# Shell Scripting – Part 1

Le Yan/Alex Pacheco

HPC User Services @ LSU

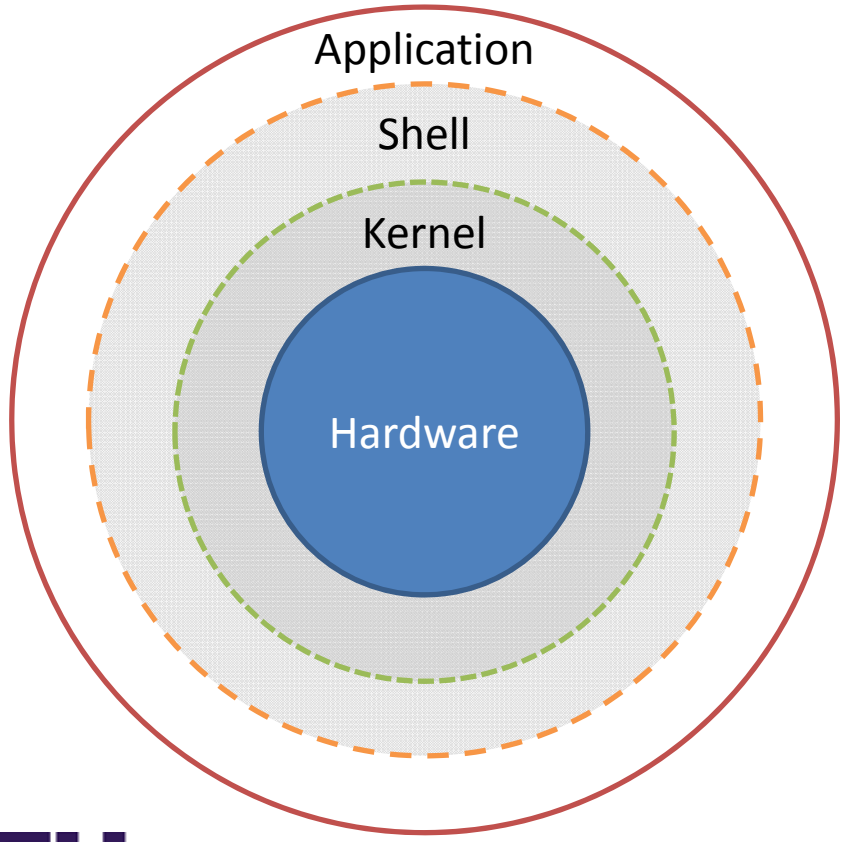
# Shell Scripting

- Part 1 (today)
  - Simple topics such as creating and executing simple shell scripts, arithmetic operations, flow control, command line arguments and functions.
- Part 2 (March 4th)
  - Advanced topics such as regular expressions and text processing tools (grep, sed, awk etc.)

# Outline

- Recap of Linux 101
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Topics Preview

# What Do Operating Systems Do?




- Operating systems work as a bridge between hardware and applications
  - Kernel: hardware drivers etc.
  - Shell: user interface to kernel
  - Some applications (system utilities)

: Operating System

# Kernel

- Kernel
  - The kernel is the core component of most operating systems
  - Kernel's responsibilities include managing the system's resources
  - It provides the lowest level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its functions
  - It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls

# Shell

- Shell
  - The command line interface is the primary user interface to Linux/Unix operating systems.
  - Each shell has varying capabilities and features and the users should choose the shell that best suits their needs
  - The shell can be deemed as an application running on top of the kernel and provides a powerful interface to the system.

# Type of Shell

- sh (Bourne Shell)
  - Developed by Stephen Bourne at AT\&T Bell Labs
- csh (C Shell)
  - Developed by Bill Joy at University of California, Berkeley
- ksh (Korn Shell)
  - Developed by David Korn at AT&T Bell Labs
  - Backward-compatible with the Bourne shell and includes many features of the C shell
- **bash (Bourne Again Shell)**
  - Developed by Brian Fox for the GNU Project as a free software replacement for the Bourne shell
  - Default Shell on Linux and Mac OSX
  - The name is also descriptive of what it did, bashing together the features of sh, csh and ksh
- **tcsh (TENEX C Shell)**
  - Developed by Ken Greer at Carnegie Mellon University
  - It is essentially the C shell with programmable command line completion, command-line editing, and a few other features.

# Shell Comparison

| Software | sh | csh | ksh | bash | tcsh |
|---|---|---|---|---|---|
| Programming language | y | y | y | y | y |
| Shell variables | y | y | y | y | y |
| Command alias | n | y | y | y | y |
| Command history | n | y | y | y | y |
| Filename autocompletion | n | y* | y* | y | y |
| Command line editing | n | n | y* | y | y |
| Job control | n | y | y | y | y |

*: not by default

*http://www.cis.rit.edu/class/simg211/unixintro/Shell.html*

# File Editing

- The two most commonly used editors on Linux/Unix systems are:
  - `vi` or `vim` (vi improved)
  - `emacs`
- `vi/vim` is installed by default on Linux/Unix systems and has only a command line interface (CLI).
- `emacs` has both a CLI and a graphical user interface (GUI).
  - if emacs GUI is installed then use `emacs -nw` to open file in console
- Other editors you may come across: `kate, gedit, gvim, pico, nano, kwrite`
- To use `vi` or `emacs` is your choice, but you need to know one of them
- **For this tutorial, we assume that you already know how to edit a file with a command line editor**

# Variables

- Linux allows the use of variables
  - Similar to programming languages
- A variable is a named object that contains data
  - Number, character or string
- There are two types of variables: ENVIRONMENT and user defined
- Environment variables provide a simple way to share configuration settings between multiple applications and processes in Linux
  - Environment variables are often named using all uppercase letters
  - Example: `PATH`, `LD_LIBRARY_PATH`, `DISPLAY` etc.
- To reference a variable, prepend $ to the name of the variable, e.g. `$PATH`, `$LD_LIBRARY_PATH`
  - Example: `$PATH`, `$LD_LIBRARY_PATH`, `$DISPLAY` etc.

# Variables Names

- Rules for variable names
  - Must start with a letter or underscore
  - Number can be used anywhere else
  - Do not use special characters such as @,#,%,$
  - (again) They are case sensitive
  - Example
    - Allowed: `VARIABLE, VAR1234able, var_name, _VAR`
    - Not allowed: `1var, %name, $myvar, var@NAME`

# Editing Variables (1)

- How to assign values to variables depends on the shell

| Type | sh/ksh/bash | csh/tcsh |
|---|---|---|
| Shell | `name=value` | `set name=value` |
| Environment | `export name=value` | `setenv name=value` |

- Shell variables is only valid within the current shell, while environment variables are valid for all subsequently opened shells.

# Editing Variables (2)

- Example: to add a directory to the PATH variable

```
sh/ksh/bash: export PATH=/path/to/executable:${PATH}
csh/tcsh: setenv PATH /path/to executable:${PATH}
```

  - `sh/ksh/bash`: no spaces except between export and PATH
  - `csh/tcsh`: no "=" sign
  - Use colon to separate different paths
  - The order matters: if you have a customized version of a software say perl in your home directory, if you append the perl path to PATH at the end, your program will use the system wide perl not your locally installed version

# Basic Commands

- Command is a directive to a computer program acting as an interpreter of some kind, in order to perform a specific task
- Command prompt is a sequence of characters used in a command line interface to indicate readiness to accept commands
  - Its intent is literally to prompt the user to take action
  - A prompt usually ends with one of the characters $,%#,:,> and often includes information such as user name and the current working directory
- Each command consists of three parts: name, options and arguments

# List of Basic Commands

| Name | Function |
|---|---|
| `ls` | Lists files and directories |
| `cd` | Changes the working directory |
| `mkdir` | Creates new directories |
| `rm` | Deletes files and directories |
| `cp` | Copies files and directories |
| `mv` | Moves or renames files and directories |
| `pwd` | prints the current working directory |
| `echo` | prints arguments to standard output |
| `cat` | Prints file content to standard output |
| | |
| | |

# File Permission (1)

- Since *NIX OS's are designed for multi user environment, it is necessary to restrict access of files to other users on the system.
- In *NIX OS's, you have three types of file permissions
  - Read (r)
  - Write (w)
  - Execute (x)
- for three types of users
  - User (u) (owner of the file)
  - Group (g) (group owner of the file)
  - World (o) (everyone else who is on the system)

# File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins    4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root     16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root      4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins    4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins      12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins    4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins  627911 Aug 28 10:13 a.out
```

- The first column indicates the type of the file
  - d for directory
  - l for symbolic link
  - - for normal file

# File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins    4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root     16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root      4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins    4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins      12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins    4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins  627911 Aug 28 10:13 a.out
```

- The next nine columns can be grouped into three triads, which indicates what the owner, the group member and everyone else can do

# File Permission (2)

```
[lyan1@mike2 ~]$ ls -al
total 4056
drwxr-xr-x  45 lyan1 Admins    4096 Sep  2 13:30 .
drwxr-xr-x 509 root  root     16384 Aug 29 13:31 ..
drwxr-xr-x   3 lyan1 root      4096 Apr  7 13:07 adminscript
drwxr-xr-x   3 lyan1 Admins    4096 Jun  4  2013 allinea
-rw-r--r--   1 lyan1 Admins      12 Aug 12 13:53 a.m
drwxr-xr-x   5 lyan1 Admins    4096 May 28 10:13 .ansys
-rwxr-xr-x   1 lyan1 Admins  627911 Aug 28 10:13 a.out
```

- We can also use weights to indicate file permission
  - r=4, w=2, x=1
  - Example: rwx = 4+2+1 = 7, r-x = 4+1 = 5, r-- = 4
  - This allows us to use three numbers to represent the permission
  - Example: rwxr-xr-w = 755

# Input & Output Commands (1)

- The basis I/O statement are echo for displaying to screen and read for reading input from screen/keyboard/prompt
- **echo**
  - The `echo arguments` command will print arguments to screen or standard output, where `arguments` can be a single or multiple variables, string or numbers
- **read**
  - The `read` statement takes all characters typed until the Enter key is pressed
  - Usage: `read <variable name>`
  - Example: `read name`

# Input & Output Commands (2)

- Examples

```
[lyan1@mike2 ~]$ echo $SHELL
/bin/bash
[lyan1@mike2 ~]$ echo Welcome to HPC      training
Welcome to HPC training
[lyan1@mike2 ~]$ echo "Welcome to HPC     training"
Welcome to HPC     training
```

- By default, `echo` eliminates redundant whitespaces (multiple spaces and tabs) and replaces it with a single whitespace between arguments.
  - To include redundant whitespace, enclose the arguments within double quotes

# I/O Redirection

- There are three file descriptors for I/O streams (remember everything is a file in Linux)
  - STDIN: Standard input
  - STDOUT: standard output
  - STDERR: standard error
- 1 represents STDOUT and 2 represents STDERR
- I/O redirection allows users to connect applications
  - <: connects a file to STDIN of an application
  - >: connects STDOUT of an application to a file
  - >>: connects STDOUT of an application by appending to a file
  - |: connects the STDOUT of an application to STDIN of another application.

# I/O Redirection Examples

- Write STDOUT to file: `ls -l > ls-l.out`
- Write STDERR to file: `ls -l &2 > ls-l.err`
- Write STDERR to STDOUT: `ls -l 2>&1`
- Send STDOUT as STDIN for another application: `ls -l | less`

# Outline

- Recap of Linux 101
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Topics Preview

# Scripting Languages

- A script is a program written for a software environment that automate the execution of tasks which could alternatively be executed one-by-one by a human operator.
- Shell scripts are a series of shell commands put together in a file
  - When the script is executed, it is as if someone type those commands on the command line
- The majority of script programs are ``quick and dirty'', where the main goal is to get the program written quickly.
  - Compared to programming languages, scripting languages do not distinguish between data types: integers, real values, strings, etc.
  - Might not be as efficient as programs written in C and Fortran, with which source files need to be compiled to get the executable

# Startup Scripts

- When you login to a *NIX computer, shell scripts are automatically loaded depending on your default shell
- `sh/ksh` (in the specified order)
  - `/etc/profile`
  - `$HOME/.profile`
- `bash` (in the specified order)
  - `/etc/profile` (for login shell)
  - `/etc/bashrc or /etc/bash/bashrc`
  - `$HOME/.bash_profile` (for login shell)
  - `$HOME/.bashrc`
- `csh/tcsh` (in the specified order)
  - `/etc/csh.cshrc`
  - `$HOME/.tcshrc`
  - `$HOME/.cshrc` (if `.tcshrc` is not present)
- `.bashrc, .tcshrc, .cshrc, .bash_profile` are script files where users can define their own aliases, environment variables, modify paths etc.

# An Example

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
alias c="clear"
alias rm="/bin/rm -i"
alias psu="ps -u apacheco"
alias em="emacs -nw"
alias ll="ls -lF"
alias la="ls -al"
export PATH=/home/apacheco/bin:${PATH}
export g09root=/home/apacheco/Software/Gaussian09
export GAUSS_SCRDIR=/home/apacheco/Software/scratch
source $g09root/g09/bsd/g09.profile

export TEXINPUTS=.:/usr/share/texmf//:/home/apacheco/LaTeX//:${
    TEXINPUTS}
export BIBINPUTS=.:/home/apacheco/TeX//:${BIBINPUTS}
```

# Writing and Executing a Script

- Three steps
  - Create and edit a text file (`hello.sh`)

```
#!/bin/bash
# My First Script
echo "Hello World!"
```

  - Set the appropriate permission

```
~/Tutorials/BASH/scripts> chmod 755 hello.sh
```

  - Execute the script

```
~/Tutorials/BASH/scripts> ./hello.sh
Hello World!
```

# Components Explained

```
#!/bin/bash
# My First Script
echo "Hello World!"
```

- The first line is called the "Shebang" line. It tells the OS which interpreter to use. In the current example, bash
  - For tcsh, it would be: `#!/bin/tcsh`
- The second line is a comment. All comments begin with "#".
- The third line tells the OS to print "Hello World!" to the screen.

# Special Characters (1)

| | |
|---|---|
| # | Starts a comment line. |
| $ | Indicates the name of a variable. |
| \ | Escape character to display next character literally |
| {} | Used to enclose name of variable |
| ; | Command separator. Permits putting two or more commands on the same line. |
| ;; | Terminator in a case option |
| . | "dot" command. Equivalent to `source` (for bash only) |

# Special Characters (2)

| | |
|---|---|
| `$?` | Exit status variable. |
| `$$` | Process ID variable. |
| `[]` | Test expression. |
| `[[ ]]` | Test expression, more flexible than [] |
| `$[], $(())` | Integer expansion |
| `\|\|, &&, !` | Logical OR, AND and NOT |
| | |

# Quotation

- Double quotation
  - Enclosed string is expanded
- Single quotation
  - Enclosed string is read literally
- Back quotation
  - Enclose string is executed as a command

# Quotation - Examples

```
[lyan1@mike2 ~]$ str1="I am $USER"
[lyan1@mike2 ~]$ echo $str1
I am lyan1
[lyan1@mike2 ~]$ str2='I am $USER'
[lyan1@mike2 ~]$ echo $str2
I am $USER
[lyan1@mike2 ~]$ str3=`echo $str2`
[lyan1@mike2 ~]$ echo $str3
I am $USER
```

# Quotation – More Examples

```
#!/bin/bash

HI=Hello

echo HI           # displays HI
echo $HI          # displays Hello
echo \$HI         # displays $HI
echo "$HI"        # displays Hello
echo '$HI'        # displays $HI
echo "$HIAlex"    # displays nothing
echo "${HI}Alex"  # displays HelloAlex
echo `pwd`        # displays working directory
echo $(pwd)       # displays working directory

~/Tutorials/BASH/scripts/day1/examples> ./quotes.sh
HI
Hello
$HI
Hello
$HI

HelloAlex
/home/apacheco/Tutorials/BASH/scripts/day1/examples
/home/apacheco/Tutorials/BASH/scripts/day1/examples
~/Tutorials/BASH/scripts/day1/examples>
```

# Outline

- Recap of Linux 101
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Topics Preview

# Arithmetic Operations (1)

- You can carry out numeric operations on integer variables

| Operation | Operator |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Exponentiation | ** (bash only) |
| Modulo | % |

# Arithmetic Operations (2)

- bash
  - `$((…))` or `$[…]` commands
    - Addition: `$((1+2))`
    - Multiplication: `$[$a*$b]`
  - Or use the `let` command: `let c=$a-$b`
  - Or use the `expr` command: `c='expr $a - $b'`
  - You can also use C-style increment operators:

    `let c+=1` or `let c--`

# Arithmetic Operations (3)

- tcsh
  - Add two numbers: `@ x = 1 + 2`
  - Divide two numbers: `@ x = $a / $b`
  - You can also use the expr command: `set c = ‘expr $a % $b‘`
  - You can also use C-style increment operators:
    `@ x -= 1` or `@ x++`
- Note the use of space
  - bash: space required around operator in the `expr` command
  - tcsh: space required between @ and variable, around = and numeric operators.

# Arithmetic Operations (4)

- For floating numbers
  - You would need an external calculator like the GNU `bc`
    - Add two numbers

```
echo "3.8 + 4.2" | bc
```

    - Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

    - Call `bc` directly:

```
bc <<< “scale=5; 2/5”
```

    - Use `bc -l` to see result in floating point at max scale:

```
bc -l <<< "2/5"
```

# Arrays (1)

- bash and tcsh supports one-dimensional arrays
- Array elements may be initialized with the `variable[i]` notation: `variable[i]=1`
- Initialize an array during declaration
  - **bash**: `name=(firstname 'last name')`
  - **tcsh**: `set name = (firstname 'last name')`
- Reference an element `i` of an array `name`: `${name[i]}`
- Print the whole array
  - **bash**: `${name[@]}`
  - **tcsh**: `${name}`
- Print length of array
  - **bash**: `${#name[@]}`
  - **tcsh**: `${#name}`

# Arrays (2)

- Print length of element `i` of array `name`: `${#name[i]}`
  - Note: In **bash** `${#name}` prints the length of the first element of the array
- Add an element to an existing array
  - **bash** `name=(title ${name[@]})`
  - **tcsh** `set name = ( title "${name}")`
  - In the above **tcsh** example, title is first element of new array while the second element is the old array name
- Copy an array name to an array user
  - **bash** `user=(${name[@]})`
  - **tcsh** `set user = ( ${name} )`

# Arrays (3)

- Concatenate two arrays
  - **bash** `nameuser=(${name[@]} ${user[@]})`
  - **tcsh** `set nameuser=( "${name}" "${user}" )`
- Delete an entire array: `unset name`
- Remove an element i from an array
  - **bash**
```
unset name[i]
```
  - **tcsh**
```
@ j = $i - 1
@ k = $i + 1
set name = ( "${name[1-$j]}" "${name[$k-]}" )
```
- Note
  - **bash**: array index starts from 0
  - **tcsh**: array index starts from 1

# Arrays (4)

**name.sh**

```
#!/bin/bash

echo "Print your first and last name"
read firstname lastname

name=($firstname $lastname)

echo "Hello " ${name[@]}

echo "Enter your salutation"
read title

echo "Enter your suffix"
read suffix

name=($title "${name[@]}" $suffix)
echo "Hello " ${name[@]}

unset name[2]
echo "Hello " ${name[@]}
```

```
~/Tutorials/BASH/scripts/day1/examples> ./name.sh
Print your first and last name
Alex Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

**name.csh**

```
#!/bin/tcsh

echo "Print your first name"
set firstname = $<
echo "Print your last name"
set lastname = $<

set name = ( $firstname $lastname)
echo "Hello " ${name}

echo "Enter your salutation"
set title = $<

echo "Enter your suffix"
set suffix = "$<"

set name = ($title $name $suffix )
echo "Hello " ${name}

@ i = $#name
set name = ( $name[1-2] $name[4-$i] )
echo "Hello " ${name}
```

```
~/Tutorials/BASH/scripts/day1/examples> ./name.csh
Print your first name
Alex
Print your last name
Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
the first
Hello  Dr. Alex Pacheco the first
Hello  Dr. Alex the first
```

# Flow Control

- Shell scripting languages execute commands in sequence similar to programming languages such as C and Fortran
  - Control constructs can change the order of command execution
- Control constructs in bash and tcsh are
  - Conditionals: `if`
  - Loops: `for, while, until`
  - Switches: `case, switch`

# if statement

- An if/then construct tests whether the exit status of a list of commands is 0, and if so, execute one or more commands

**bash**

```
if [ condition1 ]; then
  some commands
elif [ condition2 ]; then
  some commands
else
  some commands
fi
```

**tcsh**

```
if ( condition1 ) then
  some commands
else if ( condition2 ) then
  some commands
else
  some commands
endif
```

- Note the space between condition and the brackets
  - bash is very strict about spaces.
  - tcsh commands are not so strict about spaces
  - tcsh uses the `if-then-else` `if-else-endif` similar to Fortran

# File Tests

| Operation | bash | tcsh |
|---|---|---|
| File exists | `if [ -e .bashrc ]` | `if ( -e .tcshrc )` |
| File is a regular file | `if [ -f .bashrc ]` | |
| File is a directory | `if [ -d /home ]` | `if ( -d /home )` |
| File is not zero size | `if [ -s .bashrc ]` | `if ( ! -z .tcshrc )` |
| File has read permission | `if [ -r .bashrc ]` | `if ( -r .tcshrc )` |
| File has write permission | `if [ -w .bashrc ]` | `if ( -w .tcshrc )` |
| File has execute permission | `if [ -x .bashrc ]` | `if ( -x .tcshrc )` |

# Integer Comparisons

| Operation | bash | tcsh |
|---|---|---|
| Equal to | `if [ 1 -eq 2]` | `if (1 == 2)` |
| Not equal to | `if [ $a -ne $b ]` | `if ($a != $b)` |
| Greater than | `if [ $a -gt $b ]` | `if ($a > $b)` |
| Greater than or equal to | `if [ 1 -ge $b ]` | `if (1 >= $b)` |
| Less than | `if [ $a -lt 2 ]` | `if ($a < 2)` |
| Less than or equal to | `if [[ $a -le $b ]]` | `if ($a <= $b)` |

# String Comparisons

| Operation | bash | tcsh |
|---|---|---|
| Equal to | `if [ $a == $b ]` | `if ($a == $b)` |
| Not equal to | `if [ $a != $b ]` | `if ($a != $b)` |
| Zero length or null | `if [ -z $a ]` | `if ($%a == 0)` |
| Non zero length | `if [ -n $a ]` | `if ($%a > 0)` |
| | | |

# Logical Operators

| Operation | Example |
|---|---|
| ! (NOT) | `if [ ! -e .bashrc ]` |
| && (AND) | `if [ -f .bashrc ] && [ -s .bashrc ]` |
| \| (OR) | `if [[ -f .bashrc \|\| -f .bash_profile ]]`<br>`if ( -e /.tcshrc && ! -z /.tcshrc )` |

# Examples

```
read a
if [[ "$a" -gt 0 &&  "$a" -lt 5 ]]; then
  echo "The value of $a lies somewhere between 0 and 5"
fi
OR
if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]; then
  echo "The value of $a lies somewhere between 0 and 5"
fi
```

```
set a = $<
if ( "$a" > 0  &&  "$a" < 5 ) then
  echo "The value of $a lies somewhere between 0 and 5"
endif
```

# Loop Constructs

- A loop is a block of code that iterates a list of commands as long as the loop control condition is evaluated to true
- Loop constructs
  - bash: `for`, `while` and `until`
  - tcsh: `foreach` and `while`

# For Loop - bash

- The `for` loop is the basic looping construct in **bash**

```
for arg in list
do
  some commands
done
```

- The `for` and `do` lines can be written on the same line:
  `for` *`arg`* `in` *`list`*`;` `do`
- `for` loops can also use C style syntax

```
for i in $(seq 1 10)
do
  touch file${i}.dat
done
```

```
for i in $(seq 1 10); do
  touch file${i}.dat
done
```

```
for ((i=1;i<=10;i++))
do
  touch file${i}.dat
done
```

# For Loop - tcsh

- The `foreach` loop is the basic looping construct in **tcsh**

```
foreach i (`seq 1 10`)
  touch file$i.dat
end
```

# While Loop

- The `while` construct tests for a condition at the top of a loop and keeps going as long as that condition is true.
- In contrast to a `for` loop, a `while` loop finds use in situations where the number of loop repetitions is not known beforehand.
- bash

```
while [ condition ]
do
  some commands
done
```

- tcsh

```
while ( condition )
  some commands
end
```

# While Loop - Example

**factorial.sh**

```
#!/bin/bash

read counter
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

**factorial.csh**

```
#!/bin/tcsh

set counter = $<
set factorial = 1
while ( $counter > 0 )
   @ factorial = $factorial * $counter
   @ counter -= 1
end
echo $factorial
```

# Until Loop

- The `until` construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of `while` loop)

```
until [ condition is true ]
do
   some commands
done
```

**factorial2.sh**

```
#!/bin/bash

read counter
factorial=1
until [ $counter -le 1 ]; do
  factorial=$[ $factorial * $counter ]
  if [ $counter -eq 2 ]; then
    break
  else
    let counter-=2
  fi
done
echo $factorial
```

- `for`, `while`, and `until` loops can be nested, to exit from the loop use the break command

**nestedloops.sh**

```
#!/bin/bash

## Example of Nested loops

echo "Nested for loops"
for a in $(seq 1 5) ; do
  echo "Value of a in outer loop:" $a
  for b in `seq 1 2 5` ; do
    c=$(($a*$b))
    if [ $c -lt 10 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 10"
      break
    fi
  done
done
echo "==================="
echo
echo "Nested for and while loops"
for ((a=1;a<=5;a++)); do
  echo "Value of a in outer loop:" $a
  b=1
  while [ $b -le 5 ]; do
    c=$(($a*$b))
    if [ $c -lt 5 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 5"
      break
    fi
    let b+=2
  done
done
echo "==================="
```

**nestedloops.csh**

```
#!/bin/tcsh

## Example of Nested loops

echo "Nested for loops"
foreach a (`seq 1 5`)
  echo "Value of a in outer loop:" $a
  foreach b (`seq 1 2 5`)
    @ c = $a * $b
    if ( $c <  10 ) then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 10"
      break
    endif
  end
end
echo "==================="
echo
echo "Nested for and while loops"
foreach a (`seq 1 5`)
  echo "Value of a in outer loop:" $a
  set b = 1
  while ( $b <= 5 )
    @ c = $a * $b
    if ( $c < 5 ) then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 5"
      break
    endif
    @ b = $b + 2
  end
end
echo "==================="
```

```
~/Tutorials/BASH/scripts/day1/examples> ./nestedloops.sh
Nested for loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
a * b = 1 * 5 = 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
a * b = 2 * 3 = 6
2 * 5 > 10
Value of a in outer loop: 3
a * b = 3 * 1 = 3
a * b = 3 * 3 = 9
3 * 5 > 10
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 10
Value of a in outer loop: 5
a * b = 5 * 1 = 5
5 * 3 > 10
----------------------

Nested for and while loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
1 * 5 > 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
2 * 3 > 5
Value of a in outer loop: 3
a * b = 3 * 1 = 3
3 * 3 > 5
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 5
Value of a in outer loop: 5
5 * 1 > 5
----------------------
```

```
~/Tutorials/BASH/scripts> ./day1/examples/nestedloops.csh
Nested for loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
a * b = 1 * 5 = 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
a * b = 2 * 3 = 6
2 * 5 > 10
Value of a in outer loop: 3
a * b = 3 * 1 = 3
a * b = 3 * 3 = 9
3 * 5 > 10
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 10
Value of a in outer loop: 5
a * b = 5 * 1 = 5
5 * 3 > 10
----------------------

Nested for and while loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
1 * 5 > 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
2 * 3 > 5
Value of a in outer loop: 3
a * b = 3 * 1 = 3
3 * 3 > 5
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 5
Value of a in outer loop: 5
5 * 1 > 5
----------------------
```

# Switching Constructs - bash

- The `case` and `select` constructs are technically not loops since they do not iterate the execution of a code block
- Like loops, however, they direct program flow according to conditions at the top or bottom of the block

**`case` construct**

```
case variable in
  "condition1")
    some command
    ;;
  "condition2")
    some other command
    ;;
esac
```

**`select` construct**

```
select variable [ list ]
do
  command
  break
done
```

# Switching Constructs - tcsh

- tcsh has the `switch` constructs

**`switch` construct**

```
switch (arg list)
  case "variable"
    some command
    breaksw
endsw
```

## dooper.sh

```
#!/bin/bash


echo "Print two numbers"
read num1 num2
echo "What operation do you want to do?"

operations='add subtract multiply divide
        exponentiate modulo all quit'
select oper in $operations ; do
  case $oper in
    "add")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      ;;
    "subtract")
      echo "$num1 - $num2 =" $[$num1 - $num2]
      ;;
    "multiply")
      echo "$num1 * $num2 =" $[$num1 * $num2]
      ;;
    "exponentiate")
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      ;;
    "divide")
      echo "$num1 / $num2 =" $[$num1 / $num2]
      ;;
    "modulo")
      echo "$num1 % $num2 =" $[$num1 % $num2]
      ;;
    "all")
      echo "$num1 + $num2 =" $[$num1 + $num2]
      echo "$num1 - $num2 =" $[$num1 - $num2]
      echo "$num1 * $num2 =" $[$num1 * $num2]
      echo "$num1 ** $num2 =" $[$num1 ** $num2]
      echo "$num1 / $num2 =" $[$num1 / $num2]
      echo "$num1 % $num2 =" $[$num1 % $num2]
    ;;
    *)
      exit
      ;;
  esac
done
```

## dooper.csh

```
#!/bin/tcsh


echo "Print two numbers one at a time"
set num1 = $<
set num2 = $<
echo "What operation do you want to do?"
echo "Enter +, -, x, /, % or all"
set oper = $<

switch ( $oper )
  case "x"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     breaksw
  case "all"
     @ sum = $num1 + $num2
     echo "$num1 + $num2 = $sum"
     @ diff = $num1 - $num2
     echo "$num1 - $num2 = $diff"
     @ prod = $num1 * $num2
     echo "$num1 * $num2 = $prod"
     @ ratio = $num1 / $num2
     echo "$num1 / $num2 = $ratio"
     @ remain = $num1 % $num2
     echo "$num1 % $num2 = $remain"
     breaksw
  case "*"
     @ result = $num1 $oper $num2
     echo "$num1 $oper $num2 = $result"
     breaksw
endsw
```

```
~/Tutorials/BASH/scripts> ./day1/examples/dooper.sh
Print two numbers
1 4
What operation do you want to do?
1) add 3) multiply  5) exponentiate  7) all
2) subtract 4) divide  6) modulo   8) quit
#? 7
1 + 4 = 5
1 - 4 = -3
1 * 4 = 4
1 ** 4 = 1
1 / 4 = 0
1 % 4 = 1
#? 8
```

```
~/Tutorials/BASH/scripts> ./day1/examples/dooper.csh
Print two numbers one at a time
1
5
What operation do you want to do?
Enter +, -, x, /, % or all
all
1 + 5 = 6
1 - 5 = -4
1 * 5 = 5
1 / 5 = 0
1 % 5 = 1
```

# Command Line Arguments (1)

- Similar to programming languages, bash and other shell scripting languages can also take command line arguments
  - Execute: `./myscript arg1 arg2 arg3`
  - Within the script, the positional parameters `$0, $1, $2, $3` correspond to `./myscript, arg1, arg2`, and `arg3`, respectively.
  - `$#`: number of command line arguments
  - `$*`: all of the positional parameters, seen as a single word
  - `$@`: same as `$*` but each parameter is a quoted string.
  - `shift N`: shift positional parameters from `N+1` to `$#` are renamed to variable names from `$1` to `$# - N + 1`
- In csh and tcsh
  - An array `argv` contains the list of arguments with `argv[0]` set to the name of the script
  - `#argv` is the number of arguments, i.e. length of `argv` array

## shift.sh

```
#!/bin/bash

USAGE="USAGE: $0 <at least 1 argument>"

if [[ "$#" -lt 1 ]]; then
   echo $USAGE
   exit
fi

echo "Number of Arguments: " $#
echo "List of Arguments: " $@
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 $*

while [ "$#"  -gt 0 ]; do
  echo "Argument List is: " $@
  echo "Number of Arguments: " $#
  shift
done
```

```
~/Tutorials/BASH/scripts/day1/examples> ./shift.sh $(seq 1 5)
Number of Arguments:  5
List of Arguments:  1 2 3 4 5
Name of script that you are running:  ./shift.sh
Command You Entered: ./shift.sh 1 2 3 4 5
Argument List is:  1 2 3 4 5
Number of Arguments:  5
Argument List is:  2 3 4 5
Number of Arguments:  4
Argument List is:  3 4 5
Number of Arguments:  3
Argument List is:  4 5
Number of Arguments:  2
Argument List is:  5
Number of Arguments:  1
```

## shift.csh

```
#!/bin/tcsh

set USAGE="USAGE: $0 <at least 1 argument>"

if ( "$#argv" < 1 ) then
   echo $USAGE
   exit
endif

echo "Number of Arguments: " $#argv
echo "List of Arguments: " ${argv}
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 ${argv}

while ( "$#argv"  > 0 )
  echo "Argument List is: " $*
  echo "Number of Arguments: " $#argv
  shift
end
```

```
~/Tutorials/BASH/scripts/day1/examples> ./shift.csh $(seq 1 5)
Number of Arguments:  5
List of Arguments:  1 2 3 4 5
Name of script that you are running:  ./shift.csh
Command You Entered: ./shift.csh 1 2 3 4 5
Argument List is:  1 2 3 4 5
Number of Arguments:  5
Argument List is:  2 3 4 5
Number of Arguments:  4
Argument List is:  3 4 5
Number of Arguments:  3
Argument List is:  4 5
Number of Arguments:  2
Argument List is:  5
Number of Arguments:  1
```

# Declare command

- Use the `declare` command to set variable and functions attributes
- Create a constant variable, i.e. read-only
  - `declare -r var`
  - `declare -r varName=value`
- Create an integer variable
  - `declare -i var`
  - `declare -i varName=value`
- You can carry out arithmetic operations on variables declared as integers

```
~/Tutorials/BASH> j=10/5 ; echo $j
10/5
~/Tutorials/BASH> declare -i j; j=10/5 ; echo $j
2
```

# Functions (1)

- Like “real” programming languages, bash has functions.
- A function is a code block that implements a set of operations, a “black box” that performs a specified task.
- Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

```
function function_name {
  command
}
OR
function_name () {
  command
}
```

## shift10.sh

```
#!/bin/bash

usage () {
  echo "USAGE: $0 [atleast 11 arguments]"
  exit
}

[[ "$#" -lt 11 ]] && usage

echo "Number of Arguments: " $#
echo "List of Arguments: " $@
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 $*
echo "First Argument" $1
echo "Tenth and Eleventh argument" $10 $11 ${10} ${11}

echo "Argument List is: " $@
echo "Number of Arguments: " $#
shift 9
echo "Argument List is: " $@
echo "Number of Arguments: " $#
```

```
~/Tutorials/BASH/scripts/day1/examples> ./shift10.sh `seq 1 2 22`
Number of Arguments:  11
List of Arguments:  1 3 5 7 9 11 13 15 17 19 21
Name of script that you are running:  ./shift10.sh
Command You Entered: ./shift10.sh 1 3 5 7 9 11 13 15 17 19 21
First Argument 1
Tenth and Eleventh argument 10 11 19 21
Argument List is:  1 3 5 7 9 11 13 15 17 19 21
Number of Arguments:  11
Argument List is:  19 21
Number of Arguments:  2
```

# Functions (2)

- You can also pass arguments to a function
- All function parameters can be accessed via `$1, $2, $3…`
- `$0` always point to the shell script name
- `$*` or $@ holds all parameters passed to a function
- $# holds the number of positional parameters passed to the function

# Functions (3)

- Array variable called FUNCNAME contains the names of all shell functions currently in the execution call stack.
- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- You can create a local variables using the local command

```
local var=value
local varName
```

- A function may recursively call itself even without use of local variables.

**factorial3.sh**

```
#!/bin/bash

usage () {
  echo "USAGE: $0 <integer>"
  exit
}

factorial() {
  local i=$1
  local f

  declare -i i
  declare -i f

  if [[ "$i" -le 2 && "$i" -ne 0 ]]; then
    echo $i
  elif [[ "$i" -eq 0 ]]; then
    echo 1
  else
    f=$(( $i - 1 ))
    f=$( factorial $f )
    f=$(( $f * $i ))
    echo $f
  fi
}

if [[ "$#" -eq 0 ]]; then
  usage
else
  for i in $@ ; do
    x=$( factorial $i )
    echo "Factorial of $i is $x"
  done
fi
```

```
~/Tutorials/BASH/scripts/day1/examples>./factorial3.sh 1 3 5 7 9 15
Factorial of 1 is 1
Factorial of 3 is 6
Factorial of 5 is 120
Factorial of 7 is 5040
Factorial of 9 is 362880
Factorial of 15 is 1307674368000
```

# Outline

- Recap of Linux 101
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Topics Preview

# Advanced Topics Preview

- Text processing commands
  - `grep` & `egrep`
  - `sed`
  - `awk`
- Regular expression (RegEx)

# grep & egrep

- `grep` is a Unix utility that searches through either information piped to it or files in the current directory.
- `egrep` is extended grep, same as `grep -E`
- Use `zgrep` for compressed files.
- Usage:

```
grep <options> <search pattern> <files>
```

# sed

- sed ("stream editor") is Unix utility for parsing and transforming text files.
- sed is line-oriented
  - It operates one line at a time and allows regular expression matching and substitution.

# awk

- The awk text-processing language is useful for such tasks as:
  - Tallying information from text files and creating reports from the results.
  - Adding additional functions to text editors like "vi".
  - Translating files from one format to another.
  - Creating small databases.
  - Performing mathematical operations on files of numeric data.
- awk has two faces:
  - It is a utility for performing simple text-processing tasks, and
  - It is a programming language for performing complex text-processing tasks.

# Further Reading

- BASH Programming

  `http://tldp.org/HOWTO/bash-Prog-Intro-HOWTO.html`
- CSH Programming
  `http://www.grymoire.com/Unix/Csh.html`
- csh Programming Considered Harmful

  `http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/`
- Wiki Books
  `http://en.wikibooks.org/wiki/Subject:Computing`

# Next Tutorial - Distributed Job Execution

- If any of the following fits you, then you might want come
  - I have to run more than one serial job.
  - I don't want to submit multiple job using the serial queue
  - How do I submit one job which can run multiple serial jobs?
- Date: Feb 25th, 2015

# Getting Help

- User Guides
  - LSU HPC: http://www.hpc.lsu.edu/docs/guides.php#hpc
  - LONI:http://www.hpc.lsu.edu/docs/guides.php#loni
- Documentation: http://www.hpc.lsu.edu/docs
- Online courses: http://moodle.hpc.lsu.edu
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900
  - Instant Messenger (AIM, Yahoo Messenger, Google Talk)
    - Add “lsuhpchelp”

# Questions?

# Exercises

1. Write a shell script to
   - Print “Hello world!” to the screen
   - Use a variable to store the greeting
2. Write a shell script to
   - Take two integers on the command line as arguments
   - Print the sum, different, product of those two integers
   - Think: what if there are too few or too many arguments? How can you check that?
3. Write a shell script to read your first and last name to an array
   - Add your salutation and suffix to the array
   - Drop either the salutation or suffix
   - Print the array after each of the three steps above
4. Write a shell script to calculate the factorial and double factorial of an integer or list of integers