Shell Scripting 3: Shell Expansions

1. Shells and Variables
2. Control Constructs
3. **Shell Expansions**
   - **parameter expansion**
   - **command substitution**
   - **arithmetic expansion**
   - **filename expansion**
   - **filename expansion quirks**
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Expansion Phases

Recall that a key aspect of Bash "interpreting" each command line prior to executing it is the various **shell expansions**:

1. **brace expansion**
2. **tilde expansion**
3. **parameter expansion**
4. **command substitution**
5. **arithmetic expansion**
6. **filename expansion** (also known as **globbing**)

These expansion are applied to each line in a shell script just as they are to each command-line you type interactively.

Some expansions are more commonly used in shell scripts than in interactive use.

Parameter Expansion

At its most basic, **parameter expansion** is *variable evaluation*: e.g., `$x` or `${x}`.

However, there are a number of **operators** that can be used within the `${}` form to modify the value returned from the parameter evaluation.

These operators are among the few functions that Bash has available for "**string processing**" (manipulating strings).

Complex string processing is generally done by invoking other programs, such as **AWK** and **SED**.

Parameter Expansion (contd.)

Key parameter expansion operators:

- `${param%pattern}` – remove shortest match against `pattern` from the RHS of `param`'s value
- `${param%%pattern}` – remove longest match against `pattern` from the RHS of `param`'s value
- `${param#pattern}` – remove shortest match against `pattern` from the LHS of `param`'s value
- `${param##pattern}` – remove longest match against `pattern` from the LHS of `param`'s value
- `${#param}` – length of `param`'s value (in chars)
- `${param/pattern/string}` – replace (longest) match against `pattern` in `param` with `string` (which may be empty)

Parameter Expansion (contd.)

Key parameter expansion operators: (contd.)

- `${param:offset:length}` – substring of `param`'s value, starting at 0-based `offset`, `length` optional
- `${@:offset:length}` – sublist of positional parameter, starting at 1-based `offset`, `length` optional
- `${!param}` – *indirect expansion*: use `param`'s *value* as the name of the variable to evaluate

The *patterns* used with the parameter expansion operators are constructed with the *filename expansion metacharacters*: `*`, `?`, and `[ ]`.

These are *not regular expression* patterns!

Parameter Expansion (contd.)

Examples:

```
file=test.text.save

echo ${file%.*} ==> test.text

echo ${file%%.*} ==> test

echo ${file#*.} ==> text.save

echo ${file##*.} ==> save

echo ${file/text/txt} ==> test.txt.save
```

Command Substitution

Command substitution allows the output of a command to be substituted into another command or stored in a variable.

There are two syntax alternatives:

- `$(command)` (newer)
- `` `command` `` (older)

E.g., get (absolute) path of program `ls`:

- `dayofweek=$(date +%A)`
- `` dayofweek=`date +%A` ``

Command substitution is widely used in shell scripting.

It is frequently used in an *assignment*, to capture the output of a command into a variable, so that it can be processed/used.

Command Substitution (contd.)

E.g., given one of the above assignments, we could do:

```
echo "Today is $dayofweek"
```

Of course we could also just do:

```
echo "Today is $(date +%A)"
```

However, if we need to have the day multiple times, repeatedly invoking command substitution will be much slower (each command will require creating a **subprocess** in which to run `date`, with its output captured by the shell, etc.).

Command substitution is not limited to *simple commands*, we frequently use *pipelines* as well:

```
os=$(grep DESCRIPTION /etc/lsb-release | cut -d= -f2 | tr -d "\"" | tr " " "_")
```

(Gets name of current Linux distro as a single word, e.g., `Mageia_5`.)

Command Substitution (contd.)

Another thing to be clear about is that what gets substituted in is what `command` writes to **standard output**.

Since only "standard output" is captured, it is often necessary to suppress *error messages* from being printed out to the terminal:

- e.g., `cmdpath=$(which $cmd 2>/dev/null)`
- get path of a command in variable `cmd` by using `which`
- if the command is not found, though, `which` will print an error message to **standard error**
- `2>/dev/null` causes **standard error** (**file descriptor** 2) to be *redirected* to a fake device that just throws data away
- note that `cmdpath` ends with *empty string* value if `which` fails

(Recall that we saw something similar with `command-test` commands!)

Command Substitution (contd.)

Note that being inside of `$( )` does not protect *parameter expansion* results from **word splitting**.

Thus, variables evaluated inside of command substitution forms must often be protected by quoting:

```
echo $(ls -l "$file")
```

Word splitting may also applied to command substitution's results, so command subsitution forms must themselves often be quoted:

```
ls -l "$(realpath "$file")"
```

The Bash reference manual says: "If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results."

Command substitutions may even be **nested** (`$()` form is easier):

```
listing=$(ls -l "$(realpath "$file")")
```

Arithmetic Expansion

While variable values are effectively stored as strings, it is possible to get them *interpreted as numbers* (integers).

`$(( math_expression ))` causes `math_expression` to be evaluated as an arithmetic (integer) expression, producing a number result.

Both numeric constants and variables can appear in expressions (variables do *not* require `$` for evaluation).

Can use most standard arithmetic operators:

- unary: `++` and `--`
- binary: `+`, `-`, `*`, `/`, `%`, `**`, `<<`, `>>`, `&`, `|`, etc.

E.g., `x=$((x + 25))`

Filename Expansion

Filename expansion was covered in the lectures on Bash.

Filename expansion uses several **metacharacters** to do *pattern matching on filenames*, allowing *multiple files* to be specified compactly.

The filename expansion metacharacters:

- `*` – matches any string, including the null/empty string
- `?` – matches any single character (but not none)
- `[...]` – matches any single enclosed character:
  `[yY]` `[abcd]` `[0123456789]`

Filename Expansion (contd.)

Filename pattern example:

- `i*-[1-5]??.{txt,text}`
- will match these files:
  - `i-123.text` (`*` matches empty string, `?`'s match 23)
  - `iabc-5ab.txt` (`*` matches abc, `?`'s match ab)
- will not match these files:
  - `i123.txt` (no dash)
  - `abc-123.txt` (doesn't start with i)
  - `ia-923.txt` (9 is not in range 1-5)
  - `ia-12.txt` (second `?` has no match)
  - `i-123.text.save` (does not end in "`txt`" or "`text`")

Filename Expansion Quirks

Filename expansions are used frequently in shell scripts, but there are a few quirks one needs to be aware of.

While the `*` meta-character "matches any string, including the null string," filename patterns that start with `*` do *not* match **hidden files** (set the `dotglob` option to change this behavior).

Thus, the following will not list hidden files (in the CWD):

```
for file in *; do
  echo "$file"
done
```

Begin filename patterns explicitly with `.` to match hidden files.

You can match *all* files with a *list* of both `.*` and `*`:

```
for file in .* *; do
  echo "$file"
done
```

Filename Expansion Quirks (contd.)

Using the pattern `.*` often leads to unexpected results, however, becase *every directory* contains `.` and `..` as the first two entries.

These will be matched by `.*`, but are often not really desired:

```
for file in .* *; do
  ls "$file"
done
```

⇒ causes the CWD to be `ls`'d twice effectively, plus parent.

Instead do:

```
for file in .* *; do
  if [[ "$file" != . && "$file" != .. ]]; then
    ls "$file"
  fi
done
```

Filename Expansion Quirks (contd.)

If a filename expansion pattern matches no files, what results is *the pattern itself* (unless `nullglob` or `failglob` options are set).

This can cause errors in scripts:

```
for file in *.text; do
  ls "$file"
done
```

⇒ `ls: cannot access *.text: No such file or directory`

It is generally a good idea to check that the match is a file:

```
for file in *.text; do
  if [[ -f "$file" ]]; then
    ls "$file"
  fi
done
```

Filename Expansion Quirks (contd.)

/'s in patterns represent *directories*,, so:

- `*/` – all the directories in CWD
- `*/*` – all the files in all the subdirectories of CWD
- `*/*/` – all the directories in all the subdirectories of CWD

Filename Expansion Shell Options

The following **shell options** affect *filename expansion* when set:

- `dotglob` – match filenames beginning with "." even if dot not explicitly in pattern (i.e., match *hidden filenames* by default)
- `extglob` – extended pattern matching features enabled
- `failglob` – patterns that fail to match filenames produce error
- `nocaseglob` – match filenames in a *case-insensitive* manner
- `nullglob` – patterns that fail to match filenames result in *null string* (rather than pattern itself)

Set/enable option with: `shopt -s OPTNAME`

Unset/disable option with: `shopt -u OPTNAME`