# INTRODUCTION TO SHELL SCRIPTING

Dr. Jeffrey Frey
University of Delaware, IT

Dare to be first.

UNIVERSITY OF DELAWARE

# GOALS — BASIC

- What is shell scripting?
- What makes shell scripting useful?
- What should you know already…

# GOALS — PART 1

- Script layout/design
- Variables
  - environment vs. local
  - types:  string, numeric
- Standard language constructs
  - conditionals
  - loops

# GOALS — PART 2

- Command substitution
  - backticks vs. dollar–parenthesis
- Advanced variables
  - advanced expansion
  - arrays
- Advanced language constructs
  - subroutines
  - case statements (branch tables)
  - and/or lists

Dare to be first.
UNIVERSITY OF DELAWARE

"…a true creator is necessity, which is the mother of our invention."

–Plato

# THE TASK AT HAND

- You've performed 1000 runs of a program and captured its output in files named `out_#.log`
  - Each component of the program prints how long it took to execute (*walltime*): `[program] walltime=#`
  - The multiple walltimes in a file must be summed
  - You wish to calculate the minimum, maximum, and average walltime

# THE TASK AT HAND

- You've performed 1000 runs of a program and captured its output in files named `out_#.log`
  1. For each file, find all `[program] walltime=#` lines and extract the number of seconds. Add the number of seconds = total
     1. If that total is the largest yet, remember it
     2. If that total is the smallest yet, remember it
     3. Add to the 'total total' walltime
  2. Show the minimum, maximum, and average walltime

# THE TASK AT HAND

- For each file, find all `[program] walltime=#` lines and extract the number of seconds, sum them

```
$ total_wall=$(grep walltime= file_1.log | sed 's/^.*walltime=//' | \
>   awk 'BEGIN{total=0;}{total+=$1;}END{print total;}')
```

  - grep extracts lines with 'walltime='
  - sed removes program name and 'walltime='
  - awk sums the walltimes

# THE TASK AT HAND

- Numerical bits:
  - Initialization:

```
$ max_wall=0; min_wall=1000000; total_total_wall=0; num_files=0;
```

  - Per-file checks:

```
$ if [ $total_wall -gt $max_wall ]; then max_wall=$total_wall; fi
$ if [ $total_wall -lt $min_wall ]; then min_wall=$total_wall; fi
$ total_total_wall=$((total_total_wall+total_wall))
$ num_files=$((num_files+1))
```

# THE TASK AT HAND

- Final analysis:

```
$ printf "Total walltime (min/max/avg): "
$ printf "%d/%d/%d\n" $min_wall $max_wall $((total_total_wall/num_files))
```

# WHAT IS SHELL SCRIPTING?

- All of the example shell code was entered at the prompt — in *interactive* mode
- If you want to reissue a command, you must edit and retype it (or copy-and-paste, etc.)
  - A number of the lines of shell code in our example must be entered 1000 times!

# WHAT IS SHELL SCRIPTING?

# WHAT IS SHELL SCRIPTING?

When promoting the Apple II at computer shows, Steve Wozniak would sit down before the show and manually enter the machine code for the BASIC interpreter — because they hadn't made a disk or tape drive for the computer yet.

**Imagine his time savings when a tape drive became available!**

# WHAT IS SHELL SCRIPTING?

When promoting the Apple II at computer shows, Steve Wozniak would sit down before the show and manually enter the machine code for the BASIC interpreter — because they hadn't made a disk or tape drive for the computer yet.

**Imagine his time savings when a tape drive became available!**

- Hence the quote from Plato:  monotonous, repetitious tasks beg for invention of shortcuts.

# WHAT IS SHELL SCRIPTING?

# WHAT IS SHELL SCRIPTING?

A shell script is a text file containing a sequence of shell commands that you would otherwise enter interactively.

# WHAT MAKES SHELL SCRIPTING USEFUL?

- Several obvious reasons:
  - Massive time-savings versus working interactively
  - Easily create your own 'commands'
  - Provides a record of the commands necessary to repeat a task.
- Far easier to debug a sequence of commands — rerun script vs. reenter every command
- The script runs in its own environment

# WHAT MAKES SHELL SCRIPTING USEFUL?

- Several obvious reasons:
  - Massive time-savings versus working interactively

Shell scripts are computer programs written in a high-level language that extend the functionality of the shell and simplify your workflow.

  - — rerun script vs. reenter every command
- The script runs in its own environment

# PREREQUISITES

- Your script skills are proportional to your proficiency with the interactive shell environment.
  - The more you know about the shell's language, the more you can do in scripts…
  - …and the more sophisticated you seek to make your scripts, the more you tend to learn about the shell's language.

# PREREQUISITES

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
  - A script is a file: has a user and group owner and user-group-other permissions
  - "Execute" bit(s) must be set for script to be directly executable

# PREREQUISITES

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
- Proficiency in a text editor
  - From the shell:  vi, vim, emacs, pico, nano, …
  - On your PC

# PREREQUISITES

- You must be aware of line encoding when editing shell scripts for Unix/Linux on Windows! Windows text files demarcate end-of-line differently than Unix/Linux, and many Unix/Linux programs will not function properly when given a Windows text file.
- The *dos2unix* command can convert the file to Unix/Linux line encoding. A better option is to use a Windows text editor that can save files with Unix/Linux line encoding.

# PREREQUISITES

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
- Proficiency in a text editor

# PREREQUISITES

- Your script skills are proportional to your proficiency with the interactive shell environment.
- Understand the Unix filesystem, privileges
- Pro

"Script" is a generic term that typically denotes an interpreted (not compiled) programming language. Scripts can be written for shells, Perl, Python, PHP, AppleScript, JavaScript, Lua, TCL, et al. **This tutorial focuses on Bash shell scripting.**

# SCRIPT LAYOUT/DESIGN

- Each line of text in the script…
  - …can have leading and trailing whitespace (which is ignored when executed)
  - …is a comment if it starts with the '#' character (and is ignored when executed)
  - …may be blank (and is ignored when executed)
  - …contains all of or a portion of a shell command
  - …can be continued on the next line if it ends with the '\' character

# SCRIPT LAYOUT/DESIGN

- Each line of text in the script…
  - …can have leading and trailing whitespace (which is ignored when executed)
  - …r (and
  - …
  - …nd
  - …can be continued on the next line if it ends with the '\' character

Bash shell scripting is a free–form programming language: whitespace usually has no significance aside from increasing the legibility of the code.

# SCRIPT LAYOUT/DESIGN

- Example:

```
#
# My first script
#


# All lines above (and this one) are ignored on execution.
egrep -r \
  '^2015-01-14' \
  /var/log/messages* \
  /var/adm/syslog*

     # All done
     echo "All done."


     exit 1
```

# SCRIPT LAYOUT/DESIGN

- What the shell sees when you execute this script:

```
egrep -r '^2015-01-14' /var/log/messages* /var/adm/syslog*
echo "All done."
exit 1
```

# SCRIPT LAYOUT/DESIGN

- Note that this shell script is truly just a sequence of commands.
  - No variables, loops, conditionals, or other programming constructs
  - This script would be functional in most any Unix/Linux shell — Bash, Csh, Tcsh, Zsh!

# SCRIPT LAYOUT/DESIGN

- Note that this shell script is truly just a sequence of commands.

```
$ bash my_first_script
egrep: /var/adm/syslog*: No such file or directory
"All done."

$ csh my_first_script
"All done."

$ tcsh my_first_script
"All done."

$ zsh my_first_script
my_first_script:7: no matches found: /var/adm/syslog*
"All done."
```

# SCRIPT LAYOUT/DESIGN

- Executing the script
  - In this example, a shell was started and asked to execute the script:

```
$ bash my_first_script
```

  - **You** need to know in which shell the script should be executed!

# SCRIPT LAYOUT/DESIGN

- Executing the script
  - A script can be made to behave like any other executable by adding a *hash-bang* (a.k.a. *shebang*)…

```
#!/bin/bash
#
# My first script
#


# All lines above (and this one) are ignored on execution.
egrep -r \
```

  - …and marking it as executable with *chmod.*

```
$ chmod u+x my_first_script
```

# SCRIPT LAYOUT/DESIGN

- Executing the script

```
$ ls -l my_first_script
-rwxr--r-- 1 user group 227 Jan 15 11:00 my_first_script

$ ./my_first_script
egrep: /var/adm/syslog*: No such file or directory
"All done."
```

# SCRIPT LAYOUT/DESIGN

- Executing the script

```
$ ls -l my_first_script
-rwxr--r-- 1 user group 227 Jan 15 11:00 my_first_script

$ ./my_first_script
egr  : /var/adm/syslog*: No such file or directory
```

*To execute this script I prefixed its name with './' — why did I do this?*

# SCRIPT LAYOUT/DESIGN

- Executing the script
  - Another often-used *hash-bang* variant:

```
#!/usr/bin/env bash
#
# My first script
#


# All lines above (and this one) are ignored on execution.
egrep -r \
```

  - */usr/bin/env* looks for 'bash' on your shell's path and uses the first one found as the interpreter for the script.



- Most often seen in Python scripts.

# SCRIPT LAYOUT/DESIGN

- Executing the script
  - Another often-used *hash-bang* variant:

```
#!/usr/bin/env bash
#
# My first script
#


# All l
egrep -
```

  - /( and
    u( ne
    script.

If this shell script were shared with multiple users, there would be no way to know which "bash" would be used to execute the script! Each user's path may differ from your own, so it's often best to not use /usr/bin/env in this way.



- Most often seen in Python scripts.

# SCRIPT LAYOUT/DESIGN

- Executing the script
  - Script can also be executed in the current shell without starting a new process with its own environment.
  - Usually used to setup path environment variables.

```
$ echo $INTEL_PATH

$ echo $MANPATH
/usr/share/man:/usr/local/share/man
$ source /opt/intel/setup_fortran.sh
$ echo $INTEL_PATH
/opt/intel/2013/sp1
$ echo $MANPATH
/opt/intel/2013/sp1/man:/usr/share/man:/usr/local/share/man
```



- Most often seen in Python scripts.

# VARIABLES

- Where would mathematics be without variables to represent unspecified values?
  - The same goes for computer programming
  - Variables help to generalize program code
    - Rather than entering all data when the program is written, variables defer value specification to the time of execution

# VARIABLES

- Command line arguments behave like variables
- Rather than editing and recompiling the 'ls' command each time I use it, I provide one or more paths:

```
$ ls /usr/bin /usr/sbin
/usr/bin:
total 366732
-rwxr-xr-x    1 root root       37000 Jun 25  2014 [
-rwxr-xr-x    1 root root       11528 Nov 11  2010 411toppm
-rwxr-xr-x    1 root root      112224 Nov 22  2013 a2p
-rwxr-xr-x.   1 root root       93848 Feb 15  2012 a2ping
-rwxr-xr-x    1 root root       50248 Jul 23 10:18 ab
    :
```



- As we will see, command line arguments actually ARE variables within a shell script

# VARIABLES

- In the Bash shell, variables use a leading '$' when referenced but are declared without it
  - Names are case sensitive

```
$ x_var=1
$ X_Var=2

$ printf "%d == %d ?\n" $x_var $X_Var
1 == 2 ?

$ printf "X_VAR = %d ?\n" $X_VAR
X_VAR = 0 ?
```

# VARIABLES

- In the Bash shell, variables use a leading '$' when referenced but are declared without it
  - Until they are explicitly exported, variables exist only within the current shell:

```
$ declare | grep x_var
x_var=1

$ /bin/bash

$ printf "%d == %d ?\n" $x_var $X_Var
0 == 0 ?

$ declare | grep x_var
$
```

# VARIABLES

- In t … hen refe…
  - U… t only within the current shell.

Want to see lots of info about the variables defined in your Bash shell? Type ***declare -p*** and hit return — be prepared for a lengthy list (you may want to pipe it through ***less***)! Check out the man page for ***declare*** and ***set*** if you're interested.

```
$ declare | grep x_var
x_var=1

$ /bin/bash

$ printf "%d == %d ?\n" $x_var $X_Var
0 == 0 ?

$ declare | grep x_var
$
```

# VARIABLES

- When your script executes, it inherits any environment variables that were defined in the parent process.
  - The script is free to change their values and re-export them.
  - This influences child processes of the script, but NOT the script's parent process.



- As a parent, I determine the rules by which I raise my children.
- When they become parents, they are free to reuse my rules or make their own.
- BUT, I cannot travel back in time and change the rules my parents used to raise me!

# VARIABLES — NUMERIC

- By default, variables are strings
  - Variables can declared as integer type to restrict their usage as such:

```
$ declare -i x_var

$ x_var=f
$ echo $x_var
0

$ x_var=-15
$ echo $x_var
-15

$ x_var=-15.6
-bash: -15.6: syntax error: invalid arithmetic operator (error token is ".6")
```



- Declaring a variable as integer restricts the values that can be assigned to it.

# VARIABLES — NUMERIC

- Arithmetic can be performed on integer variables.

```
$ declare -i x_var
$ x_var=-15

$ x_str=20
$ echo $((x_var + x_str))
5

$ x_str=0x15
$ echo $((x_var + x_str))
6
```

  - Any variables NOT declared as integer type are converted during arithmetic evaluation.



- Bash will convert strings containing hexadecimal and octal values.

# VARIABLES — NUMERIC

- Arithmetic can be performed on integer variables.

```
$ declare -i x_var
$ x_var=-15

$ x_str=20
$ echo $((x_var + x_str))
5

$ x_str=0x15
$ echo $((x_var + x_str))
6
```

*When does -15 + 15 = 6 ?*
*What might be printed if x_str = 017 ?*
*What about x_str = 2#1111 ?*

type are



- Bash will convert strings containing hexadecimal and octal values.

# VARIABLES — OPERATORS

### Arithmetic

| | |
|---|---|
| **+** | addition |
| **-** | subtraction |
| ***** | multiplication |
| **/** | division |
| **%** | modulo |
| ****** | exponentiation |

### Bitwise

| | |
|---|---|
| **&** | and |
| **\|** | or |
| **^** | xor |
| **~** | negation |
| **<<** | shift left |
| **>>** | shift right |

### Other

| | |
|---|---|
| **expr ? expr : expr** | ternary operator |
| **( expr )** | sub-expression (precedence) |

(see http://wiki.bash-hackers.org/syntax/arith_expr)

# SPECIAL VARIABLES

| | |
|---|---|
| **$?** | exit status of last program executed |
| **$$** | pid of this program |
| **$!** | pid of last-started background job |
| **$#** | number of command line aguments |
| **$*** | command line arguments as string† |
| **$@** | command line arguments as string† |
| **$0** | name of this program |
| **$1 .. $9** | command line argument 1, 2, …, 9 |
| **$_** | starts as full path to script, changes to last command's argument list |

† The $* and $@ behave differently when inside double quotes.

# SPECIAL VARIABLES

There are only nine positional variables ($1 .. $9) but many programs accept more than nine arguments. The ***shift*** command discards $1 and moves all positional arguments down one index — including moving the tenth argument into $9.

| | |
|---|---|
| | uted |
| | |
| | job |
| | ents |
| | ring† |
| **$@** | command line arguments as string† |
| **$0** | name of this program |
| **$1 .. $9** | command line argument 1, 2, …, 9 |
| **$_** | starts as full path to script, changes to last command's argument list |

† The $* and $@ behave differently when inside double quotes.

# SPECIAL VARIABLES

```
#!/bin/bash

echo "The full path to this script is $_"
echo "This script is named $0 and is running with pid $$"
echo "You provided $# arguments"
echo "What does \$_ equal now: $_"
echo "The first two arguments are $1 and $2"
echo "The full argument list:  $@"
echo "A printf with \$*:"
printf "  %s\n" "$*"
echo "A printf with \$@:"
printf "  %s\n" "$@"
```

# SPECIAL VARIABLES

```
$ my_first_script "a b c" "d e f" g h i
The full path to this script is ./my_first_script
This script is named ./my_first_script and is running with pid 44108
You provided 5 arguments
What does $_ equal now: You provided 5 arguments
The first two arguments are a b c and d e f
The full argument list:  a b c d e f g h i
A printf with $*:
  a b c d e f g h i
A printf with $@:
  a b c
  d e f
  g
  h
  i
```

# SPECIAL VARIABLES

```
$ my_first_script "a b c" "d e f" g h i
The full path to this script is ./my_first_script
This sc   pt is named ./my_first_script and is running with pid 44108
You
What
The
The
A pr
  a
A pr
  a
  d
  g
  h
  i
```

*What directory must be present on my PATH for this to have worked?*

# CONDITIONALS

- If variables can take on arbitrary values, then it is important to be able to test their value.
  - A *conditional* evaluates an expression and executes a different set of statements based upon its value.
  - Implies *branching*: out-of-sequence execution of program code

# CONDITIONALS

- Integral to using conditionals is making use of logical expressions.
  - Test the value of an integer variable
  - Test the value of a string variable
    - Treat the string as a filepath and test file metadata
  - Create complex expressions using logic operators

# CONDITIONALS

- Logical expression review:

**Integer Comparison**

| | |
|---|---|
| ***a* -eq *b*** | equal |
| ***a* -ne *b*** | not equal |
| ***a* -gt *b*** | greater than |
| ***a* -ge *b*** | greater than or equal to |
| ***a* -lt *b*** | less than |
| ***a* -le *b*** | less than or equal to |

# CONDITIONALS

- Logical expression review:

**String Comparison**

| | |
|---|---|
| ***a* = *b*** | equal |
| ***a* != *b*** | not equal |
| **-z a** | variable exists and is empty |
| **-n *a*** | variable exists and is non-empty |
| ***a*** | equivalent to '-n *a*' |

# CONDITIONALS

- Logical expression review:

**Filepath Tests†**

| | |
|---|---|
| **-r *a*** | path is readable by user |
| **-w a** | path is writable by user |
| **-x a** | path is executable by user |
| **-f *a*** | path exists and is a file |
| **-d *a*** | path exists and is a directory |
| **-e a** | path exists |

† See the "man test" for additional filepath tests.

# CONDITIONALS

- Logical expression review:

**Logic Operators**

| | |
|---|---|
| ***expr* -a *expr*** | both expressions evaluate to true |
| ***expr* -o *expr*** | either expression evaluates to true |
| **! *expr*** | logical negation |
| **( *expr* )** | compound grouping |

# CONDITIONALS

- Newer versions of Bash have extended tests
  - Double-bracket syntax
  - Less dependent on proper quoting
  - Accepts && and || operators
  - Regular expression matching

```
$ if [[ ( -n $str1 && -n $str2 ) || $int1 -gt 4 ]]; then …

$ if [[ $str1 =~ ^hpc ]]; then …
```

# CONDITIONALS

- Arithmetic expressions also valid
  - Enclose in double parentheses

```
$ if (( i < 0 )); then echo "Negative"; fi

$ if (( i**2 < 100 )); then echo "Square of $i is less than 100."; fi
```

# CONDITIONALS

- Simple conditional
  - If a logical expression is true, perform the following commands…
  - …otherwise, continue executing after those commands.

```
if expr; then
  # When expression is true, do everything up to the "fi"
    :
fi
# Continue here…
```

# CONDITIONALS

- Conditionals branch based upon an expression that evaluates to zero = true, non-zero = false
  - Most programming languages equate zero with false
  - Shell's primary job is executing programs
    - Unix programs return zero on success, non-zero on error
    - Behavior of conditionals follows this interpretation of success/failure

# CONDITIONALS

- Conditionals branch based upon an expression that evaluates to zero or non-zero (true or false).

```
#!/bin/bash

x=0

if [ $x -eq 0 ]; then
  echo "x equals 0"
fi

if [ $x -eq 1 ]; then
  echo "x does not equal 0"
fi
```

```
#!/bin/bash

x=0

if (( x == 0 )); then
  echo "x equals 0"
fi

if (( x == 1 )); then
  echo "x does not equal 0"
fi
```

# CONDITIONALS

- Conditionals branch based upon an expression that evaluates to zero or non-zero (true or false).

```
#!/bin/bash

x=0

if [ $x -eq 0 ]; then
  echo "x equals 0"
fi

if [ $x -eq 1 ]; then
  echo "x does not equal 0"
fi
```

$? = 0

$? = 1

```
#!/bin/bash

x=0

if (( x == 0 )); then
  echo "x equal
fi

if (( x == 1 )); then
  echo "x does not equal 0"
fi
```

$? = 0

$? = 1

# CONDITIONALS

- Conditionals branch based upon an expression that evaluates to zero or non-zero (true or false).

```
$ x=0

$ [ $x -eq 0 ]
$ echo $?
0

$ [ $x -eq 1 ]
$ echo $?
1
```

# CONDITIONALS

- Example:  required argument count

```
#!/bin/bash

if [ $# -lt 3 ]; then
  echo "usage:"
  echo ""
  echo "  $0 <input1> <input2> <output>"
  echo ""
  exit 1
fi

cat "$1" "$2" > "$3"
```

# CONDITIONALS

- Example: required argument count

```
#!/bin/bash

if [ $# -lt 3 ]; then
  echo "usage:"
  echo ""
  echo "  $0 <input1> <input2> <output>"
  echo ""
  exit 1
fi

cat "$1" "$2" > "$3"
```

*Why did I put the variables in quotes?*

# CONDITIONALS

- Example:  required argument count, files exist

```
#!/bin/bash

if [ $# -lt 3 ]; then
  echo "usage:"
  echo ""
  echo "  $0 <input1> <input2> <output>"
  echo ""
  exit 1
fi

if [ ! -f "$1" -o ! -r "$1" ]; then
  echo "ERROR:  invalid file specified: $1"
  exit 1
fi

if [ ! -f "$2" -o ! -r "$2" ]; then
  echo "ERROR:  invalid file specified: $2"
  exit 1
fi

cat "$1" "$2" > "$3"
```

# CONDITIONALS

- *else* : …execute this code if expression is false…

```
if expr; then
  # When expression is true, do everything up to the "else"
    :
else
  # When expression is false, do everything up to the "fi"
    :
fi
# Continue here…
```

# CONDITIONALS

- *else* : Executing code when conditional evaluates false.

```
   :
if [ ! -f "$1" ]; then
  echo "ERROR:  file does not exist: $1"
  exit 1
else
  # The file exists, but can I actually read it?
  if [ ! -r "$1" ]; then
    echo "ERROR:  file is not readable: $1"
    exit 1
  fi
fi
   :
```

- Conditionals can be *nested*

# CONDITIONALS

- *elif* : Multi-level conditional — sequence of tests versus nesting of tests

```
   :
if [ ! -f "$1" ]; then
  echo "ERROR:  file does not exist: $1"
  exit 1
elif [ ! -r "$1" ]; then
  echo "ERROR:  file is not readable: $1"
  exit 1
elif [ ! -s "$1" ]; then
  echo "WARNING:  file is empty: $1"
fi
   :
```

# LOOPS

- Scripting is useful because it minimizes one's retyping of oft-used shell code
- In like fashion, loops allow a sequence of statements to be executed zero or more times
  - Iterate over a set of items
  - Iterate a fixed number of times
  - Iterate until a conditional is satisfied
    - Iterate indefinitely

# LOOPS — SET OF ITEMS

- Given a string containing *words* separated by whitespace, perform a sequence of statements for each *word*

```
for name in words; do
  # On each iteration, variable name contains next word
     :
done
# Continue here…
```

# LOOPS — SET OF ITEMS

- Given a string containing *words* separated by whitespace, perform a sequence of statements for each *word*

```
$ for w in a b c "d e f" g "h i j"; do echo $w; done
a
b
c
d e f
g
h i j
```

# LOOPS — SET OF ITEMS

- Given a string containing *words* separated by whitespace, perform a sequence of statements for each *word*

```
#!/bin/bash

for w in "$@"; do
  echo $w
done
```

```
#!/bin/bash

for w in $@; do
  echo $w
done
```

```
$ ./echo_args a b "c d e"
a
b
c d e
```

```
$ ./echo_args a b "c d e"
a
b
c
d
e
```



- Iterate over the arguments to the script
- Illustrates the difference between quoted and unquoted use of $@

# LOOPS — FIXED COUNT

- Perform a sequence of statements, exit loop when conditional expression is **false**

```
while expr; do
  # The expr evaluated to false, do these statements…
     :
   # …then re-test expr
done
# Continue here…
```

# LOOPS — FIXED COUNT

- Perform a sequence of statements *n* times

```
#!/bin/bash

i=0
while (( $i < 5 )); do
  printf "%d\t%d\n" $i $((i*2+1))
  i=$((i+1))
done
```

```
$ ./multiply
0    1
1    3
2    5
3    7
4    9
```

# LOOPS — FIXED COUNT

- Bash also has a C-style *for* loop construct:

```
# Evaluate expr1 as an arithmetic expression
for (( expr1 ; expr2 ; expr3 )); do
  # Evaluate expr2 as an arithmetic expression and exit if false
  # Otherwise do these statements…
     :
  # …then evaluate expr3 as an arithmetic expression and loop
done
# Continue here…
```

# LOOPS — FIXED COUNT

- Bash also has a C-style *for* loop construct:

```
#!/bin/bash

for (( i=0 ; i < 5 ; i++ )); do
  printf "%d\t%d\n" $i $((i*2+1))
done
```

```
$ ./multiply
0    1
1    3
2    5
3    7
4    9
```

# LOOPS — INDEFINITE

- Loops may also run indefinitely (usually until a condition is met)
  - Exit via one or more conditionals inside loop

```
#!/bin/bash

while (( 0 == 0 )); do
  echo "Trying to ping hostname.domain.net…"
  ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
  if [ $? -eq 0 ]; then
    break
  fi
  sleep 5
done
echo "I was able to ping hostname.domain.net."
```

# LOOPS — INDEFINITE

- Loops may also run indefinitely (or until a condition is met)
  - Exit via loop's logical expression

```
#!/bin/bash

echo "Trying to ping hostname.domain.net…"
ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
while [ $? -ne 0 ]; do
  sleep 5
  echo "Trying to ping hostname.domain.net…"
  ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
done
echo "I was able to ping hostname.domain.net."
```

# LOOPS — INDEFINITE

- Related to *while* loop is *until* loop
  - Exit loop when expression is **true**

```
#!/bin/bash

echo "Trying to ping hostname.domain.net…"
ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
until [ $? -eq 0 ]; do
  sleep 5
  echo "Trying to ping hostname.domain.net…"
  ping -c 1 -t 1 hostname.domain.net > /dev/null 2>&1
done
echo "I was able to ping hostname.domain.net."
```

# PUTTING IT ALL TOGETHER

- Write a script that will accept any number of file paths.
  - For each path, determine if the path exists or not
  - If it exists, determine whether it's a directory, regular file, or other type of entity

# PUTTING IT ALL TOGETHER

1. Loop over an arbitrary number of arguments

# PUTTING IT ALL TOGETHER

```
# So long as we have at one or more arguments, do the following loop:
while [ $# -gt 0 ]; do
  # The path we're interested in is $1:

  # Discard $1 and move all other arguments down one index:
  shift
done
```

# PUTTING IT ALL TOGETHER

1. Loop over an arbitrary number of arguments
2. For each argument, does path exist?

# PUTTING IT ALL TOGETHER

```
# So long as we have at one or more arguments, do the following loop:
while [ $# -gt 0 ]; do
  # The path we're interested in is $1:
  if [ -e "$1" ]; then

  else
    echo "ERROR:  path does not exist: $1"
  fi

  # Discard $1 and move all other arguments down one index:
  shift
done
```

# PUTTING IT ALL TOGETHER

1. Loop over an arbitrary number of arguments
2. For each argument, does path exist?
3. If path exists, what kind is it?

# PUTTING IT ALL TOGETHER

```
#!/bin/bash

# So long as we have at one or more arguments, do the following loop:
while [ $# -gt 0 ]; do
  # The path we're interested in is $1:
  if [ -e "$1" ]; then
    # What is it?
    if [ -d "$1" ]; then
      file_kind='directory'
    elif [ -f "$1" ]; then
      file_kind='regular file'
    else
      file_kind='other'
    fi
    printf "%-20s %s\n" "$file_kind" "$1"
  else
    echo "ERROR:  path does not exist: $1"
  fi

  # Discard $1 and move all other arguments down one index:
  shift
done
```

# PUTTING IT ALL TOGETHER

```
$ ls -l ./file_tests
-rw-r--r--  1 frey  everyone  483 Jan 21 21:23 ./file_tests

$ chmod u+x ./file_tests

$ ls -l ./file_tests
-rwxr--r--  1 frey  everyone  483 Jan 21 21:23 ./file_tests

$ ./file_tests /opt /opt/icu /opt/icu/attic /opt/icu/attic/icu4c-4_0-src.tgz \
> /opt/non-exist /dev
directory            /opt
directory            /opt/icu
directory            /opt/icu/attic
regular file         /opt/icu/attic/icu4c-4_0-src.tgz
ERROR:  path does not exist: /opt/non-exist
directory            /dev
```

# PUTTING IT ALL TOGETHER

- Additional pieces
  - If no arguments provided, show help text
  - Options:
    - --keep-running : path doesn't exist, show error but don't exit

# PUTTING IT ALL TOGETHER

```
#!/bin/bash

# No arguments?  User doesn't know what he's doing!
if [ $# -eq 0 ]; then
  cat <<EOT
usage:

  $0 {options} [path1] {[path2] .. [pathn]}

 options:

   --keep-running        do not exit on error

EOT
  exit 1
fi

# Exit on error:
keep_running=0
```

# PUTTING IT ALL TOGETHER

```
# Check for options:
while [ $# -gt 0 ]; do
  if [[ $1 =~ ^-- ]]; then
    if [ "$1" == '--keep-running' ]; then
      keep_running=1
    elif [ "$1" == '--' ]; then
      shift
      break
    else
      echo "ERROR:  unknown option: $1"
      exit 1
    fi
  else
    break
  fi
  shift
done
```

# PUTTING IT ALL TOGETHER

```
# So long as we have one or more arguments, do the following loop:
while [ $# -gt 0 ]; do
  # The path we're interested in is $1:
  if [ -e "$1" ]; then
    # What is it?
    if [ -d "$1" ]; then
      file_kind='directory'
    elif [ -f "$1" ]; then
      file_kind='regular file'
    else
      file_kind='other'
    fi
    printf "%-20s %s\n" "$file_kind" "$1"
  else
    echo "ERROR:  path does not exist: $1"
    if [ $keep_running -eq 0 ]; then
      exit 1
    fi
  fi

  # Discard $1 and move all other arguments down one index:
  shift
done
```