
# Shell Scripting

## Variables, Arrays & Control Constructs

Alexander B. Pacheco
LTS Research Computing
September 29 2015

# Outline

# Introduction

## What is a SHELL

- The command line interface is the primary interface to Linux/Unix operating systems.
- Shells are how command-line interfaces are implemented in Linux/Unix.
- Each shell has varying capabilities and features and the user should choose the shell that best suits their needs.
- The shell is simply an application running on top of the kernel and provides a powerful interface to the system.

# Types 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 (sh).
- 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

| | sh | csh | ksh | bash | tcsh |
|---|---|---|---|---|---|
| Programming Language | ✓ | ✓ | ✓ | ✓ | ✓ |
| Shell Variables | ✓ | ✓ | ✓ | ✓ | ✓ |
| Command alias | ✗ | ✓ | ✓ | ✓ | ✓ |
| Command history | ✗ | ✓ | ✓ | ✓ | ✓ |
| Filename completion | ✗ | ★ | ★ | ✓ | ✓ |
| Command line editing | ✗ | ✗ | ★ | ✓ | ✓ |
| Job control | ✗ | ✓ | ✓ | ✓ | ✓ |

✓ : Yes

✗ : No

★ : Yes, not set by default

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

# Variables I

- A variable is a named object that contains data used by one or more applications.
- There are two types of variables, Environment and User Defined and can contain a number, character or a string of characters.
- Environment Variables provides a simple way to share configuration settings between multiple applications and processes in Linux.
- As in programming languages like C, C++ and Fortran, defining your own variables makes the program or script extensible by you or a third party
- Rules for Variable Names
  1. Variable names must start with a letter or underscore
  2. Number can be used anywhere else
  3. DO NOT USE special characters such as `@, #, %, $`
  4. Case sensitive
  5. Examples
     - Allowed: VARIABLE, VAR1234able, var_name, _VAR
     - Not Allowed: 1VARIABLE, %NAME, $myvar, VAR@NAME
- To reference a variable, environment or user defined, you need to prepend the variable name with "$" as in $VARIABLE, $PATH, etc.

# Variables II

- Its a good practice to protect your variable name within {...} such as ${PATH} when referencing it. (We'll see an example in a few slides)
- Assigning value to a variable

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

- **sh,ksh,bash** THERE IS NO SPACE ON EITHER SIDE OF =
- **csh,tcsh** space on either side of = is allowed for the `set` command
- **csh,tcsh** There is no = in the `setenv` command

# File Permissions I

- In *NIX OS's, you have three types of file permissions
  1. read (r)
  2. write (w)
  3. execute (x)
- for three types of users
  1. user
  2. group
  3. world i.e. everyone else who has access to the system

```
drwxr-xr-x.  2  user  user  4096  Jan  28  08:27  Public
-rw-rw-r--.  1  user  user  3047  Jan  28  09:34  README
```

- The first character signifies the type of the file

  `d` for directory

  `l` for symbolic link

  `-` for normal file
- The next three characters of first triad signifies what the owner can do
- The second triad signifies what group member can do

# File Permissions II

- The third triad signifies what everyone else can do

$$d\underbrace{rwx}_{u}\overbrace{r-x}^{g}\underbrace{r-x}_{o}$$

- Read carries a weight of 4
- Write carries a weight of 2
- Execute carries a weight of 1
- The weights are added to give a value of 7 (rwx), 6(rw), 5(rx) or 3(wx) permissions.
- `chmod` is a *NIX command to change permissions on a file
- To give user rwx, group rx and world x permission, the command is

  `chmod 751 filename`
- Instead of using numerical permissions you can also use symbolic mode

| | |
|---|---|
| u/g/o or a | user/group/world or all i.e. ugo |
| +/- | Add/remove permission |
| r/w/x | read/write/execute |

# File Permissions III

- Give everyone execute permission:

  ```
  chmod a+x hello.sh
  ```

  ```
  chmod ugo+x hello.sh
  ```

- Remove group and world read & write permission:

  ```
  chmod go-rw hello.sh
  ```

- Use the `-R` flag to change permissions recursively, all files and directories and their contents.

  ```
  chmod -R 755 ${HOME}/*
  ```

  What is the permission on ${HOME}?

# Input/Output I

- The command `echo` is used for displaying output to screen
- For reading input from screen/keyboard/prompt

  **bash** `read`

  **tcsh** `$<`
- The `read` statement takes all characters typed until the Enter key is pressed and stores them into a variable.

  Syntax `read <variable name>`

  Example `read name` Enter

  *Alex Pacheco*
- `$<` can accept only one argument. If you have multiple arguments, enclose the `$<` within quotes e.g. `"$<"`

  Syntax: `set <variable> = $<`

  Example: `set name = "$<"` Enter

  *Alex Pacheco*
- In the above examples, the name that you enter in stored in the variable **`name`**.
- Use the `echo` command to print the variable **`name`** to the screen

# Input/Output II

`echo $name` Enter

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

`echo Welcome to HPC  Training`↵ (more than one space between HPC and Training)

`echo "Welcome to HPC  Training"`↵

`read name`↵ or `set name = "$<"`↵

*Alex  Pacheco*↵

`echo $name`↵

`echo "$name"`↵

# Input/Output III

- You can also use the **printf** command to display output

  Syntax: `printf <format> <arguments>`

  Example: `printf "$name"`↵

  `printf "%s\n" "$name"`↵

- Format Descriptors

  | | |
  |---|---|
  | %s | print argument as a string |
  | %d | print argument as an integer |
  | %f | print argument as a floating point number |
  | \n | print new line |

  you can add a width for the argument between the % and {s,d,f} fields
  %4s, %5d, %7.4f

- The **printf** command is used in **awk** to print formatted data (more on this later)

# I/O Redirection

- There are three file descriptors for I/O streams
  1. STDIN: Standard Input
  2. STDOUT: Standard Output
  3. STDERR: Standard Error
- 1 represents STDOUT and 2 represents STDOUT
- 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.
- Examples:
  1. write STDOUT to file: `ls -l > ls-l.out`
  2. write STDERR to file: `ls -l 2> ls-l.err`
  3. write STDOUT to STDERR: `ls -l 1>&2`
  4. write STDERR to STDOUT: `ls -l 2>&1`
  5. send STDOUT as STDIN: `ls -l | wc -l`

# Shell Scripting Basics

# What is a scripting Language?

- A **scripting language** or **script language** is a *programming language* that supports the writing of **scripts**.
- **Scripting Languages** provide a higher level of abstraction than standard programming languages.
- Compared to programming languages, scripting languages do not distinguish between data types: integers, real values, strings, etc.
- Scripting Languages tend to be good for automating the execution of other programs.
  - analyzing data
  - running daily backups
- They are also good for writing a program that is going to be used only once and then discarded.
- 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.
- The majority of script programs are "quick and dirty", where the main goal is to get the program written quickly.

# Writing your first script

## Three things to do to write and execute a script

1. Write a script
   - A shell script is a file that contains ASCII text.
   - Create a file, **hello.sh** with the following lines

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

2. Set permissions

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

OR

```
~/Tutorials/BASH/scripts> chmod a+x hello.sh
```

3. Execute the script

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

4. If you do not set execute permission for the script, then

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

# Description of the script

- My First Script

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

- The first line is called the "ShaBang" line. It tells the OS which interpreter to use. In the current example, bash
- Other options are:
  - `sh  :  #!/bin/sh`
  - `ksh :  #!/bin/ksh`
  - `csh :  #!/bin/csh`
  - `tcsh:  #!/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

| | |
|---|---|
| # | starts a comment. |
| $ | indicates the name of a variable. |
| \ | escape character to display next character literally. |
| { } | used to enclose name of variable. |
| ; | Command separator [semicolon]. Permits putting two or more commands on the same line. |
| ;; | Terminator in a case option [double semicolon]. |
| . | "dot" command [period]. Equivalent to source. This is a bash builtin. |
| $? | 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 (”$”, ”/” and ”‘”)
  - Example: `echo "$myvar"` prints the value of `myvar`
- Single Quotation `' '`
  - Enclosed string is read literally
  - Example: `echo '$myvar'` prints `$myvar`
- Back Quotation ` ` `
  - Used for command substitution
  - Enclosed string is executed as a command
  - Example: `echo ` `pwd` ` ` prints the output of the `pwd` command i.e. print working directory
  - In **bash**, you can also use `$(···)` instead of ` `···` `
    e.g. `$(pwd)` and ` `pwd` ` are the same

# Example

```
#!/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>
```

# Beyond Basic Shell Scripting

# Arithmetic Operations I

- You can carry out numeric operations on integer variables

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

- Arithmetic operations in **bash** can be done within the `$((···))` or `$[···]` commands
  - ★ Add two numbers: `$((1+2))`
  - ★ Multiply two numbers: `$[$a*$b]`
  - ★ You can also use the `let` command: `let c=$a-$b`
  - ★ or use the `expr` command: `c=‘expr $a - $b‘`

# Arithmetic Operations II

- In **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‘`
- Note the use of space

**bash** space required around operator in the `expr` command

**tcsh** space required between @ and variable, around = and numeric operators.

- You can also use C-style increment operators

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

**tcsh** `@ x -= 1` or `@ x++`

`/=`, `*=` and `%=` are also allowed.

**bash**

- The above examples only work for integers.
- What about floating point number?

# Arithmetic Operations III

- Using floating point in **bash** or **tcsh** scripts requires an external calculator like 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"
    ```
- You can also use **awk** for floating point arithmetic.

# Arrays I

- **bash** and **tcsh** supports one-dimensional arrays.
- Array elements may be initialized with the `variable[xx]` notation

  `variable[xx]=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 II

- 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 **tcsh** everything within "..." is one variable.
- 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 III

- 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-]})
```

**bash** the first array index is zero (0)

**tcsh** the first array index is one (1)

# Arrays IV

## 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, Fortran, etc.
- Control constructs can change the sequential order of commands.
- Control constructs available in **bash** and **tcsh** are
  1. Conditionals: `if`
  2. Loops: `for, while, until`
  3. Switches: `case, switch`

# `if` statement

- An **`if/then`** construct tests whether the exit status of a list of commands is 0, and if so, executes 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 "[" "]"
- **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.

# Comparison Operators

| Integer Comparison | | |
|---|---|---|
| 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 Comparison | | |
|---|---|---|
| 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)` |

# File Test & Logical Operators

| File Test Operators | | |
|---|---|---|
| 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)` |

| Logical Operators | | |
|---|---|---|
| Operation | **bash** | **tcsh** |
| Operation | **bash** | **tcsh** |
| NOT | `if [ ! -e .bashrc ]` | `if ( ! -z .tcshrc)` |
| AND | `if [ $a -eq 2 ] && [ $x -gt $y ]` | `if ($a == 2 && $x <= $y )` |
| OR | `if [[ $a -eq 2 \|\| $x -gt $y ]]` | `if ($a == 2 \|\| $x <= $y )` |

# Examples

- Condition tests using the **if/then** may be nested

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

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

- This is same as

```
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 true.
- Loop constructs available in

**bash**: `for`, `while` and `until`

**tcsh**: `foreach` and `while`

# bash: for loops

- 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 (( EXP1; EXP2; EXP3 )); do
  some commands
done
```

```
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
```

# tcsh: foreach loop

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

```
foreach arg (list)
  some commands
end
```

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

# while Construct

- The **while** construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).
- 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
```

## factorial.sh

```
#!/bin/bash

echo -n "Enter a number less than 10: "
read counter
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

## factorial.csh

```
#!/bin/tcsh

echo -n "Enter a number less than 10: "
set counter = $<
set factorial = 1
while ( $counter > 0 )
   @ factorial = $factorial * $counter
   @ counter -= 1
end
echo $factorial
```

# until Contruct (bash only)

- 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

echo -n "Enter a number less than 10: "
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
```

# Nested Loops

- `for, while & until` loops can 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 "========================"
```

# Switching or Branching Constructs I

- 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 or Branching Constructs II

- `tcsh` has the `switch` construct

## `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
```

# dooper1.sh

```
#!/bin/bash

echo "Print two numbers"
read num1 num2
echo "What operation do you want to do?"
echo "Options are add, subtract, multiply,
      exponentiate, divide, modulo and all"
read oper

  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
```

```
~/Tutorials/BASH/scripts>./day1/examples/dooper1.sh
Print two numbers
2 5
What operation do you want to do?
Options are add, subtract, multiply, exponentiate,
      divide, modulo and all
all
2 + 5 = 7
2 - 5 = -3
2 * 5 = 10
2 ** 5 = 32
2 / 5 = 0
2 % 5 = 2
```

# Command Line Arguments

- Similar to programming languages, `bash` (and other shell scripting languages) can also take command line arguments
  - `./scriptname arg1 arg2 arg3 arg4 ...`
  - `$0,$1,$2,$3, etc`: positional parameters corresponding to `./scriptname,arg1,arg2,arg3,arg4,...` respectively
  - `$#`: 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,tcsh`
  - an array `argv` contains the list of arguments with `argv[0]` set to name of 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
```

```
dyn100085:examples apacheco$ ./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
```

```
dyn100085:examples apacheco$ ./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 variable

  Syntax:

  ```
  declare -r var
  ```

  ```
  declare -r varName=value
  ```

- Create an integer variable

  Syntax:

  ```
  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 I

- Like "real" programming languages, **bash** has functions.
- A function is a subroutine, 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
}
```

# Functions II

## 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: " $#
```

```
dyn100085:examples apacheco$./shift10.sh
USAGE:./shift10.sh [atleast 11 arguments]
dyn100085:examples apacheco$./shift10.sh $(seq 1 10)
USAGE:./shift10.sh [atleast 11 arguments]
dyn100085:examples apacheco$./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
dyn100085:examples apacheco$./shift10.sh $(seq 21 2 44)
Number of Arguments:  12
List of Arguments:  21 23 25 27 29 31 33 35 37 39 41 43
Name of script that you are running: ./shift10.sh
Command You Entered:./shift10.sh 21 23 25 27 29 31 33 35
      37 39 41 43
First Argument 21
Tenth and Eleventh argument 210 211 39 41
Argument List is:  21 23 25 27 29 31 33 35 37 39 41 43
Number of Arguments:  12
Argument List is:  39 41 43
Number of Arguments:  3
```

# Functions III

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

  Syntax:

  ```
  local var=value
  ```

  ```
  local varName
  ```

# Functions IV

- 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
```

```
dyn100085:examples apacheco$./factorial3.sh $(seq 1 2 11)
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 11 is 39916800
```

# Scripting for Job Submission

## Problem Description

- I have to run more than one serial job.
  - Solution: Create a script that will submit and run multiple serial jobs.
- I don't want to submit multiple jobs using the serial queue since
  - Cluster Admins give lower priority to jobs that are not parallelized
  - The number of jobs that I want to run exceed the maximum number of jobs that I can run simultaneously
- How do I submit *one* job which can run multiple serial jobs?

## One Solution of many

- Write a script which will log into all unique nodes and run your serial jobs in background.
- Easy said than done
- What do you need to know?
  1. Shell Scripting
  2. How to run a job in background
  3. Know what the `wait` command does

```
[alp514@corona1 ~]$ cat checknodes.pbs
#!/bin/bash
#
#PBS -q normal
#PBS -l nodes=4:ppn=16
#PBS -l walltime=00:30:00
#PBS -V
#PBS -o nodetest.out
#PBS -e nodetest.err
#PBS -N testing
#PBS -M alp514@lehigh.edu
#PBS -m abe
#

export WORK_DIR=$PBS_O_WORKDIR
export NPROCS=`wc -l $PBS_NODEFILE |gawk '//{print $1}'`

NODES=(`cat ``$PBS_NODEFILE``` )
UNODES=(`uniq ``$PBS_NODEFILE``` )

echo ``Nodes Available: `` ${NODES[@]}
echo ``Unique Nodes Available: `` ${UNODES[@]}

echo ``Get Hostnames for all processes''
i=0
for nodes in ``${NODES[@]}''; do
  ssh -n $nodes 'echo $HOSTNAME '$i' ' &
  let i=i+1
done
wait

echo ``Get Hostnames for all unique nodes''
i=0
NPROCS=`uniq $PBS_NODEFILE | wc -l |gawk '//{print $1}'`
let NPROCS-=1
while [ $i -le $NPROCS ] ; do
  ssh -n ${UNODES[$i]} 'echo $HOSTNAME '$i' '
  let i=i+1
done

[alp514@corona1 ~]$ qsub checknodes.pbs
688825.corona1.cc.lehigh.edu
```

```
[alp514@corona1 ~]$ cat nodetest.out
Wed Mar 11 08:20:40 EDT 2015 : erasing contents of corona63:/scratch
Wed Mar 11 08:20:40 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:41 EDT 2015 : swap reset
Wed Mar 11 08:20:41 EDT 2015 : erasing contents of corona56:/scratch
Wed Mar 11 08:20:41 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:41 EDT 2015 : swap reset
Wed Mar 11 08:20:41 EDT 2015 : erasing contents of corona50:/scratch
Wed Mar 11 08:20:42 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:42 EDT 2015 : swap reset
Wed Mar 11 08:20:43 EDT 2015 : erasing contents of corona27:/scratch
Wed Mar 11 08:20:43 EDT 2015 : /scratch erased, resetting swap
swapon on /dev/sda6
Wed Mar 11 08:20:43 EDT 2015 : swap reset
Nodes Available:  corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63 corona63
       corona63 corona63 corona63 corona63 corona63 corona63 corona56 corona56 c
orona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56 corona56
      corona56 corona56 corona50 corona50 corona50 corona50 corona50 corona50 co
rona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona50 corona27 corona27
      corona27 corona27 corona27 corona27 corona27 corona27 corona27 corona27 cor
ona27 corona27 corona27 corona27 corona27 corona27
Unique Nodes Available:  corona63 corona56 corona50 corona27
Get Hostnames for all processes
corona27 52
corona27 59
corona27 60
corona27 57
corona27 51
corona27 62
corona27 54
corona27 48
corona27 63
corona27 58
corona27 53
corona50 43
```

```
corona50 40
corona50 38
corona50 33
corona50 34
corona50 47
corona56 31
corona63 13
corona63 6
corona56 22
corona63 9
corona63 14
corona56 16
corona56 25
corona56 23
corona56 17
corona63 5
corona63 10
corona63 8
corona63 15
corona63 3
corona50 32
corona50 44
corona56 18
corona50 36
corona50 46
corona56 27
corona50 42
corona63 1
corona63 12
corona50 45
corona50 41
corona50 35
corona56 29
corona63 7
corona56 28
corona63 11
corona56 26
corona56 21
corona63 4
```

```
corona56 24
corona63 0
corona56 20
Get Hostnames for all unique nodes
corona63 0
corona56 1
corona50 2
corona27 3
```

# Wrap Up

# References & 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`

# Hands-On Exercises

# Exercises

1. Create shell scripts to do the following
   - Write a simple hello world script
   - Modify the above script to use a variable
   - Modify the above script to prompt you for your name and then display your name with a greeting.
2. Write a script to add/subtract/multiply/divide two numbers.
3. Write a 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 script to calculate the factorial and double factorial of an integer or list of integers.

# Solution 1

## hellovariable.sh

```
#!/bin/bash

# Hello World script using a variable
STR="Hello World!"
echo $STR
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      hellovariable.sh
Hello World!
```

## helloname.sh

```
#!/bin/bash

# My Second Script

echo Please Enter your name:
read name1 name2
Greet="Welcome to HPC Training"
echo "Hello $name1 $name2, $Greet"
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      helloname.sh
Please Enter your name:
Alex Pacheco
Hello Alex Pacheco, Welcome to HPC Training
```

# Solution 2

## dosum.sh

```
#!/bin/bash

echo "Enter two integers"
read num1 num2

echo "$num1 + $num2 = " $num1 + $num2
echo "$num1 + $num2 = " $(($num1 + $num2))

let SUM=$num1+$num2
echo "sum of $num1 & $num2 is " $SUM

echo "$num1/$num2 = " $(echo "scale=5;$num1/
      $num2" | bc)
echo "$num2/$num1 = " $(bc -l <<< $num2/$num1)

exit
```

```
~/Tutorials/BASH/scripts/day1/solution> ./dosum.
      sh
Enter two integers
5 7
5 + 7 =  5 + 7
5 + 7 =  12
sum of 5 & 7 is  12
5/7 =  .71428
7/5 =  1.40000000000000000000
```

## doratio.csh

```
#!/bin/tcsh

echo "Enter first integer"
set num1 = $<
set num2 = $<

echo "$num1 / $num2 = " $num1 / $num2

@ RATIO = $num1 / $num2
echo "ratio of $num1 & $num2 is " $RATIO

set ratio=`echo "scale=5 ; $num1/$num2" | bc`
echo "ratio of $num1 & $num2 is " $ratio

exit
```

```
~/Tutorials/BASH/scripts/day1/solution> ./
      doratio.csh
Enter first integer
5
7
5 / 7 =  5 / 7
ratio of 5 & 7 is  0
ratio of 5 & 7 is  .71428
```

# Alternate Solution 2

```
#!/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
```

```
#!/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
```

# Solution 3

## 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/solution> ./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/solution> ./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
```

# Solution 4

## fac2.sh

```
#!/bin/bash

echo "Enter the integer whose factorial and
      double factorial you want to calculate"
read counter
factorial=1
i=$counter
while [ $i -gt 1 ]; do
    factorial=$[ $factorial * $i ]
    let i-=1
done

i=$counter
dfactorial=1
until [ $i -le 2 ]; do
    dfactorial=$[ $dfactorial * $i ]
    let i-=2
done

echo "$counter! = $factorial & $counter!! =
      $dfactorial"
```

## fac2.csh

```
#!/bin/tcsh

echo "Enter the integer whose factorial and
      double factorial you want to calculate"
set counter = $<
@ factorial = 1
@ i = $counter
while ( $i > 1 )
    @ factorial = $factorial * $i
    @ i--
end

@ i = $counter
@ dfactorial = 1
while ( $i >= 1 )
    @ dfactorial = $dfactorial * $i
    @ i = $i - 2
end

echo "$counter\! = $factorial & $counter\!\! =
      $dfactorial"
```

# fac3.sh: Alternate Solution 4 (bash only)

```
#!/bin/bash

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

factorial() {
    local i=$1
    local f
    local type=$2

    declare -i i
    declare -i f

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

if [[ "$#" -eq 0 ]]; then
    usage
else
    for i in $@ ; do
  x=$( factorial $i single )
  y=$( factorial $i double )
  echo "$i! = $x & $i!! = $y"
    done
fi
```