shell scripting!

The bash Shell Program

- The command-line interface you type into is a Bash Shell
  - It's specifically a running instance of the program found at **`/bin/bash`**

- **`man bash`** Description: "`Bash` is an `sh-compatible` command language interpreter..."
  1. "that executes commands read from the standard input"
     - That's you typing and pressing enter.
  2. "or from a file"
     - Shell scripts - today's focus!

- "sh-compatible command language"
  - `sh` was the original Unix **<u>sh</u>**ell program of the 70s and 80s
  - `bash` was released in 1989 and rose to prominence in the 90s
  - `sh` was the Bourne Shell named after inventor Stephen Bourne
  - `bash` is the Bourne Again Shell

What's the big deal about Shell Scripts?

- You can combine commands into a single script
  - Scripts can have variables, branched logic (if-else), loops, and functions, too!

- Automate common tasks you manually perform into a script
  - Such as commands you want to run each time you login or make a commit
  - Build steps in a programming project
  - Data scraping or processing pipelines
  - Backing up data

- Put a *facade* over programs you haver specific use cases for which require many common command-line arguments
  - Simple example this semester: **pandoc -o <name>.pdf <name>.md**
  - Could write a facade script to run the above command with: **md2pdf <name>**

Hello World - Hands-On

- In your container:

1. Make a directory named **shell-scripts** in /mnt/learncli/workdir
2. Change your working directory to **shell-scripts**
3. Open a new file named **hello-world** in **vim**
4. Add the following lines to the file:

```
target="world"
echo "hello, ${target}"
echo "\${#} : ${#}"
echo "\${@} : ${@}"
echo "\${1} : ${1}"
```

5. **Save** the file and **quit** vim
6. Run your script: **bash hello-world**

Variables and String Interpolation

- An assignment statement initializes or reassigns a variable
  - Variables are dynamically typed
  - Since you're scripting a in textual environment *most* variables are strings

- String interpolation substitutes string variables into other strings
  - Example: **`"a_variable is ${a_variable}"`**
  - When the string is evaluated the token **`${a_variable}`** is substituted with the variable named `a_variable`'s value

- String interpolation works with double quoted strings
  - Single quoted strings would treat the contents literally and not substitute
  - For more documentation on bash strings: https://www.tldp.org/LDP/abs/html/quotingvar.html

Script Argument Variables

- Suppose you execute `bash hello-world go heels`
  - The bash interpreter evaluates your script **hello-world**
  - The first argument **go** is held in the argument variable **${1}**
  - The second argument **heels** is held in the argument variable **${2}**

- The special variable **${0}** holds the *path* to the executed script file

- The special variable **${@}** holds all arguments space separated

Comments

- Comments in Bash scripts begin with a #

- You can add comments to ends of lines if there is a leading space

```
$ echo hello #this is a comment
$ echo hello#this isn't a comment
```

- Just like comments in other languages, this text is ignored by the interpreter and is for the humans reading the code

Shell Script Execution

- So far, we've executed our shell script files via **bash**:

```
$ bash [script-file] [arg_1] [arg_2] [arg_N]
```

- A convention allows us to execute *scripts* as if they're *programs*:

```
$ [script-file] [arg_1] [arg_2] [arg_N]
```

- To make use of this convention, you need to understand three ideas:
    1. How to write a **shebang** line in a script
    2. How to give a script executable permission with **chmod**
    3. The **${PATH}** environment variable and its lookup logic

#! - The Shebang line

- Starting the first line of a script file with #! is called a shebang
  - Also called a shebang, hash bang, hash-pling, pound-bang per Wikipedia.

- Immediately following the `#!` is an absolute path to an interpreter
  - For example, if writing a Bash script: **`#!/bin/bash`**

- What's going on here?
  - **#!** is a human readable byte pair of ASCII values: 0x23 0x21
  - When the operating system function responsible for loading a new program reads in a program file it first checks the initial bytes.
  - If it finds a shebang **#!** it will treat the file as a script, not a binary program.
  - It then reads the path to the interpreter **(eg /bin/bash)** and loads the interpreter program which in turn processes the script.

Executable Permission with chmod

- Files in Unix-like systems have settings which control their permissions **modifiers**
  - Can someone **read**, **write**, or **execute** the file?
  - The **`ls -l`** `lists` file entries with their permission flags
  - For example: - rwx r-x r-x

- Permissions in Unix-like systems are classified in three ways:
  - What can the *owner* associated with the file do with it?
  - What can users in the *group* associated with the file do with it?
  - What can any user on the system do with it?
  - More complex permissions are available through access-control lists

- All we're concerned with in this exercise is whether you can **execute** a file as a program
  - Thinking about permissions management in a multi-user system is beyond our scope
  - Important in scenarios like classroom servers where many users can login to it

- The **chmod** program **ch**anges permissions **mod**ifiers on file entries
  - In general, searching for "how do I give permission (read|write|execute) to (owner|group|anyone)" will lead you to the correct arguments to use

- **`chmod +x <file>`**
  - enables the executable permission for everyone on a given file -- and is what we want in this case: **chmod +x hello-world**
  - You can now run the script via a relative or absolute path like **./hello-world**

The ${PATH} Environment Variable

- Run the **printenv** command to see all **Environment Variables** established in your shell session and look for **PATH** and **PWD**

- **PATH** should look something like:

  ```
  /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
  ```

- Notice between the :'s is a sequence of paths:

  ```
  /usr/local/sbin
  /usr/local/bin
  ...
  /bin
  ```

- When you run a command that begins with a simple identifier (such as echo, ls, man, grep, hello-world and so on), as opposed to a path (either absolute or ./hello-world) each of the directories on your PATH is checked to see if an executable file of that name is found in that directory. If it is found, then that program will be loaded.

- You can change your PATH just like any other variable:
  - `PATH="${PATH}:${PWD}" - add the absolute path to the current working directory to PATH`
  - This change will only last for the current session, we'll look at how to make it persistent later

- The program **which <program>** will use the lookup PATH to find the absolute path to a program, if it exists, and print it out. Useful for debugging.

Steps on hello-world script:

1. Add the shebang line at the top of the file: **#!/bin/bash**

2. Set executable permission on the file: **chmod +x hello-world**

3. Add the directory these toy scripts to lookup PATH:
   PATH="${PATH}:${PWD}"

4. Try running your program as if it were any other: **hello-world**

Next Exercise: md2pdf

- Let's write a facade script for our use of **pandoc**:
  - **pandoc -o <file>.pdf <file>.md**

- Goal is to be able to run the following command and it results in the above command:
  - **md2pdf <file>**

- In vim, open a new file named md2pdf and add the contents:

```
#!/bin/bash
file=${1}
pandoc -o "${file}.pdf" "${file}.md"
```

- Save, then add the executable bit with **chmod +x md2pdf**

- Finally, make a simple markdown file in the working dir named **demo.md** and then run: **md2pdf demo**
  - The file demo.pdf should have been produced!

- What happens when you run md2pdf without an argument or with a filename to a non-existent markdown file?
  - Let's improve the script!

Conditionals (if/else) depend on truthiness

- The syntax for an if-then-else statement in bash:

```
if <command>
then
   <then body>
elif <command>
then
   <else if body>
else
   <else body>
fi
```

- Just like in programming languages in the C family, the else if (elif) and else branches are not required. More else if branches can be added, as well.

Exit Codes

- When programs **exit** they generate an exit status code you can use to determine its success or failure
    - Zero conventionally means a program exited successfully - 0 is truthy in if/else
    - Non-zero means some exceptional case (such as an error) - non-0 is falsey

- The Bash shell stores the exit status code of the last program run in the special variable ${?}
    - Demo:

```
ls
echo "${?}" // prints 0
foobar // prints foobar: command not found
echo "${?}" // prints 127
```

- The manual page of a program will tell you about its status codes

The test Program

- All unix-like systems have a program named test that does things like:
  - compare two strings
  - check for the existence of a file
  - check for the existence of a directory
  - negate a boolean expression, form compound expressions, and so on

- Documentation of the kinds of tests you can perform are in **man test**
  - Rely on documentation when you need to conditionally test in bash! Too many (strange) options.

- Example:

```
learncli$ test "${PATH}" = "${PWD}"
learncli$ echo "${?}"

learncli$ test "${PATH}" = "${PATH}"
learncli$ echo "${?}"
```

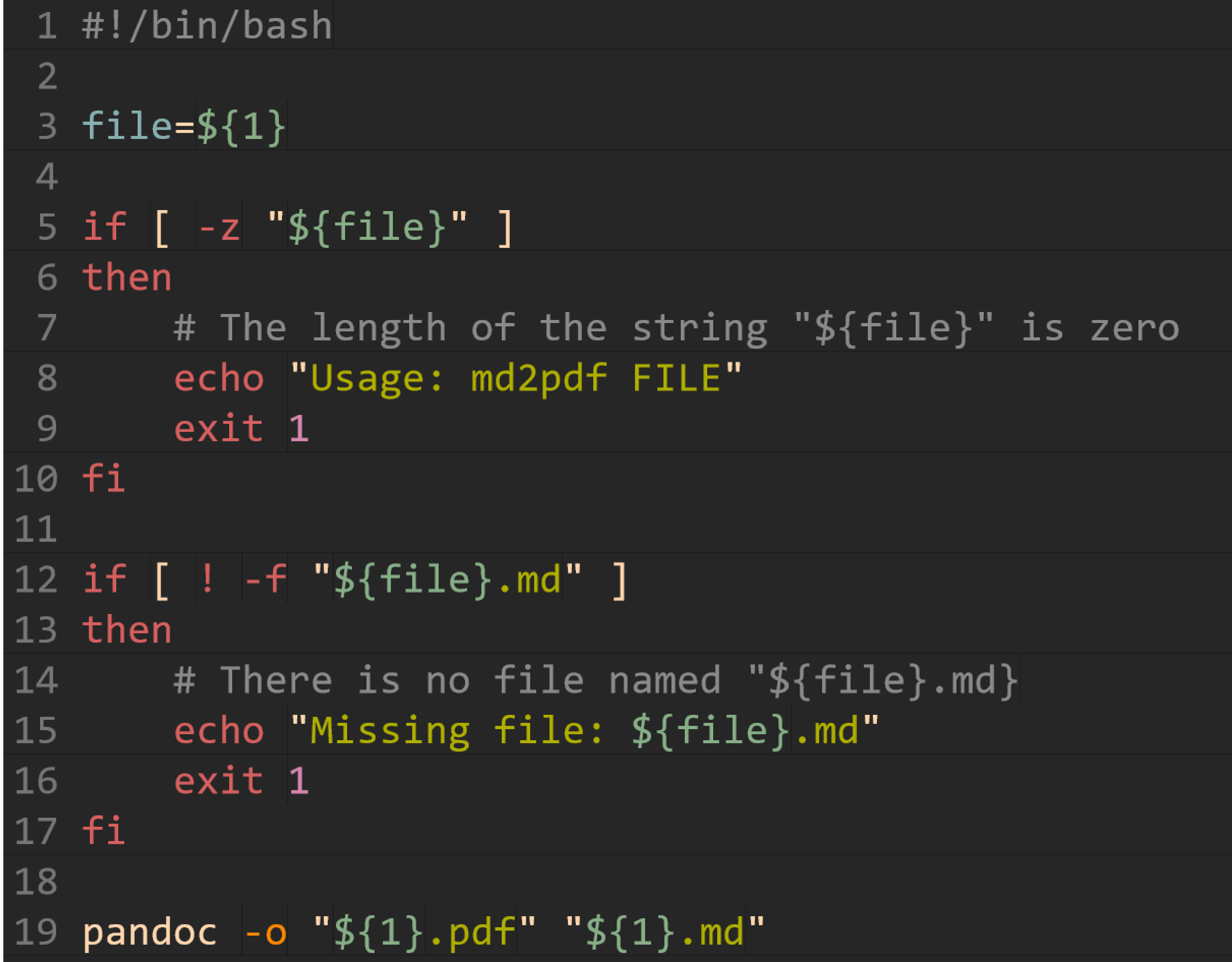
The [is a built-in alias for `test`

- Writing if statements using the test program is verbose:
  - **`if test "${foo}" = "bar"`**
- So there's a builtin named **[** that is effectively an alias for test. The only requirement is that the last argument you provide it is a].
  - **`if [ "${foo}" = "bar" ]`**
- This allows you to write more natural looking if statements
  - But be careful! You must have a space between the opening [and the first argument and the] and the last argument! This is still a *command* after all.

- The -z option tests if a string's length is zero

- Exit is a built-in (you use it to quit your session). Providing a number argument after exit sets the exit status code of the shell script itself. Good practice to exit with a non-zero status in exceptional cases.

- The -f option tests if a string is a path to a file.



```bash
#!/bin/bash

file=${1}

if [ -z "${file}" ]
then
    # The length of the string "${file}" is zero
    echo "Usage: md2pdf FILE"
    exit 1
fi

if [ ! -f "${file}.md" ]
then
    # There is no file named "${file}.md}
    echo "Missing file: ${file}.md"
    exit 1
fi

pandoc -o "${1}.pdf" "${1}.md"
```

> bash

functions & for-in loops !

Bash Functions

- Syntax for function definition:

```
funcname () {
    commands
}
```

- Inside of the function body, ${1}, ${2}, ..., ${@} are for accessing parameters
  - Parameters ***are not*** declared as part of the function definition!

- Syntax for calling a function:
  - `funcname [arg₁] [arg₂] ...`

- Notice a function call has same form as executing a program or script!
  - Functions have higher precedence than programs, so be careful!
  - Remove a function definition: **`unset -f fname`**

- Docs: https://www.gnu.org/software/bash/manual/html_node/Shell-Functions.html

for-in loops

- General syntax:

```
for <name> in <strings...>
do
    commands*
done
```

- Iterates once per **whitespace separated *string*** with ***name*** bound to a string

- When combined with *shell expansions* (next) this construct is a workhorse for looping over files and outputs of programs

- Bash has other kinds of loops, as well: until, while, for (more general)
  - Full documentation: https://www.gnu.org/software/bash/manual/html_node/Looping-Constructs.html#Looping-Constructs

> bash expansions!

Shell Expansions

- When a command is evaluated it is:
    1. Split into tokens, e.g.: foo bar "baz boz" is 3 tokens: *foo*, *bar*, *"baz boz"*
    2. Each token is *expanded,* then quotes are removed
    3. The command is *then* interpreted

- **Important**: Expansions occur *before* programs are executed and given arguments.

- You've already one kind of expansion:
    - Shell Parameter Expansion with ${var}

- Others we will explore: filename globbing, braces, command substitution.

- Full documentation: https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html

Shell Expansion: Filename Globbing

- When an *unquoted path* is expanded in the shell, special pattern matching characters cause Bash to search the filesystem for matches.
  - The pattern matching *is not* regular expression based, full docs:
    - https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html#Pattern-Matching

- Two commonly useful pattern characters:
  - * matches any string
    - For example: **echo /bin/*grep**
  - ? matches any single character
    - For example: **echo /bin/???**

- Big idea: *one pattern string* can expand to *many matched path strings*
  - Convince yourself of this with: **./for-in-demo /bin/*grep**
  - This concept is often simply referred to as "globbing"

- Useful with for-in loops to run commands over all files based on extensions like *.java

Shell Expansion: Braced Lists

- When used in a non-quoted string, curly braces with a *list* of N comma-separated strings expands into N separate strings where one string from the list is substituted in.

- Examples:
  - **b{a,o}r** expands to two strings: **bar bor**
  - **{f,b}{a,o}{r,z}** expands to 2^3 strings: **far faz for foz bar baz bor boz**

- Commonly useful when making directory structures with common prefixes:
  - Rename a file: **mv path/to/{old_name,new_name}.c**
  - Make sub-directories: **mkdir -p project-name/{src,test,bin}**
  - The above command expands to:
    mkdir -p project-name/src project-name/test project-name/bin

Shell Expansion: Command Substitution

- With command substitution, another *command* is executed in a subshell and its output is substituted.

- Form: **"$(command [arg**$_1$**] [arg**$_2$**]...)"**
  - Unlike variable substitution, parenthesis are used to surround the command.
  - Like variable substitution, safest bet is doing so inside double quoted strings.

- Examples:
  - **echo "I am $(whoami)"**
  - **some_var="The current date is $(date)"**

- Can be combined with for-in loops in powerful ways!
  - **for path in "$(find | grep 'md$'); do echo "${path}"; done**

> bash

variable scopes !

Variable Scopes

- Three levels of variable scope in shell scripts:

1. Environment - global variables inherited by child processes
   - Assigned using **export** builtin, eg: **export GIT_AUTHOR_NAME="Kris Jordan"**
     - When you run **git** it is given the environment variables of your shell
   - Other examples: PATH, EDITOR, HOME,
   - Useful for: configuration settings, API keys, production vs. development

2. Global - global variables of the current session / script
   - Default scope of variable in a script (and functions!)
   - Assigned normally, eg: global_var="Some Value"

3. Local - variables accessible only within a function
   - Specially declared inside of function using **local** built-in: **local i = 0**

Sourcing vs. Evaluating a Shell Script

- When you give the command **bash [scriptname]**
  - A separate, child bash process begins, reads [scriptname], and interprets it
  - You are evaluating the script through a separate process from your interactive command-line interface (CLI)
- When you give the command **source [scriptname]**
  - The **source** builtin evaluates [scriptname] in the *same process* as your CLI
  - It's as if you typed each line into your current shell prompt
  - The variables, functions, aliases, and so defined in [scriptname] are now available
- Demo:

```
bash /mnt/learncli/.bash_profile
echo "${GIT_AUTHOR_NAME}"  # Outputs nothing
source /mnt/learncli/.bash_profile
echo "${GIT_AUTHOR_NAME}" # Outputs value set in .bash_profile
```

Aside: Executable scripts in other languages...

- Bash is a scripting language
  - So are Python, JavaScript, Ruby, PHP, and so on
  - You can write scripts in these languages, too!

- The shebang controls which language's interpreter is used
  - JavaScript (node) and Python (python3) are installed on our container

- Example shebangs:
  - #!/usr/bin/python3
  - #!/usr/bin/node

- The code that follows can then be written in the language of the shebang's interpreter