Chapter 3. Special Characters

What makes a character *special*? If it has a meaning beyond its *literal meaning*, a meta-meaning, then we refer to it as a *special character*. Along with commands and keywords, *special characters* are building blocks of Bash scripts.

Special Characters Found In Scripts and Elsewhere

#

Comments. Lines beginning with a # (with the exception of #!) are comments and will *not* be executed.

```
# This line is a comment.
```

Comments may also occur following the end of a command.

```
echo "A comment will follow." # Comment here.
#                             ^ Note whitespace before #
```

Comments may also follow whitespace at the beginning of a line.

```
     # A tab precedes this comment.
```

Comments may even be embedded within a pipe.

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Delete lines containing '#' comment character.
           sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Excerpted from life.sh script
```

A command may not follow a comment on the same line. There is no method of terminating the comment, in order for "live code" to begin on the same line. Use a new line for the next command.

Of course, a quoted or an escaped # in an echo statement does *not* begin a comment. Likewise, a # appears in certain parameter-substitution constructs and in numerical constant expressions.

```
echo "The # here does not begin a comment."
echo 'The # here does not begin a comment.'
```

```
echo The \# here does not begin a comment.
echo The # here begins a comment.

echo ${PATH#*:}       # Parameter substitution, not a comment.
echo $(( 2#101011 ))  # Base conversion, not a comment.

# Thanks, S.C.
```

The standard quoting and escape characters (" ' \) escape the #.

Certain pattern matching operations also use the #.

;

Command separator [semicolon]. Permits putting two or more commands on the same line.

```
echo hello; echo there


if [ -x "$filename" ]; then    #  Note the space after the semicolon.
#+                  ^^
  echo "File $filename exists."; cp $filename $filename.bak
else   #                       ^^
  echo "File $filename not found."; touch $filename
fi; echo "File test complete."
```

Note that the ";" sometimes needs to be *escaped*.

;;

Terminator in a case option [double semicolon].

```
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

;;&, ;&

Terminators in a *case* option (version 4+ of Bash).

.

"dot" command [period]. Equivalent to source (see Example 15-22). This is a bash builtin.

.

"dot", as a component of a filename. When working with filenames, a leading dot is the prefix of a "hidden" file, a file that an ls will not normally show.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

When considering directory names, *a single dot* represents the current working directory, and *two dots* denote the parent directory.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

The *dot* often appears as the destination (directory) of a file movement command, in this context meaning *current directory*.

```
bash$ cp /home/bozo/current_work/junk/* .
```

Copy all the "junk" files to $PWD.

.

"dot" character match. When matching characters, as part of a regular expression, a "dot" matches a single character.

"

partial quoting [double quote]. *"STRING"* preserves (from interpretation) most of the special characters within *STRING*. See Chapter 5.

'

full quoting [single quote]. *'STRING'* preserves all special characters within *STRING*. This is a stronger form of quoting than *"STRING"*. See Chapter 5.

,

comma operator. The *comma operator* [1] links together a series of arithmetic operations. All are evaluated, but only the last one is returned.

```
let "t2 = ((a = 9, 15 / 3))"
# Set "a = 9" and "t2 = 15 / 3"
```

The *comma* operator can also concatenate strings.

```
for file in /{,usr/}bin/*calc
#             ^    Find all executable files ending in "calc"
#+                 in /bin and /usr/bin directories.
do
        if [ -x "$file" ]
        then
          echo $file
        fi
done

# /bin/ipcalc
# /usr/bin/kcalc
# /usr/bin/oidcalc
# /usr/bin/oocalc


# Thank you, Rory Winston, for pointing this out.
```

,,,

Lowercase conversion in *parameter substitution* (added in version 4 of Bash).

\

escape [backslash]. A quoting mechanism for single characters.

\X *escapes* the character *X*. This has the effect of "quoting" *X*, equivalent to *'X'*. The \ may be used to quote " and ', so they are expressed literally.

See Chapter 5 for an in-depth explanation of escaped characters.

/

Filename path separator [forward slash]. Separates the components of a filename (as in `/home/bozo/projects/Makefile`).

This is also the division arithmetic operator.

`

command substitution. The **`command`** construct makes available the output of **command** for assignment to a variable. This is also known as backquotes or backticks.

:

null command [colon]. This is the shell equivalent of a "NOP" (*no op*, a do-nothing operation). It may be considered a synonym for the shell builtin true. The ":" command is itself a *Bash* builtin, and its exit status is *true* (0).

```
:
echo $?   # 0
```

Endless loop:

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done

# Same as:
#    while true
#    do
#      ...
#    done
```

Placeholder in if/then test:

```
if condition
then :   # Do nothing and branch ahead
else     # Or else ...
   take-some-action
fi
```

Provide a placeholder where a binary operation is expected, see Example 8-2 and default parameters.

```
: ${username=`whoami`}
# ${username=`whoami`}   Gives an error without the leading :
#                        unless "username" is a command or builtin...

: ${1?"Usage: $0 ARGUMENT"}     # From "usage-message.sh example script.
```

Provide a placeholder where a command is expected in a here document. See Example 19-10.

Evaluate string of variables using parameter substitution (as in Example 10-7).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
#  Prints error message
#+ if one or more of essential environmental variables not set.
```

Variable expansion / substring replacement.

In combination with the > redirection operator, truncates a file to zero length, without changing its permissions. If the file did not previously exist, creates it.

```
: > data.xxx   # File "data.xxx" now empty.

# Same effect as   cat /dev/null >data.xxx
# However, this does not fork a new process, since ":" is a builtin.
```

See also Example 16-15.

In combination with the >> redirection operator, has no effect on a pre-existing target file (**`: >> target_file`**). If the file did not previously exist, creates it.

This applies to regular files, not pipes, symlinks, and certain special files.

May be used to begin a comment line, although this is not recommended. Using # for a comment turns off error checking for the remainder of that line, so almost anything may appear in a comment. However, this is not the case with :.

```
: This is a comment that generates an error, ( if [ $x -eq 3] ).
```

The ":" serves as a field separator, in `/etc/passwd`, and in the $PATH variable.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

A *colon* is acceptable as a function name.

```
:()
{
  echo "The name of this function is "$FUNCNAME" "
```

```
  # Why use a colon as a function name?
  # It's a way of obfuscating your code.
}

:

# The name of this function is :
```

This is not portable behavior, and therefore not a recommended practice. In fact, more recent releases of Bash do not permit this usage. An underscore _ works, though.

A *colon* can serve as a placeholder in an otherwise empty function.

```
not_empty ()
{
  :
} # Contains a : (null command), and so is not empty.
```

!

reverse (or negate) the sense of a test or exit status [bang]. The ! operator inverts the exit status of the command to which it is applied (see Example 6-2). It also inverts the meaning of a test operator. This can, for example, change the sense of *equal* (=) to *not-equal* (!=). The ! operator is a Bash keyword.

In a different context, the ! also appears in indirect variable references.

In yet another context, from the *command line*, the ! invokes the Bash *history mechanism* (see Appendix L). Note that within a script, the history mechanism is disabled.

*

wild card [asterisk]. The * character serves as a "wild card" for filename expansion in globbing. By itself, it matches every filename in a given directory.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

The * also represents any number (or zero) characters in a regular expression.

*

arithmetic operator. In the context of arithmetic operations, the * denotes multiplication.

** A double asterisk can represent the exponentiation operator or extended file-match *globbing*.

?

test operator. Within certain expressions, the ? indicates a test for a condition.

In a double-parentheses construct, the ? can serve as an element of a C-style *trinary* operator. [2]

```
condition?result-if-true:result-if-false

(( var0 = var1<98?9:21 ))
#                ^ ^

# if [ "$var1" -lt 98 ]
# then
#   var0=9
# else
#   var0=21
# fi
```

In a parameter substitution expression, the ? tests whether a variable has been set.

?

wild card. The ? character serves as a single-character "wild card" for filename expansion in globbing, as well as representing one character in an extended regular expression.

$

Variable substitution (contents of a variable).

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

A $ prefixing a variable name indicates the *value* the variable holds.

$

end-of-line. In a regular expression, a "$" addresses the end of a line of text.

${}

Parameter substitution.

$' ... '

Quoted string expansion. This construct expands single or multiple escaped octal or hex values into ASCII [3] or Unicode characters.

$*, $@

positional parameters.

$?

exit status variable. The $? variable holds the exit status of a command, a function, or of the script itself.

$$

process ID variable. The $$ variable holds the *process ID* [4] of the script in which it appears.

()

command group.

```
(a=hello; echo $a)
```

A listing of commands within *parentheses* starts a subshell.

Variables inside parentheses, within the subshell, are not visible to the rest of the script. The parent process, the script, cannot read variables created in the child process, the subshell.

```
a=123
( a=321; )

echo "a = $a"   # a = 123
# "a" within parentheses acts like a local variable.
```

array initialization.

```
Array=(element1 element2 element3)
```

{xxx,yyy,zzz,...}

Brace expansion.

```
echo \"{These,words,are,quoted}\"   # " prefix and suffix
# "These" "words" "are" "quoted"
```

```
cat {file1,file2,file3} > combined_file
# Concatenates the files file1, file2, and file3 into combined_file.

cp file22.{txt,backup}
# Copies "file22.txt" to "file22.backup"
```

A command may act upon a comma-separated list of file specs within *braces*. [5] Filename expansion (globbing) applies to the file specs between the braces.

No spaces allowed within the braces *unless* the spaces are quoted or escaped.

```
echo {file1,file2}\ :{\ A," B",' C'}

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

{a..z}

Extended Brace expansion.

```
echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
# Echoes characters between a and z.

echo {0..3} # 0 1 2 3
# Echoes characters between 0 and 3.


base64_charset=( {A..Z} {a..z} {0..9} + / = )
# Initializing an array, using extended brace expansion.
# From vladz's "base64.sh" example script.
```

The *{a..z}* extended brace expansion construction is a feature introduced in version 3 of *Bash*.

{}

Block of code [curly brackets]. Also referred to as an *inline group*, this construct, in effect, creates an *anonymous function* (a function without a name). However, unlike in a "standard" function, the variables inside a code block remain visible to the remainder of the script.

```
bash$ { local a;
              a=123; }
bash: local: can only be used in a
function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (value inside code block)
```

```
# Thanks, S.C.
```

The code block enclosed in braces may have I/O redirected to and from it.

Example 3-1. Code blocks and I/O redirection

```
#!/bin/bash
# Reading lines in /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "First line in $File is:"
echo "$line1"
echo
echo "Second line in $File is:"
echo "$line2"

exit 0

# Now, how do you parse the separate fields of each line?
# Hint: use awk, or . . .
# . . . Hans-Joerg Diers suggests using the "set" Bash builtin.
```

Example 3-2. Saving the output of a code block to a file

```
#!/bin/bash
# rpm-check.sh

#  Queries an rpm file for description, listing,
#+ and whether it can be installed.
#  Saves output to a file.
#
#  This script illustrates using a code block.

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
```

```
fi

{ # Begin code block.
  echo
  echo "Archive Description:"
  rpm -qpi $1       # Query description.
  echo
  echo "Archive Listing:"
  rpm -qpl $1       # Query listing.
  echo
  rpm -i --test $1  # Query whether rpm file can be installed.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 can be installed."
  else
    echo "$1 cannot be installed."
  fi
  echo              # End code block.
} > "$1.test"       # Redirects output of everything in block to file.

echo "Results of rpm test in file $1.test"

# See rpm man page for explanation of options.

exit 0
```

Unlike a command group within (parentheses), as above, a code block enclosed by {braces} will *not* normally launch a subshell. [6]

It is possible to iterate a code block using a non-standard *for-loop*.

{}

placeholder for text. Used after xargs `-i` (*replace strings* option). The {} double curly brackets are a placeholder for output text.

```
ls . | xargs -i -t cp ./{} $1
#            ^^         ^^

# From "ex42.sh" (copydir.sh) example.
```

{} \;

pathname. Mostly used in find constructs. This is *not* a shell builtin.

> Definition: A *pathname* is a *filename* that includes the complete path. As an example, `/home/bozo/Notes/Thursday/schedule.txt`. This is sometimes referred to as the *absolute path*.

The ";" ends the `-exec` option of a **find** command sequence. It needs to be escaped to protect it from interpretation by the shell.

[ ]

test.

Test expression between **[ ]**. Note that **[** is part of the shell *builtin* test (and a synonym for it), *not* a link to the external command `/usr/bin/test`.

[[ ]]

test.

Test expression between [[ ]]. More flexible than the single-bracket [ ] test, this is a shell keyword.

See the discussion on the [[...]] construct.

[ ]

array element.

In the context of an array, brackets set off the numbering of each element of that array.

```
Array[1]=slot_1
echo ${Array[1]}
```

[ ]

range of characters.

As part of a regular expression, brackets delineate a range of characters to match.

$[...]

integer expansion.

Evaluate integer expression between $[ ].

```
a=3
b=7

echo $[$a+$b]   # 10
echo $[$a*$b]   # 21
```

Note that this usage is *deprecated*, and has been replaced by the ((...)) construct.

(( ))

integer expansion.

Expand and evaluate integer expression between (( )).

See the discussion on the ((...)) construct.

> &> >& >> < <>

redirection.

`scriptname >filename` redirects the output of `scriptname` to file `filename`. Overwrite `filename` if it already exists.

`command &>filename` redirects both the `stdout` and the `stderr` of `command` to `filename`.

This is useful for suppressing output when testing for a condition. For example, let us test whether a certain command exists.

```
bash$ type bogus_command &>/dev/null



bash$ echo $?
1
```

Or in a script:

```
command_test () { type "$1" &>/dev/null; }
#                                    ^

cmd=rmdir            # Legitimate command.
command_test $cmd; echo $?   # 0


cmd=bogus_command    # Illegitimate command
command_test $cmd; echo $?   # 1
```

`command >&2` redirects `stdout` of `command` to `stderr`.

`scriptname >>filename` appends the output of `scriptname` to file `filename`. If `filename` does not already exist, it is created.

[i]<>filename opens file `filename` for reading and writing, and assigns file descriptor i to it. If `filename` does not exist, it is created.

process substitution.

(command)>

<(command)

In a different context, the "<" and ">" characters act as string comparison operators.

In yet another context, the "<" and ">" characters act as integer comparison operators. See also Example 16-9.

<<

redirection used in a here document.

<<<

redirection used in a here string.

<, >

ASCII comparison.

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo -n "this does not necessarily imply anything "
  echo "about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

\<, \>

word boundary in a regular expression.

```
bash$ grep '\<the\>' textfile
```

|

pipe. Passes the output (`stdout`) of a previous command to the input (`stdin`) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | sh
#  Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".


cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

A pipe, as a classic method of interprocess communication, sends the `stdout` of one process to the `stdin` of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a *filter*, a command that transforms its input for processing. [7]

`cat $filename1 $filename2 | grep $search_word`

For an interesting note on the complexity of using UNIX pipes, see the UNIX FAQ, Part 3.

The output of a command or commands may be piped to a script.

```
#!/bin/bash
# uppercase.sh : Changes input to uppercase.

tr 'a-z' 'A-Z'
#  Letter ranges must be quoted
#+ to prevent filename generation from single-letter filenames.

exit 0
```

Now, let us pipe the output of **ls -l** to this script.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

The `stdout` of each process in a pipe must be read as the `stdin` of the next. If this is not the case, the data stream will *block*, and the pipe will not behave as expected.

```
cat file1 file2 | ls -l | sort
# The output from "cat file1 file2" disappears.
```

A pipe runs as a child process, and therefore cannot alter script variables.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

If one of the commands in the pipe aborts, this prematurely terminates execution of the pipe. Called a *broken pipe*, this condition sends a *SIGPIPE* signal.

>|

force redirection (even if the noclobber option is set). This will forcibly overwrite an existing file.

||

OR logical operator. In a test construct, the || operator causes a return of 0 (success) if *either* of the linked test conditions is true.

&

Run job in background. A command followed by an & will run in the background.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

Within a script, commands and even loops may run in the background.

Example 3-3. Running a loop in the background

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # First loop.
do
  echo -n "$i "
done & # Run this loop in background.
       # Will sometimes execute after second loop.

echo   # This 'echo' sometimes will not display.

for i in 11 12 13 14 15 16 17 18 19 20   # Second loop.
do
  echo -n "$i "
done

echo   # This 'echo' sometimes will not display.

# ======================================================

# The expected output from the script:
# 1 2 3 4 5 6 7 8 9 10
```

```
# 11 12 13 14 15 16 17 18 19 20

# Sometimes, though, you get:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (The second 'echo' doesn't execute. Why?)

# Occasionally also:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (The first 'echo' doesn't execute. Why?)

# Very rarely something like:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# The foreground loop preempts the background one.

exit 0

#  Nasimuddin Ansari suggests adding    sleep 1
#+ after the   echo -n "$i"   in lines 6 and 14,
#+ for some real fun.
```

A command run in the background within a script may cause the script to hang, waiting for a keystroke. Fortunately, there is a remedy for this.

&&

AND logical operator. In a test construct, the && operator causes a return of 0 (success) only if *both* the linked test conditions are true.

-

option, prefix. Option flag for a command or filter. Prefix for an operator. Prefix for a default parameter in parameter substitution.

```
COMMAND -[Option1][Option2][...]

ls -al

sort -dfu $filename

if [ $file1 -ot $file2 ]
then #      ^
  echo "File $file1 is older than $file2."
fi

if [ "$a" -eq "$b" ]
then #    ^
  echo "$a is equal to $b."
fi
```

```
if [ "$c" -eq 24 -a "$d" -eq 47 ]
then #    ^              ^
  echo "$c equals 24 and $d equals 47."
fi


param2=${param1:-$DEFAULTVAL}
#               ^
```

--

The *double-dash* `--` prefixes *long* (verbatim) options to commands.

`sort --ignore-leading-blanks`

Used with a Bash builtin, it means the *end of options* to that particular command.

This provides a handy means of removing files whose *names begin with a dash*.

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname


bash$ rm -- -badname

bash$ ls -l
total 0
```

The *double-dash* is also used in conjunction with set.

`set -- $variable` (as in Example 15-18)

-

redirection from/to `stdin` or `stdout` [dash].

```
bash$ cat -
abc
abc

...

Ctl-D
```

As expected, **cat -** echoes stdin, in this case keyboarded user input, to stdout. But, does I/O redirection using - have real-world applications?

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Move entire file tree from one directory to another
# [courtesy Alan Cox <a.cox@swansea.ac.uk>, with a minor change]

# 1) cd /source/directory
#    Source directory, where the files to be moved are.
# 2) &&
#   "And-list": if the 'cd' operation successful,
#    then execute the next command.
# 3) tar cf - .
#    The 'c' option 'tar' archiving command creates a new archive,
#    the 'f' (file) option, followed by '-' designates the target file
#    as stdout, and do it in current directory tree ('.').
# 4) |
#    Piped to ...
# 5) ( ... )
#    a subshell
# 6) cd /dest/directory
#    Change to the destination directory.
# 7) &&
#   "And-list", as above
# 8) tar xpvf -
#    Unarchive ('x'), preserve ownership and file permissions ('p'),
#    and send verbose messages to stdout ('v'),
#    reading data from stdin ('f' followed by '-').
#
#    Note that 'x' is a command, and 'p', 'v', 'f' are options.
#
# Whew!



# More elegant than, but equivalent to:
#   cd source/directory
#   tar cf - . | (cd ../dest/directory; tar xpvf -)
#
#     Also having same effect:
# cp -a /source/directory/* /dest/directory
#     Or:
# cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
#     If there are hidden files in /source/directory.


bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#  --uncompress tar file--      | --then pass it to "tar"--
#  If "tar" has not been patched to handle "bunzip2",
```

```
#+ this needs to be done in two discrete steps, using a pipe.
#  The purpose of the exercise is to unarchive "bzipped" kernel source.
```

Note that in this context the "-" is not itself a Bash operator, but rather an option recognized by certain UNIX utilities that write to `stdout`, such as **tar**, **cat**, etc.

```
bash$ echo "whatever" | cat -
whatever
```

Where a filename is expected, `-` redirects output to `stdout` (sometimes seen with **`tar cf`**), or accepts input from `stdin`, rather than from a file. This is a method of using a file-oriented utility as a filter in a pipe.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

By itself on the command-line, file fails with an error message.

Add a "-" for a more useful result. This causes the shell to await user input.

```
bash$ file -
abc
standard input:              ASCII text



bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Now the command accepts input from `stdin` and analyzes it.

The "-" can be used to pipe `stdout` to other commands. This permits such stunts as prepending lines to a file.

Using diff to compare a file with a *section* of another:

```
grep Linux file1 | diff file2 -
```

Finally, a real-world example using `-` with tar.

Example 3-4. Backup of all files changed in last day

```
#!/bin/bash

#  Backs up all files in current directory modified within last 24 hours
#+ in a "tarball" (tarred and gzipped file).
```

```
BACKUPFILE=backup-$(date +%m-%d-%Y)
#                Embeds date in backup filename.
#                Thanks, Joshua Tschida, for the idea.
archive=${1:-$BACKUPFILE}
#  If no backup-archive filename specified on command-line,
#+ it will default to "backup-MM-DD-YYYY.tar.gz."

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."


#  Stephane Chazelas points out that the above code will fail
#+ if there are too many files found
#+ or if any filenames contain blank characters.

# He suggests the following alternatives:
# ----------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      using the GNU version of "find".


#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#         portable to other UNIX flavors, but much slower.
# ----------------------------------------------------------------


exit 0
```

Filenames beginning with "-" may cause problems when coupled with the "-" redirection operator. A script should check for this and add an appropriate prefix to such filenames, for example `./-FILENAME`, `$PWD/-FILENAME`, or `$PATHNAME/-FILENAME`.

If the value of a variable begins with a `-`, this may likewise create problems.

```
var="-n"
echo $var
# Has the effect of "echo -n", and outputs nothing.
```

-

previous working directory. A **cd -** command changes to the previous working directory. This uses the $OLDPWD environmental variable.

Do not confuse the "-" used in this sense with the "-" redirection operator just discussed. The interpretation of the "-" depends on the context in which it appears.

-

Minus. Minus sign in an arithmetic operation.

=

Equals. Assignment operator

```
a=28
echo $a   # 28
```

In a different context, the "=" is a string comparison operator.

+

Plus. Addition arithmetic operator.

In a different context, the + is a Regular Expression operator.

+

Option. Option flag for a command or filter.

Certain commands and builtins use the + to enable certain options and the - to disable them. In parameter substitution, the + prefixes an alternate value that a variable expands to.

%

modulo. Modulo (remainder of a division) arithmetic operation.

```
let "z = 5 % 3"
echo $z  # 2
```

In a different context, the % is a pattern matching operator.

~

home directory [tilde]. This corresponds to the $HOME internal variable. ~bozo is bozo's home directory, and **ls ~bozo** lists the contents of it. ~/ is the current user's home directory, and **ls ~/** lists the contents of it.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo
```

```
bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

current working directory. This corresponds to the $PWD internal variable.

~-

previous working directory. This corresponds to the $OLDPWD internal variable.

=~

regular expression match. This operator was introduced with version 3 of Bash.

^

beginning-of-line. In a regular expression, a "^" addresses the beginning of a line of text.

^, ^^

Uppercase conversion in *parameter substitution* (added in version 4 of Bash).

Control Characters

change the behavior of the terminal or text display. A control character is a **CONTROL** + **key** combination (pressed simultaneously). A control character may also be written in *octal* or *hexadecimal* notation, following an *escape*.

Control characters are not normally useful inside a script.

- **Ctl-A**

  Moves cursor to beginning of line of text (on the command-line).

- **Ctl-B**

Backspace (nondestructive).

- **Ctl-C**

  Break. Terminate a foreground job.

- **Ctl-D**

  Log out from a shell (similar to exit).

  EOF (end-of-file). This also terminates input from `stdin`.

  When typing text on the console or in an *xterm* window, **Ctl-D** erases the character under the cursor. When there are no characters present, **Ctl-D** logs out of the session, as expected. In an *xterm* window, this has the effect of closing the window.

- **Ctl-E**

  Moves cursor to end of line of text (on the command-line).

- **Ctl-F**

  Moves cursor forward one character position (on the command-line).

- **Ctl-G**

  BEL. On some old-time teletype terminals, this would actually ring a bell. In an *xterm* it might beep.

- **Ctl-H**

  Rubout (destructive backspace). Erases characters the cursor backs over while backspacing.

```
#!/bin/bash
# Embedding Ctl-H in a string.

a="^H^H"                  # Two Ctl-H's -- backspaces
                          # ctl-V ctl-H, using vi/vim
echo "abcdef"             # abcdef
echo
echo -n "abcdef$a "       # abcd f
#  Space at end  ^              ^  Backspaces twice.
echo
echo -n "abcdef$a"        # abcdef
#  No space at end              ^ Doesn't backspace (why?).
                          # Results may not be quite as expected.
echo; echo
```

```
# Constantin Hagemeier suggests trying:
# a=$'\010\010'
# a=$'\b\b'
# a=$'\x08\x08'
# But, this does not change the results.

########################################

# Now, try this.

rubout="^H^H^H^H^H"       # 5 x Ctl-H.

echo -n "12345678"
sleep 2
echo -n "$rubout"
sleep 2
```

- **Ctl-I**

  Horizontal tab.

- **Ctl-J**

  Newline (line feed). In a script, may also be expressed in octal notation -- '\012' or in hexadecimal -- '\x0a'.

- **Ctl-K**

  Vertical tab.

  When typing text on the console or in an *xterm* window, **Ctl-K** erases from the character under the cursor to end of line. Within a script, **Ctl-K** may behave differently, as in Lee Lee Maschmeyer's example, below.

- **Ctl-L**

  Formfeed (clear the terminal screen). In a terminal, this has the same effect as the clear command. When sent to a printer, a **Ctl-L** causes an advance to end of the paper sheet.

- **Ctl-M**

  Carriage return.

  ```
  #!/bin/bash
  # Thank you, Lee Maschmeyer, for this example.
  ```

```
read -n 1 -s -p \
$'Control-M leaves cursor at beginning of this line. Press Enter. \x0d'
          # Of course, '0d' is the hex equivalent of Control-M.
echo >&2   #  The '-s' makes anything typed silent,
          #+ so it is necessary to go to new line explicitly.

read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'
          #  '0a' is the hex equivalent of Control-J, linefeed.
echo >&2

###

read -n 1 -s -p $'And Control-K\x0bgoes straight down.'
echo >&2   #  Control-K is vertical tab.

# A better example of the effect of a vertical tab is:

var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'
echo "$var"
#  This works the same way as the above example. However:
echo "$var" | col
#  This causes the right end of the line to be higher than the left end.
#  It also explains why we started and ended with a line feed --
#+ to avoid a garbled screen.

# As Lee Maschmeyer explains:
# -------------------------
#  In the [first vertical tab example] . . . the vertical tab
#+ makes the printing go straight down without a carriage return.
#  This is true only on devices, such as the Linux console,
#+ that can't go "backward."
#  The real purpose of VT is to go straight UP, not down.
#  It can be used to print superscripts on a printer.
#  The col utility can be used to emulate the proper behavior of VT.

exit 0
```

- **Ctl-N**

  Erases a line of text recalled from *history buffer* [8] (on the command-line).

- **Ctl-O**

  Issues a *newline* (on the command-line).

- **Ctl-P**

Recalls last command from *history buffer* (on the command-line).

- **Ctl-Q**

  Resume (**XON**).

  This resumes `stdin` in a terminal.

- **Ctl-R**

  Backwards search for text in *history buffer* (on the command-line).

- **Ctl-S**

  Suspend (**XOFF**).

  This freezes `stdin` in a terminal. (Use Ctl-Q to restore input.)

- **Ctl-T**

  Reverses the position of the character the cursor is on with the previous character (on the command-line).

- **Ctl-U**

  Erase a line of input, from the cursor backward to beginning of line. In some settings, **Ctl-U** erases the entire line of input, *regardless of cursor position*.

- **Ctl-V**

  When inputting text, **Ctl-V** permits inserting control characters. For example, the following two are equivalent:

```
echo -e '\x0a'
echo <Ctl-V><Ctl-J>
```

  Ctl-V is primarily useful from within a text editor.

- **Ctl-W**

  When typing text on the console or in an xterm window, **Ctl-W** erases from the character under the cursor backwards to the first instance of whitespace. In some settings, **Ctl-W** erases backwards to first non-alphanumeric character.

- **Ctl-X**

  In certain word processing programs, *Cuts* highlighted text and copies to *clipboard*.

- **Ctl-Y**

  Pastes back text previously erased (with **Ctl-U** or **Ctl-W**).

- **Ctl-Z**

  Pauses a foreground job.

  Substitute operation in certain word processing applications.

  EOF (end-of-file) character in the MSDOS filesystem.

Whitespace

functions as a separator between commands and/or variables. Whitespace consists of either *spaces*, *tabs*, *blank lines*, or any combination thereof. [9] In some contexts, such as variable assignment, whitespace is not permitted, and results in a syntax error.

Blank lines have no effect on the action of a script, and are therefore useful for visually separating functional sections.

$IFS, the special variable separating *fields* of input to certain commands. It defaults to whitespace.

> **Definition:** A *field* is a discrete chunk of data expressed as a string of consecutive characters. Separating each field from adjacent fields is either *whitespace* or some other designated character (often determined by the $IFS). In some contexts, a field may be called a *record*.

To preserve *whitespace* within a string or in a variable, use quoting.

UNIX filters can target and operate on *whitespace* using the POSIX character class [:space:].

Notes

[1] An *operator* is an agent that carries out an *operation*. Some examples are the common arithmetic operators, **+ - * /**. In Bash, there is some overlap between the concepts of *operator* and keyword.

[2] This is more commonly known as the *ternary* operator. Unfortunately, *ternary* is an ugly word. It doesn't roll off the tongue, and it doesn't elucidate. It obfuscates. *Trinary* is by far the more elegant usage.

[3] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. This is a system for encoding text characters (alphabetic, numeric, and a limited set of symbols) as 7-bit numbers that can be stored and manipulated by computers. Many of the ASCII characters are represented on a standard keyboard.

[4] A *PID*, or *process ID*, is a number assigned to a running process. The *PID*s of running processes may be viewed with a ps command.

Definition: A *process* is a currently executing command (or program), sometimes referred to as a *job*.

[5] The shell does the *brace expansion*. The command itself acts upon the *result* of the expansion.

[6] Exception: a code block in braces as part of a pipe *may* run as a subshell.

```
ls | { read firstline; read secondline; }
#  Error. The code block in braces runs as a subshell,
#+ so the output of "ls" cannot be passed to variables within the block.
echo "First line is $firstline; second line is $secondline"  # Won't work.

# Thanks, S.C.
```

[7] Even as in olden times a *philtre* denoted a potion alleged to have magical transformative powers, so does a UNIX *filter* transform its target in (roughly) analogous fashion. (The coder who comes up with a "love philtre" that runs on a Linux machine will likely win accolades and honors.)

[8] Bash stores a list of commands previously issued from the command-line in a *buffer*, or memory space, for recall with the builtin *history* commands.

[9] A linefeed (*newline*) is also a whitespace character. This explains why a *blank line*, consisting only of a linefeed, is considered whitespace.
