# EECS 2031

## Software Tools

Module 2 – Shell scripts

YORK UNIVERSITÉ UNIVERSITY

# What Is a Shell?

- A program that interprets your requests to run other programs
- Most common Unix shells:
  - Bourne shell (**`sh`**)
  - C shell (**`csh`** - **`tcsh`**)
  - Korn shell (**`ksh`**)
  - Bourne-again shell (**`bash`**)
- In this course we focus on Bourne shell (**`sh`**)

# A note of caution

- The default shell in your EECS account is not the Bourne shell
- For all the examples in this set of slides to work interactively, first run **`sh`**
- Many things are similar between the different kinds of shell, but there are important differences

# The Bourne Shell

- A high level programming language
- Processes groups of commands stored in files called scripts
- Includes
  - Variables
  - Control structures

# Shell scripts

- Text files that contain one or more shell commands
- `#` indicates a comment
  - Except on line 1 when followed by `!`

```
% cat welcome
#!/bin/sh
echo 'Hello World!'
```

# Shell scripts must be executable

```
% welcome
welcome: Permission denied.
% chmod 744 welcome
% ls -l welcome
-rwxr--r-- 1 bil …
% welcome
Hello World!
```

# Just like a command

```
% welcome > greet_them

% cat greet_them

Hello World!
```

# Shell Script Variables

- All shell variables store strings
- **There are no numeric values in a shell script!**
- There are five possible types of shell script variables…

# 1. Command-line arguments

- **`$0`** is the name of the script
- **`$1`** is the first command-line argument
- **`$2`** is the second command-line argument
- …
- **`$#`** is the number of arguments
- See scripts **`showargs`** and **`chex`**

# 2. Process-related variables

- `$?` is the exit status of the last command
- 0 – successful execution
- Non-zero – Something went wrong
- See script **`igrep`**

# Redirection tricks

- Want to run a command to check its exit status and ignore the output?

```
diff f1 f2 > /dev/null
```

- Want to redirect standard error and standard output?

```
diff f1 f2 >& /dev/null
```

# 3. Environment variables

- Contain information about the system
- Available in all shells
- Examples: **`USER`**, **`HOME`**, **`PATH`**
- To display your environment variables, type **`printenv`**

# 4. Shell Variables

- Used to tailor the current shell
- Examples: **`cwd`**, **`prompt`**
- To display your shell variables, type **`set`**

# 5. User Variables

- Variable name: combination of letters, numbers, and underscore character ( _ ) that do not start with a number
- Avoid existing commands and shell/ environment variables
- Assignment: **`name=value`**
- **No space around the equal sign!**

# User Variables

- To use a variable: **`$varname`**
- Operator **`$`** tells the shell to substitute the value of the variable name
- See script **`ma`**

# if Statement

```
if condition
then
    command(s)
elif condition_2
then
    command(s)
else
    command(s)
fi
```

**then** and **else** need to be on a separate line!!

# Conditions

- A condition in a shell script is designated in one of two equivalent ways:

1. Using the **`test`** command

```
test $name = "bil"
```

2. Using the square bracket notation

```
[ $name = "bil" ]
```

**Spaces are important!**

# File conditions

| | |
|---|---|
| `-e arg` | True if file arg exists |
| `-f arg` | True if arg is an ordinary file |
| `-d arg` | True if arg is a directory |
| `! -d arg` | True if arg is not a directory |

# File conditions

| | |
|---|---|
| `-r arg` | True if arg is readable |
| `-w arg` | True if arg is writable |
| `-x arg` | True if arg is executable |
| `-s arg` | True if size of arg is larger than 0 |

# Numeric conditions

| Condition | Java Equivalent |
|---|---|
| `n1 -eq n2` | `n1 == n2` |
| `n1 -lt n2` | `n1 < n2` |
| `n1 -gt n2` | `n1 > n2` |
| `n1 -le n2` | `n1 <= n2` |
| `n1 -ne n2` | `n1 != n2` |
| `n1 -ge n2` | `n1 >= n2` |

# `if - then - else` scripts

- `if_else`
- `check_file`
- `check_file2`
- `chkex`
- `chkex2`

# case Statement

```
case variable in
pattern1) command(s);;
pattern2 | pattern3) command(s);;
…
patternN) command(s);;
*)        command(s);; #otherwise
esac
```

- Patterns can contain wildcards
- See script **caseex**

# `for` loops

```
for variable in list
do
    command(s)
done
```

- **`variable`** is a user variable
- *`list`* is a sequence of strings separated by spaces

# `for` scripts

- `fingr`
  - `$*` stands for all command-line arguments
- `fsize`
- `makeallex`

# **while** loops

```
while condition
do
    command(s)
done
```

- See script **whileex**

# `until` loops

```
until condition
do
    command(s)
done
```

- See script **`grocery`**

# **`break`** and **`continue`**

- Interrupt loops (**`for`**, **`while`**, **`until`**)
- **`break`** jumps to the statement after the nearest **`done`** statement
  - terminates execution of the current loop
- **`continue`** jumps to the nearest **`done`** statement
  - brings execution back to the top of the loop
- See script **`breakex`**

# Reading User Input

- Syntax: **`read varname`**
  - No dollar sign
- Reads from standard input
- Waits for the user to enter something followed by <RETURN>
- Stores what is read in user variable
- To use the input: **`echo $varname`**
- See scripts **`greeting`**, **`doit`**

# Reading User Input

- More than one variable may be specified
- Each word will be stored in separate variable
- If not enough variables for words, the last variable stores the rest of the line
- See script **`read3`**
  - Note use of **`printf`** instead of **`echo`**

# More on command-line arguments

- **`$1, $2, …`** normally store command line arguments.
- Their values can be changed using the **`set`** command

```
set newarg1 newarg2 …
```

- See script **`setparam`**

# Shifting arguments

- To parse command-line arguments one can also use the **`shift`** operator
- Shifts contents of **`$2`** into **`$1`**, **`$3`** into **`$2`** …
- Eliminates argument that used to be in **`$1`**
- After a shift, the argument count stored in **`$#`** is automatically decreased by one
- Allows access to 10th argument and beyond
- See script **`shiftex`**, **`my_copy`**

# All Command Line Arguments

- Both `$*` and `$@` get substituted by all the command line arguments
- They are different when double-quoted
  - `"$@"` expands such that each argument is quoted as a separate string
  - `"$*"` expands such that all arguments are quoted as a single string
- See script **`displayargs`**

# Quoting issues

- What if I want to output a dollar sign?
- Two ways to prevent variable substitution:

```
echo '$dir'

echo \$dir
```

- Note: `echo "$dir"` is the same as `echo $dir`

# User Variables and Quotes

- If value contains no space, no need to use quotes: **`dir=/usr/include/`**
- Unless you want to protect the literal **`$`**
- See script **`quotes`**

# User Variables and Quotes

- If value contains one or more spaces:
- Use single quotes for NO variable substitution
  - A dollar sign is a dollar sign
- Use double quotes for variable substitution
  - A dollar sign followed by a variable name will be substituted by the variable value

# Back quotes

- Enclosing a command invocation in back quotes (the character usually to the left of 1) results in the whole invocation substituted by the output of the command

```
% dateVar=`date`

% echo $dateVar

Mon 16 Sep 2019 10:29:26 EDT
```

- See scripts **quotes2**, **twodirs**

# Arithmetic Operations Using `expr`

- The shell is not intended for numerical work
- However, the **`expr`** utility may be used for simple arithmetic operations on integers

```
sum=`expr $1 + $2`
```

- Note: spaces are required around the operator **+** (but not allowed around the equal sign)
- See script **`cntx`**

# Shell Script Functions

- Syntax:

```
function_name()
{
   command(s)
}
```

- Allows for structured shell scripts
- See script **`funcex`**