CS2211b

Software Tools and Systems Programming

Week 5b

Shell Scripts: Part 2

Week 4b Review

Exit Status

- The exit status of a program is represented as an integer between 0 and 255.
- An **exit status of 0** indicates that the program exited **successfully**.
- An **exit status of 1 to 255** indicates that some kind of **failure** occurred. The value is normally some kind of error code to give us a clue as to what went wrong.

```
[dservos5@cs2211b ~]$ ls badfilename
ls: cannot access badfilename: No such file or directory
[dservos5@cs2211b ~]$ echo $?
2
```

Week 4b Review

Exit Status

- The exit status of a program is represented as an integer between 0 and 255.
- An **exit status of 0** indicates that the program exited **successfully**.
- An **exit status of 1 to 255** indicates that some kind of **failure** occurred. The value is normally some kind of error code to give us a clue as to what went wrong.

```
[dservos5@cs2211b ~]$ grep 'duck' textbook.txt
[dservos5@cs2211b ~]$ echo $?
1
```

Week 4b Review

Exit Status

- The exit status of a program is represented as an integer between 0 and 255.
- An **exit status of 0** indicates that the program exited **successfully**.
- An **exit status of 1 to 255** indicates that some kind of **failure** occurred. The value is normally some kind of error code to give us a clue as to what went wrong.

```
[dservos5@cs2211b ~]$ grep 'programmers' textbook.txt
22 reading for systems programmers.
[dservos5@cs2211b ~]$ echo $?
0
```

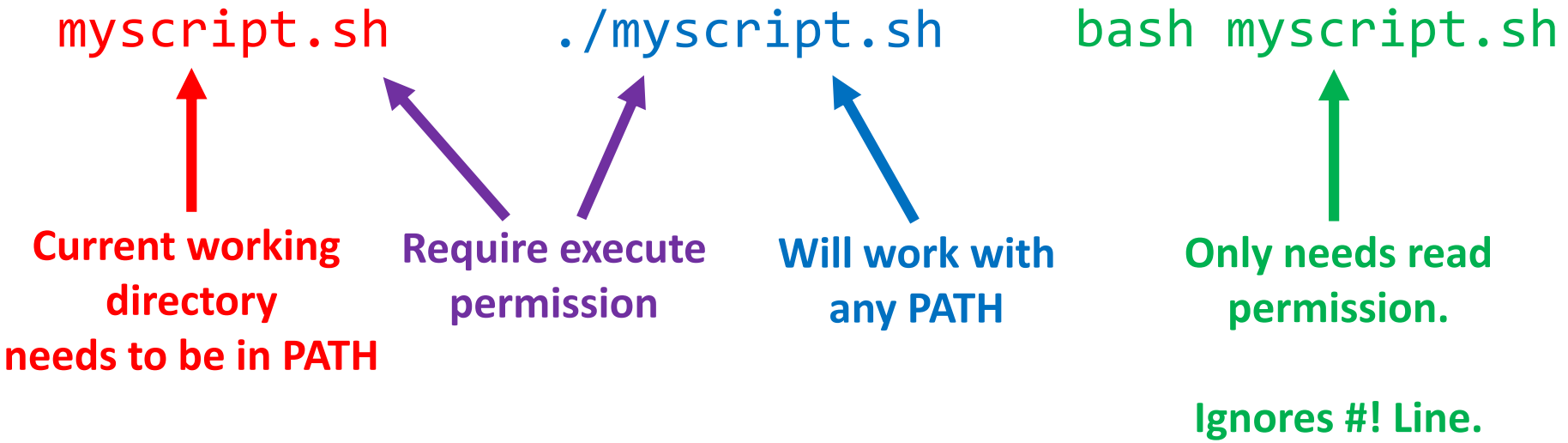
Week 4b Review

Running Shell Scripts

```
myscript.sh      ./myscript.sh      bash myscript.sh
```

Week 4b Review

Running Shell Scripts



Week 4b Review

User Input via stdin

```
read [options] varname …
```

```
#!/bin/bash
echo -n "Input a number: "
read mynum
echo "Your number is: $mynum"
```

```
#!/bin/bash
read -p "Input age and name: " age name
echo "Your name is, $name."
echo "You are $age years old."
```

Week 4b Review

Integer Arithmetic with expr

```
[dservos5@cs2211b ~]$ expr 20 + 30 - 5
45

[dservos5@cs2211b ~]$ expr 5 / 3
1

[dservos5@cs2211b ~]$ expr \( 50 - 10 \) \* 2
80
```

Week 4b Review

Integer Arithmetic with expr

```
[dservos5@cs2211b ~]$ expr 20 + 30 - 5
45
```

```
[dservos5@cs2211b ~]$ expr 5 / 3
1
```

Integer division
Only integer arguments and results allowed

```
[dservos5@cs2211b ~]$ expr \( 50 - 10 \) \* 2
80
```

Need to escape some operators
Each part of the expression has to be an argument to expr. Some operators conflict with shell wildcards and need to be escaped.

Week 4b Review

Decimal Arithmetic with bc

```
[dservos5@cs2211b ~]$ echo '5 / 3' | bc
1

[dservos5@cs2211b ~]$ echo 'scale=3; 5 / 3' | bc
1.666

[dservos5@cs2211b ~]$ echo 'scale=4; sqrt(5+6)/(1.5)^3' | bc
.9826
```

Week 4b Review

Decimal Arithmetic with bc

Integer division by default
Need to set scale for decimals

```
[dservos5@cs2211b ~]$ echo '5 / 3' | bc
1

[dservos5@cs2211b ~]$ echo 'scale=3; 5 / 3' | bc
1.666

[dservos5@cs2211b ~]$ echo 'scale=4; sqrt(5+6)/(1.5)^3' | bc
.9826
```

- **Does not need spaces between arguments**
- **Supports more functions and operations (sqrt and power)**
- **Can take decimal input (1.5)**
- **Takes expression via stdin and not as arguments**

Shell Scripts: Part 2

Bash Arithmetic

- The **bash** shell supports a few built in commands that allow us to do arithmetic without outside programs like bc.
- This syntax is only supported by bash and ksh.
- Should be avoided if you wish to create portable shell scripts (ones that work in other shells).
- `let` builtin evaluates an integer arithmetic expression and assigns the result to a shell variable.
- **Syntax:**

```
let expression …
```

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Each arithmetic expression to `let`, needs to be a single argument. Recall that you can use quotes to include spaces in a single argument.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

`let` assigns the value to the shell variable mentioned in the expression. In this case, b is assigned the result of 10 – 12, which is -2.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Like `expr`, `let` does integer arithmetic. Only takes integer numbers and division returns integer results. In this case, 3/4 would result in 0.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

`let` supports the `++` and `--` per/postfix operators. Increments or decrements the variable by one. If x is 5, it will now be 6.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

`let` uses the ** notation for power. This is the same as the ^ operator in bc. 2**3 is equivalent to 2^3, equals 8.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

You can reference a shell variable in a `let` expression by its name (without the $).

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

You can reference a shell variable in a `let` expression by its name (without the $).

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 +

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

-e option to echo allows us to include line breaks (\n) in our string.

Bash Arithmetic

Example 1:

/cs2211/week5/letex.sh

```
#!/bin/bash

let a=4+9
let "b = 10 - 12"
let c=3/4

x=5
y=2

let x++
let "z = y * 10 + 2**3"

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Output:

```
$ letex.sh
a = 13
b = -2
c = 0
x = 6
y = 2
z = 28
```

(( )) Notation

- `(( expression ))` notation is equivalent to `let` in most cases.
- Considered more portable than `let` as it is specified by POSIX.
- Still not supported by `csh` and `tcsh`.
- Full list of operators and functions supported by ((can be found here:

http://wiki.bash-hackers.org/syntax/arith_expr

(( )) Notation

Example 2: Redo Example 1 using (( ))

/cs2211/week5/ex2.sh

```
#!/bin/bash

((a=4+9))
(( "b = 10 - 12" ))
(( c=3/4 ))

x=5
y=2

((  x++  ))
(("z = y * 10 + 2**3" ))

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Output:

```
$ ex2.sh
a = 13
b = -2
c = 0
x = 6
y = 2
z = 28
```

(( )) Notation

Example 2: Redo Example 1 using (( ))

/cs2211/week5/ex2.sh

```
#!/bin/bash

((a=4+9))
(( "b = 10 - 12" ))
(( c=3/4 ))

x=5
y=2

(( x++ ))
(("z = y * 10 + 2**3" ))

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Spaces between ((and)) do not matter.

However, spaces between (and (or) and) do matter. Example, this is not allowed:

```
(   (x++)   )
```

Must have zero spaces between (and (.

```
y = 2
z = 28
```

Arithmetic Expansion

- Rather than having `let` or `((` set the value of a shell variable directly, we can use arithmetic expansion to return the result.
- Uses `$(( expression ))` notation.
- Unlike in `let` and `((`, we cannot use quotes.
- **Examples:**

```
[dservos5@cs2211b ~]$ echo $((10+5-7))
8

[dservos5@cs2211b ~]$ echo $(( 9 / 2 ))
4

[dservos5@cs2211b ~]$ x=5
[dservos5@cs2211b ~]$ echo $(( x*2 ))
10
```

Arithmetic Expansion

- Rather than having `let` or `((` set the value of a shell variable directly, we can use arithmetic expansion to return the result.
- Uses `$(( expression ))` notation.
- Unlike in `let` and `((`, we cannot use quotes.
- **Examples:**

```
[dservos5@cs2211b ~]$ echo $(("1+2"))
-bash: "1+2": syntax error: operand expected (error token is
""1+2"")

[dservos5@cs2211b week5]$ ls ex$((1+1)).sh
ex2.sh
```

Arithmetic Expansion

- Rather than having `let` or `((` set the value of a shell variable directly, we can use arithmetic expansion to return the result.
- Uses `$(( expression ))` notation.
- Unlike in `let` ar
- **Examples:**

Cannot use quotes around the arithmetic expression with the `$((` notation.

```
[dservos5@cs2211b ~]$ echo $(("1+2"))
-bash: "1+2": syntax error: operand expected (error token is
""1+2"")
```

```
[dservos5@cs2211b week5]$ ls ex$((1+1)).sh
ex2.sh
```

Arithmetic Expansion

- Rather than having `let` or `((` set the value of a shell variable directly, we can use arithmetic expansion to return the result.
- Uses `$(( expression ))` notation.
- Unlike in `let` and `((`, we cannot use quotes.
- **Examples:**

```
[dservos5@cs2211b ~]$
-bash: "1+2": syntax e
""1+2"")
```

`$((` will be replaced with the result (expanded) just like a shell variable in other commands.

```
[dservos5@cs2211b week5]$ ls ex$((1+1)).sh
ex2.sh
```

Arithmetic Expansion

Example 3: Redo Example 1 using $(( ))

/cs2211/week5/ex3.sh

```
#!/bin/bash

a=$((4+9))
b=$(( 10 - 12 ))
c=$(( 3/4 ))

x=5
y=2

((x++))
z=$(( y * 10 + 2**3 ))

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

Output:

```
$ ex3.sh
a = 13
b = -2
c = 0
x = 6
y = 2
z = 28
```

Arithmetic Expansion

Example 3: Redo Example 1 using $(( ))

/cs2211/week5/ex3.sh

```
#!/bin/bash

a=$((4+9))
b=$(( 10 - 12 ))
c=$(( 3/4 ))

x=5
y=2

((x++))
z=$(( y * 10 + 2**3 ))

echo -e "a = $a\nb = $b\nc = $c"
echo -e "x = $x\ny = $y\nz = $z"
```

No longer using quotes.
Would cause error if we did.

Output:

```
$ ex3.sh
a = 13
b = -2
c = 0
x = 6
y = 2
z = 28
```

Arithmetic

	`expr`	`bc`	`let`	`(( ))`	`$(( ))`
Portable	✓	✓ (if installed)	✗	✓ (not in csh/tcsh)	✓ (not in csh/tcsh)
Decimal Numbers	✗	✓	✗	✗	✗
Power Operator	✗	✓ ^	✓ **	✓ **	✓ **
sqrt Function	✗	✓ sqrt	✗	✗	✗
Sets Variables Directly	✗	✗	✓	✓	✓ (can, but probably not what you want)
Returns Result	stdout	stdout	✗	✗	expansion
Expression via stdin	✗	✓	✗	✗	✗
Expression via Arguments	One argument for each part of expression	✗	One Argument	One Argument	One Argument

Command Line Arguments

- Similarly to the other UNIX/Linux commands we have seen to date, shell script can also take arguments from the command line.
- **Example:**

```
myshellscript.sh arg1 arg2 arg3 …
```

- We can access these arguments using the special shell variable **$n** where **n** is a number from 0 to 9 such that:
  - $0 the name of the file/script
  - $1 the first argument
  - $2 the second argument
  - ….
  - $9 the ninth argument

This is called a "positional parameter".

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

No arguments given.

```
$ args.sh
0: ./args.sh
1:
2:
3:
4:
5:
6:
7:
8:
9:
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "
echo "
echo "
echo "
echo "
echo "
echo "
echo "
echo "
```

Output:

No arguments given.

```
$ args.sh
0: ./args.sh
1:
```

$0 is always the name of the script.

Path will change based on where/how we run the script from. E.g.

```
[dservos5@cs2211b ~]$ /cs2211/week5/args.sh
0: /cs2211/week5/args.sh
```

Name will be different if we use a link rather than call the script directly.

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

No arguments given.

```
$ args.sh
0: ./args.sh
1:
2:
3:
4:
5:
6:
7:
8:
9:
```

$n variables are empty if no n^{th} argument is given.

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh 5 hello world 1.3
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh 5 hello world 1.3
0: ./args.sh
1: 5
2: hello
3: world
4: 1.3
5:
6:
7:
8:
9:
```

hello world is considered to be two arguments as it contains a space.

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh 5 "hello world" 1.3
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh 5 "hello world" 1.3
0: ./args.sh
1: 5
2: hello world
3: 1.3
4:
5:
6:
7:
8:
9:
```

Using quotes makes the shell treat hello world as a single argument.

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

More than 9 arguments

```
$ args.sh A B C D E F G H I J K L M
```

Command Line Arguments

Example 4:

/cs2211/week5/args.sh

```
#!/bin/bash

echo "0: $0"
echo "1: $1"
echo "2: $2"
echo "3: $3"
echo "4: $4"
echo "5: $5"
echo "6: $6"
echo "7: $7"
echo "8: $8"
echo "9: $9"
```

Output:

```
$ args.sh A B C D E F G H I J K L M
0: ./args.sh
1: A
2: B
3: C
4: D
5: E
6: F
7: G
8: H
9: I
```

Only showing first 9 arguments.

How do we get more?

`echo $10` outputs:
`A0`

Why?

Command Line Arguments

- We have a few more special variables related to arguments available to us:
  - $* all the arguments as a single string.
  - $# the number of arguments given to the script.

- `shift` command:
  - Shifts all arguments to the left by one:

    $1 = $2, $2 = $3, $3 = $4, and so on
  - The value of $1 is lost.
  - $# is decremented by one.
  - $* no longer contains the first argument.
  - $0 is not changed (still contains script name).

Command Line Arguments

Example 5: Write a shell script that prints of its arguments (even if there are more than 9) and also states how many arguments were given.

Example Input/output:

```
$ myecho.sh A B C D E F G H I J K L M N O P Q R S T
A B C D E F G H I J K L M N O P Q R S T
20 arguments were given.

$ myecho.sh Weeks of coding can save you hours of planning
Weeks of coding can save you hours of planning
9 arguments were given.

$ myecho.sh "Weeks of coding" "can save you" "hours of planning"
Weeks of coding can save you hours of planning
3 arguments were given.
```

Command Line Arguments

Example 5: Write a shell script that prints of its arguments (even if there are more than 9) and also states how many arguments were given.

/cs2211/week5/myecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."
```

Command Line Arguments

Example 5: Write a shell script that prints of its arguments (even if there are more than 9) and also states how many arguments were given.

/cs2211/week5/myecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."
```

$* contains all arguments.

Command Line Arguments

Example 5: Write a shell script that prints of its arguments (even if there are more than 9) and also states how many arguments were given.

/cs2211/week5/myecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."
```

$# contains the number of arguments.

Command Line Arguments

Example 5: Write a shell script that prints of its arguments (even if there are more than 9) and also states how many arguments were given.

/cs2211/week5/myecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."
```

What if we add a `shift`?

Command Line Arguments

Example 5b: What if we add a `shift`?

/cs2211/week5/myshiftecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."

shift

echo $*
echo "$# arguments were given."
```

Output:

```
$ myshiftecho.sh A B C D E F
```

Command Line Arguments

Example 5b: What if we add a `shift`?

/cs2211/week5/myshiftecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."

shift

echo $*
echo "$# arguments were given."
```

Output:

```
$ myshiftecho.sh A B C D E F
A B C D E F
6 arguments were given.
B C D E F
5 arguments were given.
```

Command Line Arguments

Example 5b: What if we add a `shift`?

/cs2211/week5/myshiftecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."

shift

echo $*
echo "$# arguments were given."
```

Output:

```
$ myshiftecho.sh "A B C" D E F
```

Command Line Arguments

Example 5b: What if we add a `shift`?

/cs2211/week5/myshiftecho.sh

```
#!/bin/bash

echo $*
echo "$# arguments were given."

shift

echo $*
echo "$# arguments were given."
```

Output:

```
$ myshiftecho.sh "A B C" D E F
A B C D E F
4 arguments were given.
D E F
3 arguments were given.
```

Conditional Statements

- Conditional statements like **`if`** are used to "test" something:
  - In Java, C and Python, these type of statements test the result of a Boolean expression like 5 > 6.
  - In shell scripts, the **`if`** statement only checks the **exit status** of a command (if it is successful or not).
- **Syntax:**

```
if command
then
   commands_to_run_if_successful
fi
```

Conditional Statements

Examples:

```
if grep UNIX /cs2211/week3/textbook.txt > /dev/null
then
        echo 'UNIX was found!'
fi

if ls /usr/bin/bc > /dev/null 2>&1
then
        echo 'The bc command was found!'
fi

if who | grep dservos5; then
        echo 'dservos5 is online.'
fi
```

Conditional Statements

Examples:

```
if grep UNIX /cs2211/week3/textbook.txt > /dev/null
then
        echo 'UNIX was found!'
fi

if ls /usr/bin/bc > /dev/null 2>&1
then
        echo 'The bc commar
fi

if who | grep dservos5; then
        echo 'dservos5 is online.'
fi
```

We can put then on the same line if we use a ;

Conditional Statements

- As with other programming/scripting languages, the `if` statement in shell scripts supports **else** and **else if** type constructs.
- **Syntax:**

```
if command1; then
    commands_to_run_if_command1_successful
elif command2; then
    commands_to_run_if_command2_successful
…
elif commandN; then
    commands_to_run_if_commandN_successful
else
    commands_to_run_otherwise
fi
```

Conditional Statements

Examples:

```
if who | grep dservos5; then
      echo 'dservos5 is online.'
else
      echo 'Coast is clear!'
fi


if grep UNIX textbook.txt; then
      cp textbook.txt unixtextbook.txt
elif grep DOS textbook.txt; then
      cp textbook.txt dostextbook.txt
else
      cp textbook.txt unkowntextbook.txt
fi
```

Conditional Statements

- As in the command line, we can use the `||` and `&&` operators in `if` statements.
- These act like AND and OR operators in other languages.
- Exit status of `command1 || command2` is 0 if either `command1` or `command2` has an exit status of 0.
- Exit status of `command1 && command2` is 0 if BOTH command1 and command2 have an exit status of 0.
- Syntax:

```
if command1 || command2; then
    run_if_either_successful
fi
```

```
if command1 && command2; then
    run_if_both_successful
fi
```

Conditional Statements

- That's great for exit statuses but how do we check the result of Boolean expressions?
- Yet again, there is a command for that: **`test`**
- The `test` command evaluates a logical comparison (e.g. 5 > 4) and returns a 0 exit status if true, and a nonzero status if false.
- Can also be used to check if files/directories exists, their status, size, etc.
- **Syntax:**

```
test EXPRESSION1 option EXPRESSION2
test option EXPRESSION
```

Conditional Statements

Options:

Arithmetic Tests

Options/Syntax	Description
<INTEGER1> **-eq** <INTEGER2>	True, if the integers are **equal**.
<INTEGER1> **-ne** <INTEGER2>	True, if the integers are **NOT equal**
<INTEGER1> **-le** <INTEGER2>	True, if the first integer is **less than or equal** to the second one.
<INTEGER1> **-ge** <INTEGER2>	True, if the first integer is **greater than or equal** to the second one.
<INTEGER1> **-lt** <INTEGER2>	True, if the first integer is **less than** the second one.
<INTEGER1> **-gt** <INTEGER2>	True, if the first integer is **greater than** the second one.

Conditional Statements

Example 6: Write a shell script that takes two arguments and returns the largest.

Conditional Statements

Example 6: Write a shell script that takes two arguments and returns the largest.

/cs2211/week5/findmax.sh

```
#!/bin/bash

if test $1 -gt $2; then
        echo $1
else
        echo $2
fi
```

Output:

```
$ findmax.sh 5 9
9

$ findmax.sh 7 -8
7

$ findmax.sh 7 -8.4
./findmax.sh: line 3: test: -8.4:
integer expression expected
-8.4
```

Conditional Statements

Example 6: Write a shell script that takes two arguments and returns the largest.

/cs2211/week5/findmax.sh

```
#!/bin/bash

if test $1 -gt $2; then
        echo $1
else
        echo $2
fi
```

Output:

```
$ findmax.sh 5 9
9

$ findmax.sh 7 -8
7

$ findmax.sh 7 -8.4
./findmax.sh: line 3: test: -8.4:
integer expression expected
-8.4
```

test does not support decimals.

Conditional Statements

Example 6: Write a shell script that takes two arguments and returns the largest.

/cs2211/week5/findmax.sh

```
#!/bin/bash

if test $1 -gt $2; then
        echo $1
else
        echo $2
fi
```

Output:

```
$ findmax.sh 5 9
9

$ findmax.sh 7 -8
7

$ findmax.sh 7 -8.4
./findmax.sh: line 3: test: -8.4:
integer expression expected
-8.4
```

Why was -8.4 output?

Conditional Statements

Options:

File Tests

Options/Syntax	Description
-e <FILE>	True if <FILE> exists.
-f <FILE>	True, if <FILE> exists and is a **regular** file.
-d <FILE>	True, if <FILE> exists and is a **directory**.
-L <FILE>	True, if <FILE> exists and is a **symbolic link** (can also use -h).
-r <FILE>	True, if <FILE> exists and is **readable**.
-w <FILE>	True, if <FILE> exists and is **writable**.
-x <FILE>	True, if <FILE> exists and is **executable**.
-s <FILE>	True, if <FILE> exists and has size bigger than 0 (**not empty**).
<FILE1> **-ef** <FILE2>	True, if <FILE1> and <FILE2> refer to the **same device and inode numbers**.

Conditional Statements

Example 7: Check if the regular file *duck.txt* exists in the current working directory and is not a directory. Do not use `ls`.

Conditional Statements

Example 7: Check if the regular file *duck.txt* exists in the current working directory and is not a directory. Do not use `ls`.

/cs2211/week5/findduck.sh

```
#!/bin/bash

if test -f duck.txt; then
        echo 'I found the duck!'
else
        echo 'No ducks here.'
fi
```

Output:

```
[dservos5@cs2211b week5]$ findduck.sh
No ducks here.
[dservos5@cs2211b ~]$ /cs2211/week5/findduck.sh
I found the duck!
```

Conditional Statements

Options:

String Tests

Options/Syntax	Description
-z <STRING>	True, if <STRING> is **empty**.
-n <STRING>	True, if <STRING> is **not empty** (this is the default operation).
<STRING1> **=** <STRING2>	True, if the strings are **equal**.
<STRING1> **!=** <STRING2>	True, if the strings are **not equal**.
<STRING1> **<** <STRING2>	True if <STRING1> sorts **before** <STRING2> lexicographically.
<STRING1> **>** <STRING2>	True if <STRING1> sorts **after** <STRING2> lexicographically.

Conditional Statements

Example 8:

/cs2211/week5/ex8.sh

```
#!/bin/bash

if test "$HOSTNAME" = "cs2211b.gaul.csd.uwo.ca"; then
        /cs2211/bin/ducksay "QUACK QUACK!"
else
        echo "ERROR: Not running on the course server!"
        exit 1
fi
```

If the script is run on the course server (i.e. hostname is *cs2211b.gaul.csd.uwo.ca*) then run the `ducksay` program that is only available on the course server.

Otherwise print an error and exit with a status code of 1.

Conditional Statements

Example 8:

/cs2211/week5/ex8.sh

```
#!/bin/bash

if test "$HOSTNAME" = "cs2211b.gaul.csd.uwo.ca"; then
        /cs2211/bin/ducksay "QUACK QUACK!"
else
        echo "ERROR: Not running on the course server!"
        exit 1
fi
```

Good practice to put double quotes around variables that contain strings. What if it had a space in it?

Would get an error like: *./ex8.sh: line 3: test: too many arguments* without the quotes.

Conditional Statements

Example 8:

/cs2211/week5/ex8.sh

```
#!/bin/bash

if test "$HOSTNAME" = "cs2211b.gaul.csd.uwo.ca"; then
        /cs2211/bin/ducksay "QUACK QUACK!"
else
        echo "ERROR: Not running on the course server!"
        exit 1
fi
```

Spaces are required between expressions and options. Each one has to be an argument to `test`. Just like with `expr`.

Conditional Statements

Example 8:

/cs2211/week5/ex8.sh

```
#!/bin/bash

if test "$HOSTNAME" = "cs2211b.gaul.csd.uwo.ca"; then
        /cs2211/bin/ducksay "QUACK QUACK!"
else
        echo "ERROR: Not running on the course server!"
        exit 1
fi
```

Exits the shell script right away and returns the given exit status. Default exit status (if you do not use `exit`) is 0.

It is good practice to always return a nonzero exit status if an error occurred.

Conditional Statements

Options:

Misc syntax

Options/Syntax	Description
! <EXPR>	True, if <EXPR> is false (NOT).
(<EXPR>)	Group a <EXPR> (for precedence).
-v <VARIABLENAME>	True if the variable <VARIABLENAME> has been set.
<EXPR1> -a <EXPR2>	True, if <EXPR1> and <EXPR2> are true (AND).
<EXPR1> -o <EXPR2>	True, if either <EXPR1> or <EXPR2> is true (OR).

More Options and a Good Guide on the Test Command:

http://wiki.bash-hackers.org/commands/classictest

Conditional Statements

Options:

Misc syntax

Options/Syntax	
! <EXPR>	
(<EXPR>)	
-v <VARIABLENAME>	True if the variable <VARIABLENAME> has been set.
<EXPR1> -a <EXPR2>	True, if <EXPR1> and <EXPR2> are true (AND).
<EXPR1> -o <EXPR2>	True, if either <EXPR1> or <EXPR2> is true (OR).

Spaces are mandatory here.

Also likely need to escape the ()s. E.g. \(and \).

More Options and a Good Guide on the Test Command:

http://wiki.bash-hackers.org/commands/classictest

Conditional Statements

[ ] Notation

- Most shells support the [ ] notation as a shorthand for the `test` command.
- **Syntax:**

```
[ option EXPRESSION ]
[ EXPRESSION1 option EXPRESSION2 ]
```

- **Example:**

```
if [ ! "$USER" = "dservos5" ]; then
        echo "Who are you?"
else
        echo "Welcome back Daniel!"
fi
```

Conditional Statements

[ ] Notation

- Most shells support the [ ] notation as a shorthand for the te

MUST HAVE spaces after [and before] in this notation.

- **Syntax:**

```
[ option EXPRESSION ]
[ EXPRESSION1 option EXPRESSION2 ]
```

- **Example:**

```
if [ !"$USER" = "dservos5" ]; then
        echo "Who are you?"
else
        echo "Welcome back Daniel!"
fi
```

Conditional Statements

Example 9: Complex Example

Write a shell script named *8ball.sh* which takes a single argument, a filename. *8ball.sh* should use the special shell variable $RANDOM which returns a random number between 0 - 32767 to choose a line from the input file to print.

Print an error if the file does not exist or is not a regular file.

Assume file is less than 32767 lines long and contains at least one line.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

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

First we need to do some error checking.

Check if the first argument is a valid regular file using the -f option to `test`.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi
```

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi
```

Using shorthand [] notation

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi
```

NOT

Checks if file does NOT exist.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi
```

If the file does not exist, print an error message and exit with status code 1.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi
```

Before we can generate a random number we need to know how many lines are in the file (so we know how big the random number can be).

We can use wc for this.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`
```

Now we need to get a random number between 0 and 32767 from $RANDOM and store it for later use.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`

x=$RANDOM
```

$RANDOM is just a special shell variable. It does not take any arguments to change its range.

We will have to do some math to make it between 1 and the number of lines in $1.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`

x=$RANDOM
x=$(( x % lines + 1 ))
```

Use modulus operator to make $x between 1 and the number of lines in $1.

Using $(( )) arithmetic expansion notation.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`

x=$RANDOM
x=$(( x % lines + 1 ))
```

Now $x should contain the line number we want. Just have to use head and `tail` to get the text on that line.

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`

x=$RANDOM
x=$(( x % lines + 1 ))

head -$x $1 | tail -1
```

Conditional Statements

Example 9: Complex Example

/cs2211/week5/8ball.sh

```
#!/bin/bash

if [ ! -f $1 ]; then
        echo "Bad file name!"
        exit 1
fi

lines=`cat $1 | wc -l`

x=$RANDOM
x=$(( x % lines + 1 ))

head -$x $1 | tail -1
```

Output:

```
$ 8ball.sh 8ball.txt
As I see it, yes

$ 8ball.sh 8ball.txt
Most likely

$ 8ball.sh 8ball.txt
You may rely on it

$ 8ball.sh 8ball.txt
Cannot predict now
```

Loop Statements

For loop

- `for` loops allow the repetition of a command for a specific set of values.
- Similar to for each loops in other languages like Java and Python.
- **Syntax:**

```
for var in value1 value2 …
do
   commands_to_run
done
```

Loop Statements

For loop

- `for` loops allow the repetition of a command for a specific set of values.
- Similar to for each loops in other languages like Java and Python.
- **Syntax:**

```
for var in value1 value2 …; do
   commands_to_run
done
```

Can put do on same line with semicolon (;) just like with `then` in `if` statements.

Loop Statements

For loop

- `for` loops allow the repetition of a command for a specific set of values.
- Similar to for each loops in other languages like Java and Python.
- **Syntax:**

```
for var in value1 value2 …; do
   commands_to_run
done
```

Loop is run once for each value listed.

List of values can be the result of expansion (variable or command).

Loop Statements

For loop

- `for` loops allow the repetition of a command for a specific set of values.
- Similar to for each loops in other languages like Java and Python.
- **Syntax:**

```
for var in value1 value2 …; do
   commands_to_run
done
```

The value of the variable `var` is updated each iteration of the loop to the next value in the list.

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Nested statements are allowed in shell scripts (goes for if statements as well).

First loop runs for values 1 to 9, updating `$i` each time to the next value. Second loop does the same but for variable `$j`.

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Multiplies the current value of `$i` and `$j` and stores the result in `$value`.

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Prints the value of `$value` and then a tab character. Unlike `echo`, `printf` does not print a line break by default.

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

This echo just adds a line break.

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
```

Example Output:

```
[dservos5@cs2211b week5]$ timestable.sh
1       2       3       4       5       6       7       8       9
2       4       6       8       10      12      14      16      18
3       6       9       12      15      18      21      24      27
4       8       12      16      20      24      28      32      36
5       10      15      20      25      30      35      40      45
6       12      18      24      30      36      42      48      54
7       14      21      28      35      42      49      56      63
8       16      24      32      40      48      56      64      72
9       18      27      36      45      54      63      72      81
```

Loop Statements

Example 10: Print out a multiplication table

/cs2211/week5/timestable.sh

```
#!/bin/sh

for i in 1 2 3 4 5 6 7 8 9; do
    for j in 1 2 3 4 5 6 7 8 9; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Typing out each number in the set of variables is a pain. What if we wanted to run our loop a thousand times?

Loop Statements

Example 10: Updated with seq

/cs2211/week5/updatedtimestable.sh

```
#!/bin/sh

for i in `seq 1 9`; do
    for j in `seq 1 9`; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

We can use the seq command to create the list of variables for us.

`seq FIRST LAST`

Loop Statements

Example 10: Updated with seq

/cs2211/week5/updatedtimestable.sh

```
#!/bin/sh

for i in `seq 1 9`; do
    for j in `seq 1 9`; do
        value=$((i * j))
        printf "$value\t"
    done
    echo
done
```

Backquotes in these lines are important. Need to get the output of this command.

Without the quotes, the for loop would interpret seq 1 9 as values to loop through.

Loop Statements

Example 11: Find the total word count of all files in the given directory. Print an error if the directory does not exist or no arguments are given.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Error checking using `if` statements and `test`.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then

fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Use `ls` to get a list of the files in the directory.

Save the list to the `$filelist` variable.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Start our total word count at 0

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $
total=0

for file in $fileList; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Run this loop once for each file in `$filelist`.

Update the value of `$file` to the next file each loop iteration.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Count the number of words in $file and store the result in $count.

Note that we need to give the dir path ($1/) as it is not our current working directory.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Update the value of $total by adding the value of $count to it.

We could also do:
((total+=count))

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
        exit 1
elif [ ! -d $1 ]; then
        echo "Error: Not a directory."
        exit 2
fi

filelist=`ls $1`
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Finally, output the total wordcount.

Example 11:

/cs2211/week5/wordcount.sh

```
#!/bin/bash

if [ $# -ne 1 ]; then
        echo "Error: should take exactly one argument."
```

Example Output:

```
[dservos5@cs2211b week5]$ wordcount.sh countdir
17
```

```
total=0

for file in $filelist; do
        count=`cat $1/$file | wc -w`
        ((total=total+count))
done

echo $total
```

Loop Statements

Example 12: Take a word as an argument and print all of the files in the current directory with that word (ignore hidden files). No error checking required.

Loop Statements

Example 12: Take a word as an argument and print all of the files in the current directory with that word (ignore hidden files). No error checking required.

/cs2211/week5/findfile.sh

```
#!/bin/bash

for file in *; do
    if grep "$1" $file > /dev/null 2>&1; then
        echo $file
    fi
done
```

Output:

```
[dservos5@cs2211b week5]$ findfile.sh duck
ex8bad1.sh
ex8.sh
findduck.sh
```

Loop Statements

Example 12: Take a word as an argument and print all of the files in the current directory with that word (ignore hidden files). No error checking required.

/cs2211/week5/findfi

```
#!/bin/bash

for file in *; do
    if grep "$1" $file > /dev/null 2>&1; then
        echo $file
    fi
done
```

We can use shell wild cards here to create a list of files, like we do in the command line.

Output:

```
[dservos5@cs2211b week5]$ findfile.sh duck
ex8bad1.sh
ex8.sh
findduck.sh
```

Loop Statements

While Loop

- `while` loops keep repeating the statements they contain until a given command is no longer successful.
- Frequently, this command will be the test command or the [ ] shorthand, used to test some condition.
- **Syntax:**

```
while command
do
   commands_to_run
done
```

```
while command; do
   commands_to_run
done
```