Shell Scripting 5: Bash Development

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. **Bash Development**
   - **Bash programming style**
   - **debugging shell scripts**
6. Practical Advice
7. Bash Details and Advanced Features

Becoming a Bash Programmer

One of the hardest aspects of learning a new programming language is learning the best **idioms** for that language.

Often, those new to a language instead try to force code into models they are used to from other languages.

Bash is quite different from C-family languages, so if you try to emulate Java/C approaches, you will often end up being more verbose or less efficient than necessary.

For example, *arrays* and *numeric for loops* are much used in C-family languages, but are generally *not* likely to be the most direct way to do things in Bash.

A key reason is that Bash supports *lists* and the `for-in` list mapping construct, and both command-line arguments and file expansions produce lists.

Becoming a Bash Programmer (contd.)

Suppose invocations of your script have the following syntax:

```
myprog  FILE ARG...
```

If you need to loop through all the args and process each of them, the simplest way to do this in Bash is:

```
file=$1  #save FILE
shift  #remove FILE from the arguments list
#Loop through each supplied ARG:
for arg in "$@"; do
  ...process $arg...
done
```

This is the Bash way to process command-line arguments!

Becoming a Bash Programmer (contd.)

Since Java/C programmers don't have the experience (or option) of using lists, they will probably be tempted to make use of arrays and a numeric for loop:

```
args_arry=("$@")  #turn arguments list into an array
#Loop through each supplied ARG, now in args_arr:
for (( i=1; i<=${#args_arr[@]}; i++ )); do
  ...process $args_array[i]...
done
```

This is clearly neither the easiest nor most efficient way to do things in Bash!

It also clearly marks you as an inexperienced (or incompetent) Bash programmer.

Becoming a Bash Programmer (contd.)

On the other hand, sometimes people new to Bash end up making frequent (and unnecessary) use of fairly esoteric Bash elements.

A good example is the `eval` builtin:

- `eval [ARG...]`
- Combine `ARG`s into a single string, execute as a shell command.

While `eval` enables one to do some pretty slick things, it will generally be required only *extremely rarely* (consider that very few programming languages have a comparable capability).

In particular, one does not want to do:

```
eval ls -l
```

when the following is completely identical in functionality:

```
ls -l
```

Becoming a Bash Programmer (contd.)

Here is an example when `eval` is required:

```
# Print the capital letters from A to Z:
char_decimal=65  #ASCII code for A
while [[ char_decimal -le 90 ]]; do
  char_octal=$(echo "ibase=10; obase=8; $char_decimal" | bc)
  eval char="$'\\${char_octal}'"
  echo $char
  char_decimal=$((char_decimal + 1))
done
```

Key is to construct form `$'\nnn'` which represents the character with octal value `nnn`, and then `eval` it to get the actual character.

Developing Shell Scripts

Programs in C/C++/Java and other languages must be **compiled** before they can be run.

Bash programs are **interpreted**, so do not require compilation.

However, they can still contain both **syntax errors** and **logic errors**.

Bash provides an **interactive environment** in which you can *evaluate individual lines of code*.

You should test each line of code by running it in a shell before you commit it to a script file.

If you follow this approach, Bash scripts should contain fewer initial errors than with compiled languages

Debugging Shell Scripts

Unlike a compiled language, script **syntax errors** will manifest themselves *when the script is run*.

A syntax error will produce results like:

```
myscript: line 22: syntax error near unexpected token ‘else’
```

For such a message, start at line 22 of the script file and *look backwards through the script* to see what is wrong to make Bash think an `else` is not valid where it is on line 22.

The above message resulted from forgetting the `;` before `then`:

```
if [[...]] then
  ...
else
  ...
fi
```

Debugging Shell Scripts (contd.)

Bash can be run in **debugging mode** to assist in debugging scripts.

If Bash is started with the `-x` option, a *trace* of each command plus its arguments gets printed to standard output after the command has been expanded but before it is executed.

You can run a script with this option like:

```
bash -x myscript
```

The `set` builtin can also be used to turn debugging on and off for different portions of the script:

```
set -x  #debugging on
...code to debug...
set +x  #debugging off
```

Debugging Shell Scripts (contd.)

Remember that the `set` builtin can be used to set the *positional parameters* for testing code lines in the interactive environment:

```
set A B C
for param in "$@"; do echo $param; done
```

$\Rightarrow$

```
A
B
C
```