Shell Scripting 4: Functions & Arrays

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. **Functions and Arrays**
   - **functions**
   - **arrays**
   - **calling AWK and SED**
   - **advanced I/O**
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Functions

Bash supports the use of **functions** (**subroutines**).

A function can be defined using any of the following syntax variations:

- `function` *name* `() { ...body...; }`
- `function` *name* `{ ...body...; }`
- *name* `() { ...body...; }`

Functions are called and executed just like simple commands (but are run in the *current shell process*—not a **subprocess**).

Inside the **function body**, *arguments* to the function call can be accessed using the **positional parameters** `$1`, `$2`, etc.

(The original shell positional parameters are *restored* upon return from the function.)

Functions (contd.)

Function to create a required directory if it does not exist:

```
# Usage: ensure_dir DIRECTORY
function ensure_dir ()
{
  if [[ ! -e "$1" ]]; then
    #DIRECTORY does not exist so make it:
    if ! mkdir "$1"; then
      echo "Failed to mkdir $1!"
      return 1  #failure return from function
    fi
  elif [[ ! -d "$1" ]]; then
    #DIRECTORY exists, but as non-directory file:
    echo "Error: $1 exists but is not a directory!"
    exit 1  #failure exit from entire script
  fi
  return 0  #success return from function
}
```

Functions (contd.)

Bash functions differ from C-family language functions/methods:

- their *"parameter list"* is not explicit in the definition
- they run in the *same shell context* as the caller
- their return value is *exit status*

While `ensure_dir` must be called with a single argument, `DIRECTORY`, this is not made explicit in the syntax of the function's definition.

Running in the "same shell context" means that variables defined *outside* the function can be accessed *inside* the function, and any changes to their values will be reflected upon return.

(The only exception to "same shell context" is that *positional parameters* are bound to the function arguments on entry, and restored upon return.)

Functions (contd.)

Since you cannot return *values* from Bash functions, changing the values of external variables is the only method available to return info.

Functions can be **recursive**, but may require care due to running in the same shell context (which is different from most languages).

Variables *local to the function* in context may be created with a `local` **declaration**:

```
local cwd=$PWD
```

Arrays

Bash supports **one dimensional arrays**:

- zero-based indexing
- no size limits, dynamically expandable
- elements need not be indexed/assigned contiguously
- can use `declare` to make array variable:

  ```
  declare -a arr
  ```

- can also just assign elements:

  ```
  arr[0]=ford; arr[1]=gm; arr[2]=honda...
  ```

- can initialize entire array with paren notation:

  ```
  arr=(ford gm honda...)
  ```

- can initialize entire array with paren notation and list:

  ```
  elements="ford gm honda..."
  arr=(${elements})
  ```

Arrays (contd.)

Array access:

- retrieving array element value:
  `${arrvar[1]}`
- get entire array of values as single word:
  `"${arrvar[*]}"`
- get entire array of values as separate words:
  `"${arrvar[@]}"`
- get sub-array of values as separate words:
  `"${arrvar[@]:offset:length}"`
- get length of array (number of elements):
  `${#arrvar[*]}`
- get length of an element:
  `${#arrvar[1]}`

Arrays (contd.)

Array examples:

Initializing 10 array elements to be index + 5:

```
for ((i=0; i<9; i++)); do
  arr[$i]=$((i+5))
done
```

Non-contiguous elements:

```
arr[0]=a
arr[10]=b
echo ${#arr[*]}
echo ${arr[*]}
```

⇒ `2`

⇒ `a b`

Calling AWK and SED

Bash has limited *string processing* capabilities.

Because of this, the programs **AWK** and **SED** are often used in shell scripts to do complex string processing.

Basic usage is as follows:

- `awk -e AWK_CODE FILE`
- `... | awk -e AWK_CODE`
- `sed -e SED_CODE FILE`
- `... | sed -e SED_CODE`

Calling AWK and SED (contd.)

Examples:

- Get device for mountpoint `/backup`:
  ```
  dev=$(awk -e '/\/backup/{print $1}' /etc/fstab)
  ```
- Get file size:
  ```
  size=$(ls -l "${file}" | awk -e '{print $5}')
  ```
- Compress whitespace in a file:
  ```
  sed -e  's/[[:blank:]]\+/ /g' "${file}"
  ```
- Replace spaces with underscores in filename:
  ```
  newname=$(echo "$file" | sed -e 's/ /_/g')
  ```

See the awk and sed man pages plus slides for further info.

Advanced I/O

Redirections are normally *temporary*, in that they apply only to one command (that might be run in a new *subprocess(es)*).

E.g., using `nl` to number lines from a file:

```
nl -ba < "test.text"
```

E.g., same thing using `read` and a loop:

```
n=1
while read line; do
  printf "%3d: %s\n" $((n++)) "$line"
done < "test.text"
```

Advanced I/O (contd.)

The `exec` builtin can be used to (permanently) apply redirections to the current shell process.

This capability can be used to *open files for reading/writing*:

- open file for reading:

  ```
  exec FD< FILENAME
  ```

- open file for writing:

  ```
  exec FD> FILENAME
  ```

- where:
  - `FD` is a **file descriptor** (non-negative integer, *file handle*)
  - `FILENAME` is the file name/path

Advanced I/O (contd.)

Now, reading and printing numbered lines from a file can be done:

```
exec 3< "test.text"
n=1
while read -u3 line; do
  printf "%3d: %s\n" $((n++)) "$line"
done
```

Note that this does not change *standard input*, we simply use `exec` to open the file on FD 3 and read from that FD.

Opening a file for output and periodically writing to it:

```
exec 4> "test.text"
...
echo ... >&4
...
echo ... >&4
```

Advanced I/O (contd.)

`exec` can also be used to (temporarily) change standard input/output:

```
exec 3<&0  #save stdin in FD 3
exec 4>&1  #save stdout in FD 4

exec 0<"input.text"   #redirect stdin to input.text
exec 1>"output.text"  #redirect stdout to output.text

...code using stdin and stdout, now redirected to files...

exec 0<&3 3<&-  #restore stdin and remove FD 3
exec 1>&4 4>&-  #restore stdout and remove FD 4
```

Advanced I/O (contd.)

Other advanced I/O features are **here documents** and **here strings**, which allow *standard input* to be supplied from within a script file itself.

Here document:

```
cat <<-end
        line one
        line two
        end
```

(<<- vs. just << causes leading <tab> characters in lines to be ignored, so can indent for readability—but must use tabs.)

Advanced I/O (contd.)

Here string:

```
cat <<<$'line one\nline two'
```

So, instead of using `echo` to start a pipeline:

```
echo "$var" | awk '{print $2}'
```

we could use a *here string*:

```
awk '{print $2}' <<<"$var"
```