Shell Scripting 1: Scripts & Variables

Shells Review

Shells provide the **command line interface** (**CLI**) to an OS: users type commands to a *shell* program running in a *terminal window*, and the shell interacts with the OS **kernel** and other programs to carry out the commands.

Shells are **command interpreters**:
they parse what the user types, and may rewrite/expand various shell "shorthands," and finally carry out the prescribed actions.

Bash (**Bourne Again SHell**) is the default Linux shell.

Bash is a GNU extension of the original **Bourne shell** (**sh**).

Other shells are available: **`csh`**, **`tcsh`**, **`tcsh`**, **`zsh`**.

Shells Review (contd.)

Interacting with a shell via the CLI is termed **interactive use**.

Bash has a number of features designed specifically to enhance user productivity for *interactive use*:

- **command-line editing**
- **tab completion**
- **command history**
- **prompt** customization

In addition to supporting interactive use, most shells provide **control constructs** and other features that support writing simple **programs** using the shell language.

These features make Bash a type of **programming language**.

Shell Scripts

Programs written in shell languages are called **shell scripts**.

A **scripting language** is a type of programming language that allows for **rapid development** of simple programs, due to:

- being **interpreted** rather than **compiled**
- using simple **syntax** and **data typing** mechanisms
- having **automatic memory management**

Scripting languages have become popular, and several non-shell scripting languages are in wide use: PERL, Python, PHP, JavaScript.

Shell Scripts (contd.)

Shell scripts get their power because they are able to invoke all the same programs/commands/utilities that you can invoke interactively.

This allows one to implement complex tasks without having to write everything *"from scratch."*

Many Linux GUI programs invoke command-line programs, and these same programs can be called from shell scripts too.

Much of the startup of a Linux system used to involve running shell scripts (e.g., **service scripts** in `/etc/rc.d/init.d`).

Shell Scripts (contd.)

System administrators often write shell scripts to *automate tasks* that are done repeatedly.

They can be handy for normal users as well, when certain tasks need to be done repeatedly.

Shell scripts are also handy for those infrequent tasks that require difficult to remember commands.

You simply save the required commands in a script file, which you can easily invoke when needed.

Creating a Shell Script

A shell script is just a **text file**, which you can create with any **text editor** you want (though some editors are **syntax aware**).

Shell scripts can be executed just like a **binary** (**compiled**) program, as long as:

- the *first line* of the script file uses the following notation to tell the OS what interpreter to use:
  `#!/bin/bash`
  (the `#!` must be the first two characters in the file)
- the user has *execute permission* for the script file

Note that `#` is the **comment character**, and normally everything that follows on a line is ignored (`#!` is special).

Shell Script Example

Shell script to backup `~/Documents` to some directory:
(if script file is `backup-docs`, call like: `backup-docs ~/backups`)

```
#!/bin/bash

# Make certain that directory for backup exists:
if [[ ! -d "$1" ]]; then
    echo "Invalid directory for backup: $1"
    exit 1
fi

# Get date and  hostname:
date=$(date "+%y%m%d")
hostname="${HOSTNAME%%.*}"
backup_filename="Documents-${hostname}-${date}.tgz"

# Perform backup:
cd ~
tar -czf "$1/${backup_filename}" Documents

exit
```

Bash Syntax Basics

A programming language's **syntax** defines words and structure required for a program to be legal/valid in the language.

There are some critical differences between Bash's syntax and that of the C-family languages.

These differences often catch new Bash programmers!

Among the most important differences are those involving **line breaks** and **whitespace**.

(A **line break** is produced by a **newline/linefeed** character.)

(**Whitespace** refers to a sequence of space and tab characters.)

Bash Syntax Basics (contd.)

In C/C++/Java, *line break and whitespace differences generally are not meaningful*:

- programs could be written as a *single line* if desired (that's why you have to have `;`'s at the ends of most lines)
- whitespace is added only to improve readability:
  - `x=1;` is equivalent to `x = 1 ;`
  - `if(x==y)` is equivalent to `if ( x == y )`

This means that there are many different ways that the exact same code can appear in C/C++/Java.

Bash Syntax Basics (contd.)

In Bash, *line break and whitespace differences often are meaningful:*

- a *line break* has much the same effect as a ";" in C/C++/Java
- *whitespace* differences can change semantics:
  - `x=1` assigns 1 to variable x
  - `x = 1` invokes command x with arguments = and 1
  - `${x}` evaluates a variable to get its value
  - `${ x }` is an error
- *adjacent placement* produces **concatenation**:
  - `${x}${y}` produces a single (concatenated) result
  - `${x} ${y}` produces two separated results

Variables

In programming languages, **variables** are used to temporarily store a value during the run of the program.

Programming languages like C/C++/Java require that variables be created via **declarations** before they can be used:

```
int x;
```

Declarations also define the **type** of value a variable can store.

The value of a variable is changed with an **assignment statement**:

```
x = 10;
```

(In this context, “=” is the **assignment operator**, not the **equality** relationship.)

Variables (contd.)

In Bash, variables *do not have types* (mostly) and are *only rarely declared*.

Values are effectively kept as *strings*, which may be interpreted as integers depending on the context.

A **variable** is usually created by an **assignment**:

- `x=10` (lack of whitespace is important)
- `x=` or `x=""` assigns the default/empty value

(Again, note that "`x = 10`" is *not* a legal assignment, due to the whitespace around the =.)

The `unset` command will *delete* a variable:

```
unset x
```

Variables (contd.)

In languages like C/C++/Java, using a variable in an *expression* causes the variable to be **evaluated**.

Evaluating a variable means that its **value** is retrieved and then substituted for the variable's name:

```
int y = x + 100;
```

In Bash, variables are *not automatically evaluated*.

Instead, retrieving a variable's value requires explicitly **evaluating** the variable.

The synatx for evaluating a variable is:

- `${var}` (technically, this is called **parameter expansion**)
- `$var` (shorthand form, if end of variable name is clear)

Variables (contd.)

Example to demonstrate:

```
x=10
echo x      #prints out "x"
echo ${x}   #prints out "10"
```

Variable names can contain only alphanumeric characters plus underscores, and must begin with an alphabetic character or underscore

Variable names are *case sensitive*.

Standard style is to use *lowercase* variable names.

(Recall that **environment variables** are given *uppercase* names.)

Parameters

In Bash, **parameters** are *"entities that can store values."*

There are three types of parameters:

- **variables** – denoted by *names*
- **positional parameters** – denoted by *integers*
- **special parameters** – denoted by *special characters*

Positional parameters hold the *command-line arguments* passed to the shell script when it is run:

- e.g., if execute: `myscript norm 50`
- $0 will give the script name: `myscript`
- $1 will give the first argument: `norm`
- $2 will give second argument: `50`

Parameters (contd.)

A positional parameter of more than a single digit must be enclosed in braces when evaluated (e.g., `${10}`).

Positional parameters' values *cannot be changed via assignment*.

The `set` builtin can be used to set them however.

E.g., the following sets the positional parameters 1, 2, 3 to have the values A, B, C:

```
  set A B C
```

(This is primarily useful for interactively testing code.)

The `shift` builtin can be used to *remove* one or more of the positional parameters "from the left."

E.g., `shift` alone makes `$2` into `$1`, `$3` into `$2`, etc.

Parameters (contd.)

There are various **special parameters**:

- `$*` and `$@` both produce a **list** of the script arguments, but they differ if used inside of *double quotes*:
  - `"$*"` produces a *single word* from all the arguments
  - `"$@"` produces a *list* from the arguments
- `$#` gives the number of script arguments
- `$$` gives the process ID (PID) of the shell
- `$?` gives the last command's exit status

Quoting

Quoting disables the special meaning/treatment for certain words and characters.

For example, spaces usually are **word separators**, so filenames containing spaces must be quoted.

There are three types of quotes:

- **double quotes** (`""`): `"file with spaces"`
- **single quotes** (`''`): `'file with spaces'`
- **escapes** (`\`): `file\ with\ spaces`

There are three different mechanisms because they have slightly different behavior.

Also, you can use single quotes inside of double quotes, but you cannot use single quotes inside of single quotes, etc.

Quoting (contd.)

Double quotes preserve literal characters, except:

- `$` and ` retain their special meaning
- `\` retains its special meaning when it precedes:
  `$`, `, `"`, `\`, or *<newline>*
- special parameters `*` and `@` have special meaning
  when inside of double quotes

E.g., `"$variable"` evaluates `variable` and double quotes its value.

This is very handy if the value contains spaces, and you want it treated as a unit:

```
  file="filename with spaces"
  ...
  ls -l "$file"
```

Quoting (contd.)

Single quotes preserve literal characters
(a single quote cannot occur within single quotes).

The **backslash** is the **escape character**, preserving the literal value of the next character, except:

- \\<*newline*> is treated as a **line continuation** (meaning it is effectively removed/ignored)

Please note that Bash does not follow the **C escape notation** for special characters, e.g., `\n` for a *newline* character in a `char`/string.

However, the form `$'`*string*`'` produces a string with *escaped characters* replaced as in C strings:

- `\n` by <*newline*>, `\r` by <*return*>, `\t` by <*tab*>, etc.
- e.g., `$'\n'` is a newline character (`"\n"` is *not*)