# COMP20200 Unix Programming

## Lecture 21

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

School of Computer Science, University College Dublin, Ireland

24/04/2023

# Lecture 21 - Overview

- Functions
- Arrays
- Bash wildcards
- Job control

# Bash Functions

- Shell functions are a way to group commands using a single name for the group.
- **Functional decomposition**: decompose a problem into high-level functions, decompose each of the high-level functions into lower-level functions, and so on.

# Functional Decomposition: Properties

- **Abstraction**
  - We can focus on individual building blocks rather than the whole structure.
- **Modularity**
  - Wrote a handy, generalized function? Use it in your other scripts!
- **Readability**
  - Smaller code blocks are easier to wrap your mind around.
- **Provability**
  - Decompose a problem into a set of provable functions.
  - Prove the incorrectness of a function using tests.

# Writing Functions Well

- Keep functions small.
- They should do one thing. They should do it well. They should do it only.
- **Stepdown Rule:** The statements within a function are all at the same level of abstraction. Supports top-down narrative of your program.

# Functions in Bash

There are two ways to define functions:

```
function name {
    commands
    return
}
```

```
name () {
    commands
    return
}
```

- Spaces between curly braces and commands are required!
- Shell function definitions must appear in the script before they are called.

# Function Parameters

Passing parameters to a function is similar to shell script parameters.

**example.sh:**

```
#!/bin/bash
echo $1
function func {
  echo $1
}
func "hello function"

$ chmod u+x ./example.sh && ./example.sh "hello script"
hello script
hello function
```

# Return Parameters

- The variable "$?" contains the return status of a function.
- The keyword **return** is used to return the status.
- If the keyword is not specified and the function executes successfully, "$?" will be 0.

**Example:**

```
#!/bin/bash
function func {
    return 1
}
func
echo "Return code is $?"
```

**Output:** Return code is 1

# Script Variables: Global and Function Scopes

- The scope of a variable is the part of the program where it is visible.
- Scope rules define the visibility rules for names in a programming language.

# Static and Dynamic Scopes

- When the scope rules depend only on the syntactic structure of the program, the language has **static scope**.
  - Java, C, C++
- When the scope rules depend on the control flow at runtime, the language has **dynamic scope.**
- Also called the rule of most recent association.
  - Lisp, Perl, Bash

# Static Scope

```
x=1
f () {
  print x
}

g () {
   x = 100
   f()
}

f()
g()
```

**Output:** 1 1

# Static Scope

- Static scope allows the determination of all the environments present in a program simply by reading its text.
- Due to this, programmers have a better understanding of their program.
- They can connect every occurrence of a name to its correct declaration by observing the textual structure of the program and without worrying about runtime behavior.
- It helps the compiler to perform correctness tests and code optimisations.

# Dynamic Scope

```
x=1
f () {
  print x
}

g () {
  x = 100
  f()
}

f()
g()
```

**Output:** 1 100

# Dynamic Scope

- Dynamic scope makes programs difficult to read.
- And therefore is largely absent from the modern languages.

# Scoping in Bash

- Bash uses dynamic scoping.
- By default, variables are declared globally, meaning that they can be accessed and modified from anywhere in the script.
- Local variables are defined only for the context in which it was created.

# Global Scope and Local Scope

```
#!/bin/bash
VAR="global variable"
function func {
    local VAR="local variable"
    echo $VAR
}
func
echo $VAR
```

**Output:**

local variable

global variable

# Arrays

# Defining Arrays

An array is a variable containing multiple values.

There are three different ways to create an array:

- `declare -a arrayname`
  - Explicit declaration, empty until modified
- `arrayname[index number]=value`
  - Puts value in the specified position of a new array
- `arrayname=(value1 value2 ... valueN)`
  - Creates an array using the given values, indexed sequentially

Multi-dimensional arrays are not supported in *bash*.

# Accessing Arrays

Once created, can access individual elements as follows:

```
${arrayname[index]}
```

```
shell> array=('Unix Programming' COMP20200 'Lecture 15')

shell> echo ${array[0]}
Unix Programming
shell> echo ${array[1]}
COMP20200
shell> echo ${array[2]}
Lecture 15
```

# Outputing Array Contents

The special indices '@' and '*' reference all members of an array.

```
ipc=("sockets" "pipes" "shared memory" "signals")
for i in ${ipc[*]}; do echo $i; done
for i in ${ipc[@]}; do echo $i; done
for i in "${ipc[*]}"; do echo $i; done
for i in "${ipc[@]}"; do echo $i; done
```

In the first two cases, when unquoted, they print the same contents as below:

```
sockets
pipes
shared
memory
signals
```

# Outputing Array Contents

```
ipc=("sockets" "pipes" "shared memory" "signals")
for i in "${ipc[*]}"; do echo $i; done
```

The * notation prints a single word containing all the array's contents.

```
sockets pipes shared memory signals
```

The @ notation matches and prints the array's real contents.

```
ipc=("sockets" "pipes" "shared memory" "signals")
for i in "${ipc[@]}"; do echo $i; done
```

# Deleting an array

```
shell> vowels=(a e i o u)
shell> echo ${vowels[@]}
a e i o u
shell> unset vowels
shell> echo ${vowels[@]}
<no output>

shell> vowels=(a e i o u)
shell> echo ${vowels[@]}
a e i o u
shell> unset 'vowels[3]'
shell> echo ${vowels[@]}
a e i u
```

# Bash wildcards and regular expressions

# Bash Wildcards

Wildcards can be classified into two categories:

- Standard wildcards or Globbing patterns
- Regular expressions

# Globbing Patterns and Regular Expressions

Globbing patterns are defined as follows:

- Globbing is the operation that expands a wildcard pattern into the list of pathnames matching the pattern
- Globbing patterns are used by command-line utilities to work with multiple files

Regular expression (**regex**) can be defined as follows:

- A type of globbing pattern used when working with text
- A special text string for describing a search pattern

# Bash Wildcards

- * zero or more characters
- ? exactly one character
- `[abcde]` exactly one character listed
- `[a-e]` exactly one character in the given range
- `[!abcde]` any character that is not listed
- `[!a-e]` any character that is not in the given range
- {debian,linux} exactly one entire word in the options given

# Bash Wildcards - Examples

Consider you have a directory:

```
$ ls work
1.out   2.out   a.txt   b.txt   c.txt
vars2.sh   vars3.sh   vars.sh
```

- Return all files starting with "vars":

```
$ ls vars*
vars2.sh   vars3.sh   vars.sh
```

- Following commands will return nothing:

```
$ ls [0-9]??.out
ls: cannot access '[0-9]??.out': No such file or
    directory
$ ls [0-9]?.out
ls: cannot access '[0-9]?.out': No such file or
    directory
```

# Bash Wildcards - More Examples

Consider you have a directory:

```
$ ls work
1.out   2.out   a.txt   b.txt   c.txt
vars2.sh   vars3.sh   vars.sh
```

- Return all files ending with extensions “txt”, “out”:

  ```
  $ ls *.{out,txt}
  1.out   2.out   a.txt   b.txt   c.txt
  ```

- Return all files not ending with extensions “txt”, “out”:
  **Not achievable using wildcards (try regex)**

# Bash Wildcards - Caveats

- Depending what you want to remove the following may be useful:

```
rm [0-9]??.txt
```

- But if you are not sure you could try first:

```
ls [0-9]??.txt
```

- Be careful, **NEVER** do anything like:

```
rm ~/${DOCS}/*
```

*what if DOCS is not set?*

# Bash Wildcards and Regular Expressions

Consider you have a directory:

```
$ ls work
a.txt  b.txt  c.txt h.txt w.txt

#c.txt
hello world

# Shell expands w* into filenames matching the pattern
$ grep w* c.txt
<no output>

$ grep "w*" c.txt
hello world
```

# Job control

# What is Job Control?

- Ability to selectively stop (suspend) the execution of processes.
- And continue (resume) their execution at a later point.
- Shell keeps a table of currently executing jobs.
- The list of jobs can be viewed using **jobs** command.

# Job Control

Start a job asynchronously:

```
$ sleep 10 &
[1] 4991
```

- 4991 is the PID
- 1 is the job number

List the jobs:

```
$ jobs
[1]+  Running                 sleep 10 &
<after about 10 seconds>
[1]+  Done                    sleep 10
```

# Job Control - fg and bg

- The **fg** command switches a job running in the background into the foreground.
- The **bg** command restarts a suspended job, and runs it in the background.

```
$ sleep 30 &
[1] 5109
$ fg
sleep 30
^Z
[1]+  Stopped                 sleep 30
$ bg
[1]+ sleep 30 &
<after about 30 seconds>
[1]+  Done                    sleep 30
```

# Asynchronous execution - wait command

- Suspend script execution until all jobs running in background have terminated
- Until the job number or process ID specified as an option terminates
- Facilitates parallel execution of programs

waitex.sh

```
---------
#!/bin/bash

./program1 &
PID1=$!
./program2 &
PID2=$!

wait $PID1
echo $? # Exit status of program 1
wait $PID2
echo $? # Exit status of program 2
```

# Bash Scripting - Review

- Setting variables
- Executing commands in scripts
- “test” command
- Conditionals (if/then/elif/else/fi)
- Command-line arguments
- Environment variables
- Loops (while, for, until)
- Integer Arithmetic
- Functions
- Arrays
- Bash wildcards
- Job control

# To follow... Course conclusion

Q & A