General

In the following group, the filename may be replaced with the value of a variable, if desired, as long as the rules in the VARIABLES section are followed.

The redirection mechanisms can be applied to complex shell commands as a whole if the shell commands are placed within parenthesis, ie.

```
date > file
ls -l >> file
who >> file
```

is the same as

```
( date; ls -l; who ) > file
```

which runs the commands in a subshell (causes *fork()*), or

```
{ date; ls -l; who; } > file
```

which runs the commands within the current shell (no *fork()*)

< *file*

Read input from the given file instead of from the keyboard, ie. "redirect stdin from *file*".

> *file*

Write output to the given file instead of to the screen, ie. "redirect stdout to *file*".

2> *file*

Write errors to the given file instead of to the screen, ie. "redirect stderr to *file*".

2>&1

Send stderr to the same place that stdout is going. This copies the file descriptor from stdout to stderr instead of open'ing stderr again, so the two data streams share the same buffer instead of each having it's own buffer.

Double Quotes (stands for DO dollar signs)

Allow **variable substitution**, but prevent any other interpretation, such as I/O redirection, wildcards, etc.

Single Quotes (stands for SUPPRESS dollar signs)

Prevent **any** interpretation of the contents.

\ The backslash does the same thing as single quotes, but only for the immediately following character.

General

Variables are used to contain values that may change during the execution of the shell. Typically, these are filenames, data read from the keyboard or a data file, calculated values, or similar.

Variables may contain spaces, tabs, or other characters considered to be delimiters by the shell, so any time you want to use the value of a variable you should place double quotes around it.

Modifiers

The value of variables can be modified before being used if the variable name is enclosed within braces ("**{**" and "**}**") and the appropriate modifier characters are used:

${var-string}

Use the value of *var* unless it's empty; then use *string*.

${var=string}

Use the value of *var* unless it's empty; then assign *string* to *var* and use it.

${var?string}
: Use the value of *var* unless it's empty; then print *string* as an error message end terminate the shell.

${var+string}
: Use the value of *var* unless it's **NOT** empty; then use *string*.

${var#wildpat}
: Use the value of *var* after removing any characters from the front of the value which match the wildcard pattern *wildpat*. The characters matched by the pattern are the shortest possible string.

${var##wildpat}
: Same as above, but the match is the longest possible string.

${var%wildpat}
: Same as **#**, but this one trims from the end of the string instead of the beginning. It matches the shortest possible string.

${var%%wildpat}
: Same as above, but the match is the longest possible string.

$$ The process id of the shell. This is usually used for temporary filenames, such as **/tmp/$0.$$**.

$! The process id of the last job to be executed in the background.

$0 Name of the shell script that is running. It may include a pathname, so it would be more appropriate to use **basename**, or a similar construct, to remove the directory component before using it in an output statement.

```
#!/bin/ksh
PROG=$(basename $0)
USAGE="Usage:  $PROG fromfile tofile ..."
```

or

```
#!/bin/ksh
PROG=${0##*/}
USAGE="Usage:  $PROG fromfile tofile ..."
```

$1-$*n*
: These represent the command line parameters provided by the user of the script when it was executed.

$# The number of command line parameters.

"$@"
: The values of the command line parameters with double quotes around individual parameters.

The last three variables, $*n*, $#, and $@, are specific shortcuts of the general technique of accessing elements of an array:

${array[n]}
: This accesses the *n*th element of *array*.

${array[@]}
: This accesses all of the elements of *array*.

${#array[@]}
: This provides the number of elements in *array*.

Always put double quotes around variable values.

Such as `if [[ -f "$dir/$1" -o -d "$dir/$1" ]]`

This rules prevents the shell from interpreting any spaces that might appear within the value of a variable (obviously, if a variable **cannot** contain spaces, tabs, or newlines, then you needn't use double quotes — an example would be **$$** or **$#**). It also prevents errors when a variable is empty and the value is used in a location where a value is **required**. This is common inside [[and]] and when changing the command line parameter list via **set --**.

Always use ((and)) for numeric tests.

Such as `if (( count < high ))`

Always use [[and]] for string and file tests.

Such as `[[ "$var1" = "$var2" -o -r "$file" ]]`

Never put spaces around the equal sign in variable assignments.

Such as `typeset -i count=10`, or `progname="$0"`

Always put spaces everywhere else!

Such as around the fields within [[and]] or ((and)). Always put spaces on both sides of command names and *keywords* such as **if**, **then**, **else**, **elif**, **fi**, **while**, **for**, **select**, **do**, **done**, **case**, **in**, **esac**, **let**, and any other built-in or regular (external) commands.