

CHAPTER 3

Generic Concepts

Wildcards, Quoting and Redirection Operators

3.1 WILDCARDS

UNIX has special meaning for some characters such as `*`, `?`, `.`, `/`, `[`, `]`. Words in the commands that contain these characters are treated as patterns (model) for filenames. The word is expanded into a list of file names, according to the type of pattern. If we want shell not to expand these characters then we have to pre-pend `\` before them. This way we can make these characters to get escape from shell's normal interpretation is known as escaping and thus these characters are called as escape characters.

The following expansions are made by most shells, including bash.

- `*` matches any string (including null)
- `?` Matches any single character.
- `/` root directory
- `.` any character

For example, consider the command `expr` with which we can do command line calculations. It works nicely with `+`, `-`, `/`. However, if we use `*` then we will get an error such as syntax error. In some shells, `*` is considered as all the filenames of the C.W.D. Then, command line calculators such as `expr`, `bc`, etc., may give undigestable results. If we don't want such a thing to take place, we have to pre-pend this character `*` with back slash (`\`) character. This is called as escaping. For example, see the following workout on our terminal.

```
$ expr 2 + 3
5
$ expr 2 * 3
expr: syntax error
$ expr 10 / 2
5
$ X=10
$ Y=5
```

```
$ echo `expr $X + $Y`
15
$ echo `expr $X - $Y`
5
$ echo `expr $X * $Y`
expr: syntax error
$ echo `expr $X \* $Y`
50
$ echo `expr $X / $Y`
2
```

Let us assume that we wanted to calculate $X(X+Y)$. The following command gave error. When we pre-pend paranthesis with back slash, we have got the results.

```
$ echo `expr $X \* ( $X + $Y )`
expr :syntax error near unexpected token `(`
$ echo `expr $X \* \( $X + $Y \)`
150
```

From the above experiments, we can see that if we use back slash before `*`, it is taking as literal `*` and calculations are done. Thus, by placing back slash before `*` we are escaping `*` from shells (OS) normal meaning or interpretation. It is also true with parenthesis as experimented above. Like that other characters are also having some special meaning. For them also we can apply back slash if we want them to be taken literally.

Example :

Assuming that the current directory contains the files

```
tmp
tmp1
tmp2
tmp10
tmpx
```

The pattern `*1*` matches the files tmp1 and tmp10.

The pattern `t???` matches tmp1 and tmp2

The pattern `tmp[0-9]` matches with tmp1 and tmp2

The pattern `tmp[!0-9]` matches with tmpx only

The pattern `tmp[a-z]` matches with tmpx only

The pattern `tmp*` matches with all files.

This models can be used with any command.

```
$ls -l tmp[0-9]
$rm tmp*
$cp *.c /tmp
$cp a*.c /tmp
```

Displays details of files tmp1 and tmp2 only

Deletes all files whose names starts with tmp.

All C language files of C.W.D are copied to /tmp.

All C language files of C.W.D whose primary name starts with 'a' are copied to /tmp.

```
$mv a?.c /tmp
```

```
$rm a[0-9].c
```

All C language files of C.W.D whose primary name is of two characters length and starts with 'a' are moved to /tmp.

This removes C files of C.W.D whose primary name is of two characters length and first character is 'a' and second character is any digit. Here, [0-9] signifies any digit.

3.2 THE ECHO COMMAND

We can use echo command to display some thing on the screen. Its syntax is given as :

echo OPTIONS strings or variables

For example, the following command will display "Hello How are you".

echo Hello How are You

It has the following options :

- n** do not output the trailing newline
- e** enable interpretation of backslash escapes
- E** disable interpretation of backslash escapes (default)
- help** display this help and exit
- version** output version information and exit

If -e is in effect, the following sequences are recognized :

- \ONNN** the character whose ASCII code is NNN (octal)
- ** backslash
- \a** alert (BEL)
- \b** backspace
- \c** suppress trailing newline
- \f** form feed
- \n** new line
- \r** carriage return
- \t** horizontal tab
- \v** vertical tab

However, remember that your shell may have its own version of echo, which usually supersedes the version described here.

3.3 QUOTING

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the shell metacharacters has special meaning to the shell and must be quoted if it is to represent itself. When the command history expansion facilities are being used the *history expansion* character, usually '!', must be quoted to prevent history expansion. There are three quoting mechanisms: the *escape character*, single quotes, and double quotes. We have already explained about escaping. Now, let us discuss about others.

Single Quotes

Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Double Quotes

Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of '\$', '\', and, when history expansion is enabled, '!'. The characters '\$' and '\' retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: '\$', '\', '"', '\', or newline. Within double quotes, backslashes that are followed by one of these characters are removed. Backslashes preceding characters without a special meaning are left unmodified. A double quote may be quoted within double quotes by preceding it with a backslash. If enabled, history expansion will be performed unless an '!' appearing in double quotes is escaped using a backslash. The backslash preceding the '!' is not removed. The special parameters '*' and '@' have special meaning when in double quotes.

For example see the following workout at the command prompt.

```
$X = Rao
$echo '$X'
$X
$echo $X
Rao
```

We may find that when enclosed between double quotes, X value gets expanded.

We do have another important thing. That is we can quote between back quotes also. When we enclose between back quotes, the argument gets expanded and also gets executed.

For example, snap shot of our experiment with these operators is given in Figure 3.1.

```

RITCH$
RITCH$
RITCH$
RITCH$
RITCH$
RITCH$X=ls
RITCH$'$X'
bash: $X: command not found
RITCH$"$X"
aa  aaaa  employee  m2  m4  mm1  mmmm  nbv  pp  rrr  x2
aaa  abc  hist      m3  m4  mm2  mmo  pal  ppp  x1  xx
RITCH$'$X'
bash: aa: command not found
RITCH$echo '$X'
$X
RITCH$echo "$X"
ls
RITCH$echo '$X'
aa aaa aaaa abc employee hist m2 m3 m4 mm mm1 mm2 mmmm mmo nbv pal pp ppp rrr x1
x2 xx
RITCH$echo '$Xaa'
$Xaa
RITCH$echo "$Xaa"
RITCH$echo '$Xaa'
RITCH$'$Xaa'
>
bash: command substitution: line 1: unexpected EOF while looking for matching ''
bash: command substitution: line 2: syntax error: unexpected end of file
RITCH$'$Xaa'
bash: $Xaa: command not found
RITCH$"$Xaa"
bash: : command not found
RITCH$ $Xaa
RITCH$

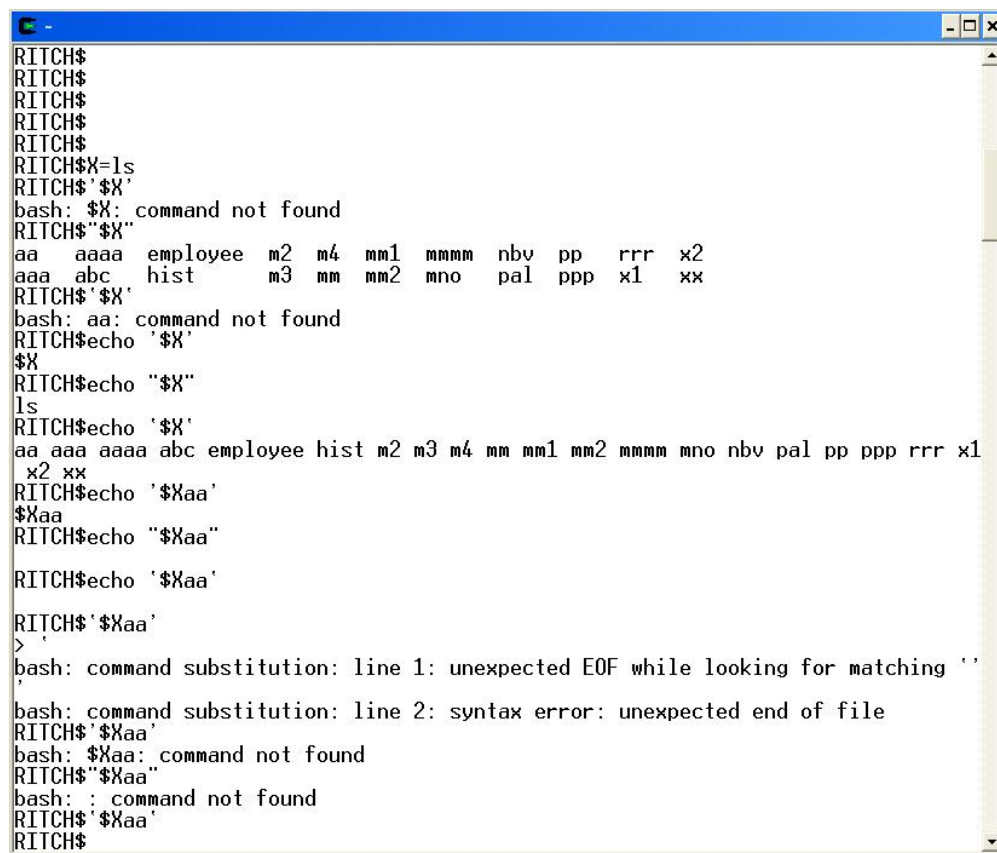
```

Figure 3.1. Differences between and single and double quotes

We have defined a variable `X` and assigned value `ls` to it. When we enclose between single quotes at the dollar prompt, shell considered `$X` as the name of a command and it tried to execute. As we don't have any such command, we have got an error. Similarly, when we have enclosed between double quotes, `$X` got expanded. Thus, it became `ls` and got it executed by the shell. When we have enclosed between back quotes, `$X` got expanded as `ls` and `ls` got executed and its results (files of current directory such as `aa`) is taken by the shell as if we have entered at the dollar prompt. As `aa` is not a command. Thus, we are getting an error.

Similarly, `$X` in between all the three type of quotes are use with `echo` command. When enclosed between single quotes we have got the results as `$X` indicating not variable value substitution. In the case of double quotes, variable value got substituted. Thus, we have got result as `"ls"`. Where as in the case of back quotes, substitution took place and the result (`ls`) got executed also.

See other experiment in Figure 3.2 and see whether you can derive anything or not.



```

RITCH$
RITCH$
RITCH$
RITCH$
RITCH$
RITCH$X=ls
RITCH$'$X'
bash: $X: command not found
RITCH$"$X"
aa  aaaa  employee  m2  m4  mm1  mmmm  nbv  pp  rrr  x2
aaa  abc   hist     m3  mm  mm2  mno   pal  ppp  x1  xx
RITCH$'$X'
bash: aa: command not found
RITCH$echo '$X'
$X
RITCH$echo "$X"
ls
RITCH$echo '$X'
aa aaa aaaa abc employee hist m2 m3 m4 mm mm1 mm2 mmmm mno nbv pal pp ppp rrr x1
x2 xx
RITCH$echo '$Xaa'
$Xaa
RITCH$echo "$Xaa"
RITCH$echo '$Xaa'
RITCH$'$Xaa'
>
bash: command substitution: line 1: unexpected EOF while looking for matching `''
'
bash: command substitution: line 2: syntax error: unexpected end of file
RITCH$'$Xaa'
bash: $Xaa: command not found
RITCH$"$Xaa"
bash: : command not found
RITCH$`$Xaa`
RITCH$

```

Figure 3.2. Quoting examples

Single quotes (`' '`) operate similarly to double quotes, but do not permit referencing variables, since the special meaning of `$` is turned off. Within single quotes, *every* special character except ``` gets interpreted literally. Consider single quotes ("full quoting") to be a stricter method of quoting than double quotes ("partial quoting"). Since even the escape

character (\) gets a literal interpretation within single quotes, trying to enclose a single quote within single quotes will not yield the expected result.

3.4 REDIRECTION OPERATORS

For any program whether it is developed using C, C++ or Java, by default three streams are available known as input stream, output stream and error stream (see Figure 3.3). In programming languages, to refer to these streams some symbolic names are used (system defined variables).

For example

- In C, stdin, stdout and stderr.
- In C++, cin, cout, and cerr.
- In Java, System.in, System.out and System.err.

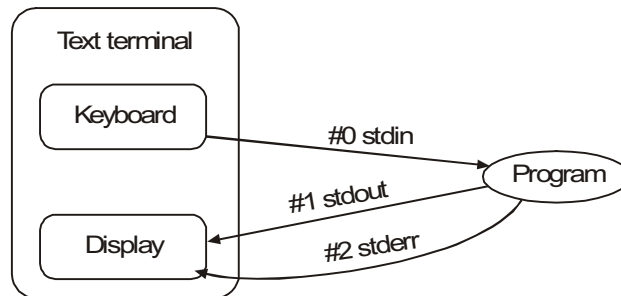


Figure 3.3. Redirection operators

By default is input is from keyboard and output and error are sent to monitor. With the help of redirection operators, we can send them to a file or to a device.

UNIX, supports the following redirection operators such as [also see Table 1]

- standard output operator
- < standard input operator
- << here the document
- >> appending operator

Operator	Function
<	Redirect standard input from file
<<	Redirect standard input from command source
>	Redirect standard output to file
>!	Redirect standard output and overwrite file
>>	Redirect standard output and append to file
>>!	Redirect standard output to file or append to file
>&	Redirect standard output/error to file
>>&	Redirect standard output/error and append to file
>>&!	Redirect standard output/error to file or append to file
	Pipe standard output to standard input
&	Pipe standard output/error to standard input

Standard Input, Output Redirection operators

UNIX supports input, output redirection. We can send output of any command to a file by using > operator.

Example :

command >aaa

Output of the given command is sent to the file. First, file aaa is created if not existing otherwise its content is erased and then output of the command is written.

cat aa >aaaa

Here, aaaa file contains the content of the file aa.

We can let a command to take necessary input from a file with < operator (standard input operator).

cat<aa

This displays output of file aa on the screen.

cat aa aa1 aa3>aa12

This creates the file aa12 which contains the content of all the files aa, aa1 and aa3 in that order.

cat <aa >as

This makes cat command to take input from the file aa and write its output to the file as. That is, it works like a cp command.

UNIX has a nice (intelligent) command line interface. Thus, all the following commands work in the same manner.

cat <aa >as

cat >as <aa

<aa cat >as

<aa >as cat

>as cat <aa

>as <aa cat

This discussion is meaningful with any command including the ones created by us from C or C++ programs. For example, consider the following C program which takes three integers and writes their values.

```
#include<stdio.h>
int main()
{
    int x,y,z;
    scanf("%d%d%d", &x, &y, &z);
    printf("%d\n%d\n%d\n", x, y, z);
    return 0;
}
```

Let the file name be a.c and by using the either of the following commands, its machine language file a is created.

```
gcc -o a a.c
```

```
cc -o a a.c
```

When we start this program a by simply typing a at the dollar prompt, it takes 3 values and displays the given values on the screen.

```
a>res
```

On some machines, we will not be in a position to run the above program 'a', by simply typing its name at the command prompt. In that situation, we can run the same by type :

```
./a
```

Otherwise, we can change the PATH environment variable such that it contains . (C.W.D) in it. This can be achieved by executing the following command.

```
PATH=.: $PATH
```

We will know more about this in the next chapters.

This program takes three values interactively and writes the same into file res. You can check by typing **cat res**.

```
a<res
```

This command takes necessary input from the file **res** and displays the results on the screen.

```
a <res >as
```

```
a >as <res
```

```
<res a >as
```

```
<res >as a
```

```
>as a <res
```

```
>as <res a
```

All, of these commands takes three values from the file res and write the same in the file as.

A note on file descriptors

As mentioned above, for any program by default three streams are available known as: standard input, standard output, and standard error. In UNIX operating system point of view, they are referred as 0,1 and 2; which are called as file descriptors. We can as well use them with redirection operators.

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is '<', the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is '>', the redirection refers to the standard output (file descriptor 1).

The word following the redirection operator in the following descriptions, unless otherwise noted, is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, filename expansion, and word splitting. If it expands to more than one word, Bash reports an error.

Note that the order of redirections is significant. For example, the command

`ls > dirlist 2>&1`

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file *dirlist*, while the command

`ls 2>&1 > dirlist`

directs only the standard output to file *dirlist*, because the standard error was made a copy of the standard output before the standard output was redirected to *dirlist*

Now consider a simple program, *b.c*, which gives both standard output and standard error.

```
#include<stdio.h>
int main()
{
    fprintf(stdout, "This is standard output\n");
    fprintf(stderr, "This is standard error\n");
    return 0;
}
```

Let us compile and run the same.

`gcc -o bb b.c`

`./bb`

We get the result as :

This is standard output

This is standard error

Now if we execute the following command, only standard output is redirected to the file *x1*.

`$/bb 1>x1`

This is standard error

`$cat x1`

This is standard output

`./bb >x2 2>&1`

`$cat x2`

This is standard output

This is standard error

```
$/bb 2>&1 1>x2
```

```
cat x2
```

```
This is standard output
```

```
This is standard error
```

```
$/bb >& x2
```

```
$cat x2
```

```
This is standard output
```

```
This is standard error
```

```
$/bb 2>x2 1>&2
```

```
$cat x2
```

```
This is standard output
```

```
This is standard error
```

```
$/bb 2>&1 >x3
```

```
$cat x3
```

```
This is standard output
```

C Shell and its descendants :

It is slightly more difficult to direct Standard Error separately from standard output. The command `ls >& mydir` will direct *both* standard output and standard error to `mydir`.

To create different files for the standard output and the error messages, you must use a sub-shell command. What you need to do is direct standard output to one file, then redirect both standard out and standard error to a second file. This will result in standard output messages sent to one file and the standard error messages to the second file. The code to perform this looks like :

(myprog > outfile) >& errorlog

The way this works is that the code inside the parentheses will be executed first, (the term for this is a sub-shell.) Within the sub-shell, the standard output data stream is directed to the file `outfile`. When the sub-shell finishes its work, the second redirection sends the data from both standard out and standard error to the file `errorlog`. However, all that is left in that output stream are the error messages, because the data intended for standard out has already been redirected inside the sub-shell.

This is a somewhat cumbersome, albeit elegant solution to meet the need to divide the two output streams.

The >> and << Operators

Similarly, `>>` operator can be used to append standard output of a command to a file.

Example

```
command>>aaa
```

This makes, output of the given command to be appended to the file `aaa`. If the file `aaa` is not existing, it will be created afresh and then standard output is written.

Here the document operator(<<)

This is used with shell programs. This signifies that the data is here rather in a separate file. This type of redirection instructs the shell to read input from the current source until a line containing only *word* (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command.

The format of here-documents is :

```
<<["]word
      here-document
delimiter
```

No parameter expansion, command substitution, arithmetic expansion, or filename expansion is performed on *word*. If any characters in *word* are quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. If *word* is unquoted, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion. In the latter case, the character sequence `\newline` is ignored, and `\` must be used to quote the characters `\`, `$`, and ```.

If the redirection operator is `<<-`, then all leading tab characters are stripped from input lines and the line containing *delimiter*. This allows here-documents within shell scripts to be indented in a natural fashion.

Example :

```
$cat<<END
This will display
Whatever we type
Interactively on the screen again
END
$
```

The above workout displays whatever we have typed till the string "END". Make sure that we enter the string "END" on a fresh line.

```
$cat<<END >outputfile
This will display
Whatever we type
Interactively on the screen again
END
$
```

The above command writes whatever we have typed till "END" string into the file "outputfile".

```
$ grep Rao<<end
I like PP Reddy
I know Mr. PN Rao since 1987
I wanted to see Raj today
Mr. Rao, please see me today
end
$
I know Mr. PN Rao since 1987
Mr. Rao, please see me today
```

The above sequence of commands when executed at the dollar prompt, we will get those lines having rao as output of grep command. Here, by using << operator we are mentioning that the data is directly available here. The command takes input till we enter 'end'.

Another interesting thing about "here documents" is how they handle shell variables. If you use the form shown above, the shell will expand any shell variables located within the text for the **here document**, but if you add the protected slash to the ending mark, the shell will **NOT** expand shell variables. The following two code snippets differ only in the way the ending mark is coded, however the output is significantly different :

```
#!/bin/sh
var1="Hello"
var2="How are you"
cat << end
$var1 $var2 Rao
end
```

We will learn more about shell scripts later. However, assume that the above statements are in a file pqr. The following series of steps are executed. Thus, the output of the script above looks like :

Hello How are you Rao

```
$chmod u+x pqr
$./pqr
Hello How are you Rao
```

Now, consider the following script.

```
#!/bin/sh
var1="Hello "
var2="How are you"
cat << \end
$var1 $var2 Rao
end
```

Notice the slash before the ending mark. Because of that slash, the shell will not expand any shell variables in the **here document**, as we see from the output shown below :

\$var1 \$var2 Rao

Even though **here documents** are usually only used in scripts, they are very powerful tools.

Bash handles several filenames (given below) specially when they are used in redirections :

/dev/fd/fd	If <i>fd</i> is a valid integer, file descriptor <i>fd</i> is duplicated.
/dev/stdin	File descriptor 0 is duplicated.
/dev/stdout	File descriptor 1 is duplicated.
/dev/stderr	File descriptor 2 is duplicated.

/dev/tcp/host/port

If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open a TCP connection to the corresponding socket.

/dev/udp/host/port

If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open a UDP connection to the corresponding socket.

A failure to open or create a file causes the redirection to fail.

Redirections using file descriptors greater than 9 should be used with care, as they may conflict with file descriptors the shell uses internally.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified.

The general format for redirecting input is :

[n]<word

Redirecting Output

Redirection of output causes the file whose name results from the expansion of *word* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

The general format for redirecting output is :

[n]>[[]word

If the redirection operator is '>', and the noclobber option to the set built-in has been enabled, the redirection will fail if the file whose name results from the expansion of *word* exists and is a regular file. If the redirection operator is '>|', or the redirection operator is '>' and the noclobber option is not enabled, the redirection is attempted even if the file named by *word* exists.

Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of *word* to be opened for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

The general format for appending output is :

`[n]>>word`

Redirecting Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word*.

There are two formats for redirecting standard output and standard error :

`&>word`

and

`>&word`

Of the two forms, the first is preferred. This is semantically equivalent to

`>word 2>&1`

Appending Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be appended to the file whose name is the expansion of *word*.

The format for appending standard output and standard error is :

`&>>word`

This is semantically equivalent to

`>>word 2>&1`

3.5 CONCLUSIONS

First, we have introduced to wildcards, quoting, echo command to understand the generic behavior of Shell. Redirection operators are explained in detail with live examples. The examples given can be experimented with out any confusion.