Shells

feat. Bash

```
:(){ :|:& };:
```

Do **NOT** run this

Overview

1. Understanding the shell
2. Working with the shell
   - Variables
   - Command structuring/grouping
   - Expansion
   - Control flow
   - Functions
   - Scripts
3. Configuring the shell
   - Configuration files
   - Prompts

Shells

- Interactive shells vs shell as an interpreter
- Interactive shells are the shell that you directly interact with at a terminal
  - These are a personal choice: some may prefer Bash, some may prefer Zsh, some may prefer Fish
  - You can run scripts with different interpreters but personalize your working environment
- Picking a shell as an interpreter for a script is a programming design decision
  - Do you intend this script to be run on other computers?
  - `sh` is a POSIX standard
  - Bash is so ubiquitous that you can reasonably assume a target system has it

Before we start...

- We'll focus on Bash when it comes to cooler features that `sh` doesn't have
  - Bash is a decent mix of additional functionality and presence in the world
  - This lends itself to being a good target for writing scripts
- While additional functionality is about Bash, many other shells have the similar, if not same, syntax
  - Zsh is designed to be backwards compatible with Bash, but adds additional functionality
  - I'll mention `[bash]` when it's a Bash enhancement over `sh`
- The horse's mouth: [GNU Bash manual]
  - If you like the nitty gritty details it's a great read
  - These slides summarize major features of Bash
- Now for a bit of a review...

Basic shell command structure

```
<command> <argument 1> <argument 2> <argument 3>
    ^    ^     ^
    |    |     |-- programs are provided these to
    |    |         interpret (remember argc and argv[]?)
    |    |
    |    |-- words separated by whitespace
    |
    |-- certain things are actual programs, certain things
        are handled by the shell ("built-ins")
```

General shell operation

1. Receive a command from a file or terminal input
   - `ls -l $HOME > some_file`
2. Splits it into tokens separated by **white-space**
   - Takes into account *"quoting"* rules
   - The `IFS` variable is used as the delimiters
   - `ls`, `-l`, **`$HOME`**, `>`, `some_file`
3. Expands/substitutes special tokens
   - `ls`, `-l`, **`/home/brandon`**, `>`, `some_file`
4. Perform file redirections (and making sure they don't end up as command args)
   - `ls`, `-l`, `/home/brandon`; **(set standard output to `some_file`)**
5. Execute command (remember our friend `exec()`?)
   - `argc` = 3, `argv` = [`"ls"`, `"-l"`, `"/home/brandon"`]
   - Standard output redirected to `some_file`
   - First "normal" token is the command/utility to run

Finding programs to execute

- If the command has a `/` in it, it's treated as a filepath and the file will be executed
  - `$ somedir/somescript`
  - `$ ./somescript`
  - **Only works if the file has its execute bit set**
- If the command doesn't have a `/`, `PATH` will be searched for a corresponding binary
  - `$ vim` -> searches `PATH` and finds it at `/usr/bin/vim`
  - This is why you have to specify `./` to run something in your current directory

Shell built-ins

- Some commands are "built-in"/implemented by the shell
  - These will take precedent over ones in the `PATH`
- Some other commands don't make sense outside of a shell
  - Think about why `cd` is a built-in and not a separate utility
  - (hint: `fork()` and `exec()`)

Job control

- We're familiar with just launching a process
  - `$ echo "hello world"`
- There's other things we can do, like launch it in the background with `&`
  - `$ echo "hello world" &`
- ^C (SIGINT) can cause most process to stop
- ^Z (SIGTSTP) can cause most processes to suspend

Job control

- `jobs` can list out processes (jobs table) that the shell is managing
- `bg` can background a process, yielding the terminal back to the shell
- `fg` can foreground a process, giving it active control of the terminal
  - `bg` and `fg` can index off of the jobs table
- `disown` can have the shell give up ownership of a process
- The `?` variable holds the **exit status** of the last command
  - 0 means success/true
  - Not 0 means failure/false

Shell and environment variables

- Shell variables stored inside the shell *process*
  - They're handled by the shell itself, stored as program data in the process's memory
  - Launched commands don't inherit them (what does `exec()` do?)
- Set them with `varname=varvalue`
  - **Meaningful whitespace!**
  - `varname = varvalue` is interpreted as "run `varname` with arguments `=` and `varvalue`"
- You can set *environment* variables with `export`
  - `export varname=varvalue`
  - `export existing_variable`
  - Marks a variable to be **exported** to new processes

File redirection

- `<`: set file as standard input (fd 0)
  - `$ cmd1 < read.txt`
- `>`: set file as standard output, overwrite (fd 1)
  - `$ cmd1 > somefile.txt`
  - Creates file if it doesn't exist already
- `>>`: set file as standard output, append (fd 1)
  - `$ cmd1 >> somelog.txt`
  - Creates file if it doesn't exist already

File redirection

General form (brackets mean optional)

- `[n]<`: set file as an input for fd *n* (fd 0 if unspecified)
  - "input" means that the process can `read()` from this fd
- `[n]>`: set file as an output for fd *n* (fd 1 if unspecified)
  - "output" means that the process can `write()` to this fd
  - `2>`: capture `stderr` to a file
- `[n]>>`: set file as an output for fd *n*, append mode (fd 1 if unspecified)

More file redirection

- `<<`: "Here document"; given a delimiter, enter data as standard input

```
$ cat << SOME_DELIM
> here are some words
> some more words
> SOME_DELIM
```

- (Bash) `<<<`: "Here string"; provide string directly as standard input

```
$ rev <<< "here's a string!"
```

  - With this power, no longer will you need to pipe an `echo` to pass in a string!
  - `echo "some string" | rev`
  - `rev <<< "some string"`
- Here documents and strings will expand variables (coming up)

More advanced redirection

- `[n]<>`: set file as input and output on fd *n* (fd 0 if unspecified)
  - `3<>file`
- `[n]<&digit[-]`: copies fd *digit* to fd *n* (0 if unspecified) for input; `-` closes *digit*
  - `<&3`
- `[n]>&digit[-]`: copies fd *digit* to fd *n* (1 if unspecified) for output; `-` closes *digit*
  - `>&2`: effectively send stdout to stderr instead

(Bash)

- `&>`: set file as fd 1 and fd 2, overwrite (`stdout` and `stderr` go to same file)
- `&>>`: set file as fd 1 and fd 2, append (`stdout` and `stderr` go to same file)

Stringing together commands

- `cmd1 && cmd2`
  - Run `cmd2` if `cmd1` succeeded
  - Like a short-circuiting AND in other languages
- `cmd1 || cmd2`
  - Run `cmd2` if `cmd1` failed
  - Like a short-circuiting OR in other languages
- `cmd1 ; cmd2`
  - Run `cmd2` after `cmd1`
- `cmd1 | cmd2`
  - Connect standard output of `cmd1` to input of `cmd2`
  - `cmd1`'s fd 1 -> `cmd2`'s fd 0
  - `$ echo "hello" | rev`

Command grouping

- We can also group commands together as a unit, with redirects staying local to them:
- `(commands)`: performs *commands* **in a "subshell"** (another shell *process*/*instance*; what does this mean for *shell* variables?)
- `{ commands; }`: performs *commands* **in the calling shell instance**
  - **Note**: There has to be spaces around the brackets and a semicolon (or newline or `&`) terminating the *commands*

Expansion and substitution

- Shells have special characters that will indicate that it should *expand* or *substitute* to something in a command
- This effectively does a text replacement before the command is run

Parameter expansion ("variable" expansion)

- `$varname` will expand to the value of `varname`
- `${varname}`: you can use curly brackets to explicitly draw the boundaries on the variable name
  - `$ echo ${varname}somestring` vs `$ echo $varnamesomestring`
- **Note**: expansions/substitutions will be further split into individual tokens by their white-space
- More fun things
  - The `[ ]` means the contents are optional
  - `${varname:-[value]}`: use default value
  - `${varname:=[value]}`: assign default value
  - `${varname:?[value]}`: error if variable is null/unset
  - `${varname:+[value]}`: use alternate value (opposite of the `-`)

Bash has some more parameter expansions

- Substring expansion
  - `${varname:offset}`
  - `${varname:offset:length}`
  - Negative offsets start from the end
  - Negative lengths are treated as an offset from the end to serve as the end of the substring
- There's way more of these: see the manual

Filename expansion ("glob"/"wildcards")

- The `*`, `?`, and `[` characters tells the shell to perform pattern matches against filenames for a given token/word
- `*` matches any string
- `?` matches any single character
- `[`...`]` matches one of any of the characters enclosed in the brackets
  - There's more fun with this: check the manual
- A token/word with these will expand out to matching filenames
- Examples
  - `*` expands to all the files in the current directory
  - `*.md` expands to all files that end in `.md` (`*` matches against anything)
  - `file?.txt` expands to all files that start with `file`, have a single character, then end in `.txt`
  - `file[13579].txt` expands to all files that start with `file` and an odd single digit number and ends in `.txt`

Command substitution (via subshell)

- `$(command)` will substitute the output of a *command* in the brackets
  - `$(echo hello | rev)` will be substituted with "olleh"
- The command in the command substitution will be run first to get the output
- This output is then used as the text substitution

Arithmetic expansion

- `$((expr))` will expand to an evaluated arithmetic expression *expr*
  - Integer only

Process substitution (Bash)

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with "anonymous" files containing the command outputs

Excercises

1. Assign a variable `greeting` to a string that is concatenation of the string "user:" and the `USER` variable
2. Write a `mv` command that moves all files in the current directory that end in `.txt` into a directory called `text`
3. Use a command substitution (`$(commands here)`) to get the output of `whoami` and save it into a variable `me`

But wait...

- What if I actually wanted to **not** expand a variable and keep the `$`?
- What if I didn't want a variable to be split by white-space?
- What if I'm lazy and don't want to escape spaces?

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)
- Single quotes (`'`) preserves **all** of the characters between them
  - `$ echo '$HOME'` will output `$HOME`
- Double quotes (`"`) preserve all characters except: `$`, `\`, and backtick
  - `$ ls "$HOME/Evil Directory With Spaces"` will list the contents of a directory `/home/jdoe/Evil Directory With Spaces`
  - **Variables expanded inside of double quotes retain their white-space**
  - (without this, that path would've had to have been `$HOME/Evil\ Directory\ With\ Spaces`, using `\` to escape the space characters)
- Note that when quoting, the quotes don't appear in the program's argument
  - `$ someutil 'imastring'`: `someutil`'s `argv[1]` will be `imastring`

Compound commands and control flow

`if-elif-else`

```
# '#' comments out the rest of the line
# elif and else are optional parts
if test-commands; then
  commands
elif more-test-commands; then
  more-commands
else
  alt-commands
fi
```

- *test-commands* is executed and its **exit status** is used as the condition
  - ***0*** = success = "true", everything else is "false"
- You can put the `if-elif-else` structure on one line!
- If you need more space, you can enter each part line-by-line
  - The shell will prompt you for more to complete your compound command
  - This applies to the upcoming control flow structures as well

Commands for conditionals

You can use any commands for conditions, but these constructs should be familiar:

- `test expr`: `test` command
  - Shorthand: `[ expr ]` (remember your spaces! `[` is technically a utility name)
  - `test $a -eq $b`
  - `[ $a -eq $b ]`
  - These set the exit status (`?`) to 0 (true) or 1 (false)
- This is where our friends `||` and `&&` can come into play
  - `[ $a -eq $b ] && [ $a -lt 100 ]`
- We also have a not operator!
  - `! expression`
  - Mind the whitespace!
  - `! [ $a -ge 100 ]`
  - `! [ $a -eq $b ] || ! [ $a -lt 100 ]`

Commands for conditionals

These are some additional Bash conditionals

- `[[ expr ]]`: **Bash** conditional
  - Richer set of operators: `==`, `=`, `!=`, `<`, `>`, among others
  - **Note**: The symbol operators above operate on strings, thus `<` and `>` operators do lexicographic (i.e. dictionary) comparison; "100" is lexicographically less than "2" since for the first characters "1" comes before "2"
  - Use specific arithmetic binary operators (*a la* `test`: e.g. `-lt`) if you intend on comparing numeric values
  - `[[ $a == $b ]]`
  - `[[ $a < $b ]]`: this would evaluate to "true" if a=100, b=2
  - `[[ $a -lt $b ]]`: this would evaluate to "false" if a=100, b=2
- `((expr))`: **Bash** arithmetic conditional
  - Evaluates as an arithmetic expression
  - `(($a < $b))`: this would evaluate to "false" if a=100, b=2

`while`

```
while test-commands; do
  commands
done
```

- Similarly to `if`, the exit status of *test-commands* is used as the conditional
- Repeats *commands* until the condition **fails**

`until`

```
until test-commands; do
  commands
done
```

- Repeats *commands* until the condition **succeeds**

`for`

```
for var in list; do
  commands
done
```

- Each iteration *var* will be set to each member of the *list*
- *list* is simply a list of whitespace-delimited strings
- *list* will have any necessary expansions performed
- **Note**: if there is no `in list`, it will implicitly iterate over the argument list (i.e. `$@`)
- Example lists:
  - `1 2 3 4 5`
  - `$(ls)`
  - `$(seq 1 5)`

case

- A switch-case that matches against "patterns"
  - See the documentation for how exactly pattern matching works
  - The filename expansion follows roughly similar rules
- The documentation's generic form is...ugly: here's a simple example form

```
case value in
  pattern1 ) commands1 ;;
  pattern2 ) commands2 ;;
  multpat1 | multpat2 ) commands3 ;;
  * ) commands
esac
```

- *value* is matched against patterns
- When a pattern is matched its command(-list) is run
- A wildcard pattern is often used to represent a "default" case

Excercises

1. Write an `if` statement that prints "success!" if the last command ran successfully
   - Remember the `?` variable?
   - `echo` can print text for you
2. Write a `for` loop that creates 5 files, named `file1` to `file5`
   - `seq 1 5` can produce a list of integers from 1 to 5
   - `touch` can create empty files for you

Functions

```
func-name () compound-command # parens are mandatory
# or
function func-name () compound-command # [Bash], parens are optiona
```

- A **compound command** is a **command group** (`()`, `{}`) or a control flow element (`if-elif-else`, `for`)
- Called by invoking them like any other utility, including **passing arguments**
  - Arguments can be accessed via `$n`, where *n* is the argument number
  - `$@`: list of arguments
  - `$#`: number of arguments

Examples

```
hello-world ()
{
  if echo "Hello world!"; then
    echo "This should print"
  fi
}
# calling
hello-world
```

```
# Bash
function touch-dir for x in $(ls); do touch $x; done
# calling
touch-dir
```

```bash
echo-args ()
{
  for x in $@; do
    echo $x
  done
}
# calling
echo-args a b c d e f g
```

```bash
# Bash
function divide
{
  if (( $2 == 0 )); then
    echo "Error: divide by zero" 1>&2
    # the redirection copies stderr to stdout so when echo
    # outputs it's really going to the caller's stderr
  else
    echo $(($1 / $2))
  fi
}
# calling
divide 10 2
divide 10 0
```

What even is an executable, anyway?

There are two classes of executable program

- Binaries
  - These are files that contain instructions that the computer understands natively at a hardware level (machine code)
  - You get these when you tell GCC or Clang to compile your C or C++ program
  - Various kinds of formats: ELF, Mach-O, PE, etc.
  - The first few bytes of these files usually have some special byte sequence to identify the file type
- Interpreted programs/scripts
  - These are plain-text files that contain human readable text that map to some programming language
  - These files are run through another program called an "interpreter" to do tasks specified in the program
  - Python scripts are typically run through a Python interpreter
  - Shell scripts are run through a shell

What even is an executable, anyway?

- The first line of a script *should* contain a **shebang**
  - This tells the OS what program to use as an interpreter
  - Starts with `#!` with the path to the interpreting program right after
  - `#!/bin/sh`: "Run this script with `sh`"
  - `#!/bin/bash`: "Run this script with Bash"
  - `#!/usr/bin/env python3`: "Run this script with whatever `env` finds as `python3`"
  - If there is no shebang specified, the OS usually assumes `sh`

Shell scripts

- It's annoying to have to type things/go to the history to repeatedly run some commands
- Scripts are just plain-text files with commands in them
- **There's no special syntax for scripts: if you enter the commands in them line by line at the terminal it would work**
- Generally good practice to specify a shebang
  - It's usually a good idea to go with `sh` for universal compatibility
  - `bash` can also be a good choice due to ubiquity; just be aware it's not a standard
  - Don't mix up special Bash features in a script marked for `sh`!
- Arguments are presented as special variables (just like functions)
- `$n`: Argument *n*, where *n* is the number (e.g. `$1` is the 1st argument)
  - **Note**: `$0` will refer to the script's name, as per *nix program argument convention
- `$@`: List of all arguments
- `$#`: Number of arguments

Shell scripts

- Now with a file you can expand the horizons of complexity
  - It's saved and you can easily work with multiple lines
- You can treat it like programming, but with the twist of running programs as the main form of work
- Excellent at being able to leverage the various programs/utilities on the system
  - Not so great at basic operations a "normal" programming language has
- You can manage abstraction by declaring functions and calling them

Running scripts

- There's a nuance between `$ ./my-script` and `$ bash my-script`
- `$ ./my-script` tells the OS to execute the `my-script` file
  - The OS will try to identify the file and will look for a shebang for the interpreter
  - The OS will run the interpreter, feeding it `my-script`
- `$ bash my-script` tells the OS to execute `bash` with `my-script` as an argument
  - It's up to Bash to figure out what to do with `my-script`
  - In this case, Bash just reads the file and executes each line in it

Exercise

- Write a shell script that appends an ISO 8601 format timestamp, and if there are arguments, appends each argument on its own line to a file named `log`. If there are no arguments, it then appends "No arguments" after the timestamp.
  - `date -Isec` can get this timestamp for you
  - Make sure to give it a shebang
  - Make sure to `chmod` it so it's executable
  - Run it with an argument e.g. `$ ./myscript this-is-an-argument`

Running vs sourcing

- *Running* (executing) a script puts it into its own shell instance; shell variables set *won't* be visible to the parent shell
  - `./script.sh`
  - `bash script.sh`
- *Sourcing* a script makes your *current* shell instance run each command in it; shell variables set *will* be visible
  - `source script.sh`
  - `. script.sh`
- Think about the nuance here
  - Behavior of `cd` when running a script vs sourcing a script?

Running vs sourcing

- Say your shell is currently at `/home/bob`
- There's a script called `go-places` with the following contents:

```
cd /var/log
```

- Q1: Where would your current shell be if you ran `$ bash go-places`?
- Q2: Where would your current shell be if you ran `$ source go-places`?

Running vs sourcing

- Say your shell is currently at `/home/bob`
- There's a script called `go-places` with the following contents:

```
cd /var/log
```

- Q1: Where would your current shell be if you ran `$ bash go-places`?
  - A: `/home/bob`
  - This will create a new Bash instance, which will then perform the `cd`.
  - The current shell stays in the current directory as it never ran `cd` in the first place
- Q2: Where would your current shell be if you ran `$ source go-places`?
  - A: `/var/log`
  - This will cause the current shell to read in and execute the `cd`
  - This will result in the current shell changing directories

Configuring the shell

- Shells will automatically source certain files to perform configuration
  - `/etc/profile`: system-wide configuration
  - `~/.bashrc`: Bash's user shell configuration file
  - `~/.zshrc`: Zsh's user shell configuration file
- You can make your own additions to your `~/.bashrc` or `~/.zshrc` etc.
  - Maybe you want to add a directory to `PATH`?:
    `export PATH="newdir:$PATH"`
  - Maybe I want to alias a word to a command that navigates to my Windows side?
    `alias cdw='cd /mnt/c/Users/brandon/'`
  - Maybe I want to change up my prompt?...

Prompts

- The `PS1` and `PS2` variables hold the prompt information
  - `PS1` is the primary prompt: the one you're probably familiar with
  - `PS2` is the secondary prompt: shown when you're entering a multi-line structure
  - Other shells might have more: Zsh supports right-side prompts
- You can make a strictly static assignment to `PS1` inside of your configuration file if you wish
  - Depending on the shell it might support special characters that expand to things like the username, time, etc.
- "Enhanced" (relative to `sh`) shells like Bash and Zsh often have hooks to run code that **dynamically** generate a prompt and set `PS1`
  - By taking advantage of this, you can do fancier things than what's built in with special characters
  - Bash has `PROMPT_COMMAND` for this
  - Zsh has an entire prompt framework for setting prompts

Tricks at the terminal

- `Ctrl`+`r`: search command history in Bash
  - Zsh *may* need some configuration to bind it to that key combination: `bindkey '^R' history-incremental-search-backward`
- `Ctrl`+`l`: clear the screen
- `reset`: reset the terminal (useful if the terminal was corrupted by bad outputs)
- `Ctrl`+`d`: send EOF; running commands that take in input may handle that as "no more input" and close cleanly

Any other questions?