Filenames and Pathnames in Shell: How to do it Correctly

David A. Wheeler

2015-10-09 (original version 2010-05-19)

Many Bourne shell scripts (as run by bash, dash, ash, ksh, and so on) do ***not*** handle filenames and pathnames correctly on Unix-like/POSIX systems. Some shell programming books teach it wrongly, and even the POSIX standard sometimes gets it wrong. Thus, many shell scripts are buggy, leading to surprising failures and in some cases security vulnerabilities (see the "Secure Programming for Linux and Unix HOWTO" section on filenames, CERT's "Secure Coding" item MSC09-C, CWE 78, CWE 73, CWE 116, and the CWE/SANS Top 25 Most Dangerous Programming Errors). This is a real problem, because on Unix-like systems (e.g., Unix, Linux, or POSIX) shells are universally available and widely used for lots of basic tasks.

This essay shows common *wrong* ways to handle filenames and pathnames in Bourne shells, and gives a summary of how to do it correctly for the impatient. It then walks through rationale so you can *understand* why common techniques do not work... and why the alternatives do. I presume that you already know how to write Bourne shell scripts.

The basic problem is that today most Unix-likes allow filenames to include almost *any* bytes. That includes newlines, tabs, the escape character (including escape sequences that can execute commands when displayed), other control characters, spaces (anywhere!), leading dashes (-), shell metacharacters, and byte sequences that aren't legal UTF-8 strings. So your scripts could be fail or even be subverted if you ever unarchive "tar" or "zip" files from someone else, examine directories with files created by someone else, or simply create files yourself that contain shell metacharacters (like space or question mark).

This is not a *just* a shell problem. Lots of code in *all* languages (not just shell), and at least some GUI toolkits, do not handle all permitted filenames and pathnames correctly. Some GUI toolkits (e.g., file-pickers) presume that filenames are always in UTF-8 and never contain control characters, even though neither are necessarily true.

However, this flaw in Unix-like kernels (allowing dangerous filenames) combines with additional weaknesses in the Bourne shell language, making it even *more* difficult in shell to correctly handle filenames and pathnames. I think shell is a reasonable language for short scripts, when properly used, but the excessive permissiveness of filenames turns easy tasks into easily-done-wrong tasks. A few small changes would make it much easier to write secure code for handling filenames for all languages including shell. So if your script may handle unarchived files, or files created by different user or mobile app, then your script needs to handle this botched situation. Tools like shellcheck can help you find some of these problems, but not all of them, and you can use such tools more effectively if you understand the problem.

First, though, some key terminology. A *pathname* is used to identify a particular

file, and may include zero or more "/" characters. Each pathname component (separated by "/") is a officially called a filename; pathname components (aka filenames) cannot contain "/". So officially "/usr/bin/sh" is a pathname, with pathname components (filenames) inside it, that refers to a particular file. (Note: on Cygwin, "\" is a synonym for "/", so it also separates pathname components.) In practice, many people use the term "filename" to mean both pathname components (which are officially filenames) and entire pathnames. Neither pathname components nor full pathnames can contain the NUL character (\0), because that is the terminator, and pathname components also cannot include "/"; those turn out to be the only rules you can really count on today.

1. How to do it wrongly

First, let's go through some examples that are *wrong*, because the first step to fixing things is to know what's broken. These examples assume default settings (e.g., there is no "set -f" or "IFS=..."):

```
cat * > ../collection  # WRONG
```

This is wrong. If a filename in the current directory begins with "-", it will be misinterpreted as an option instead of as a filename. For example, if there's a file named "-n", it will suddenly enable cat's "-n" option instead if it has one (GNU cat does, it numbers the lines). In general you should **never** have a glob that begins with "*" — it should be prefixed with "./". Also, if there are no (unhidden) files in the directory, the glob pattern will return the pattern instead ("*"); that means that the command (cat) will try to open a file with the improbable name "*".

```
for file in * ; do  # WRONG
  cat "$file" >> ../collection
done
```

Also wrong, for the same reason; a file named "-n" will fool the `cat` program, and if the pattern does not match, it will loop once with the pattern itself as the value.

```
cat $(find . -type f) > ../collection  # WRONG
```

Wrong. If any pathname contains a space, newline, or tab, its name will be split (file "a b" will be incorrectly parsed as two files, "a" and "b"). If a pathname contains a globbing character like *, the shell will try to expand it, potentially creating additional problems. Also, if the find command matches no files, the command will be run with no parameters; on many commands (like cat) this will cause the program to hang on input from standard input (you can fix this by appending pathname /dev/null, but many people do not know to do that).

```
( for file in $(find . -type f) ; do  # WRONG
    cat "$file"
  done ) > ../collection
```

Wrong, for similar reasons. This breaks up pathnames that contain space, newline, or tab, and it incorrectly expands pathnames if the pathnames themselves contain characters like “*”.

```
( find . -type f |   # WRONG
  while read file ; do cat "$file" ; done ) > ../collection
```

Wrong. This works if a pathname has spaces in the middle, but it won’t work correctly if the pathname begins or ends with whitespace (they will get chopped off). Also, if a pathname includes “\”, it’ll get corrupted; in particular, if it ends in “\”, it will be combined with the next pathname (trashing both). In general, using “read” in shell without the “-r” option is usually a mistake, and in many cases you should set `IFS=""` just before the read.

```
( find . -type f | xargs cat ) > ../collection # WRONG
```

Wrong. By default, xargs’ input is *parsed*, so space characters (as well as newlines) separate arguments, and the backslash, apostrophe, double-quote, *and* ampersand characters are used for quoting. According to the POSIX standard, you have to include the option `-E ""` or underscore *may* have a special meaning too. Note that many of the examples in the POSIX standard xargs section are wrong; pathnames with spaces, newlines, or many other characters will cause many of the examples to fail.

```
( find . -type f |
  while IFS="" read -r file ; do cat "$file" ; done ) \
         > ../collection # WRONG
```

Wrong. Like many programs, this assumes that you can have list of pathnames, with one pathname per line. But since pathnames can internally include newline, all simple line-at-a-time processing of pathnames is wrong! This construct is fine if pathnames can’t include newline, but since many Unix-like systems permit, attackers are happy to use this false assumption as an attack.

```
cat $file
```

Wrong. If $file can contain whitespace, then it could broken up and interpreted as multiple file names, and if $file starts with dash, then the name will be interpreted as an option. Also, if $file contains metacharacters like “*” they will be expanded first, producing the wrong

set of filenames.

2. Doing it correctly: A quick summary

So, how can you process pathnames *correctly* in shell? Here's a quick summary about how to do it correctly, for the impatient who "just want the answer".

2.1. Basic rules

1. Double-quote all variable references and command substitutions unless you are *certain* they can only contain alphanumeric characters or you have specially prepared things (i.e., use `"$variable"` instead of `$variable`). In particular, you should practically always put `$@` inside double-quotes; POSIX defines this to be special (it expands into the positional parameters as *separate* fields even though it is inside double-quotes).
2. Set IFS to just newline and tab, if you can, to reduce the risk of mishandling filenames with spaces. Use newline or tab to separate options stored in a single variable. Set IFS with `IFS="$(printf '\n\t')"`
3. Prefix all pathname globs so they cannot expand to begin with "-". In particular, never start a glob with "?" or "*" (such as "*.pdf"); always prepend globs with something (like "./") that cannot expand to a dash. So never use a pattern like "*.pdf"; use "./*.pdf" instead.
4. Check if a pathname begins with "-" when accepting pathnames, and then prepend "./" if it does.
5. Be careful about displaying or storing pathnames, since they can include newlines, tabs, terminal control escape sequences, non-UTF-8 characters (or characters not in your locale), and so on. You can strip out control characters and non-UTF-8 characters before display using `printf '%s' "$file" | LC_ALL=POSIX tr -d '[:cntrl:]' | iconv -cs -f UTF-8 -t UTF-8`
6. Do *not* depend on always using "`--`" between options and pathnames as the primary countermeasure against filenames beginning with "-". You have to do it with *every* command for this to work, but people will *not* use it consistently (they never have), and many programs (including `echo`) do not support "`--`". Feel free to use "`--`" between options and pathnames, but only as an *additional* optional protective measure.
7. Use a template that is known to work correctly; below are some tested templates.
8. Use a tool like shellcheck to find problems you missed.

2.2. Template: Using globs

```
# Correct portable glob use: use "for" loop, prefix glob, check for existence:
# (remember that globs normally do NOT include files beginning with "."):
for file in ./* ; do        # Prefix with "./*", NEVER begin with bare "*"
  if [ -e "$file" ] ; then  # Make sure it isn't an empty match
    COMMAND ... "$file" ...
  fi
done
```

```
# Correct portable glob use, including hidden files (beginning with "."):
for file in ./* ./.[!.]* ./..?* ; do        # Prefix with "./*"
  if [ -e "$file" ] ; then  # Make sure it isn't an empty match
    COMMAND ... "$file" ...
  fi
done

# Correct glob use, simpler but requires nonstandard bash extension nullglob:
shopt -s nullglob  # Bash extension, so globs with no matches return empty
for file in ./* ; do        # Use "./*", NEVER bare "*"
  COMMAND ... "$file" ...
done

# Correct glob use, simpler but requires nonstandard bash extension nullglob;
# you can do things on one line if you can add /dev/null as an input.
shopt -s nullglob  # Bash extension, so globs with no matches return empty
COMMAND ... ./* /dev/null
```

2.3. Template: Using find

The find command is great for recursively processing directories. Typically you would specify other parameters to find (e.g., select only normal files using “-type f”, or skip filenames “.” and “..”), which is not shown here. Below are the forms that always work (though some require nonstandard extensions or fail with Cygwin), followed by simpler ones with serious limitations.

2.3.1. Always works

```
# Simple find -exec; unwieldy if COMMAND is large, and creates 1 process/file:
find . -exec COMMAND... {} \;

# Simple find -exec with +, faster if multiple files are okay for COMMAND:
find . -exec COMMAND... {} \+

# Use find and xargs with \0 separators
# (nonstandard common extensions -print0 and -0. Works on GNU, *BSDs, busybox)
find . -print0 | xargs -0 COMMAND

# Head-busting, but it works portably.  Use '\'' for single-quote in command.
# Runs a subshell, so variable values are lost after each iteration:
find . -exec sh -c '
 for file do
    ...  # Use "$file" not $file
 done' sh {} +

# find... while loop, requires find (-print0) and shell (read -d) extensions.
# Fails on Cygwin; in while loops, filenames ending in \r \n and \n look =.
# Variable values may be lost unset because loop may run in a subshell.
find . -print0 | while IFS="" read -r -d "" file ; do ...
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done

# while + find with process substitution.
# Requires nonstandard read -d (bash ok) and find -print0.
# Rquires nonstandard process redirection <(...); bash, zsh, and ksh 93
# have process redirection, but dash and ksh 88 do not. Also,
```

```
# the underlying system must have named pipes (FIFOs) or /dev/fd.
# Fails on Cygwin; in while loops, filenames ending in \r \n and \n look =.
# Variables *do* retain their value after the loop ends, and
# you can read from stdin (change the 4s to another number if fd 4 is needed):
while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done 4< <(find . -print0)

# Named pipe version.
# Requires nonstandard extensions to find (-print0) and read -d (bash ok);
# underlying system must inc. named pipes (FIFOs).
# Fails on Cygwin; in while loops, filenames ending in \r \n and \n look =.
# Variables *do* retain their value after the loop ends, and
# you can read from stdin (change the 4s to something else if fd 4 needed).
mkfifo mypipe
find . -print0 > mypipe &
while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done 4< mypipe

# Use the author's encodef program.
# Variables *do* retain their value after the loop ends.
# This version is POSIX portable, but you must have encodef. In practice
# you can often use "-print0" instead of POSIX "-exec printf '%s\0' {} \;"
for encoded_pathname in $(find . -exec printf '%s\0' {} \; | encodef ) ; do
  file="$(encodef -dY -- "$encoded_pathname")" ; file="${file%Y}"
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done
```

2.3.2. Limitations

It is sometimes easier to not fully handle pathnames, especially if you are trying to write portable shell code. However, that code can quickly become a security vulnerability if you use it to examine expanded archives (such as zip or tar files), or examine a directory with files created by another (e.g., a remote filesystem, a virtual machine controlled by someone else or an attacker, another mobile app, etc.). Here are examples (and their limitations):

```
# Okay if pathnames can't contain tabs or newlines; beware the assumption:
IFS="$(printf '\n\t')"
set -f # Needed for filenames with *, etc; see below on saving/restoring
for file in $(find .) ; do
  COMMAND "$file" ...
done

# Okay if pathnames can't contain tabs or newlines; beware the assumption:
IFS="$(printf '\n\t')"
set -f # Needed for filenames with *, etc; see below on saving/restoring
COMMAND $(find .) /dev/null

# Okay if pathnames can't contain newlines; beware the assumption.
# Also, this makes stdin inaccessible, and variables may not stay set.
find . | while IFS="" read -r file ; do ...
  COMMAND "$file" # Use "$file" not $file everywhere.
done

# You can securely use the above approaches, even if directories have
```

```
# evil filenames, if you can skip evil filenames.   For example, here's how to
# skip pathnames with embedded control chars, including newline and tab:
IFS="$(printf '\n\t')"
controlchars="$(printf '*[\001-\037\177]*')"
set -f # Needed for filenames with *, etc; see below on saving/restoring
for file in $(find . ! -name "$controlchars") ; do
  COMMAND "$file" ...
done

# Skip pathnames with embedded control chars, including newline and tab:
IFS="$(printf '\n\t')"
controlchars="$(printf '*[\001-\037\177]*')"
set -f # Needed for filenames with *, etc; see below on saving/restoring
COMMAND $(find . ! -name "$controlchars") /dev/null

# Here's one way to quickly exit a program if a filename
# contains a control character (including tabs, newlines, and ESC) or DEL.
# My thanks to Michael Thayer for this suggestion.
expr "$filename" : "`printf '.*[\01-\037\0177*?]'`" && exit 1
```

2.4. Template: Building up a variable

There's no easy portable way to handle multiple arbitrary filenames in one variable and then directly use them. Shell arrays work, but can be tricky to use in this case and are not portable. I suggest forbidding filenames with tabs and newlines; then you can easily use those characters as separators like this:

```
# If you build up options in a string, use tab|newline to separate filenames
IFS="$(printf '\n\t')"
tab="$(printf '\t')"
command_options="-x${tab}-y"
# If you want to put pathnames in built-up string, prevent tab|newline
# in the pathname, use "set -f", and then you can use an unquoted variable.
# E.g., presuming that $file doesn't contain tab|newline, -F $file is:
command_options="${options}-F${tab}${file}"
set -f # Needed for filenames with *, etc; see below on saving/restoring
mycommand $command_options "$another_pathname"
```

2.5. Template: Saving and restoring "set -f"

Sometimes you need to disable file globbing in shell, especially when receiving information from find. POSIX includes various portable mechanisms to disable and re-enable file globbing in shell. The "set -f" command disables file globbing. You can use "set -f" to disable file globbing, and "set +f" to re-enable it. But what if you want to use "set -f" to disable file globbing temporarily, and later restore whatever it was before? One way is to put the "set -f" and what it depends on in a subshell; that works, but then variable settings are lost once the subshell is done. You can also save and restore shell option settings by doing this:

```
oldSetOptions=$(set +o)             # Save shell option settings
... (set -f, etc.)
eval "$oldSetOptions" 2> /dev/null  # Restore shell option settings
```

3. Rationale for the basic rules

Here is the rationale for each of the basic rules.

3.1. Double-quote parameter (variable) references and command substitutions

As described by any Bourne shell programming book, *always* use double-quotes (") to surround variable references and command substitutions, unless you are certain they can only produce alphanumeric characters or you have specially prepared things. The dangerous characters are whitespace or shell pathname expansion (glob) characters like "*", because unquoted variable references and command substitutions undergo shell field splitting and pathname expansion:

- Field splitting splits a word into multiple words; by default they are split by space, tab, or newline. This expansion can be controlled or disabled by setting the variable `IFS`, but you have to specially set it.
- Pathname expansion looks for filename patterns, and if they are found, splits up a word into multiple words for each filename. If the unquoted variable reference or command substitution produces a character like `*`, shell will normally try to replace that with a list of the filenames that match the pattern. This expansion can be disabled using "`set -f`".

The good news is that once you get into the habit, this is an easy style rule to follow. Even if you know that they can only produce characters that will not cause problems, quoting is a good idea, since the script might change in the future. It is easy to remember "alphanumeric characters okay" than a more complicated rule, and if you allow more than alphanumeric characters, it is likely that the variable will eventually allow dangerous characters. Lots of scripts already follow this rule, so while it's annoying, it's not too bad. Here are some examples:

Don't use	Instead use
`$file`	`"$file"`
`$(pwd)`	`"$(pwd)"`
`$(dirname $file)`	`"$(dirname "$file")"`

By the way, it turns out that the POSIX spec is unclear whether or not field splitting applies to arithmetic expansion in shell; most (but not all) implementations do apply field splitting in this case.

3.2. Set IFS to just newline and tab at the start of each script

One of the first non-comment commands in every shell script should be:

```
IFS="$(printf '\n\t')"
# or:
IFS="`printf '\n\t'`"
# Widely supported, POSIX added http://austingroupbugs.net/view.php?id=249
IFS=$'\n\t'
```

This sets the IFS variable so that the "space" character is no longer an input field separator, and thus only newline and tab are field separators. If you need to run on

really old systems the second form with backquotes is better, but the first one is easier to read, POSIX compliant, and very portable - it works on any system not in a museum.

This doesn't help security very much, but it does help reliability. If you make a mistake in your script, and the script encounters a pathname (or other data) with a space, your script is more likely to work correctly. Filenames with tabs and newlines are almost never used except by attackers, but users often use spaces; doing this will prevent file splitting from unintentionally splitting up filenames with spaces. So if you forget to surround a variable reference with double-quotes, or use a for loop with a simple command substitution, it is less likely to fail. It is also really easy to do this; just add one line near the top.

The recommended IFS command sets newline and then tab. It is harder to do it in the other order in some shells, because $(...) consumes trailing newlines. The easy way is to use `IFS=$'\t\n'`, which is widely supported but has only recently been added to POSIX.

You can still build a list of command options inside a single shell variable, even when space isn't in IFS. You just need to use tab or newline to separate parameters, and *not* space. You can even embed pathnames with spaces in this variable, since spaces are no longer field separators.

You might also want to put "`set -eu`" or at least "`set -u`" at the beginning of your scripts, along with setting IFS. This does nothing for pathnames, but these *can* help detect other script errors.

By the way, the need for proper quoting is not limited to Bourne shells. The Windows shell also requires proper quoting, and improper quoting can lead to vulnerabilities. A user merely needs to create filenames with characters such as ampersands, and an improperly-quoted shell program might end up running it. For example, imagine if an attacker can create a directory of the form "name&command_to_execute", say on a fileserver. Then a Windows script which fails to quote properly (e.g., it has `ECHO %CD%` or `SET CurrentPath=%CD%` without putting double-quotes around `%CD%`) would end up running the command of the attacker's choosing.

3.3. Prefix all globs so they cannot expand to begin with "-"

A "glob" is a pattern for pathname matching like "*.pdf". Whenever you use globbing to select files, *never* begin with a globbing character (typically the characters "*", "?", or "["). If you're starting from the current directory, prefix the glob with "./" like this:

```
cat ./*                  # Use this, NOT "cat *" ... Must have 1+ files.
for file in ./* ; do     # Use this, NOT "for file in *" (beware empty lists)
  ...
done
```

This is important because almost all commands will interpret a string beginning

with dash as an option, not as a filename, until they see something that does not begin with dash. Globs are expanded by the shell into a list of filenames, and dash is earlier in the sort order compared to before alphanumerics, so it is easy for attackers to make this happen.

If you always prefix pathnames (e.g., those acquired through globs), then pathnames starting with "-" will always be handled correctly. Globbing is often the easiest way to handle all files, or a subset of them, in a specific directory, but you need to make sure you do it correctly.

3.4. Check if a pathname begins with "-" when accepting pathnames, and then prepend "./" if it does

Similar to the previous rule, if you read in a pathname, as early as possible see if it begins with "-"... if it does, prepend "./". This eliminates this source of pathnames that are confused as option flags.

3.5. Be careful about displaying or storing pathnames

Filter or encode pathnames before displaying them. The biggest problem is that pathnames could contain control characters that control the terminal and/or the GUI display, causing nasty side-effects on display. Displaying pathnames can even cause a security vulnerability in some situations (!). If you must display pathnames, consider encoding or stripping out control characters first (many ls implementations do this when the output is a terminal). You can strip out the control characters this way:

```
printf '%s' "$file" | LC_ALL=POSIX tr -d '[:cntrl:]'
```

In addition, you have no way of knowing for certain what the pathname's character encoding is, so if you got a pathname from someone else, and they do not use UTF-8 (including ASCII), you're likely to end up with garbage mojibake.

In practice, what most people do is exchange pathnames and hope that they are UTF-8. If you both use the same locale, you could use that instead, but UTF-8 is the only encoding in wide use for Unix pathnames that can handle arbitrary languages. Most modern GUI toolkits presume that filenames are UTF-8, even though nothing actually ensures that this is true. If you must display pathnames, consider forcing them to display as UTF-8. I encourage you to *always* encode pathnames in UTF-8... but beware that nothing actually enforces this common convention. Thus, you will want to enforce it yourself where you can.

One way you can avoid displaying non-UTF-8 filenames in shell is to try to convert them to UTF-8 using *iconv*. The iconv program is in POSIX, and it can strip out characters not in a given encoding. Sadly, the encodings that must be supported by iconv are not standardized. Still, GNU iconv supports UTF-8, and other systems are likely to do so, so this will probably work:

```
printf '%s' "$file" | iconv -cs -f UTF-8 -t UTF-8
```

A common approach for storing pathnames in files, or to transmit them in data

formats, is to separate them with newlines and/or tabs. Sadly, this does not work in the general case, since pathnames can include both characters. You need to forbid such nasty filenames, escape them, or use \0 to separate the pathnames. If you can forbid them, that is the easiest... but you may not have that option.

3.6. Do not depend on "--"

Many books, and the POSIX standard, mistakenly advocate using "`--`" between the options and pathnames as the primary method to deal with filenames beginning with "-". This is *impractical* and *bad advice*:

1. For "`--`" to work, all maintainers would have to faithfully use "`--`" in practically every command invocation. That just doesn't happen in real life, even after decades of people trying. People forget it all the time; no one is that consistent, especially since code seems to work without it. Very few commands *require* it, after all.
2. You can't do it anyway, even if you were perfectly consistent; many programs and commands do not support "`--`". POSIX even explicitly *forbids* echo from supporting "`--`", and echo must support "-n" (and GNU coreutils echo supports other options too).

Thus, as a practical matter you need to do something else; by always prefixing filenames if they start with dash, as recommended earlier, the problem disappears.

Do feel free to use "`--`" between options and pathnames, when you can do it, as an *additional* protective measure. But using "`--`" as your primary (or only) mechanism for dash-prefixed filenames is bad idea. You are better off prefixing the pathnames when you get the pathname, since then you only have to do it once per pathname. Once you prefix the pathname it doesn't matter if you remember "`--`" or not; it just works correctly.

3.7. Yes, there's more

Sadly, there's more. There are two major ways to get sets of pathnames in the shell, glob patterns and the find command. Globs are primarily useful for a short list of unhidden filenames in one directory; find is useful for other situations, including recursively descending into subdirectories.

In both cases you have to worry around what happens when there are zero matches. If you just gave "command" and something that gave a list of filenames, most commands will hang while trying to read from standard input. The easy solution in this case is to add "/dev/null" to the end... assuming you can do that.

The next two sections examine glob patterns and the find command in turn.

4. Globbing

Globbing is a simple language specifically designed for filename handling, primarily to create lists of unhidden files in a particular directory. In this language,

“*” matches all non-hidden files in the current directory, “*.pdf” matches all non-hidden files in the current directory ending in “.pdf” - and so on.

The good news about globbing in shell is that glob expansion is built into the shell and done *after* field (IFS) expansion. Thus, as long as you directly use globs as command parameters or as part of a “for” loop, you will have no problem with pathnames containing whitespace or control characters (since they will not undergo field expansion). There is also no challenge getting the information back into shell; the shell is doing the processing.

However, if a pathname begins with “-”, glob will dutifully expand it, confusing any command later. As noted above, the recommended solution is to always prefix a glob with something that does not begin with dash, such as “./”.

Remember that globbing normally skips hidden files (those beginning with “.”). Often that is what you want. If you want the hidden files in a directory instead, you may want to use “find” instead. You can get the hidden files with a glob by adding two more globbing patterns:

```
.[!.]* ..?*
```

In many cases even a simple glob *could* fail to match, and adding globbing patterns to find hidden files makes this even more likely... which leads us to the problem of handling empty pathname lists.

4.1. Beware of globs if there might be empty lists of pathnames

Beware of globbing if there might be no matches with the pattern (and this is often the case). By default, if a glob like `./*.pdf` matches no files, then the original glob pattern will be returned instead.

This is almost never what you want. E.g., in a “for” loop this will cause the loop to execute once, but with the pattern instead of a pathname! Similarly, if you use a glob on a command line, such as `cat ./*pdf`, the result will be a request to open a non-existent file... which is almost never what you want.

You can use use globbing in a `for` loop, even if it might not match anything, using one of two approaches. One approach, which is completely portable, is to re-test for the existance of the file before using it in the loop:

```
for file in ./* ; do        # Use this, NOT "for file in *"
  if [ -e "$file" ] ; then  # Make sure it exists and isn't an empty match
    COMMAND ... "$file" ...
  fi
done
```

This is both ugly and a little inefficient (you have to re-test each file again). There are also pathological cases where the pattern doesn’t match but there is a file that is identical to the unmatched pattern (though for typical patterns that can’t happen), so you have to check your pattern to see if that could happen.

A more efficient but nonstandard solution for empty matches is to use a nonstandard shell extension called "null globbing". Null globbing fixes this by replacing an unmatched pattern with nothing at all. In bash you can enable nullglob with "`shopt -s nullglob`". In zsh, you can use `setopt NULL_GLOB` for the same result. Then this will work correctly:

```
shopt -s nullglob  # Bash extension, so that empty glob matches will work
for file in ./* ; do        # Use this, NOT "for file in *"
    COMMAND ... "$file" ...
done
```

Null globbing can work well on the command line too, but there's a catch. If all patterns might be empty, you have to include at least one file (such as /dev/null) that is okay to include, *or* it needs to be okay to run the command without any pathname arguments. Thus, you can use "cat ./*.pdf /dev/null".

Another problem with globbing is that if the list of matches is too long, on some older shells it will *also* fail. In short, in robust scripts, globbing should normally be used only as a "for" loop's list.

4.2. The globstar extension

Traditional globbing is only useful when you want to process files in a particular directory. Some shells have added a nonstandard "globstar" extension, but it's both nonstandard and has various limitations. I discuss it here, but you probably want to use `find` (discussed next).

With the globstar extension, the pattern "**" returns every pathname (including directories) in the current directory, recursively; it omits dot files, doesn't descend into dot dirs, and sorts the file list.

Bash version 4 recently added this, but you must enable it with "`shopt -s globstar`". The zsh shell originally came up with this, and ksh93 was the first to copy it (but in ksh you have to enable it with "set -G"). Note that there's no standard way to invoke it!

If you use this in a `for` loop list and combine it with nullglob, you can handle absolutely all pathnames easily and efficiently, including the empty case. That sounds great, but watch the fine print... I think there are many reasons to avoid this right now. It's nonstandard, and gives you little control over the recursion. Most importantly, at least some implementations have trouble if there are links in the directories. Bash 4, at least, can get stuck in infinite loops if there are links. In many cases, `find` is currently the better approach for reliably doing recursive descent into directories.

5. Find

If you want to process files beyond what normal globbing can do (e.g., recursively handle directories), or you don't like the limitations on having to re-check for non-matches, use `find`. Re-implementing accurate directory traversal in the shell is

possible, but both painful and silly; you would have to deal with symbolic links, hard links, renames during traversal, and other problems. The `find` tool is designed to handle this job; let it do its job. Sometimes you want to retrieve pathnames from programs other than find; in those cases the issues tend to be very similar.

An advantage of `find` is that it has lots of options for controlling how you process files and directories. You can use options to limit it to one directory, determine the ordering, and so on. It normally processes all files (including hidden ones), but you can use this pattern to skip hidden files (omit the "?" to skip directory "."):

```
find . -name '.?*' -prune -o ....
```

The `find` command is always passed a starting directory, and it always returns values beginning with that directory. Thus, as long as the starting directory doesn't begin with "-" you won't have a problem with leading "-".

If you can directly use the find "-exec" option to run the command you want to use with the file, that is the easy way. Sadly, this is awkward to do in many cases, so you often want to get pathames from find back into the shell.

A challenge with find (and any other external program) is that it is more complicated to portably get information back into the shell. Pathnames can embed newlines; that means that reading filenames line-by-line fails (e.g, by read), splitting using IFS and newline cannot work directly, and command command substitution $(...) is awkward because it strips away trailing newlines. There are additional problems with Cygwin; Cygwin sometimes silently maps ending \r\n into \n, but it is legal to have filenames that differ only because one ends in \r\n and the other ends in \n; As a result, some constructs for handling arbitrary filenames do not work on Cygwin. Pathnames can also embed tabs and spaces, which by default causes unwanted field splitting. The usual approach is to separate pathnames with \0, but the mechanisms to handle this are not in the POSIX standard (e.g., options for "read" or "xargs" to handle such things).

Some of these approaches use "read", which is tricky to use. You typically need to use the "-r" option (so backslash is considered part of the line, and not an escape mechanism; otherwise filenames containing backslashes will cause problems). You also typically have to set IFS to be empty for the read command; otherwise, a pathname that includes IFS characters at the end of a filename would be corrupted (see the POSIX.1-2008 specification lines 103920-103925). If you want to read pathnames separated by \0 you'll need another option, the discussion on null-separated pathnames gives more detail.

You'd like to use simple "for" loops, but by default if there is a pathname (glob) expansion character like "*" in a pathname, find will return the "*". If the shell receives a glob expansion character from a command substitution, by default the character will be *re-expanded* by the shell. You cannot just quote that expansion, either; that would make it appear that the list is just one pathname. Thus, in many cases you need to use "set -f" to disable expansion of filenames.

You could combine xargs with find using a pipe, and use newlines to separate pathnames, but **don't do it**. The problem is that xargs interprets many characters in surprising ways, so it's hard to use xargs correctly when using newlines as separators. The correct *portable* way to use xargs with newline separators requires that you pipe pathnames through another command like `sed` to do character substitutions. The result is complicated, hard to read, rediculously inefficient, and isn't better than many other alternatives (e.g., it doesn't handle newlines either); here it is:

```
find . | sed -e 's/[^A-Za-z0-9]/\\&/g' | xargs -E "" COMMAND # DO NOT DO
```

6. Using null-separated pathnames

If you want to exchange pathnames (in their full generality) between programs, or store them for later, a common solution is to use byte 0 (aka \0 or null) to separate pathnames. This works because pathnames, by definition, cannot include byte 0.

This is very useful, and it works nicely, but note that there are downsides to this approach:

- The easy way to use this convention requires the use of nonstandard extensions that are not a part of the POSIX standard, so the result is non-standard and less portable. Still, many toolsets have a few tool extensions to support this, including the GNU, *BSD, and busybox toolsets. Hopefully someday POSIX will add this.
- The option names to use this convention today (when available) are jarringly inconsistent. In particular, `perl` uses `-0`, while the GNU tools options are as follows: `sort -z`, `find -print0`, `xargs -0`, and `grep` accepts either `-Z` or `--null`. The equivalent option in the bash shell `read` command is `-d ""` (aka empty delimeter).
- This convention is supported by only a few tools (e.g., the GNU toolset includes support, but in only a few of its tools).
- This format is more difficult to view and modify, in part because so few tools support it, compared to the line-at-a-time format that is widely supported.
- Most shells cannot store byte 0 in a variable at all. You can't even pass such null-separated lists back to the shell via command substitution; `cat $(find . -print0)` and similar "for" loops don't work. Even the POSIX standard's version of "read" can't use \0 as the separator (POSIX's read has the -r option, but not bash's -d option).

But if you want maximum generality when recursing into subdirectories, this is a common and relatively painless way to do it.

7. Encoding pathnames

It **is** possible to encode pathnames so that all pathnames can be handled. There is no standard POSIX mechanism for doing this encoding, unfortunately.

encodef is a small utility I wrote that can encode and decode filenames in a few formats. With it, you can do this:

```
# This version is POSIX portable; in practice
# you can often use "-print0" instead of "-exec printf '%s\0' {} \;"
for encoded_pathname in $(find . -exec printf '%s\0' {} \; | encodef ) ; do
  file="$(encodef -d -Y -- "$encoded_pathname")" ; file="${file%Y}"
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done
```

8. A quick aside about newline

Newline can be a little tricky to get into a shell variable. You can't do:

```
newline="$(printf '\n')"
```

Because after the $(...) command is executed, any trailing newline is removed.

One alternative is:

```
newline='
'
```

But this can get corrupted by programs that change the encoding of file end-of-lines.

The following is a standards-compliant trick to get newline into a variable:

```
newline="$(printf '\nX')"
newline="${newline%X}"
```

That is a pain, obviously. More recently, POSIX added support for `$'...'`. Most shells, though not all, already support it. On a shell that does, you can do this:

```
newline=$'\n'
```

9. Could the POSIX standard be changed to make file processing easier?

The POSIX standard could (and should!) be modified to make it easier to to handle the outrageously permissive pathnames that are permitted today. Basically, we need extensions to make globbing and find easier to use.

9.1. Globbing

There are two basic problems with globbing:

1. Globbing in shell returns junk (the pattern) when there are zero matches. There should be a shell option (typically called a "nullglob" option) so an empty list is returned if nothing matches and there was at least one metacharacter. Oddly enough, the underlying glob() function has an option that's close to this, but there's no standard way for shells to take advantage of it! Bash, ksh, and others have support, but not in a common standard way, and glob() doesn't support exactly what is needed either (bugid:247).
2. Globbing normally replies pathnames beginning with "-". There should be an option that when set prepends "./" to any glob result that begins with "-". This should be an option for glob(), as well as for the shell. I think the standard

should also state that implementations *may* enable this by default. Not all real-world commands support "--", and users often forget to add it; we need to have a mechanism to automatically deal with pathnames beginning with "./" if you need them.

9.2. Find / null separators

There also needs to be standard way to use `find` with arbitrary pathnames. The normal way to handle this is by separating pathnames with the null (\0) character; a few changes would simplify this:

1. Extend existing commands to generate or use null-separated pathname lists. At the least, add "find -print0" (bugid:243) and "xargs -0" (bugid:244) since these are already widely implemented. For consistency, I think "-0" should be the standard option name for null-separated lists. It'd be useful to add "grep -0" (GNU grep accepts either -Z or --null) and "sort -0" (GNU sort uses -z). Once added, this common template would be portable:

   ```
   find . -print0 | xargs -0 COMMAND
   ```

2. Extend the shell's `read` so that it can easily read null-separated streams. (bugid:245) Bash can do this today, but it's painful; the command is `IFS="" read -d "" -r` which is overly complicated I believe there should be a new "-0" option for read, which says "ignore IFS, and just read until the next \0 byte" (Here's a bash 4.1 patch). You can then do this (which makes it easier to have long command sequences, as long as you don't need stdin):

   ```
   find . -print0 | while read -0 file ; do ... done
   ```

3. Extend the shell so that its `for` loop can handle a null-separated list. This one is harder; it's not obvious how to do this. My current theory is that there be a new shell option 'nullseparator"; when enabled, IFS is ignored, and instead \0 is the input seperator. Then, extend the shell's `for` loop syntax so that if you say `then in` instead of `in`, this mode is temporarily enabled while the list is processed (the original setting is then restored). (I originally had `null in`, but using `then in` means that no new keywords are needed.) You could then do this:

   ```
   for file then in $(find . -print0) do ... done
   ```

As a side note, it'd be nice if the `$'...'` construct was standard, as it makes certain things easier (bugid:249).

10. If pathnames were limited, would it be better?

Shell programming is remarkably easy in many cases; what's sad is that this common case (file processing) is far complicated than it needs to be. This is *not* a problem limited to shell; while shell is especially tricky, it is difficult to correctly process POSIX pathnames in *all* languages.

Fundamentally, the rules on pathnames are too permissive. Extending POSIX would make it somewhat easier, and we should do that. However, It would be *much*

simpler if systems imposed a few simple rules on pathnames, such as prohibiting control characters (bugid:251), prohibiting leading “-”, and requiring pathnames to be UTF-8. Then you could always print pathnames safely, and these “normal” shell constructs would always work:

```
# This works if pathnames never begin with "-" and nullglob is enabled:
for file in *.pdf ; do ... done           # Use "$file" not $file
# This works if pathnames have no control chars and IFS is tab and newline:
set -f
for file in $(find .) ; do ... done        # Use "$file" not $file
```

A good general principle in security is that you should whitelist input, and only accept patterns that pass the whitelist. However, currenly most kernels have *no* mechanism for whitelisting file creation; they just create whatever garbage comes to mind. Since they accept essentially anything, it becomes much harder to guard against the data later.

I think that we should *both* extend the POSIX standard *and* limit the permitted pathnames. Not all systems will limit pathnames, so we need standard mechanism for them. But the new standard mechanisms simply can’t be as simple as restricting pathnames; restricting pathnames makes systems far easier to use correctly.

Please see my paper on fixing Unix/Linux filenames for more about this.

I’ve also done some work on how to encode/decode pathnames/filenames; see the encodef home page for more information.

But for now, this is how to handle pathnames properly in shell programs.

Feel free to see my home page at http://www.dwheeler.com. You may also want to look at my paper Why OSS/FS? Look at the Numbers! and my book on how to develop secure programs. And, of course, my paper on fixing Unix/Linux/POSIX filenames.