# Shell Mistakes

## 0. Introduction

Bourne shell is a useful scripting language but it turns out that there are a number of easy mistakes to make when using it, whether interactively or in a script. This file attempts to describe some of them, demonstrate that they are indeed mistakes, and show how to avoid them.

### 0.1 Assumptions

Basic familiarity with UNIX and Bourne shell is assumed: the reader will probably have written simple shell scripts already, and if not will at least know what the idea of a script is. This isn't a complete shell tutorial.

### 0.2 Typographical Conventions

Examples are displayed boxed:

```
Example
```

In transcripts of shell sessions, user input is displayed `specially`, to distinguish it from the output of the computer:

```
sfere$ echo wibble
wibble
```

This will only work if your browser supports style sheets. The distinction between user input and computer output will usually be reasonably obvious from context, however.

## 1. Redundant cat

It seems to be extremely common to use "cat" to feed input into some filter. For example:

```
cat file | grep searchstring
```

The effect of this is to create a pipe, executing "cat file" with its output redirected to the pipe and "grep searchstring" with its input redirected from the pipe. But you can eliminate the entire invocation of "cat" by just redirecting grep's input to come from the file:

```
grep searchstring < file
```

Not only does this avoid the time-wasting invocation of "cat", but (depending on the program you are running) it can sometimes make the process more efficient still by giving the program more direct access to the file. (If performance is really vital than you should test the various possibilities rather than speculating that one is faster than another, though.)

If what you want is to have the input file at the start rather than the end, you can do that too:

```
< file grep searchstring
```

In this case you can also just pass the name of the file to grep:

```
grep searchstring file
```

In other cases that might change the behaviour, though.

### 1.1 Exceptions

Very occasionally it does make sense to put in the "cat": if the process that reads the input behaves differently depending on whether it is a regular file, a pipe, a terminal, etc, then it might be that turning it into a pipe via the "cat" command makes sense.

If you think that use of cat like this boils down to a question of style, read the section error handling, with particular reference to the remarks regarding pipelines.

---

## 2. Redundant ls

A similar case is the redundant invocation of "ls". This is usually in order to get a list of filenames, for example:

```
for f in `ls *.html`; do
  process "$f"
done
```

There are at least two mistakes here. First, if any of the files have spaces in the name, then each word of the filename will be processed as a separate argument. Watch:

```
sfere$ touch "a file with a space in the name"
sfere$ for x in `ls *`; do echo "[$x]"; done
[a]
[file]
[with]
[a]
[space]
[in]
[the]
[name]
```

Secondly if any of the files are directories then you will get the contents of the directories instead of the name of the directory. Like this:

```
sfere$ mkdir somedir
sfere$ touch somedir/1
sfere$ touch somedir/2
sfere$ touch 1
sfere$ for x in `ls s*`; do echo "[$x]"; done
[1]
[2]
```

Notes:

- "somedir" isn't listed, though it matches the pattern
- "1" is listed, though it doesn't match the pattern
- "2" is listed, though not only does it not match the pattern but there isn't even a file called "2" in the current directory.

The problems this causes might be more clearly illustrated by an example which actually tries to do something with the files:

```
sfere$ for x in `ls s*`; do ls -l "$x"; done
```

```
-rw-r--r--    1 richard  richard         0 Apr 10 19:48 1
ls: 2: No such file or directory
```

Sometimes you know that none of the files have spaces in the name; and sometimes you know that there are no subdirectories matching your pattern. In such cases, these problems wouldn't arise.

However, even then, there's a quicker and easier way to achieve the desired effect, which is to take advantage of the fact that "for" actually knows about filename patterns anyway:

```
for f in *.html; do
  process "$f"
done
```

### 2.1 Excluding Directories

This still includes any subdirectories that match the pattern. If the contents of the loop should apply to directories anyway, or produce a harmless error message when passed a directory, then this may not be a problem. If you need to separately eliminate them then you must test each one explicitly:

```
for f in *.html; do
  if test ! -d "$f"; then
    process "$f"
  fi
done
```

A variant on this is to instead of ignoring directories, to ignore anything that isn't a regular (i.e. ordinary) file:

```
for f in *.html; do
  if test -f "$f"; then
    process "$f"
  fi
done
```

## 3. Missing Quoting

Shell's quoting syntax is arcane but unfortunately to write reliable scripts it's necessary to use it fully. The problem is quite simple: if you don't use quoting in the right place, then your script won't behave properly. For example:

```
sfere$ touch "a file with spaces in the name"
sfere$ for x in *; do ls $x; done
ls: a: No such file or directory
ls: file: No such file or directory
ls: with: No such file or directory
ls: spaces: No such file or directory
ls: in: No such file or directory
ls: the: No such file or directory
ls: name: No such file or directory
```

The answer in this case is fairly simple: quote variable expansions in "double quotes":

```
sfere$ for x in *; do ls "$x"; done
a file with spaces in the name
```

The full rules for quoting apply both to scripts and to commands entered interactively.

In the examples below we'll create a file called "test file", with a space in the name, using various different quoting styles. If you use no quoting at all, then the command used to create the file will see two separate arguments, thus creating two files rather than one file with a space in the name:

```
sfere$ touch test file
sfere$ ls -l
total 0
-rw-r--r--    1 richard  richard         0 Apr  7 13:43 file
```

```
-rw-r--r--    1 richard  richard         0 Apr  7 13:43 test
```

(They appear in reverse order because ls sorts filenames lexically by default.)

## 3.1 Backslashes

Outside of any other form of quoting, a backslash ("\") may be used to quote any character except newline. In this example, the space character is preceded with a backslash to make the shell treat "test file" as a single argument, creating a file with a space in the name:

```
sfere$ touch test\ file
sfere$ ls -l
total 0
-rw-r--r--    1 richard  richard         0 Apr  7 13:11 test file
```

Sometimes the quoted character is referred to as being "escaped" and the backslash as an "escape character".

The one exception is that a backslash followed by a newline doesn't quote the newline: it removes it (and the backslash) entirely, allowing a command to span multiple lines. For example:

```
sfere$ echo foo\
> bar
foobar
sfere$
```

Notes:

- The "> " is output from the shell, not user input
- The user didn't type a space after the newline
- No space was seen by the shell after the newline - "foobar" is a single word

If you have reconfigured your shell's prompting, then you may get something different from "> ".

## 3.2 Single Quotes

Any character in a string contained in 'single quotes' has no special meaning. (The exception of course is that the single quote character itself cannot appear in such a string.) For example:

```
sfere$ touch 'test file'
sfere$ ls -l
total 0
-rw-r--r--    1 richard  richard         0 Apr  7 13:39 test file
```

It is perfectly permissible to quote newlines this way; for example:

```
sfere$ echo 'first line
> second line'
first line
second line
```

Here, the echo command sees a single argument, which happens to have a newline in it.

## 3.3 Double Quotes

"Double quotes" are the most flexible kind of quoting. Certain characters, like space and tab, which have special meaning outside of quotes lose that meaning inside double quotes. However other characters retain their special meaning, or change slightly.

The simple example of creating our test file looks like this:

```
sfere$ touch "test file"
sfere$ ls -l
total 0
-rw-r--r--    1 richard  richard         0 Apr  7 13:46 test file
```

The set of characters that retain special meaning inside double quotes is as follows:

| Character | Description |
|---|---|
| $ | The dollar symbol retains its special meaning of starting an expansion or substitution of some sort. |
| ` | The backquote retains its special meaning of executing a command and substituting the output into the string. |
| \ | The backslash can be used to quote only certain characters; in particular $, `, ", \ and newline. If it is used before any other character then it is *not* removed. |

## 3.4 Special Behaviour of $@

$@ is a built-in shell variable that contains the names of arguments to the shell script. Superficially it is identical to $*, which also contains the argument values. However it behaves slightly differently when quoted.

For the examples here we'll assume the arguments to the script are "arg 1" and "arg 2" - both containing spaces. You can simulate this interactively with the "set" command:

```
sfere$ set "arg 1" "arg 2"
```

If you just use $* to get at the parameters, then you fall victim to quoting problems; the arguments get split into separate words at each space character:

```
sfere$ for x in $*; do echo "[$x]"; done
[arg]
[1]
[arg]
[2]
```

If you try to quote $* then the result is no better - now you get all the arguments combined into a single word:

```
sfere$ for x in "$*"; do  echo "[$x]"; done
[arg 1 arg 2]
```

(The character that is used to join each word together is taken from the first character of the value of $IFS. This is worth knowing for cases where what you want is all of the script arguments joined together into one.)

An unquoted $@ is no different to $*. However, if you double-quote it then you get the desired result:

```
sfere$ for x in "$@"; do  echo "[$x]"; done
[arg 1]
[arg 2]
```

Notes:

- This behaviour only occurs if $@ appears inside double quotes
- If there are no arguments to the script, then "$@" expands to no words at all, rather than to a single empty string.
- If there are other characters in the string, then they will be joined to the first or last word resulting from the expansion of $@.
- In modern shells, the `in "$@"` is redundant; however Solaris /bin/sh still requires it.

As an example of the final point:

```
sfere$ for x in "S $@ E"; do  echo "[$x]"; done
[S arg 1]
[arg 2 E]
```

## 3.5 Nested Quotes

Quotes don't nest as such: putting a single quote character inside a double-quoted string doesn't start a single-quoted string, it just stands for itself.

However, if you use shell commands such as "eval" and "trap" then you will find that you have to write quotes within quotes sometimes; it's very easy to get confused if you do this, as the inner quoting characters themselves must be quoted properly.

The approach to take is decide exactly what the command you want eval or trap to execute is; then considering it as a string of characters, rather than as a command, add quoting as necessary to remove special meaning from all special characters.

Consider this command:

```
echo \\ `echo spong`
```

This already contains an escaped backslash; and it contains a command substitution. If you run it directly it produces the following output:

```
sfere$ echo \\ `echo spong`
\ spong
```

If you want to run it via eval (which evaluates its argument as if it were a single shell command) then it must be quoted appropriately. Any of the three quoting styles can be used in this case.

If you decide to use backslashes then you must quote at least the characters that get special interpretation. This doesn't work if there are any newlines in the string. If in doubt as to whether any given character needs quoting, quote it.

In this case, we'll put a backslash before each space, backslash and backquote, and not bother with the other characters. This still applies "inside" the backquotes: at the point we are applying the quoting, this is just a sequence of characters, not a shell command at all - it only gets interpreted later. Each backslash has to be escaped separately for the same reason, causing the number of backslashes required to produce a single backslash in the eventual output to double up from two to four. The result is not very readable:

```
eval echo\ \\\\\ \`echo\ spong\`
```

If you choose double quotes, then any backslash, dollar, backquote or double quote in the string must be escaped with a backslash; all other characters remain unchanged. Newlines are acceptable.

```
eval "echo \\\\ \`echo spong\`"
```

That's a bit better: we don't have to worry about quoting spaces now. However, the quoted backslashes still double up to four. In a more complex example it would rapidly become unmanageable.

If you opt to use single quotes, then you can include newlines in the string, but not single quotes themselves. In this example, this is quite obviously the winner, being far more readable than the alternatives:

```
eval 'echo \\ `echo spong`'
```

Double quotes are the only way to get completely general quoting, however.

## 3.6 Changing Quoting

It's possible to change quoting style in the middle of a word, for example:

```
sfere$ echo "foo bar"wibble
foo barwibble
```

Although this is occasionally useful, it isn't the clearest of syntaxes.

## 3.7 Quoting Summary

| Quote | Exceptions | Notes |
|---|---|---|
| \ | newline | Escapes any single character other than newline; behaves differently inside quotes. \ + newline is removed entirely, to spread a command over multiple lines. |
| ' | ' | Quote any character except ' itself. Newlines stand for themselves. |
| " | ", $, `, \ | Directly quote any character except " itself, plus $, ` and \; all of these must be escaped. Newlines stand for themselves. |
| \ when inside "" | all but ", $, `, \ | Escapes $, `, \, ". \ + newline is removed entirely, spreading a string across multiple lines. For any other character \ has no effect (and is not removed). |

# 4. Error Handling

Things go wrong. Users mis-spell filenames; discs get full; programs crash. It's hard to consider every possibility, but scripts are often written to completely ignore all errors, sometimes leading to very destructive failure modes.

For example, imagine a script which modifies a file "in-place".

```
sfere$ ls -l t
-rwxr-xr-x    1 richard  richard        98 Apr  7 15:16 t
sfere$ cat t
#! /bin/sh
for file; do
  sed < "$file" > "$file.tmp" 's/foo/bar/g'
  mv "$file.tmp" "$file"
done
```

This works fine under normal circumstances:

```
sfere$ cat input
wibble foo spong
sfere$ ./t input
sfere$ cat input
wibble bar spong
```

But what happens if we run out of disc space? Well, the sed command will get as far as reading in data, but when it tries to write it out it will get a write error. If it got as far as opening the output file, then the result will be that input.tmp will now be an empty file - and so when we invoke mv, we'll replace the input file with a completely empty file.

```
sfere$ cat input
wibble foo spong
sfere$ ./t input
sed: Couldn't close {standard output}: No space left on device
sfere$ cat input
sfere$
```

We know something's gone wrong, because sed produced an error message, but our script has destroyed our input file! Hopefully we have a backup, but it would be better not to have lost it in the first place.

(The error message appears to refer to closing the file rather than writing to it - this is because the I/O library used by sed buffers output internally before actually writing it to disc, so the last few bytes of the output only get written out when the output file is closed at the end of the program.)

What can we do about this? Ideally we'd like sed to tell us when it fails to write some or all of its output and prevent the mv command from occurring. And sed does tell us when it fails: conventionally when a program succeeds, it exits with status zero; if any error occurs it exits with some nonzero status.

After executing a command, the shell puts the exit status into the built-in variable $?. We can check that this is zero after each command. For example:

```
sfere$ cat t
#! /bin/sh
for file; do
  sed < "$file" > "$file.tmp" 's/foo/bar/g'
  if [ $? = 0 ]; then
    mv "$file.tmp" "$file"
  fi
done
sfere$ ./t input
sed: Couldn't close {standard output}: No space left on device
sfere$ echo $?
0
sfere$ cat input
wibble foo spong
sfere$ ls
input  input.tmp  t
```

Problems with this script are:

- You have to remember to check the exit status of every command that might go wrong. For a small script, this may not seem like much work; for a large script it could triple the line count.
- It doesn't make this script itself exit with a nonzero status (so some further script that invoked it wouldn't be easily able to tell whether it succeeded or failed).
- It leaves the temporary file lying around.

A quick solution to the first of these problems can be found in the -e shell option. If this option is turned on then if any command exits with nonzero status, the entire script is immediately terminated. For example:

```
sfere$ cat t
#! /bin/sh
set -e
for file; do
  sed < "$file" > "$file.tmp" 's/foo/bar/g'
  mv "$file.tmp" "$file"
done
sfere$ ./t input
sed: Couldn't close {standard output}: No space left on device
sfere$ echo $?
4
sfere$ cat input
wibble foo spong
```

Notes:

- "set -e" doesn't in fact make the script exit if *any* command exits with nonzero status: the exceptions are basically those that make while, until, if, &&, || and ! work. (But see below.)
- Not all commands exit with a status 0 for success and something else for errors. For example, GNU diff returns a status of 0 to mean no differences, 1 for differences found and other values to indicate an error. In such cases you must explicitly check the value $? to determine whether the command succeeded.
- Sometimes it isn't appropriate to exit on a failure. Often such cases can be best handled with "if"; sometimes it is more appropriate to temporarily turn off "-e" (with set +e) for a few commands, then turn it back on again.

## 4.1 Cleaning Up After Errors

That temporary file is still lying around, wasting disc space and looking untidy; although here it's just one file, it's not unheard of to see thousands of redundant temporary files lying around, and this can become a serious problem.

One way to ensure it gets cleaned up is to use the "trap" command, as follows:

```
sfere$ cat t
#! /bin/sh
set -e
for file; do
  trap 'rm -f "$file.tmp"' 0
  sed < "$file" > "$file.tmp" 's/foo/bar/g'
  mv "$file.tmp" "$file"
done
sfere$ ./t input
sed: Couldn't close {standard output}: No space left on device
sfere$ echo $?
4
sfere$ ls
input  t
```

Note the nesting of quotes in the "trap" command. This causes the exit handler to be set without any expansion going on - so it will be the exact characters **rm -f "$file.tmp"**. The quotes and expansion of $file will be processed only when the handler is executed.

The reason for this is that if we expanded the filename early, then we'd have to escape any spaces in the filename, which is messy to get right and (as shown) not actually necessary.

Also, note that the "trap" command appears inside the loop rather than outside the loop: otherwise, when the script is invoked with no arguments, the file ".tmp" will be deleted.

An approach that avoids the need for nested quoting is to use a shell function to do the cleanup:

```
#! /bin/sh
set -e

cleanup() {
  rm -f "$file.tmp"
}

for file; do
  trap cleanup 0
  sed < "$file" > "$file.tmp" 's/foo/bar/g'
  mv "$file.tmp" "$file"
done
```

Note that some (obsolete) shells don't support functions, so this approach isn't guaranteed to be completely portable.

## 4.2 Pipelines

Unfortunately "set -e" doesn't work properly with pipelines, as only the exit status of the last command in a pipeline is checked by the shell: for all other commands, it is ignored.

```
sfere$ cat u
#! /bin/sh
set -e
true | false | true
echo "reached"
sfere$ ./u
reached
sfere$ echo $?
0
```

("false" is a command which always exits with nonzero status.)

There's not much that can portably be done about this, other than avoiding pipelines or accepting that some errors will not be automatically detected. The latter approach is often appropriate for interactive commands, but only rarely for scripts.

Pipelines can be avoided using temporary files to hold intermediate data (but this may be inappropriate if the intermediate data is large, or needs to be processed immediately) or perhaps by setting up named pipes between commands.

GNU bash has a PIPESTATUS variable which can be used to pick up the exit status of each command in a pipeline:

```
sfere$ true|false|true
sfere$ echo ${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]}
0 1 0
```

One wrinkle that is worth knowing: if a command in a pipeline terminates before it has read all its input from the previous command, the previous command might receive the SIGPIPE signal. If it does not handle it then it will exit with nonzero status, even though the error is expected. For example:

```
sfere$ find|head -1
.
sfere$ echo ${PIPESTATUS[0]} ${PIPESTATUS[1]}
141 0
```

Here, everything worked as expected - find didn't get an error trying to read all the data, it only exited nonzero because head didn't read beyond the first line. So, be careful of SIGPIPE when looking at exit statuses.

Notes:

- The value 141 above is actually 128+13, where 13 is the numeric value of SIGPIPE and 128 is the value the shell adds to signal numbers to indicate that a process died due to a signal rather than exiting "normally". SIGPIPE might be different from 13 on other systems.
- If you use bash features in a script, you must start it with "#! /bin/bash" (or whatever is appropriate if bash is installed somewhere else) rather than "#! /bin/sh".
- Some programs install a handler for SIGPIPE, or ignore it. This can make it hard to distinguish between "safe" failure due to SIGPIPE and fatal errors due to some other problem.

## 4.3 Reliable Writes

In the example above we wrote to a temporary file and renamed it into place, as we still needed the input file around while we were generating the output file. This isn't always the case, but the approach remains a good one.

Imagine a script which rewrites your resolv.conf when your computer gets a new DHCP lease. The part of it that rewrites the file might look something like this:

```
echo "search $mydomain" > /etc/resolv.conf
echo "nameserver $nameservers" >> /etc/resolv.conf
```

But consider what happens if something goes wrong while this is running - if the disc fills up, if the power fails, etc. Your resolv.conf will be be empty or incomplete and whatever information about nameserver addresses, etc, was in the old version will have been destroyed.

A common approach is to take a backup copy before writing the new one:

```
cp /etc/resolv.conf /etc/resolv.conf.bak
```

Although this means the information can be easily recovered, we can actually do better by writing to a temporary file in the same directory and then renaming it into place:

```
echo "search $mydomain" > /etc/resolv.conf.tmp
echo "nameserver $nameservers" >> /etc/resolv.conf.tmp
mv /etc/resolv.conf.tmp /etc/resolv.conf
```

Now resolv.conf always has either the old data or the new data in it - there is not a single moment when it is incomplete.

The reason that the file should be in the same directory is because that's the easiest way to guarantee that it's on the same filesystem; this is useful because mv across filesystems is in fact implemented as a copy, which has all the disadvantages of the original version.

There is a downside to this approach: if the file to be written is a symlink then it will be replaced by an ordinary file. Also, if (through hard links) the file has more than one name in the file system, then the connection between the names will be broken. The former problem can be solved by following the link to find the "real" name of the file, then latter is not really possible to work around - if this is a problem for you then the next best bet is to take a backup of the file before rewriting it and put up with it being incomplete if something does go wrong.

---

# Appendix A: Other Material

## A.1 References

The Single UNIX Specification (http://www.opengroup.org/onlinepubs/7908799/) has a section on Shell Command Language (http://www.opengroup.org/onlinepubs/007908799/xcu/chap2.html).

Most UNIX systems will respond to the command "man sh" with the documentation for their shell.

At http://language.perl.com/ppt/v7/sh.1 is a copy of the man page for the UNIX v7 implementation of /bin/sh.

## A.2 Faking Errors

I had to fake up some of the errors above. The technique I used to do this was to create a shared object to simulate the error and insert it into the running program with LD_PRELOAD. The source file is werror.c; it works on my Debian system.

---

# Administrivia

RJK | Contents