Today

- Shell/Bash scripting

1

Review: Unix Commands

- How do you configure bash?
- How is PATH used?
- What is an alias? How do you define it? Where can you define? How can you see your aliases? How can you delete it?
- How do you redirect input? Output?
- What is your favorite text editor?
- What does a shell script look like?
  - How do you run a shell script?

2

Classifications of Shell Commands

Recall: A shell script is a **text** file that contains shell or UNIX *commands*

- Programs/Executables
  - Most programs that are part of the OS in **`/bin`, `/usr/bin`**
- Built-in commands
- Functions
- Aliases

```
$ type cat
cat is /usr/bin/cat
$ type ls
ls is aliased to `ls --color=auto'
$ type cd
cd is a shell builtin
$ type if
if is a shell keyword
```


5

Classifications of Shell Commands

All work the same in taking parameters and exit status

- Programs/Executables
  - Most programs that are part of the OS in **`/bin`, `/usr/bin`**
- Built-in commands
- Functions
- Aliases

6

Built-in Commands

- Built-in commands are internal to the shell and do not create a separate process
- Commands are built-in because:
  - They are intrinsic to the language (`exit`)
  - They produce side effects on the current process (`cd`)
  - They perform faster
    - No fork/exec

Important Built-in Commands

`exit`	Quit the shell
`exec`	Replaces shell with program
`cd`	Change working directory
`shift`	Rearrange positional parameters
`set`	Set positional parameters
`wait`	Wait for background process to exit
`umask`	Change default file permissions
`eval`	Parse and execute string

Check out `cd`:
```
1. which ls
2. which cd
```

Important Built-in Commands

`time`	Run command and print times
`export`	Put variable into environment
`trap`	Set signal handlers
`continue`	Continue in loop
`break`	Break in loop
`return`	Return from function
`:`	True
`.`	Read file of commands into current shell

Comments

- Comments begin with an #
- Comments end at the end of the line
- Comments can begin whenever a token begins
- Our text editors should help you with syntax highlighting
- Examples:

```
# This is a comment
# and so is this
grep foo bar # this is a comment
grep foo bar# this is not a comment
```

Add a comment at 2nd line in your script that describes what your script does

Variables

- To set:
  `name=value` ← Notice no spaces around =
  - Variables are *untyped*
- To use: `$var`
- Variables can be local or environment
  - Environment variables are part of UNIX and can be accessed by child processes
- To turn *local* variable into *environment* var:
  `export variable`

Variable Example

```
#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
echo '$MESSAGE'
echo "$MESSAGE"
```

Prints variable
Prints literally
Prints variable

variable.sh

Using Environment Variables

```
#!/bin/bash

echo I am $USER
echo "I live at $HOME"
```

- Both statements would work, with or without quotes
- Better practice: with quotes

env_var.sh

Parameters

- A parameter is one of the following:
  - A *positional* parameter, starting from 0
  - A *special* parameter
- To get the value of a parameter: **`${param}`**
  - Can be part of a word (**`abc${foo}def`**)
  - Works within double quotes
- The **{}** can be omitted for simple variables, special parameters, and single digit positional parameters

Positional Parameters

- The arguments to a shell script
  - `$0, $1, $2, $3 …`
  - Parameter 0 is the name of the shell or the shell script
- The arguments to a shell *function*
- Arguments to the `set` built-in command
  - `set this is a test`
    - `$1=this, $2=is, $3=a, $4=test`
- Manipulated with `shift`
  - `shift 2`
    - `$1=a, $2=test`

Example with Parameters

Script

```
#!/bin/sh

# Parameter 1: file
# Parameter 2: how_many_lines
head -$2 $1
```

Invocation:

```
$ bash toplines /usr/share/dict/words 3
A
A's
AMD
```

Special Parameters

Parameter	Meaning
$#	Number of positional parameters
$-	Options currently in effect
$?	Exit value of last executed command
$$	Process number of current process
$!	Process number of background process
$*	All arguments on command line from 1 on
"$@"	All arguments on command line Individually quoted "$1" "$2" …; good if parameters contain spaces

params.sh

Exit Status

- $? : exit status of the most recently executed command

```
run_some_command
EXIT_STATUS=$?
```

- 0 for exit status means that command executed successfully/normally
  - Anything else means there was an error

Special Characters

- The shell processes the following characters specially unless quoted:
  - `| & ( ) < > ; " ' $ ` space tab newline`
- The following are special whenever patterns are processed:
  - `* ? [ ]`
- The following are special at the beginning of a word:
  - `# ~`
- The following is special when processing assignments:
  - `=`

Command Substitution: ``

- Used to turn the output of a command into a string
- *Used to create arguments or variables*

```
$ date
Thu Jan 20 22:47:27 EST 2022
$ NOW=`date`
$ echo $NOW
Thu Jan 20 22:47:31 EST 2022
$ PATH=`genPath`:$PATH
```

Compound Commands

- Multiple commands
  - Separated by semicolon or newline
- Command groupings
  - pipelines
- Subshell

```
( command1; command2 ) > file
```

- Boolean operators
- Control structures

Control Structures Summary

- `if … then … fi`
- `while … done`
- `until … do … done`
- `for … do … done`
- `case … in … esac`

Control Structures: `if`

```
if expression
then
        command1
        …
else
        command2
        …
fi
```


What is an expression?

- Any UNIX command
- Evaluates to true if the exit code is 0, false if the exit code > 0
- Special command **`/bin/test`** handles most common expressions:
  - String compare
  - Numeric comparison
  - Check file properties
- **`[ ]`** often a built-in version of **`/bin/test`** for syntactic sugar

Examples

```
if test $USER = "sprenkles"
then
        echo "I know you"
else
        echo "I don't know you"
fi
```

know.sh

```
if [ -f /tmp/stuff ] && \
    [ `wc -l /tmp/stuff | cut -f1 -d" "` -gt 10 ]
then
        echo "The file has more than 10 lines in it"
else
        echo "The file is nonexistent or small"
fi
```

filesize.sh

Boolean Operators

- Exit value of a program is a number
  - 0 means success
  - anything else is a failure code
- cmd1 && cmd2
  - executes cmd2 if cmd1 is successful
- cmd1 || cmd2
  - executes cmd2 if cmd1 is not successful

Send output to black hole (Can't be read)

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Mon Jan 17 15:32:05 EST 2020
```

test Summary

- String based tests

`-z string`	Length of string is 0
`-n string`	Length of string is not 0
`string1 = string2`	Strings are identical
`string1 != string2`	Strings differ
`string`	string is not NULL

- Numeric tests

`int1 –eq int2`	First int equal to second
`int1 –ne int2`	First int not equal to second
`-gt, -ge, -lt, -le`	greater, greater/equal, less, less/equal

test Summary

- File tests

`-r file`	File exists and is readable
`-w file`	File exists and is writable
`-f file`	File is regular file (exists)
`-d file`	File is directory
`-s file`	File exists and is not empty

- Logic

`!`	Negate result of expression
`-a, -o`	And operator, or operator
`( expr )`	Groups an expression

What does this code do?

```
ARGS=1      # Number of arguments expected
# Exit value if incorrect number of args passed.
E_BADARGS=65

test $# -lt $ARGS && echo "Usage: `basename $0` <arg1>" && \
exit $E_BADARGS
```

- Add appropriate code to `toplines`