*

Returns the base filename portion of the named *pathname*, stripping off all the directories

dirname *pathname*

Returns the directory portion of the named *pathname*, stripping off the filename

Arithmetic and String Manipulation

expr *

The "expr" command takes an numeric or text pattern expression as an argument, evaluates it, and returns a result to stdout.

The original Bourne shell had no built-in arithmetic operators. E.g.

```
expr 2 + 1
```

```
expr 2 '*' '(' 21 + 3 ')'
```

Used with text strings, "expr" can match regular expressions and extract sub expressions. Similar functionality can be achieved with sed. e.g.

```
expr SP99302L.Z00 : '[A-Z0-9]\{4\}\([0-9]\{3\}\)L\.'
```

dc

Desk Calculator - an RPN calculator, using arbitrary precision arithmetic and user-specified bases. Useful for more complex arithmetic expressions than can be performed internally or using expr

bc

A preprocessor for dc which provides infix notation and a C-like syntax for expressions and functions.

Merging files

paste

Merge lines from multiple files into tab-delimited columns.

join

Perform a join (in the relational database sense) of lines in two sorted input files.

References, Resources, Man pages etc.

The standard man pages for [sh](#) and [ksh](#) are quite complete, but not easy to learn from. The following is a sampling of the many available books on the subject. The Bolsky and Korn book might be viewed as the standard "reference". The Blinn book is Bourne shell, but everything in it should work for either shell.

The links are to publisher's web sites, or Amazon.com. Some links are also given to the example scripts provided with the books.

Books

- *The New KornShell Command And Programming Language*, by Morris I. Bolsky, David G. Korn (Contributor). [More info](#)
- *Learning the Korn Shell, 2nd Edn.* by Bill Rosenblatt and Arnold Robbins. [More info](#)
- *Korn Shell Programming by Example*, by Dennis O'Brien, David Pitts (Contributor). [More info](#)
- *The Korn Shell Linux and Unix Programming Manual (2nd Edn)* by Anatole Olczak. [More info](#)
- *Portable Shell Programming: An Extensive Collection of Bourne Shell Examples* by Bruce Blinn. [More info](#)
Examples from this book can be [downloaded](#) for study.
- *Linux Shell Scripting with Bash* by Ken O. Burtch. [More info](#)
- *Unix Shell Programming* by Stephen Kochan and Patrick Wood (third Edition). [More info](#)
- *Teach yourself Shell Programming in 24 Hours*, by S. Veeraraghavan. SAMS 2nd Edn. (2002) [More info](#)
- *Mastering Unix Shell Scripting* by Randal K. Michael, Wiley (2003) [More info](#) Light on basics, but develops scripting through examples. Ksh only. Examples can be [downloaded](#) from the Wiley site (www.wiley.com/legacy/compbooks/michael/).
- *Wicked Cool Shell Scripts* by Dave Taylor, No Starch Press (2004) [More info](#) Develops scripting entirely through examples, drawn from Linux and OSX in addition to traditional Unix. Recommended, but not for beginners. Examples can be [downloaded](#) from the Intuitive site (www.intuitive.com/wicked/wicked-cool-shell-script-library.shtml).
- *Unix Power Tools*, by S. Powers, J. Peek, T. O'Reilly, M. Loudikes et al. [More info](#)

Online Resources

- [Shelldorado](http://www.shelldorado.com) (<http://www.shelldorado.com>)
Lots of links to scripting resources
- [Kornshell](http://www.kornshell.com) (<http://www.kornshell.com>)
The official Korn shell home page, with download links.
- [Mac OSX Unix tutorial](http://www.osxfaq.com/Tutorials/LearningCenter/) (<http://www.osxfaq.com/Tutorials/LearningCenter/>)
Good resource on advanced use of OSX and Unix shell scripting in general

Unix-like shells and utilities for Microsoft Windows

- [U/Win](http://www.research.att.com/sw/tools/uwin/) (<http://www.research.att.com/sw/tools/uwin/>)
A free port of ksh and Unix command line utilities, plus Windows DLL for Unix compatability. Developed by AT&T Research.
- [Cygwin](http://www.cygwin.com/) (<http://www.cygwin.com/>)
A free Linux-like environment for Windows. Provides bash, command line utilities and DLLs. Developed by RedHat. An X server is also available.
- [MKS Toolkit](http://www.mkssoftware.com/) (<http://www.mkssoftware.com/>)
A commercial ksh clone and command line utilities, plus DLL for Unix compatability. An X server is also available.
- [Microsoft Services for UNIX](http://www.microsoft.com/windows/sfu/) (<http://www.microsoft.com/windows/sfu/>)
A POSIX environment for Windows, with ksh, csh, command line tools, libraries and software development tools. Developed by Interix and bought by Microsoft. Free download.