POSIX shell best practices

ivanb.neocities.org/blogs/y2024/posix

Maximum portability for a POSIX shell script

Script invocation

Shebang '**#!**' is a human-readable instance of magic byte string. It triggers the operating system's program loader mechanism to parse the rest of the line as an **interpreter directive** and launch the specified interpreter program. Some of the operating systems have a length limit of 30 charcters for this **interpreter directive**. Others require a mandatory **space** character after '**#!**' character sequence. Best practice is to always add **space** character after shebang.

'/bin/sh' is usually a symbolic link for 'dash':

```
#! /bin/sh
```

Or call 'dash' directly:

```
#! /usr/bin/dash
```

printf instead of echo

'echo' is implemented differently in each shell, never use 'echo', use '**printf**' instead:

```
printf '%s\n' 'Hello world.'
```

Redirections first

To avoid ambiguity, use redirections before commands. This is also the order in which a shell interpets command string internally:

```
>&2 printf '%s\n' 'Message containing an error, or information should be
directed to error stream /dev/stderr.'
>|listing.txt 2>/dev/null ls -lhAFR
>|listing.txt 2>&1 ls -lhAFR
>|listing.txt 2>|errors.txt ls -lhAFR
>>listing.txt printf 'Current user is: "%s".\n' "$LOGNAME"
```

Making a text file using 'cat' command (on the last empty line press 'Control-D'):

```
>|thefile.txt cat
>>thefile.txt cat
```

Counting the number of lines in a file with 'wc' command:

```
<listing.txt wc -l
```

Sorting the result of 'ls' command in dictionary order and storing it in file using 'sort' command:


```
ls | >|sorted.txt sort -d
```

Vertical bar **|** after redirection operator overrides **noclobber** shell option, read the manual:

```
man dash
```

Make use of file descriptors

Instead of opening and closing a file every time you need to append text to it, call '**exec**' to open a file on available **file descriptor**, **№9** for example, append text to it with '**>&9**', then close it when you're done with '**exec 9>&-**':

```
exec 9>>listing.txt
>&9 printf '%s\n' 'Some text'
>&9 printf '%s\n' 'More text'
exec 9>&-
```

Open file on **file descriptor №8**, read a line from it into a variable, do something with it, repeat, close **file descriptor №8**:

```
exec 8<listing.txt
<&8 read -r -- var
printf 'Do something with %s\n' "$var"
<&8 read -r -- var
printf 'Do something else with %s\n' "$var"
exec 8<&-
```

Interactive shell requires **file descriptors №0, №1, №2** to be open at all times, they must contain unique links to shell's **standard input**, **standard output** and **standard error**. They can be linked with other files only in a subshell **( ), $( )**. Non-interactive shell (script) allows them to be linked with other files, for example '**exec 2>&1**'.

Permanently send error messages to a file instead of **standard error** (non-interactive shell, or subshell):

```
exec 2>|errors.txt
```

File descriptor №0 is **standard input**, it can be used as '**0<**', or '**<**'.

File descriptor №1 is **standard output**, it can be used as '**1>**', or '**>**'.

File descriptor №2 is **standard error**, it can be used as '**2>**'. A special form of '**2>&1**' instructs the shell to send the error stream to **File descriptor №1** just once (use 'exec' for permanent effect):

```
>|listing.txt 2>&1 ls -lhAFR
```

The above code is equivalent to:

```
>|listing.txt 2>|listing.txt ls -lhAFR
```

Other variants are possible: '**1>&2**' - same as '**>&2**'; '**7>&8**'; 'exec **6<&0**' - permanently saves unique link pointing to shell's **standard input** in **file descriptor №6**; 'exec **0<file.txt**' - **file descriptor №0** is now pointing to file.txt, which is open for reading, 'read' shell command will now read from file.txt instead of **standard input**; 'exec **0<&6**' will restore **file descriptor №0** to it's original state.

File descriptors from №6 upto №9 are **always available**. №3 and №4 are **unavailable** on some old and some proprietary systems (they are used by shell itself).

File descriptor №5 is **used by bash shell**. Never use **file descriptor №5**, even if you are using other shell such as 'dash'. If 'dash' was invoked from 'bash' and you start to use **file descriptor №5**, there will be a lot of issues including a possible crash and data loss.

Command grouping

Commands between curly braces **{ }** are grouped and will have a single return value for the whole group. Return value of the command is stored in variable **$?**. **0** means success, any non-zero value - an error. Curly braces **{ }** must be separated from commands by spaces. Semicolon **;** is required after the last command if it is on the same line as closing curly brace **}**.

```
{ df;lsblk;ls; }
printf '%d\n' "$?"
```

Commands between parentheses **( )** are executed in a subshell with separate environment and will have a single return value for the whole group. Return value of the command is stored in variable **$?**. **0** means success, any non-zero value - an error. Environment and options of parent shell are unaffected by the subshell:

```
x=44
(set -f; x=117; printf '%d\n' "$x"; set -o)
printf '%d\n' "$x"
set -o
(ls nosuchfile)
printf '%d\n' "$?"
```

In the example above, in subshell 'noglob' is turned on and 'x' is set to 117. In parent shell both of those changes have no effect. However, subshell returns an error code after it's execution as any other command, which is stored in a variable '$?'. Quotes in subshell have no effect on quotes in parent shell:

```
theuser="$(printf '%s\n' "$LOGNAME")"
```

Using subshells **$( )** to get the **full path** of currently running script file:

```
scriptRoot="$(cd "$(dirname "$0")"; pwd)"
printf '%s\n' "$scriptRoot"
```

The 'if' command

Curly braces **{ }** are **not** a part of '**if**' command. Curly braces are command grouping construct. Square brackets **[ ]** are **not** a part of '**if**' command. Square brackets are a second form of '**test**' command. Parentheses **( )** are **not** a part of '**if**' command. Parentheses are a subshell construct.

'**if**' command checks the return status '**$?**' of a command, or group of commands, and if it is **0**, the 'true' branch is executed, otherwise the 'false' branch is executed. The command does **not** need to be placed on the same line as the '**if**' command:


```
if
  >/dev/null 2>&1 ping -c1 neocities.org
then
  printf '%s\n' 'Connection works.'
fi
```

Examples:

```
if >/dev/null ls; then printf '%s\n' 'Found it!'; else printf '%s\n'
'Nope!';fi
if 2>/dev/null ls nosuchfile; then printf '%s\n' 'Found it!'; else printf
'%s\n' 'Nope!';fi
```

In place of 'ls' in the example above can be any command, group of commands, or commands in a subshell:

```
if (var=3
  test $var -gt 2)
then
  printf '%s\n' 'TRUE'
else
  printf '%s\n' 'FALSE'
fi
```

Group of commands:

```
if { var=3
  test $var -lt 2
}
then
  printf '%s\n' 'TRUE'
else
  printf '%s\n' 'FALSE'
fi
```

'**test**' is the same as '**[ ]**':

```
if test 5 -lt 7; then printf '%s\n' 'TRUE';fi
if [ 5 -lt 7 ]; then printf '%s\n' 'TRUE';fi

test 5 -lt 7 && printf '%s\n' 'TRUE'
[ 5 -lt 7 ] && printf '%s\n' 'TRUE'
```

Functions

A **function** has the form **NAME() COMMAND**. As with '**if**', '**COMMAND**' can be any command, group of commands, or commands in a subshell. The curly braces **{ }**, parentheses **( )**, or square brackets **[ ]** from the '**COMMAND**' are **not** a part of the **function**.

'**COMMAND**' can be a single command:

```
myfunc() printf 'Current user is: "%s".\n' "$LOGNAME"
myfunc
```

If commands are executed in a group, the function will change the environment of a parrent shell:

```
var=9
myfunc() { set -f; var=11; printf '%d\n' "$var"; set -o; }
myfunc
printf '%d\n' "$var"
set -o
```
In the example above, in a group of commands, 'noglob' is turned on and 'var' is set to 11. In parent shell both of those changes persist. You will need to turn 'noglob' off at the end of the '**COMMAND**' definition or after it's execution, and also to restore the original value of 'var'. Remember that '**COMMAND**' is **not** a **function**'s body.

```
set +f; var=9
```
If commands are executed in a subshell, the function will not affect the environment and options of a parrent shell:

```
var=20
myfunc() (set -f; var=50; printf '%d\n' "$var"; set -o)
myfunc
printf '%d\n' "$var"
set -o
```
In the example above, in a subshell 'noglob' is turned on and 'var' is set to 50. In parent shell both of those changes have no effect. There is **no** need to turn 'noglob' off at the end of the '**COMMAND**' definition or to save and restore the original value of 'var' inside the body of '**COMMAND**'. Remember that '**COMMAND**' is **not** a **function**'s body.

The 'while' command, and others ('until', etc.)

Exactly as with the '**if**' command, the curly braces **{ }**, parentheses **( )**, or square brackets **[ ]** are **not** a part of the '**while**' command.

'**while**' with group of commands **{ }**:

```
x=1
while { x=$(($x + 1)); test $x -le 9; }; do printf '%d\n' "$x"; done
printf '%d\n' "$x"
```
'**while**' with group of commands **{ }** and commands in a subshell **( )**:

```
x=1
while { x=$(($x + 1)); test $x -le 9; }; do (printf '%s ' 'The value is:';
printf '%d\n' "$x") done
```
In the examples above, the parentheses **( )** act as command separators, however the curly braces are **not** command separators and the commands need to be separated by semicolons **;**. Also the curly braces **{ }** need to be separated by spaces from the commands.

'**read**' command in combination with '**while**' command can accept **input** redirecton from a file, or **here-document** and can redirect it's **output** to a file:

```
while IFS='' read -r -- var; do printf '%s\n' 'Do something with $var.';
done <listing.txt while IFS='' read -r -- var; do printf '%s\n' 'Do
```



```
something with $var.'; done <listing.txt >|out.txt
```
Example of '**while read**' inside a **function**:

```
myfunc() (while IFS='' read -r -- var; do printf '%s\n' 'Do something with
$var.'; done <"$1" >|"$2")
myfunc listing.txt out.txt
```
Example of '**while read**' with **here-document**:

```
while IFS='' read -r -- var; do printf '%s\n' "Do something with ${var}.";
done <<EOF >|out.txt
$LOGNAME
$SHELL
$LC_CTYPE
EOF
```

Here-documents

Here-document is a section of a source code file that is treated as if it were a separate file. There are 3 variants of **here-documents**: <<EOF, <<-EOF, <<\EOF. EOF can be **any** unique word. There should be **no** spaces between << and EOF. The end of **here-document** is specified by a separate line consisting of only 'EOF' word.

<<EOF form. Quotes **' "** and number signs **#** are **safe** to use. Backquote, dollar sign and backslash **` $ ** need to be escaped with backslash ****.

```
<<EOF >|out.txt cat
text
$LOGNAME
$(date)
`who`
EOF
```
<<-EOF form. Same as the **<<EOF**, except it will delete the leading **Tab** characters from the body of **here-document**. Spaces persist, only **Tab** characters can be safely used for **indentation** of text in this form.

```
<<-EOF >|out.txt cat
                text
    spaces as indentation are used in this line
        $LOGNAME
$(date)
                        `who`
EOF
```
<<\EOF form. Shell will **not** interpret **any** special characters.

```
<<\EOF >|out.txt cat
text
$LOGNAME
$(date)
`who`
\
EOF
```

<<-\EOF combined form. Only in this order -\, << and EOF must be typed **without** space in-between: **<<-\EOF**. Shell will **not** interpret **any** special characters, **and** all leading **Tab** characters will be cleared.

```
<<-\EOF >|out.txt cat
                text
    spaces as indentation are used in this line
        $LOGNAME
$(date)
                        `who`
EOF
```

Embedded **here-documents** (indented with **Tab** characters):

```
<<-EOF >|out.txt cat
        text
        $(<<-END cat
                something else
        END
        )
        another text
EOF
```

To perform some actions as other user on the system (**here-documents** are indented with **Tab** characters):

```
<<-EOF su otheruser
        other_user's_password
        <<-END cat

                Do something
                as other user.
        END
        exit
EOF
```

Shell settings

These shell **options** are desirable to use inside the shell script:

- **set -o errexit**, short version: **set -e**
- **set -o nounset**, short version: **set -u**
- **set -o noclobber**, short version: **set -C**
- **set +o noglob**, short version: **set +f**

errexit cause the shell to exit immediately if any untested command fails. There may be some issues with **errexit** on some proprietary systems, where **errexit** may cause buggy behavior: shell exits during '**for**' loop.

nounset cause the shell to exit immediately if it is attempting to expand a variable that is not set. On some old and/or proprietary systems, **nounset** may cause the shell to crash, sometimes only when **nounset** is used in combination with **errexit** (set -eu).

Both **errexit** and **nounset** should be avoided if maximum **portability** is paramount.

noclobber will prevent the shell from overwriting existing files with '**>**' redirection. Use '**>|**' to overwrite existing files.

It's always wise to explicitly enable pathname expansion with **set +o noglob**, or **set +f**. If needed, **noglob** can be disabled on per-function basis, when the command is running in a subshell **( )**.

Variables

: - **no-op** operator. It's exit status is always **0**, it does nothing, except expanding variables. Usage of **:** in shell script:

```
var1="$1" var2="$2" var3="$3"
: "${var1:=default 1} ${var2:=default 2} ${var3:=default 3}"
```

${var:=default value} - if '**var**' is **unset**, or contains an **empty string**, then the value of '**var**' will be set to '**default value**'.

IFS variable contains input field separators. Default value is space, tab, newline. To better handle file names containing spaces, **IFS** can be changed to ommit the space character:

```
IFS="$(printf -- '\n\t')"
printf "$IFS" | od -bc
```

With '**read**' command, the **IFS** variable should always be set to an empty string **''**, and **-r** option should be specified to disable backslash **** escapes:

```
IFS='' read -r -p 'Input from user: ' -- var
printf '%s\n' "$var"
```

$@ expands to the positional parameters, starting from **$1**. When the expansion occurs within double quotes, each positional parameter expands as a separate argument. On some proprietary systems, if there are no arguments, **$@** will expand to exactly 1 argument containing an empty string **''**. To avoid this buggy behavior, **${1+"$@"}** should be used instead: if **$1** is **set**, it's value will be replaced by all arguments, otherwise nothing will happen. With no arguments, **$1** is always **unset**.

```
printf '%s\n' "${1+"$@"}"
```

$USER variable is deprecated. Use **$LOGNAME** instead.

```
printf '%s\n' "$LOGNAME"
```

Arithmetics

A lot of POSIX compliant shell implementations do **not** support advanced arithmetics. Things like **: $((var += 1))** will most likely fail on most shells, especially on proprietary operating systems. The rules of **portable arithmetics** are:

- Always use dollar sign **$** when referencing variables inside the **$(( ))** construct.
- Never try to assign a variable inside the **$(( ))** construct.

```
portable=54
portable=$(( $portable + 34 ))
printf '%d\n' "$portable"
```

Bitwise operations are much faster than mathematic operations:

When looking for odd and even numbers, bitwise AND **&** is much faster than modulus operator **%**. Every odd number AND 1 will result in **1**, every even number AND 1 will result in **0**.

```
printf '%d\n' "$(( 33 & 1 ))"
printf '%d\n' "$(( 30 & 1 ))"
for x in 1 2 3 4; do test $(( $x & 1 )) -eq 1 && printf '%s\n' 'green' ||
printf '%s\n' 'red'; done
```

Bitwise XOR **^** can be used to check number equality. Any number XOR itself will result in **0**.

```
printf '%d\n' "$(( 30 ^ 30 ))"
```

```
x=0
while
{
  x=$(( $x + 1 ))
  test $(( $x ^ 5 )) -ne 0
}
do
  printf '%s is %d.\n' 'The value' "$x"
done
```

Bitwise left shift **<<** and right shift **>>** are much faster than multiplication, division and other mathematic operations.

```
printf '%d\n' "$(( 512 << 4 ))"
printf '%d\n' "$(( 8192 >> 5 ))"
```

Behavior of 'logical AND' and 'logical OR'

Caution required when logical AND '**&&**' and logical OR '**||**' are used:

```
true || false && printf '%s\n' 'UNEXPECTED'
```

Compare the result from an example above with the result of the same operation done in **Tcl** programming language:

```
#! /usr/bin/env tclsh
expr {true || false && [puts UNEXPECTED; string cat 1]}
```

The proper behavior is when 'UNEXPECTED' never be printed in this situation.

Miscellaneous

'**rlwrap**' program can be used to conveniently run POSIX compliant shell in interactive mode:

```
rlwrap dash
```

Generating alphanumeric passwords:


```
</dev/urandom tr -cd '[:alnum:]' | fold -w 62 | head -n 4
</dev/urandom tr -cd '[:alnum:]' | fold -w 62 | sed 4q
```

Searching for a substring in a string:

```
string='POSIX shell programming'
substring='hell programming'

case "$string" in
  *"$substring"*)
    printf 'Found the "%s" in "%s".\n' "$substring" "$string"
  ;;
  *)
    printf '%s\n' 'Substring is not found.'
  ;;
esac
```

CSI (Control Sequence Introducer) **sequences** can be used to change the text appearance, or clear screen:

```
tRed='\033[31m'
tNorm='\033[0m'
errorM()
{
  >&2 printf -- "\n\tERROR: ${tRed}%s${tNorm}\n\n" "$1"
  exit 1
}
>/dev/null 2>&1 ls '/mnt/nosuchfile' || errorM 'Drive is not mounted!'
```

Clearing the screen with **CSI** is much faster than using the '**clear**' program. It can be used to speed up shell script where you need to constantly refresh the screen. Example using **mksh** shell script with POSIX-incompatible syntax.

Clearing the visible portion of the terminal emulator's window with **CSI**:

```
>&2 printf -- '\033[1;1H\033[0J'
```

Clearing the entire terminal emulator's scrollback buffer with **CSI**:

```
>&2 printf -- '\033[3J\033[1;1H\033[0J'
```

Setting terminal emulator's window title with **non-CSI** sequence:

```
>&2 printf -- "\033]0;${PWD}\07"
```