Shell Scripting 2: Control Constructs

1. Shells and Variables
2. **Control Constructs**
   - **input and output**
   - **exit status**
   - **command types**
   - **for and for-in loops**
   - **while loops**
   - **if-then-else**
   - **test-commands and expressions**
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Basic Input/Output

Bash has just one input command and two output commands:

`read [options] [var...]`

- reads *one line* from *standard input* (e.g., terminal)
- if var(s) given, assigns first word to first var and so forth, rest to last var
- if no vars, line assigned to variable `Reply`

`echo [-n] [arg...]`

- print arguments to *standard output* (e.g., terminal)
- `-n` option suppresses trailing $<newline>$

`printf format [arg...]`

- print arguments to *standard output* (terminal)
- `format` is a **format string** as in C's `printf()`

Exit Status

Recall that every Linux process (program) must return a *non-negative integer* that indicates its **exit status**:

- 0 (zero) means process completed *successfully*
- non-zero value means process *failed*
  (the particular value can denote the reason for failure)

Shell scripts return exit status with the `exit` builtin:

- `exit 0` (success)
- `exit 1` (failure, standard value)
- `exit` (returns status of last command run)

If a script does not end with `exit`, the script returns the exit status of its final command.

Command Types

There are several **command types** in Bash:

- **simple command** – command name along with its options and arguments:
  - `ls -lA Documents`
- **pipeline** – sequence of *simple commands* separated by |'s:
  - `grep printf lab.c | wc -l`
- **list** – sequence of *simple commands* and/or *pipelines* separated by ";" or "&&" or "||":
  - `gcc -o lab1 lab1.c && ./lab1`
- **compound command** – constructed using the various shell **control constructs** (for, if-else, etc.):
  - `for f in *; do ls -l ${f}; done`

Indenting Bash Code

Program code is **indented** according to certain patterns in order to enhance readability for humans.

While not necessary to be able to run the code, unindented code can be extremely difficult to understand and get correct.

As with most programming languages, you may see several alternative indentation patterns being used with Bash.

Syntax for the Bash control constructs is shown with the common indentation patterns that are preferred by the instructor.

If you wish to use alternative indentation patterns, make certain you understand when a *line break* or ";" is required by the syntax.

Looping: For

Looping constructs allow some code to be executed *multiple times*.

The code being executed each time through the loop is called the loop **body**.

A **for loop** is generally used to execute the body code a *definite number of times*.

For loops use an **index variable** that gets incremented/decremented until it reaches a certain value.

Often, the index variable is used in the body code

Looping: For (contd.)

For loop syntax:

```
for ((expr1 ; expr2 ; expr3)); do
  command1
  ...
done
```

Example: print sequence of numbers:

```
for ((x=1; x<=10; x++)); do
  echo $x
done
```

This would print out the numbers 1 through 10, each on a separate line.

Looping: For-In

A **for-in loop** is also referred to as a **mapping loop**.

It allows one to do something with each element of a **list**.

For-in loops are heavily used in shell scripting because **lists** are common from **filename expansion** and CLI arguments (`"$@"`).

For-in loop syntax:

```
for variable in list; do
  command1
  ...
done
```

Maps down **list**: successively binds **var** to each "word" in **list**, executing the body commands with each binding.

Looping: For-In (contd.)

Example: make backup copies of some files:

```
for file in *.text; do
  cp $file $file.save
done
```

Suppose the the CWD contained the files `file1.text`, `file2.text`, and `file3.text`.

`*.text` would expand to the *list:*

```
file1.text file2.text file3.text
```

The above for-in loop would set the variable `file` to `file1.text` and execute the `cp` command, then set `file` to `file2.text` and execute the `cp` command, and so forth.

Looping: For-In (contd.)

A `for-in` can be done on single line if working interactively:

```
for var in list; do command1; command2; done
```

Some examples for interactive use:

```
for n in 1 2 3; do touch test$n; done
```

```
for n in $(seq 1 10); do touch test$n; done
```

```
for f in *.text; do touch "$f"; done
```

```
for f in *.txt; do mv "$f" "${f%.txt}.text"; done
```

Looping: While

While loops are used when we want to keep executing body code while some **condition** is true.

While loop syntax:

```
while test-command; do
  command1
  ...
done
```

A **test-command** can be:

- a *command* – true if command has success **exit status**
- `[[ expression ]]` – true if **expression** is true

Test-Commands will be discussed further below.

Looping: While (contd.)

Example: reading lines from the terminal:

```
while read line; do
  echo "$line" > user-input.text
done
```

This while loop would read lines from the terminal, and write each out to the file `user-input.text`.

(Review **output redirection** (>) from the Bash lectures if necessary.)

Example: similar to for loop:

```
x=1
while [[ $x -le 10 ]]; do
  x=$((x + 1))
done
```

Looping: Until

Bash also provides **until loops** as a syntactic variant of *while loops*.

Until loop syntax:

```
until test-command; do
  command1
  ...
done
```

We say it is a syntactic variant, because the above could have been written as a *while* loop:

```
while ! test-command; do
  command1
  ...
done
```

Looping: break and continue

The `break` and `continue` builtins are available to use with the looping constructs:

- `break [N]` – break from Nth enclosing for/while/until
- `continue [N]` – resume next iteration of Nth enclosing for/while/until

Branching: If-Then-Else

Branching constructs allow you to execute code only if a certain condition is true or not true.

If-then syntax:

```
if test-command; then
  command1
  ...
fi
```

If-then-else syntax:

```
if test-command; then
  command1
  ...
else
  command1
  ...
fi
```

Branching: If-Then-Else (contd.)

Example: create a directory if it does not exist:

```
if [[ ! -d tempdir ]]; then
  mkdir tempdir
fi
```

Example: change `x`'s value depending on its current value:

```
if [[ $x -gt 10 ]]; then
  x=$((x - 10))
else
  x=$((x + 1))
fi
```

Branching: If-Elif

if-elif or **if-elif-else**, are like if-then-else but:

- with one or more instances of:

```
elif test-command; then
  command1
  ...
```

- optionally finishing with:

```
else
  command1
  ...
fi
```

Branching: If-Elif (contd.)

Examples:

```
if [[ $x -gt 10 ]]; then
  x=$((x*2))
elif [[ $x -gt 5 ]]; then
  x=$((x+5))
else
  x=$((++x))
fi
```

Alternatives: Case

A set of *alternatives* can be handled with **case**, which is similar (but more powerful) than C's **switch**.

case syntax:

```
case word in
  pattern [| pattern]...) commands-list ;;
  ...
  [*) commands-list ;;]
esac
```

Example:

```
case "$response" in
  y|Y|yes|YES) echo "OK" ;;
  n*|N*) exit 0 ;;
  *) echo "Invalid Response" ;;
esac
```

Test-Commands

Notice that in the above *control constructs*, we have `test-command` where you might expect to see `expression`.

Bash differs from C-family languages in that the conditional control constructs (if, while, etc.) *do not evaluate an expression*.

Rather, they *execute a command*, and use its *exit status* to determine true/false:

- 0 (success) means true
- >0 (failure) means false

These commands are what we have shown as `test-command`:
a command is executed, its exit status determines which branch to take or whether to continue looping.

Test-Commands (contd.)

One can use *any command* as a test-command:

- the command's exit status will determine branching/looping
- the command's actions get performed as a **side effect**

Example: see if a file contains some string:

```
if grep --quiet "$string" "$file"; then
  #file contains string:
  ...
fi
```

Example: read from file until hit file-end:

```
while read input; do
  #next line from file is in variable input:
  ...
done
```

Test-Commands (contd.)

Note that any *output* from a command being used as a test-command will be printed out, including *error messages*.

It is often necessary/desirable to suppress any error output—or even all output—when using a command to test a condition.

Can suppress error messages (stderr) with "`2>/dev/null`":

```
if metaflac "$file" 2>/dev/null; then
  ...(process flac file tags)...
else
  echo "Failed to retrieve FLAC file tags!"
  exit 1
fi
```

Test-Commands (contd.)

Can suppress all output (stdout + stderr) with "`&>/dev/null`":

```
if ! which metaflac &>/dev/null; then
  echo "metaflac program must be installed!"
  exit 1
fi
```

Test-Command Expressions

Test-commands are often written using one of *three special notations*, which effectively *turn expressions into commands*:

- `[ conditional_expression ]`
- `[[ conditional_expression ]]`
- `(( arithmetic_expression ))`

`[ expression ]` is the older style of test, whose options are more limited, with somewhat stranger syntax.

`[[ expression ]]` is the newer style of test, which uses a more standard syntax and has more functionality, so preferred.

`(( expression ))` evaluates an arithmetic expression:

- 0/true status, if the expression evaluates to non-zero
- 1/false status, if the expression evaluates to zero

Test-Command Expressions (contd.)

Notes:

- *whitespace* is generally required between `expression` and the brackets (e.g., `[[` and `]]`)
- any value besides the empty string (`""`) is considered true inside of `[ ]` or `[[ ]]` (e.g., `[[ false ]]` is true)

There are *three classes of expressions* that can occur inside of `[ ]` or `[[ ]]`:

- **string comparisons**
- **arithmetic comparisons**
- **file conditions**

String Comparison Expressions

String comparison expressions compare or test strings:

- `==` – binary equality operator (can just use `=`)
  (RHS argument can be a *globbing pattern*)
- `!=` – binary inequality operator
- `>` – **lexicographic ordering** ("sorting") comparison
- `<` – same
- `-n` – unary operator tests if non-null/non-empty string
- `-z` – unary operator tests if null/empty string

(Remember that variable/parameter values are effectively stored as strings.)

Arithmetic Comparison Expressions

Arithmetic comparison expressions compare values as *integers*:

- `-eq` – $=$
- `-ne` – $\neq$
- `-lt` – $<$
- `-le` – $\leq$
- `-gt` – $>$
- `-ge` – $\geq$

E.g., `[[ $x -gt 5 ]]`

File Condition Expressions

File condition expressions test/compare file characteristics.

Unary file conditions: (partial list)

- `-e` file – file exists
- `-f` file – file exists and is regular
- `-d` file – file exists and is a directory
- `-s` file – file exists and has size greater than zero

Binary file conditions:

- file1 `-nt` file2 – file1 newer than file2 (mtime)
- file1 `-ot` file2 – file1 older than file2
- file1 `-ef` file2 – file1 and file2 are same (i.e., inode)

e.g., `[[ -f "$file" ]]`

Expression Logical Operators

Expressions inside of `[[ ... ]]` can be combined using several **logical operators**:

- `! expression` – true if expression is false
- `expr1 && expr2` – true if both expressions true
- `expr1 || expr2` – true if at least one expression true
- `( expression )` – used to affect operator order

Pattern Matching Expressions

One thing that is sometimes required of expressions is to be able to check if a filename or variable value *matches some pattern*.

The == and != test conditions both allow the RHS argument to be *filename expansion pattern*.

Examples:

```
if [[ $file == *.bak ]]; ....
```

```
if [[ $x == 1*?1 ]]; ....
```

Pattern Matching Expressions (contd.)

A **regular expression** matching operator is also available for use in test-command expressions: `=~`

This is a binary operator, where the RHS is to be an *extended regular expression*.

Examples:

```
if [[ $x =~ 10*1 ]]; ....

if [[ $y =~ test.+ ]]; ....
```

true & false

`true` and `false` are commands:

- `true` simply returns 0/success
- `false` simply returns 1/failure

They are useful as **Boolean/flag** variable values, because they function as desired when used as test-commands:

```
#Exit if no regular files in CWD:
regfile=false
for file in *; do
  if [[ -f $file ]]; then
    regfile=true
  fi
done

if ! $regfile; then
  exit 1
fi
```