COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 5 - Shell Scripting

NCC Moore

McMaster University

Summer 2021

Shell Scripting

Variables and Assignment

Environment Variables

Miscellaneous Complications

Command Substitution

Conditional Control Flow

Iterative Control Flow

Errata

"As with all instruments, it is the man, not the tool that makes the difference. The more subtle the tool, the greater the difference. Skill with a shovel makes less difference than with a violin."

– Jeff Cooper

Using your Hands is LAME!

So far we have barely dipped our toes into the vast ocean that is the Bash environment. Time to level up our skillz!

- So far, every command you've used has been entered or selected manually.
- Shell scripting allows you to collect shell commands together and execute them as one unit!
  - Scripts are *fast*, executing hundreds of commands in the blink of an eye!
  - Scripts are *versitile*, anything you can do in Bash can be put in a script!
  - Scripts are *reliable*, avoiding the errors inherent in manual command entry.
- The only catch is that shell scripts take time to set up, and are somewhat harder to work with and debug than traditional programs.

Loose Scripts Sink Ships!

To create a shell script...

- Create a file with a *.sh extension, like `my_script.sh`.
- Open it in your favourite text editor (emacs of course), and give it the following contents.

```
#!/bin/bash

echo "Hello World!"
```

- To be used, the script must be made **executable**.
- The following command sets the script as executable.

```
$ chmod +x my_script.sh
```

- You can execute it the same way as any executable.

```
$ ./my_script.sh
```

The Whole Shebang!

The first line of `my_script.sh` is called a **shebang**

- The shebang is indicated by `#!`.
- This line indicates which program we wish to use to interpret the script.
- We can, for example, run python files as scripts using:

```
#!/usr/bin/python
```

Alternatively, we can pass the script into the interpretter we want directly.

```
bash my_script.sh
python3 checkers.py
```

Assigned Variability

Assign a variable with =, but *do not leave any spaces!*

```
#!/bin/bash
var1=Hello
var2=GoodBye
echo $var1 # outputs Hello
echo var2 # outputs var2
```

Here we begin entering the essential weirdness of Bash scripting.

- Notice how we're working with strings here, but there are no quotes in sight!
- Notice also that a variable is not substituted into a command unless preceeded by a $!

Keep in mind everything here also apply outside of the script file!

Don't Quote me on that!

In Bash scripting, quotes do not denote string values.

- Bash **separates arguments by whitespace**.
  - This is similar to how Haskell separates function arguments with the space character.
- Quotes allow us to include whitespace in arguments.

```
#!/bin/bash
touch some file
  # creates two files, some and file
touch "some file"
  # creates one file, "some file"
rm some file "some file"
  # deletes all of the above files
```

Single vs Double Quotes

- **Single quotes** preserve the input verbatim.
  - This is useful for certain commands like `grep`, where the input parameters use symbols which overlap with symbols used in Bash.
- **Double quotes** preserve input, but permit substitutions.

```
#!/bin/bash
var='Hello World'
echo '$var' # outputs $var
echo "$var" # outputs Hello World
```

I'd Like to Get Some Input...

The following **special variables** are reserved for managing arguments to your Bash scripts.

$1 - $9	The first nine supplied arguments
$@	All supplied arguments
$#	The number of arguments supplied

```
#!/bin/bash
if [ $# -eq 2 ]; then
  echo $1
  echo $2
else
  echo "Incorrect Number of Inputs"
fi
```

I'd Like to Get Some Input... (cont.)

You can provide arguments to a Bash script the same way you'd provide arguments to any other Bash command!

```
$ ./test.sh
Incorrect Number of Inputs
$ ./test.sh Hello World
Hello
World
```

Script it and They Will Come!

Here are some more special variables:

`$0`	the name of the Bash script
`$$`	process id of the current script
`$USER`	username of the user executing the script
`$HOSTNAME`	hostname of the machine the script is running on
`$RANDOM`	produces a random number
`$HOME`	home path of the user executing the script
`$PATH`	directories at which Bash can find your executable binaries

Some of these values are **environment variables**, which have special functions within the Bash shell.

Bash Startup Scripts

A common part of manual installation in Linux requires setting environment variables for the program you're installing. To do that, we need access to Bash's startup routines!

- When Bash starts up, it begins by looking for the script `/etc/profile`, and executing it if it exists.
  - Changes here effect all users, but require super user privileges.
- The next script Bash looks for depends on whether or not the shell is a **login shell**. If you entered a password, you're in a login shell!
  - In a non-login shell, it will run `~/.bashrc` if the file exists.
  - In a login shell, Bash looks for `~/.bash_profile`, `~/.bash_login`, or `~/.profile`, in that order, and executes the first one it finds that works.
- In both cases, changes do not effect other users, and super user privileges are not required.

Saving the Environment

In order to set environment variables, all we need to do is add them to one of Bash's startup scripts.

- We already know how to assign variables in a bash script.

```
ENVVAR=/path/to/some/directory
```

Variables assigned this way are scoped to the Bash shell that created them, but are not transferred!

```
export ENVVAR=/path/to/some/directory
```

Using the `export` command makes the variable available to the creating Bash session, *as well as all subprocesses!*

Example: Extending $PATH

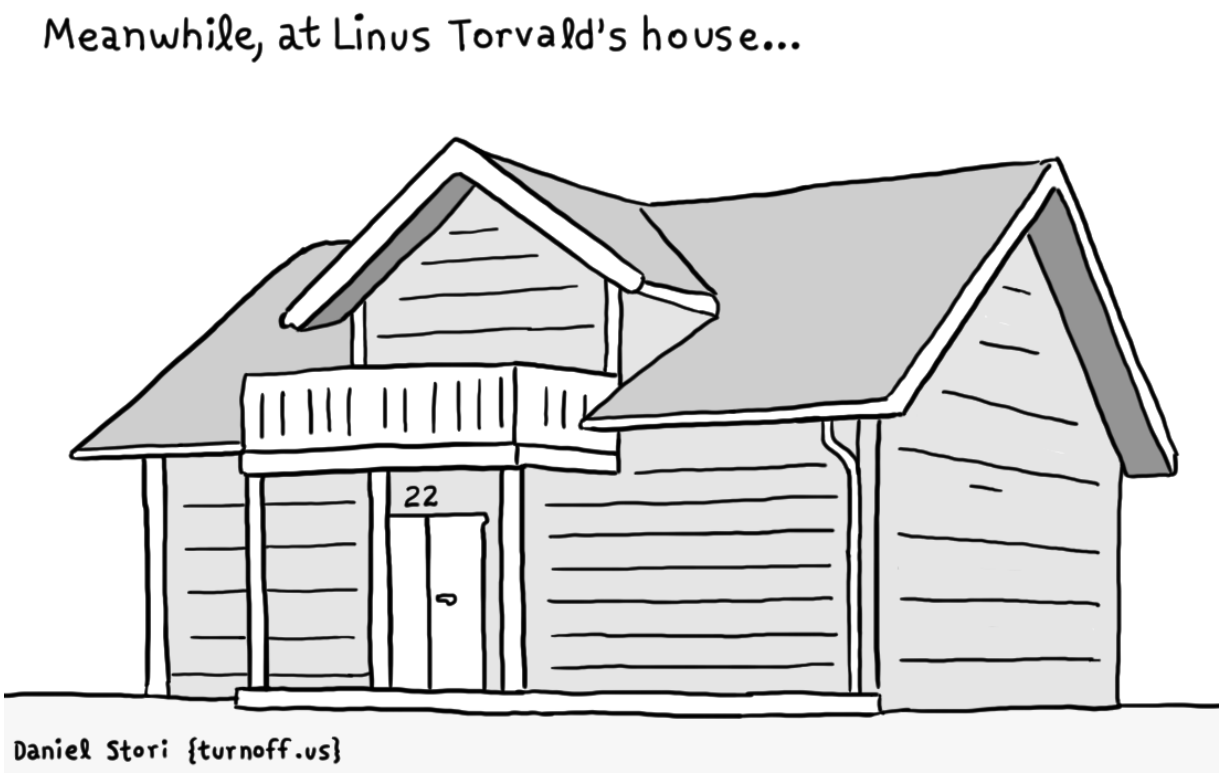
Recall that Bash uses $PATH to look for executable programs.

- If you are adding a new program, and you don't want to have to navigate to the folder in order to run it, add it to $PATH!

```
export PATH=$PATH:path/to/some/directory
```

Add the above line to ~/.bash_profile or similar.

- Calling echo will show the different paths $PATH looks through is a **colon separated list**.
- The above construction is similar to Python's += assignment operator.
- Any executable binaries at the specified directory can be used as commands!
- The only catch is that you have to restart your Bash session so the changes take effect.



"To be a nemesis, you have to actively try to destroy something, don't you? Really, I'm not out to destroy Microsoft. That will just be a completely unintentional side effect."

Beware Of Whitespace

Whitespace (spaces, tabs and newlines) have more semantic content in Bash than any other language (except perhaps Haskell).

- Think about the way commands are structured. A space denotes **argument application!**
- Bash takes every character *literally*!

```
#!/bin/bash
var2=Hello    # GOOD!
var1 = Hello # EVIL!
```

Bash has No Types!

Any programmer not utterly ruined by Python knows that variables have types!

```
1  int var1 = 1; // This is an integer
2  char[] var2 = "Hello World!"; // This is a string
3  char var3 = '!'; // This is a character
```

In Bash however,

```
var1=1 # This is some text
var2="Hello World!" # This is some text
var3='!' # This is some text
```

It is more correct to think of variables in Bash as being more like Macros in C than Variables, in that they perform *direct character substitution, regardless of syntactic construction!*

Concatenation

Because of this, concatenation does not require an operator.

- In Python, we concatenate strings like so:

```
1 var1 = "Hello, " + "World!"
```

In Bash, we just perform adjacent character substitutions:

```
var1="Hello, "
var2="World!"
echo $var1$var2
```

Again, this is much closer to Macro programming than working with variables in a regular programming language.

Command Substitution

If you want to assign a variable to the output of a command, you need to use **command substitution**.

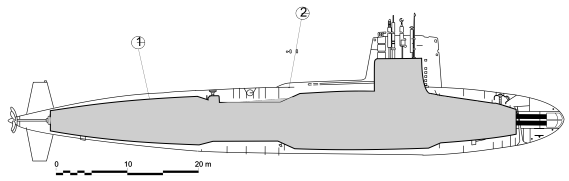
```
#!/bin/bash

count=$(ls -l | grep -v total | wc -l)
echo "Number of files in directory is $count"
```

This allows you to redirect the results of **stdout** to intermediate variables, and then back into **stdin**.

- You can also do this with **pipes**, which we will be covering a bit later in this course.

Subshells

Code executed inside of a script is considered to be a **subshell** of the shell you execute it from.



- Variables within a shell are not automatically available within subshell processes spawned by a shell.
- Think of it like global variables in Python. To use them within a function, you have to declare them!
- Use the `export` command to accomplish this.

```
#!/bin/bash
VAR=Hello
export VAR
./script.sh # VAR is now availabe in script.sh
```

Expressions of Dissatisfaction

You may be asking yourself, "OK, but where's the MATH!?"

- Integer arithmetic must be performed within the **double parenthesis** envirnoment: `$(( math goes here ))`
- This is really just syntactic sugar for the `expr` command:

```
RESULT1=$(( 1 + 5 ))
RESULT2=$(expr 1 + 5) # both accomplish the same thing
```

The full listing of available expressions is in the `expr` manual page!

If Statement Syntax

```
if [ <some test> ] ; then
   # some commands
fi

if [ <some test> ] && [ <another test> ] ; then
  # some commands
else
  # some more commands
fi

if [ <some test> ] || [ <another test> ] ; then
  # some commands
elif [ <some other test> ]
  # some more commands
fi
```

The Test Command

```
if [ -n "Hello" ] ; then
  echo "Hello is bigger than zero"
fi
```

- The square brackets [ ] are a reference to the `test` command.
- A full listing of the sorts of tests you can perform is in the `test` man page. It's worth a read!
- The `test` command can also be used as follows.

```
if test -n "Hello" ; then
  echo "Hello is bigger than zero"
fi
```

Comparators!

This is a non-exhaustive list of comparisons available in `test`.

Operator	Data Type	Description
`! x`	Expression	x is true
`-n x`	String	Length of x is ¿ 0
`x = y`	String	x and y are equal
`x != y`	String	x and y are not equal
`x -eq y`	Integer	x and y are equal
`x -gt y`	Integer	x is greater than y
`x -lt y`	Integer	x is greater than y
`-e x`	Item	Item exists
`-f x`	File	x exists and is a regular file
`-d x`	Directory	x exists and is a directory
`-x x`	File	x exists and is executable

Iterating the Concept

- `for / in`

```
INPUT="one two three"
for item in $INPUT ; do
  echo $item
done
```

- C-style `for`

```
for ((i=0 ; i < 10 ; i++)) ; do
  echo "Counter: $i"
done
```

- `while`

```
COUNT=0
while [ "$COUNT" -lt 10 ] ; do
  echo "$COUNT"
  COUNT=$(( $COUNT + 1 ))
done
```

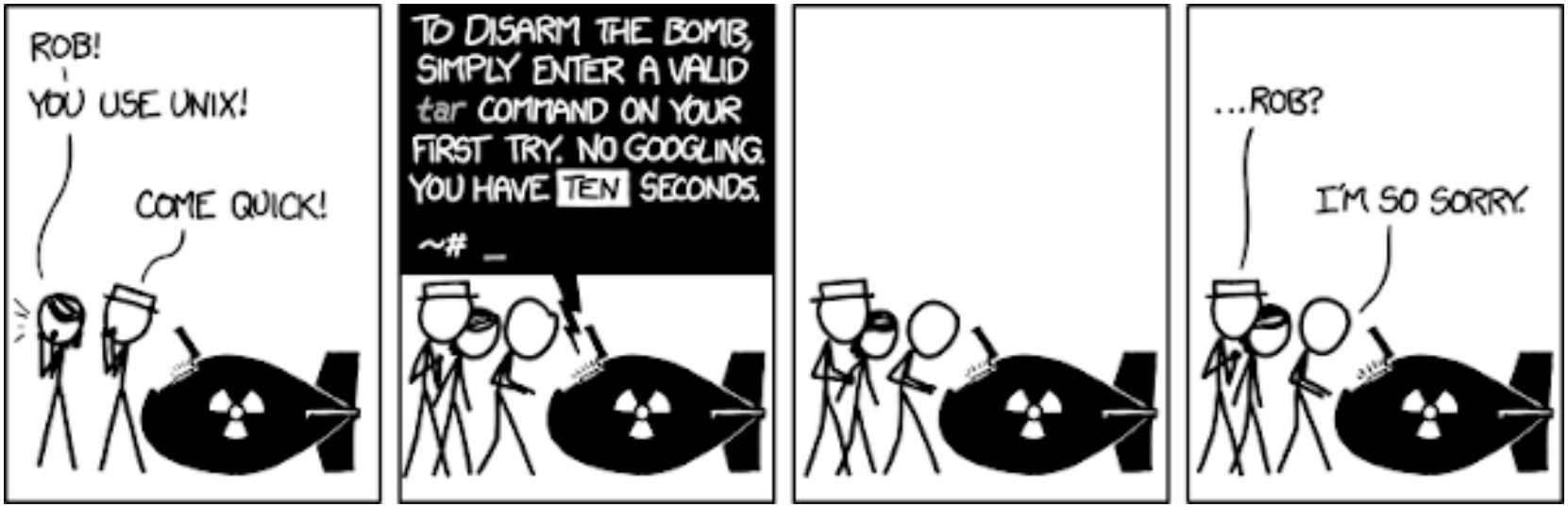
Internal Field Separation

By default, Bash separates inputs and arguments by *whitespace*.

```
#!/bin/bash

IFS=$":"
INPUT="a:b:c:d"
for field in $INPUT; do
    echo $field
done
unset IFS

INPUT="a b c d"
for field in $INPUT; do
    echo $field
done
```

Try commenting the line `unset IFS` and see how the output changes!

The Last Slide Comic



Credits

The contents of these slides were liberally borrowed (with permission) from slides from the Winter 2020 offering of 1XA3 (by Curtis D'Alves), and the Winter 2021 offering of 1XC3 (by Dr. Kevin Browne).